

# **INF05010 - Algoritmos avançados**

## **Notas de aula**

Marcus Ritt  
`mrpritt@inf.ufrgs.br`

7 de Agosto de 2013

Universidade Federal do Rio Grande do Sul  
Instituto de Informática  
Departamento de Informática Teórica



Versão 4736 do 2013-08-07, compilada em 7 de Agosto de 2013. Obra está licenciada sob uma [Licença Creative Commons](#) (Atribuição–Uso Não-Comercial–Não a obras derivadas 3.0 Brasil).



# Conteúdo

<b>1. Algoritmos em grafos</b>	<b>3</b>
1.1. Representação de grafos	3
1.2. Caminhos e ciclos Eulerianos	4
1.3. Filas de prioridade e heaps	6
1.3.1. Heaps binários	9
1.3.2. Heaps binomiais	12
1.3.3. Heaps Fibonacci	17
1.3.4. Rank-pairing heaps	22
1.3.5. Árvores de van Emde Boas	31
1.3.6. Tópicos	40
1.3.7. Exercícios	40
1.4. Fluxos em redes	41
1.4.1. Algoritmo de Ford-Fulkerson	42
1.4.2. O algoritmo de Edmonds-Karp	46
1.4.3. O algoritmo “caminho mais gordo” (“fattest path”)	48
1.4.4. O algoritmo push-relabel	49
1.4.5. Variações do problema	52
1.4.6. Aplicações	56
1.4.7. Outros problemas de fluxo	60
1.4.8. Exercícios	60
1.5. Emparelhamentos	62
1.5.1. Aplicações	65
1.5.2. Grafos bi-partidos	65
1.5.3. Emparelhamentos em grafos não-bipartidos	77
1.5.4. Exercícios	78
<b>2. Tabelas hash</b>	<b>79</b>
2.1. Hashing com listas encadeadas	79
2.2. Hashing com endereçamento aberto	83
2.3. Cuco hashing	85
2.4. Filtros de Bloom	87
<b>3. Algoritmos de aproximação</b>	<b>89</b>
3.1. Problemas, classes e reduções	89

3.2. Medidas de qualidade . . . . .	91
3.3. Técnicas de aproximação . . . . .	91
3.3.1. Algoritmos gulosos . . . . .	91
3.3.2. Aproximações com randomização . . . . .	96
3.4. Esquemas de aproximação . . . . .	98
3.5. Aproximando o problema da árvore de Steiner mínima . . . . .	100
3.6. Aproximando o PCV . . . . .	101
3.7. Aproximando problemas de cortes . . . . .	103
3.8. Aproximando empacotamento unidimensional . . . . .	107
3.8.1. Um esquema de aproximação assintótico para min-EU . . . . .	111
3.9. Aproximando problemas de sequenciamento . . . . .	113
3.9.1. Um esquema de aproximação para $P \parallel C_{\max}$ . . . . .	116
3.10. Exercícios . . . . .	117
<b>4. Algoritmos randomizados</b>	<b>119</b>
4.1. Teoria de complexidade . . . . .	119
4.1.1. Amplificação de probabilidades . . . . .	121
4.1.2. Relação entre as classes . . . . .	122
4.2. Seleção . . . . .	125
4.3. Corte mínimo . . . . .	127
4.4. Teste de primalidade . . . . .	131
4.5. Exercícios . . . . .	135
<b>5. Complexidade e algoritmos parametrizados</b>	<b>137</b>
<b>A. Material auxiliar</b>	<b>141</b>
A.1. Algoritmos . . . . .	141
<b>B. Técnicas para a análise de algoritmos</b>	<b>145</b>

# 1. Algoritmos em grafos

## 1.1. Representação de grafos

Um grafo pode ser representado diretamente de acordo com a sua definição por  $n$  estruturas que representam os vértices,  $m$  estruturas que representam os arcos e ponteiros entre as estruturas. Um vértice possui ponteiros para todo arco incidente saindo ou entrando, e um arco possui ponteiros para o início e término. A representação direta possui várias desvantagens. Por exemplo não temos acesso direto aos vértices para inserir um arco.

Duas representações simples são listas (ou vetores) não-ordenadas de vértices ou arestas. Uma outra representação simples de um grafo  $G$  com  $n$  vértices é uma *matriz de adjacência*  $M = (m_{ij}) \in \mathbb{B}^{n \times n}$ . Para vértices  $u, v$  o elemento  $m_{uv} = 1$  caso existe uma arco entre  $u$  e  $v$ . Para representar grafos não-direcionados mantemos  $m_{uv} = m_{vu}$ , i.e.,  $M$  é simétrico. A representação permite um teste de adjacência em  $O(1)$ . Percorrer todos vizinhos de um dado vértice  $v$  custa  $O(n)$ . O custo alto de espaço de  $\Theta(n^2)$  restringe o uso de uma matriz de adjacência para grafos pequenos<sup>1</sup>.

Uma representação mais eficiente é por *listas* ou *vetores* de adjacência. Neste caso armazenamos para cada vértice os vizinhos em uma lista ou um vetor. As listas ou vetores mesmos podem ser armazenados em uma lista ou um vetor global. Com isso a representação ocupa espaço  $\Theta(n + m)$  para  $m$  arestas.

Uma escolha comum é um vetor de vértices que armazena listas de vizinhos. Essa estrutura permite uma inserção e deleção simples de arcos. Para facilitar a deleção de um vértice em grafos não-direcionados, podemos armazenar junto com o vizinho  $u$  do vértice  $v$  a posição do vizinho  $v$  do vértice  $u$ . A representação dos vizinhos por vetores é mais eficiente, e por isso preferível caso a estrutura do grafo é estático [29, 34].

Caso escolhermos armazenar os vértices em uma lista dupla, que armazena uma lista dupla de vizinhos, em que os vizinhos são representados por posições da primeira lista, obtemos uma *lista dupla de arcos* (ingl. doubly connected arc list, DCAL). Essa estrutura permite uma inserção e remoção tanto de vértices quanto de arcos.

Supõe que  $V = [n]$ . Uma outra representação compacta e eficiente conhecido como *forward star* para grafos estáticos usa um *vetor de arcos*  $a_1, \dots, a_m$ . Mantemos a lista de arestas ordenado pelo começo do arco. Uma permutação  $\sigma$

---

<sup>1</sup> Ainda mais espaço consuma uma *matriz de incidência* entre vértices e arestas em  $\mathbb{B}^{n \times m}$ .

Tabela 1.1.: Operações típicas em grafos.

Operação	Lista de arestas	Lista de vértices	Matriz de adjacência	Lista de adjacência
Inserir aresta	$O(1)$	$O(n + m)$	$O(1)$	$O(1)$ ou $O(n)$
Remover aresta	$O(m)$	$O(n + m)$	$O(1)$	$O(n)$
Inserir vértice	$O(1)$	$O(1)$	$O(n^2)$	$O(1)$
Remover vértice	$O(m)$	$O(n + m)$	$O(n^2)$	$O(n + m)$
Teste $uv \in E$	$O(m)$	$O(n + m)$	$O(1)$	$O(\Delta)$
Percorrer vizinhos	$O(m)$	$O(\Delta)$	$O(n)$	$O(\Delta)$
Grau de um vértice	$O(m)$	$O(\Delta)$	$O(n)$	$O(1)$

nos dá as arestas em ordem do término. (O uso de uma permutação serve para reduzir o consumo de memória.) Para percorrer eficientemente os vizinhos de um vértice armazenamos o índice  $s_v$  do primeiro arco sainte na lista de arestas ordenado pelo começo e o índice  $e_v$  do primeiro arco entrante na lista de arestas ordenado pelo término com  $s_{n+1} = e_{n+1} = m + 1$  por definição. Com isso temos  $N^+(v) = \{a_{s_v}, \dots, a_{s_{v+1}-1}\}$  com  $\delta_v^+ = s_{v+1} - s_v$ , e  $N^-(v) = \{a_{\sigma(e_v)}, \dots, a_{\sigma(e_{v+1}-1)}\}$  com  $\delta_v^- = e_{v+1} - e_v$ . A representação precisa espaço  $O(n + m)$ .

A tabela 1.1 compara a complexidade de operações típicas nas diferentes representações.

## 1.2. Caminhos e ciclos Eulerianos

Um *caminho Euleriano* passa por toda arestas de grafo exatamente uma vez. Um caminho Euleriano fechado é um ciclo Euleriano. Um grafo é *Euleriano* caso ele possui um ciclo Euleriano que passa por cada vértice (pelo menos uma vez).

### Proposição 1.1

Uma grafo não-direcionado  $G = (V, E)$  é Euleriano sse  $G$  é conectado e cada vértice tem grau par.

**Prova.** Por indução sobre o número de arestas. A base da indução é um grafo com um vértice e nenhuma aresta que satisfaz a proposição. Supõe que os grafos com  $\leq m$  arestas satisfazem a proposição e temos um grafo  $G$  com  $m + 1$  arestas. Começa por um vértice  $v$  arbitrário e procura um caminho que nunca passa duas vezes por uma aresta até voltar para  $v$ . Isso sempre é possível porque o grau de cada vértice é par: entrando num vértice sempre podemos sair. Removendo este caminho do grafo, obtemos uma coleção de componentes



conectados com menos que  $m$  arestas, e pela hipótese da indução existem ciclos Eulerianos em cada componente. Podemos obter um ciclo Euleriano para o grafo original pela concatenação desses ciclos Eulerianos. ■

Pela prova temos o seguinte algoritmo com complexidade  $O(|E|)$  para encontrar um ciclo Euleriano na componente de  $G = (V, E)$  que contém  $v \in V$ :

```

1   Euler( $G = (V, E), v \in V$ ) :=
2   if  $|E| = 0$  return  $v$ 
3   procura um caminho começando em  $v$ 
4   sem repetir arestas voltando para  $v$ 
5   seja  $v = v_1, v_2, \dots, v_n = v$  esse caminho
6   remove as arestas  $v_1v_2, v_2v_3, \dots, v_{n-1}v_n$  de  $G$ 
7   para obter  $G_1$ 
8   return Euler( $G_1, v_1$ ) + Euler( $G_2, v_2$ ) +  $\dots$  + Euler( $G_{n-1}, v_{n-1}$ ) +  $v_n$ 
9   // Usamos + para concatenação de caminhos.
10  //  $G_i$  é  $G_{i-1}$  com as arestas do
11  // caminho Euler( $G_{i-1}, v_{i-1}$ ) removidos, i.e
12  //  $G_i := (V, E(G_{i-1}) \setminus E(\text{Euler}(G_{i-1}, v_{i-1})))$ 

```

### 1.3. Filas de prioridade e heaps

Uma fila de prioridade mantém um conjunto de chaves com prioridades de forma que a atualizar prioridades e acessar o elemento de menor prioridade é eficiente. Ela possui aplicações em algoritmos para calcular árvores geradoras mínimas, caminhos mais curtos de um vértice para todos outros (algoritmo de Dijkstra) e em algoritmos de ordenação (heapsort).

#### Exemplo 1.1

Árvore geradora mínima através do algoritmo de Prim.

##### Algoritmo 1.1 (Árvore geradora mínima)

**Entrada** Um grafo conexo não-orientado ponderado  $G = (V, E, c)$

**Saída** Uma árvore  $T \subseteq E$  de menor custo total.

```

1   $V' := \{v_0\}$  para um  $v_0 \in V$ 
2   $T := \emptyset$ 
3  while  $V' \neq V$  do
4      escolha  $e = \{u, v\}$  com custo mínimo
5          entre  $V'$  e  $V \setminus V'$  (com  $u \in V', v \in V \setminus V'$ )
6       $V' := V' \cup \{v\}$ 
7       $T := T \cup \{e\}$ 
8  end while
```

##### Algoritmo 1.2 (Prim refinado)

Implementação mais concreta:

```

1   $T := \emptyset$ 
2  for  $u \in V \setminus \{v\}$  do
3      if  $u \in N(v)$  then
4           $\text{value}(u) := c_{uv}$ 
5           $\text{pred}(u) := v$ 
6      else
7           $\text{value}(u) := \infty$ 
8      end if
9       $\text{insert}(Q, (\text{value}(u), u))$  { pares (chave, elemento) }
10 end for
11 while  $Q \neq \emptyset$  do
12      $v := \text{deletemin}(Q)$ 
```

```

13   T := T ∪ {pred(v)v}
14   for u ∈ N(v) do
15     if u ∈ Q e cvu < value(u) then
16       value(u) := cuv
17       pred(u) := v
18       update(Q, u, cvu)
19     end if
20   end for
21 end while

```

Custo?  $n \times \text{insert} + n \times \text{deletemin} + m \times \text{update}$ .

◇

### Observação 1.1

Implementação com vetor de distâncias:  $\text{insert} = O(1)$ <sup>2</sup>,  $\text{deletemin} = O(n)$ ,  $\text{update} = O(1)$ , e temos custo  $O(n + n^2 + m) = O(n^2 + m)$ . Isso é assintoticamente ótimo para grafos densos, i.e.  $m = \Omega(n^2)$ .

◇

### Observação 1.2

Implementação com lista ordenada:  $\text{insert} = O(n)$ ,  $\text{deletemin} = O(1)$ ,  $\text{update} = O(n)$ , e temos custo  $O(n^2 + n + mn) = O(mn)$ <sup>3</sup>.

◇

### Observação 1.3

Implementação com uma lista de  $\sqrt{n}$  blocos de  $\sqrt{n}$  elementos,  $\text{insert}$ ,  $\text{deletemin}$  e  $\text{update}$  podem ser implementados em tempo  $O(\sqrt{n})$ , logo o algoritmo de Prim e de Dijkstra tem complexidade  $O(m\sqrt{n})$ .

◇

### Exemplo 1.2

Caminhos mais curtos com o algoritmo de Dijkstra

#### Algoritmo 1.3 (Dijkstra)

**Entrada** Grafo não-direcionado  $G = (V, E)$  com pesos  $c_e \geq 0$  nas arestas  $e \in E$ , e um vértice  $s \in V$ .

**Saída** A distância mínima  $d_v$  entre  $s$  e cada vértice  $v \in V$ .

```

1   ds := 0; dv := ∞, ∀v ∈ V \ {s}
2   visited(v) := false, ∀v ∈ V

```

<sup>2</sup>Com chaves compactas  $[1, n]$ .

<sup>3</sup>Na hipótese razoável que  $m \geq n$

## 1. Algoritmos em grafos

```

3   Q := ∅
4   insert(Q, (s, 0))
5   while Q ≠ ∅ do
6       v := deletemin(Q)
7       visited(v) := true
8       for u ∈ N(v) do
9           if not visited(u) then
10              if du = ∞ then
11                  du := dv + dvu
12                  insert(Q, (u, du))
13              else
14                  du := min(dv + dvu, du)
15                  update(Q, (u, du))
16              end if
17          end if
18      end for
19  end while
```

A fila de prioridade contém pares de vértices e distâncias.

### Proposição 1.2

O algoritmo de Dijkstra possui complexidade

$$O(n) + n \times \text{deletemin} + n \times \text{insert} + m \times \text{update}.$$

**Prova.** O pré-processamento (1-3) tem custo  $O(n)$ . O laço principal é dominado por no máximo  $n$  operações insert,  $n$  operações deletemin, e  $m$  operações update. A complexidade concreta depende da implementação dessas operações. ■

### Proposição 1.3

O algoritmo de Dijkstra é correto.

**Prova.** Seja  $\text{dist}(s, x)$  a menor distância entre  $s$  e  $x$ . Provaremos por indução que para cada vértice  $v$  selecionado na linha 6 do algoritmo  $d_v = \text{dist}(s, v)$ . Como base isso é correto para  $v = s$ . Seja  $v \neq s$  um vértice selecionado na linha 6, e supõe que existe um caminho  $P = s \cdots xy \cdots v$  de comprimento menor que  $d_v$ , tal que  $y$  é o primeiro vértice que não foi processado (i.e. selecionado na

linha 6) ainda. (É possível que  $y = v$ .) Sabemos que

$$\begin{array}{ll}
 d_y \leq d_x + d_{xy} & \text{porque } x \text{ já foi processado} \\
 = \text{dist}(s, x) + d_{xy} & \text{pela hipótese } d_x = \text{dist}(s, x) \\
 \leq d(P) & d_P(s, x) \geq \text{dist}(s, x) \text{ e } P \text{ passa por } xy \\
 < d_v, & \text{pela hipótese}
 \end{array}$$

uma contradição com a minimalidade do elemento extraído na linha 6. (Notação:  $d(P)$ : distância total do caminho  $P$ ;  $d_P(s, x)$ : distância entre  $s$  e  $x$  no caminho  $P$ .) ■  $\diamond$

#### Observação 1.4

Podemos ordenar  $n$  elementos usando um heap com  $n$  operações “insert” e  $n$  operações “deletemin”. Pelo limite de  $\Omega(n \log n)$  para ordenação via comparação, podemos concluir que o custo de “insert” mais “deletemin” é  $\Omega(\log n)$ . Portanto, pelo menos uma das operações é  $\Omega(\log n)$ .  $\diamond$

### 1.3.1. Heaps binários

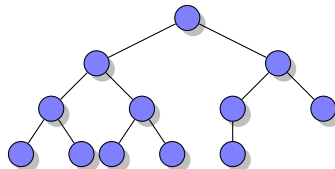
#### Teorema 1.1 (Williams [41])

Uma fila de prioridade pode ser implementado com custo  $\text{insert} = O(\log n)$ ,  $\text{deletemin} = O(\log n)$ ,  $\text{update} = O(\log n)$ . Portanto, uma árvore geradora mínima pode ser calculado em tempo  $O(n \log n + m \log n)$ .

Um *heap* é uma árvore com chaves nos vértices que satisfazem um critério de ordenação.

- *min-heap*: as chaves dos filhos são maior ou igual que a chave do pai;
- *max-heap*: as chaves dos filhos são menor ou igual que a chave do pai.

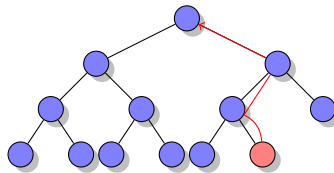
Um *heap* binário é um heap em que cada vértice possui no máximo dois filhos. Implementaremos uma fila de prioridade com um heap binário *completo*. Um heap completo fica organizado de forma que possui folhas somente no último nível, da esquerda para direita. Isso garante uma altura de  $O(\log n)$ .



## 1. Algoritmos em grafos

Positivo: Achar a chave com valor mínimo (operação findmin) custa  $O(1)$ . Como implementar a inserção? Idéia: Colocar na última posição e restabelecer a propriedade do min-heap, caso a chave é menor que a do pai.

```
1  insert(H,c) :=
2    insere c na última posição p
3    heapify up(H,p)
4
5  heapify up(H,p) :=
6    if root(p) return
7    if key(parent(p)) > key(p) then
8      swap(key(parent(p)), key(p))
9      heapify up(H, parent(p))
10   end if
```



### Lema 1.1

Seja  $T$  um min-heap. Decremente a chave do nó  $p$ . Após heapify-up( $T, P$ ) temos novamente um min-heap. A operação custa  $O(\log n)$ .

**Prova.** Por indução sobre a profundidade  $k$  de  $p$ . Caso  $k = 1$ :  $p$  é a raiz, após o decremento já temos um min-heap e heapify-up não altera ele. Caso  $k > 1$ : Seja  $c$  a nova chave de  $p$  e  $d$  a chave de  $\text{parent}(p)$ . Caso  $d \leq c$  já temos um min-heap e heapify-up não altera ele. Caso  $d > c$  heapify-up troca  $c$  e  $d$  e chama heapify-up( $T, \text{parent}(p)$ ) recursivamente. Podemos separar a troca em dois passos: (i) copia  $d$  para  $p$ . (ii) copia  $c$  para  $\text{parent}(p)$ . Após passo (i) temos um min-heap  $T'$  e passo (ii) diminui a chave de  $\text{parent}(p)$  e como a profundidade de  $\text{parent}(p)$  é  $k - 1$  obtemos um min-heap após da chamada recursiva, pela hipótese da indução.

Como a profundidade de  $T$  é  $O(\log n)$ , o número de chamadas recursivas também, e como cada chamada tem complexidade  $O(1)$ , heapify-up tem complexidade  $O(\log n)$ . ■

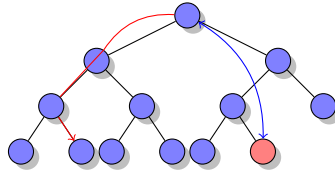
Como remover? A idéia básica é a mesma: troca a chave com o menor filho. Para manter o heap completo, colocaremos primeiro a chave da última posição na posição do elemento removido.

```
1  delete(H,p) :=
```

```

2   troca última posição com p
3   heapify down(H,p)
4
5   heapify down(H,p):=
6   if (p não possui filhos) return
7   if (p possui um filho) then
8       if  $\text{key}(\text{left}(p)) < \text{key}(p)$  then swap( $\text{key}(\text{left}(p))$ ,  $\text{key}(p)$ )
9   end if
10  { p possui dois filhos }
11  if  $\text{key}(p) > \text{key}(\text{left}(p))$  or  $\text{key}(p) > \text{key}(\text{right}(p))$  then
12      if  $\text{key}(\text{left}(p)) < \text{key}(\text{right}(p))$  then
13          swap( $\text{key}(\text{left}(p))$ ,  $\text{key}(p)$ )
14          heapify down(H, left(p))
15      else
16          swap( $\text{key}(\text{right}(p))$ ,  $\text{key}(p)$ )
17          heapify down(H, right(p))
18      end if
19  end if

```



### Lema 1.2

Seja  $T$  um min-heap. Incremente a chave do nó  $p$ . Após  $\text{heapify-down}(T, p)$  temos novamente um min-heap. A operação custa  $O(\log n)$ .

**Prova.** Por indução sobre a altura  $k$  de  $p$ . Caso  $k = 1$ ,  $p$  é uma folha e após o incremento já temos um min-heap e  $\text{heapify-down}$  não altera ele. Caso  $k > 1$ : Seja  $c$  a nova chave de  $p$  e  $d$  a chave do menor filho  $f$ . Caso  $c \leq d$  já temos um min-heap e  $\text{heapify-down}$  não altera ele. Caso  $c > d$   $\text{heapify-down}$  troca  $c$  e  $d$  e chama  $\text{heapify-down}(T, f)$  recursivamente. Podemos separar a troca em dois passos: (i) copia  $d$  para  $p$ . (ii) copia  $c$  para  $f$ . Após passo (i) temos um min-heap  $T'$  e passo (ii) aumenta a chave de  $f$  e como a altura de  $f$  é  $k - 1$ , obtemos um min-heap após da chamada recursiva, pela hipótese da indução. Como a altura de  $T$  é  $O(\log n)$  o número de chamadas recursivas também, e como a cada chamada tem complexidade  $O(1)$ ,  $\text{heapify-up}$  tem complexidade  $O(\log n)$ . ■

Última operação: atualizar a chave.

## 1. Algoritmos em grafos

```
1  update(H, p, v) :=
2    if v < key(p) then
3      key(p) := v
4      heapify up(H, p)
5    else
6      key(p) := v
7      heapify down(H, p)
8    end if
```

**Sobre a implementação** Uma árvore binária completa pode ser armazenado em um vetor  $v$  que contém as chaves. Um pontador  $p$  a um elemento é simplesmente o índice no vetor. Caso o vetor contém  $n$  elementos e possui índices a partir de 0 podemos definir

```
1  root(p) := return p = 0
2  parent(p) := return  $\lfloor (p-1)/2 \rfloor$ 
3  key(p) := return  $v[p]$ 
4  left(p) := return  $2p+1$ 
5  right(p) := return  $2p+2$ 
6  numchildren(p) := return  $\max(\min(n - \text{left}(p), 2), 0)$ 
```

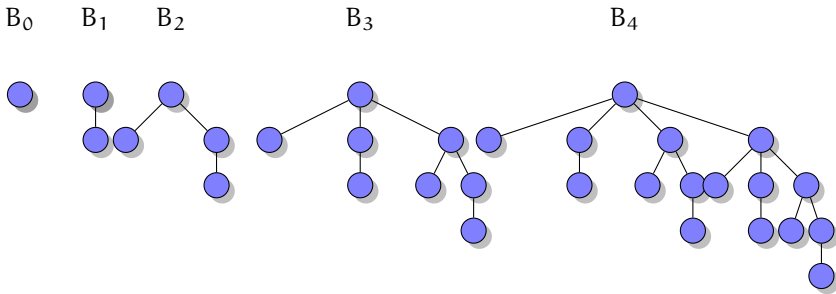
Outras observações:

- Para chamar `update`, temos que conhecer a posição do elemento no heap. Para um conjunto de chaves compactos  $[0, n)$  isso pode ser implementado usando um vetor `pos`, tal que `pos[c]` é o índice da chave  $c$  no heap.
- A fila de prioridade não possui teste  $u \in Q$  (linha 15 do algoritmo 1.2) eficiente. O teste pode ser implementado usando um vetor `visited`, tal que `visited[u]` sse  $u \notin Q$ .

### 1.3.2. Heaps binomiais

Um heap binomial é um coleção de *árvores binomiais* que satisfazem a ordenação de um heap. A árvore binomial  $B_0$  consiste de um único vértice. A árvore binomial  $B_i$  possui uma raiz com filhos  $B_0, \dots, B_{i-1}$ . O *posto* de  $B_k$  é  $k$ . Um heap binomial contém no máximo uma árvore binomial de cada posto.





### Lema 1.3

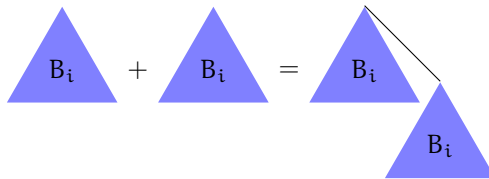
Uma árvore binomial tem as seguintes características:

1.  $B_n$  possui  $2^n$  vértices,  $2^{n-1}$  folhas (para  $n > 0$ ), e tem altura  $n + 1$ .
2. O nível  $k$  de  $B_n$  (a raiz tem nível 0) tem  $\binom{n}{k}$  vértices. (Isso explica o nome.)

**Prova.** Exercício. ■

### Observação 1.5

Podemos combinar dois  $B_i$  obtendo um  $B_{i+1}$  e mantendo a ordenação do heap: Escolhe a árvore com menor chave na raiz, e torna a outra filho da primeira. Chamaremos essa operação “link”. Ela tem custo  $O(1)$  (veja observações sobre a implementação).



◇

### Observação 1.6

Um  $B_i$  possui  $2^i$  vértices. Um heap com  $n$  chaves consiste em  $O(\log n)$  árvores. Isso permite juntar dois heaps binomiais em tempo  $O(\log n)$ . A operação é semelhante à soma de dois números binários com “carry”. Começa juntar os  $B_0$ . Caso tem zero, continua, case tem um, inclui no heap resultante. Caso tem dois o heap resultante não recebe um  $B_0$ . Define como “carry” o link dos dois  $B_0$ ’s. Continua com os  $B_1$ . Sem tem zero ou um ou dois, procede como no caso dos  $B_0$ . Caso tem três, incluindo o “carry”, inclui um no resultado, e define como “carry” o link dos dois restantes. Continue desse forma com os restantes árvores. Para heaps  $h_1, h_2$  chamaremos essa operação  $\text{meld}(h_1, h_2)$ .

◇

## 1. Algoritmos em grafos

Com a operação meld, podemos definir as seguintes operações:

- $\text{makeheap}(c)$ : Retorne um  $B_0$  com chave  $c$ . Custo:  $O(1)$ .
- $\text{insert}(h, c)$ :  $\text{meld}(h, \text{makeheap}(c))$ . Custo:  $O(\log n)$ .
- $\text{getmin}(h)$ : Mantendo um link para a árvore com o menor custo:  $O(1)$ .
- $\text{deletemin}(h)$ : Seja  $B_k$  a árvore com o menor chave. Remove a raiz. Define dois heaps:  $h_1$  é  $h$  sem  $B_k$ ,  $h_2$  consiste dos filhos de  $B_k$ , i.e.  $B_0, \dots, B_{k-1}$ . Retorne  $\text{meld}(h_1, h_2)$ . Custo:  $O(\log n)$ .
- $\text{updatekey}(h, p, c)$ : Como no caso do heap binário completo com custo  $O(\log n)$ .
- $\text{delete}(h, c)$ :  $\text{decreasekey}(h, c, -\infty)$ ;  $\text{deletemin}(h)$

Em comparação com um heap binário completo ganhamos nada no caso pessimista. De fato, a operação insert possui complexidade pessimista  $O(1)$  *amortizada*. Um insert individual pode ter custo  $O(\log n)$ . Do outro lado, isso acontece raramente. Uma análise amortizada mostra que em média sobre uma série de operações, um insert só custa  $O(1)$ . Observe que isso não é uma análise da complexidade média, mas uma análise da complexidade pessimista de uma série de operações.

### Análise amortizada

#### Exemplo 1.3

Temos um contador binário com  $k$  bits e queremos contar de 0 até  $2^k - 1$ . Análise “tradicional”: um incremento tem complexidade  $O(k)$ , porque no caso pior temos que alterar  $k$  bits. Portanto todos incrementos custam  $O(k2^k)$ . Análise amortizada: “Poupamos” operações extras nos incrementos simples, para “gastá-las” nos incrementos caros. Concretamente, setando um bit, gastamos duas operações, uma para setar, outra seria “poupada”. Incrementando, usaremos as operações “poupadas” para zerar bits. Desta forma, um incremento custa  $O(1)$  e temos custo total  $O(2^k)$ .

Uma outra forma da análise amortizada é através uma *função potencial*  $\varphi$ , que associa a cada estado de uma estrutura de dados um valor positivo (a “poupança”). O custo amortizado de uma operação que transforma uma estrutura  $e_1$  em uma estrutura  $e_2$  é  $c - \varphi(e_1) + \varphi(e_2)$ , com  $c$  o custo de operação. No exemplo do contador, podemos usar como  $\varphi(i)$  o número de bits na representação binário de  $i$ . Agora, se temos um estado  $e_1$

$$\underbrace{11 \dots 1}_p \text{ bits um } 0 \quad \underbrace{\dots}_q \text{ bits um}$$

com  $\varphi(e_1) = p + q$ , o estado após de um incremento é

$$\underbrace{00 \dots 0}_0 1 \underbrace{\dots}_q$$

com  $\varphi(e_2) = 1 + q$ . O incremento custa  $c = p + 1$  operações e portanto o custo amortizado é

$$c - \varphi(e_1) + \varphi(e_2) = p + 1 - p - q + 1 + q = 2 = O(1).$$

◇

Resumindo: Dado um série de chamadas de uma operação com custos  $c_1, \dots, c_n$  o custo amortizado da operação é  $\sum_{1 \leq i \leq n} c_i / n$ . Caso temos  $m$  operações diferentes, o custo amortizado da operação que ocorre nos índices  $J \subseteq [1, m]$  é  $\sum_{i \in J} c_i / |J|$ .

As somas podem ser difíceis de avaliar diretamente. Um método para simplificar o cálculo do custo amortizado é o *método potencial*. Acha uma *função potencial*  $\varphi$  que atribui cada estrutura de dados antes da operação  $i$  um valor não-negativo  $\varphi_i \geq 0$  e normaliza ela tal que  $\varphi_1 = 0$ . Atribui um custo amortizado

$$a_i = c_i - \varphi_i + \varphi_{i+1}$$

a cada operação. A soma dos custos não ultrapassa os custos originais, porque

$$\sum a_i = \sum c_i - \varphi_i + \varphi_{i+1} = \varphi_{n+1} - \varphi_1 + \sum c_i \geq \sum c_i$$

Portanto, podemos atribuir a cada tipo de operação  $J \subseteq [1, m]$  o custo amortizado  $\sum_{i \in J} a_i / |J|$ . Em particular, se cada operação individual  $i \in J$  tem custo amortizado  $a_i \leq F$ , o custo amortizado desse tipo de operação é  $F$ .

#### Exemplo 1.4

Queremos implementar uma tabela dinâmica para um número desconhecido de elementos. Uma estratégia é reservar espaço para  $n$  elementos, manter a última posição livre  $p$ , e caso  $p > n$  alocar uma nova tabela de tamanho maior. Uma implementação dessa ideia é

```

1 insert(x):=
2   if p > n then
3     aloca nova tabela de tamanho t = max{2n, 1}
4     copia os elementos  $x_i, 1 \leq i < p$  para nova tabela
5     n := t
6   end if
7    $x_p := x$ 
8   p := p + 1

```

## 1. Algoritmos em grafos

com valores iniciais  $n := 0$  e  $p := 0$ . O custo de insert é  $O(1)$  caso existe ainda espaço na tabela, mas  $O(n)$  no pior caso.

Uma análise amortizada mostra que a complexidade amortizada de uma operação é  $O(1)$ . Seja  $Cn$  o custo das linhas 3–5 e  $D$  o custo das linhas 7–8. Escolhe a função potencial  $\varphi(n) = 2Cp - Dn$ . A função  $\varphi$  é satisfaz os critérios de um potencial, porque  $p \geq n/2$ , e inicialmente temos  $\varphi(0) = 0$ . Com isso o custo amortizado caso tem espaço na tabela é

$$\begin{aligned} a_i &= c_i - \varphi(i-1) + \varphi(i) \\ &= D - (2C(p-1) - Dn) + (2Cp - Dn) = C + 2C = O(1). \end{aligned}$$

Caso temos que alocar uma nova tabela o custo é

$$\begin{aligned} a_i &= c_i - \varphi(i-1) + \varphi(i) = D + Cn - (2C(p-1) - Dn) + (2Cp - 2Dn) \\ &= C + Dn + 2C - Dn = O(1). \end{aligned}$$

◇

**Custo amortizado do heap binomial** Nosso potencial no caso do heap binomial é o número de árvores no heap. O custo de getmin e updatekey não altera o potencial e por isso permanece o mesmo. makeheap cria uma árvore que custa mais uma operação, mas permanece  $O(1)$ . deletemin pode criar  $O(\log n)$  árvores novas, porque o heap contém no máximo um  $B_{\lceil \log n \rceil}$  que tem  $O(\log n)$  filhos, e permanece também com custo  $O(\log n)$ . Finalmente, insert reduz o potencial para cada link no meld e portanto agora custa somente  $O(1)$  amortizado, com o mesmo argumento que no exemplo 1.3.

Desvantagem: a complexidade (amortizada) assintótica de calcular uma árvore geradora mínima permanece  $O(n \log n + m \log n)$ .

**Meld preguiçosa** Ao invés de reorganizar os dois heaps em um meld, podemos simplesmente concatená-los em tempo  $O(1)$ . Isso pode ser implementado sem custo adicional nas outras operações. A única operação que não tem complexidade  $O(1)$  é deletemin. Agora temos uma coleção de árvores binomiais não necessariamente de posto diferente. O deletemin reorganiza o heap, tal que obtemos um heap binomial com árvores de posto único novamente. Para isso, mantemos um vetor com as árvores de cada posto, inicialmente vazio. Sequencialmente, cada árvore no heap, será integrado nesse vetor, executando operações link só for necessário. O tempo amortizado de deletemin permanece  $O(\log n)$ .

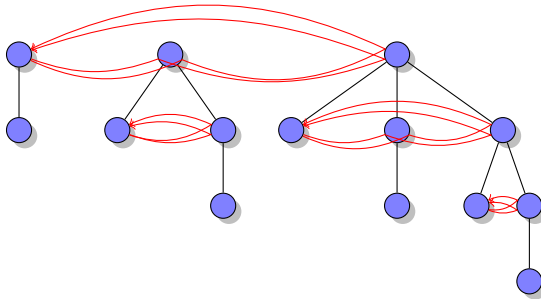
Usaremos um potencial  $\varphi$  que é o dobro do número de árvores. Supondo que antes do deletemin temos  $t$  árvores e executamos  $l$  operações link, o custo

amortizado é

$$(t + 1) - 2t + 2(t - 1) = t - 1.$$

Mas  $t - 1$  é o número de árvores depois o deletemin, que é  $O(\log n)$ , porque todas árvores possuem posto diferente.

**Sobre a implementação** Um forma eficiente de representar heaps binomiais, é em forma de apontadores. Além das apontadores dos filhos para o os pais, cada pai possui um apontador para um filho e os filhos são organizados em uma lista encadeada dupla. Mantemos uma lista encadeada dupla também das raízes. Desta forma, a operação link pode ser implementada em  $O(1)$ .



### 1.3.3. Heaps Fibonacci

Um heap Fibonacci é uma modificação de um heap binomial, com uma operação decreasekey de custo  $O(1)$ . Com isso, uma árvore geradora mínima pode ser calculada em tempo  $O(m + n \log n)$ . Para conseguir decreasekey em  $O(1)$  não podemos mais usar heapify-up, porque heapify-up custa  $O(\log n)$ .

Primeira tentativa:

- $\text{delete}(h, p)$ : Corta  $p$  de  $h$  e executa um meld entre o resto de  $h$  e os filhos de  $p$ . Uma alternativa é implementar  $\text{delete}(h, p)$  como  $\text{decreasekey}(h, p, -\infty)$  e  $\text{deletemin}(h)$ .
- $\text{decreasekey}(h, p)$ : A ordenação do heap pode ser violada. Corta  $p$  e execute um meld entre o resto de  $h$  e  $p$ .

Problema com isso: após de uma série de operações delete ou decreasekey, a árvore pode se tornar “esparso”, i.e. o número de vértices não é mais exponencial no posto da árvore. A análise da complexidade das operações como deletemin depende desse fato para garantir que temos  $O(\log n)$  árvores no heap. Consequência: Temos que garantir, que uma árvore não fica “podado”

## 1. Algoritmos em grafos

demaís. Solução: Permitiremos cada vértice perder no máximo dois filhos. Caso o segundo filho é removido, cortaremos o próprio vértice também. Para cuidar dos cortes, cada nó mantém ainda um valor booleana que indica, se já foi cortado um filho. Observe que um corte pode levar a uma série de cortes e por isso se chama de corte em cascatas (ingl. cascading cuts). Um corte em cascata termina na pior hipótese na raiz. A raiz é o único vértice em que permitiremos cortar mais que um filho. Por isso não mantemos flag na raiz.

**Implementações** Denotamos com  $h$  um heap,  $c$  uma chave e  $p$  um elemento do heap.  $\text{minroot}(h)$  é o elemento do heap que corresponde com a raiz da chave mínima, e  $\text{cut}(p)$  é uma marca que verdadeiro, se  $p$  já perdeu um filho.

```
1  insert(h, c) :=
2      meld(makeheap(c))
3
4  getmin(h) :=
5      return minroot(h)
6
7  delete(h,p) :=
8      decreasekey(h,p,-∞)
9      deletemin(h)
10
11 meld(h1,h2) :=
12     h := lista com raízes de h1 e h2 (em O(1))
13     minroot(h) :=
14         if key(minroot(h1)) < key(minroot(h2)) h1 else h2
15
16 decreasekey(h,p,c) :=
17     key(p) := c
18     if c < key(minRoot(h))
19         minRoot(h) := p
20     if not root(p)
21         if key(parent(p)) > key(p)
22             corta p e adiciona na lista de raízes de h
23             cut(p) := false
24             cascading cut(h,parent(p))
25
26 cascading cut(h,p) :=
27     { p perdeu um filho }
28     if root(p)
29         return
```

```

30  if (not cut(p)) then
31      cut(p) := true
32  else
33      corta p e adiciona na lista de raízes de h
34      cut(p) := false
35      cascading cut(h, parent(p))
36  end if
37
38  deletemin(h) :=
39      remover minroot(h)
40      juntar as listas do resto de h e dos filhos de minroot(h)
41      { reorganizar heap }
42      determina o posto máximo  $M = M(n)$  de h
43       $r_i := \text{undefined}$  para  $0 \leq i \leq M$ 
44      for toda raíz r do
45          remove r da lista de raízes
46           $d := \text{degree}(r)$ 
47          while ( $r_d$  not undefined) do
48               $r := \text{link}(r, r_d)$ 
49               $r_d := \text{undefined}$ 
50               $d := d + 1$ 
51          end while
52           $r_d := r$ 
53      end for
54      definir a lista de raízes pelas entradas definidas  $r_i$ 
55      determinar o novo minroot
56
57  link( $h_1, h_2$ ) :=
58      if ( $\text{key}(h_1) < \text{key}(h_2)$ )
59           $h := \text{makechild}(h_1, h_2)$ 
60      else
61           $h := \text{makechild}(h_2, h_1)$ 
62      cut( $h_1$ ) := false
63      cut( $h_2$ ) := false
64      return h

```

Para concluir que a implementação tem a complexidade desejada temos que provar que as árvores com no máximo um filho cortado não ficam esparsos demais e analisar o custo amortizado das operações.

## 1. Algoritmos em grafos

**Custo amortizado** Para análise usaremos um potencial de  $c_1t + c_2m$  sendo  $t$  o número de árvores,  $m$  o número de vértices marcados e  $c_1, c_2$  constantes. As operações `makeheap`, `insert`, `getmin` e `meld` (preguiçoso) possuem complexidade (real)  $O(1)$ . Para `decreasekey` temos que considerar o caso em que o corte em cascata remove mais que uma subárvore. Supondo que cortamos  $n$  árvores, o número de raízes é  $t + n$  após dos cortes. Para todo corte em cascata, a árvore cortada é desmarcada, logo temos no máximo  $m - (n - 1)$  marcas depois. Portanto custo amortizado é

$$O(n) - (c_1t + c_2m) + (c_1(t + n) + c_2(m - (n - 1))) = c_0n - (c_2 - c_1)n + c_2$$

e com  $c_2 - c_1 \geq c_0$  temos custo amortizado constante  $c_2 = O(1)$ .

Com posto máximo  $M$ , a operação `deletemin` tem o custo real  $O(M + t)$ , com as seguintes contribuições

- Linha 43:  $O(M)$ .
- Linhas 44-51:  $O(M + t)$  com  $t$  o número inicial de árvores no heap. A lista de raízes contém no máximo as  $t$  árvores de  $h$  e mais  $M$  filhos da raiz removida. O laço total não pode executar mais que  $M + t$  operações `link`, porque cada um reduz o número de raízes por um.
- Linhas 54-55:  $O(M)$ .

Seja  $m$  o número de marcas antes do `deletemin` e  $m'$  o número depois. Como `deletemin` marca nenhum vértice, temos  $m' \leq m$ . O número de árvores  $t'$  depois de `deletemin` satisfaz  $t' \leq M$  porque `deletemin` garante que existe no máximo uma árvore de cada posto. Portanto, o potencial depois de `deletemin` é  $\varphi' = c_1t + c_2m' \leq c_1M + c_2m$ , e o custo amortizado é

$$\begin{aligned} O(M + t) - (c_1t + c_2m) + \varphi' &\leq O(M + t) - (c_1t + c_2m) + (c_1M + c_2m) \\ &= (c_0 + c_1)M + (c_0 - c_1)t \end{aligned}$$

e com  $c_1 \geq c_0$  temos custo amortizado  $O(M)$ .

**Um limite para  $M$**  Para provar que `deletemin` tem custo amortizado  $\log n$ , temos que provar que  $M = M(n) = O(\log n)$ . Esse fato segue da maneira "cautelosa" com que cortamos vértices das árvores.

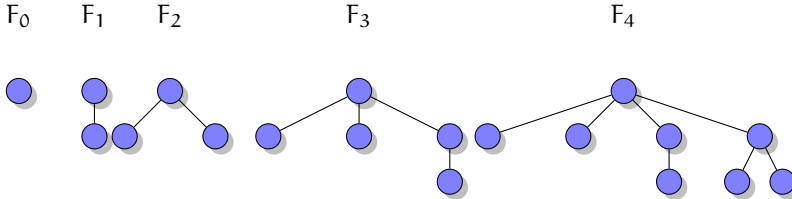
### Lema 1.4

Seja  $p$  um vértice arbitrário de um heap Fibonacci. Considerando os filhos na ordem temporal em que eles foram introduzidos, filho  $i$  possui pelo menos  $i - 2$  filhos.



**Prova.** No instante em que o filho  $i$  foi introduzido,  $p$  estava com pelo menos  $i - 1$  filhos. Portanto  $i$  estava com pelo menos  $i - 1$  filhos também. Depois filho  $i$  perdeu no máximo um filho, e portanto possui pelo menos  $i - 2$  filhos. ■

Quais as menores árvores, que satisfazem esse critério?



### Lema 1.5

Cada subárvore com uma raiz  $p$  com  $k$  filhos possui pelo menos  $F_{k+2}$  vértices.

**Prova.** Seja  $S_k$  o número mínimo de vértices para uma subárvore cuja raiz possui  $k$  filhos. Sabemos que  $S_0 = 1$ ,  $S_1 = 2$ . Define  $S_{-2} = S_{-1} = 1$ . Com isso obtemos para  $k \geq 1$

$$S_k = \sum_{0 \leq i \leq k} S_{k-2-i} = S_{k-2} + S_{k-3} + \cdots + S_{-2} = S_{k-2} + S_{k-1}.$$

Comparando  $S_k$  com os números Fibonacci

$$F_k = \begin{cases} k & \text{se } 0 \leq k \leq 1 \\ F_{k-2} + F_{k-1} & \text{se } k \geq 2 \end{cases}$$

e observando que  $S_0 = F_2$  e  $S_1 = F_3$  obtemos  $S_k = F_{k+2}$ . Usando que  $F_n \in \Theta(\Phi^n)$  com  $\Phi = (1 + \sqrt{5})/2$  (exercício!) conclui a prova. ■

### Corolário 1.1

O posto máximo de um heap Fibonacci com  $n$  elementos é  $O(\log n)$ .

**Sobre a implementação** A implementação da árvore é a mesma que no caso de heaps binomiais. Uma vantagem do heap Fibonacci é que podemos usar os nós como ponteiros – lembre que a operação `decreasekey` precisa disso, porque os heaps não possuem uma operação de busca eficiente. Isso é possível, porque sem `heapify-up` e `heapify-down`, os ponteiros mantem-se válidos.

### 1.3.4. Rank-pairing heaps

Haeupler, Sen e Tarjan [26] propõem um rank-pairing heap (um heap “emparelhando postos”) com as mesmas garantias de complexidade que um heap Fibonacci e uma implementação simplificada e mais eficiente na prática (ver observação 1.9).

**Torneios** Um *torneio* é uma representação alternativa de heaps. Começando com todos elementos, vamos repetidamente comparar pares de elementos, e promover o vencedor para o próximo nível (Fig. 1.1(a)). Uma desvantagem de representar torneios explicitamente é o espaço para chaves redundantes. Por exemplo, o campeão (i.e. o menor elemento) ocorre  $O(\log n)$  vezes. A figura 1.1(b) mostra uma representação sem chaves repetidas. Cada chave é representado somente na comparação mais alta que ele ganhou, as outras comparações ficam vazias. A figura 1.1(c) mostra uma representação compacta em forma de *semi-árvore*. Numa semi-árvore cada elemento possui um filho *ordenado* (na figura o filha da esquerda) e um filho *não-ordenado* (na figura o filho da direita). O filho ordenado é o perdedor da comparação direta com o elemento, enquanto o filho não-ordenado é o perdedor da comparação com o irmão vazio. A raiz possui somente um filho ordenado.

Cada elemento de um torneio possui um *posto*. Por definição, o posto de uma folha é 0. Uma comparação *justa* entre dois elementos do mesmo posto  $r$  resulta num elemento com posto  $r + 1$  no próximo nível. Numa comparação *injusta* entre dois elementos com postos diferentes, o posto do vencedor é definido pelo maior dos dois postos dos participantes (uma alternativa é que o posto fica o mesmo). O posto de um elemento representa um limite inferior do número de elementos que perderam contra-lo:

#### Lema 1.6

Um torneio com campeão de posto  $k$  possui pelo menos  $2^k$  elementos.

**Prova.** Por indução. Caso um vencedor possui posto  $k$  temos duas possibilidades: (i) foi o resultado de uma comparação justa, com dois participantes com posto  $k - 1$  e pela hipótese da indução com pelo menos  $2^{k-1}$  elementos, tal que o vencedor ganhou contra pelo menos  $2^k$  elementos. (ii) foi resultado de uma comparação injusta. Neste caso um dos participantes possuiu posto  $k$  e o vencedor novamente ganhou contra pelo menos  $2^k$  elementos. ■

Cada comparação injusta torna o limite inferior dado pelo posto menos preciso. Por isso uma regra na construção de torneios é fazer o maior número de comparações justas possíveis. A representação de um elemento de heap é possui quatro campos para a chave ( $c$ ), o posto ( $r$ ), o filho ordenado ( $o$ ) e o filho não-ordendo ( $u$ ):

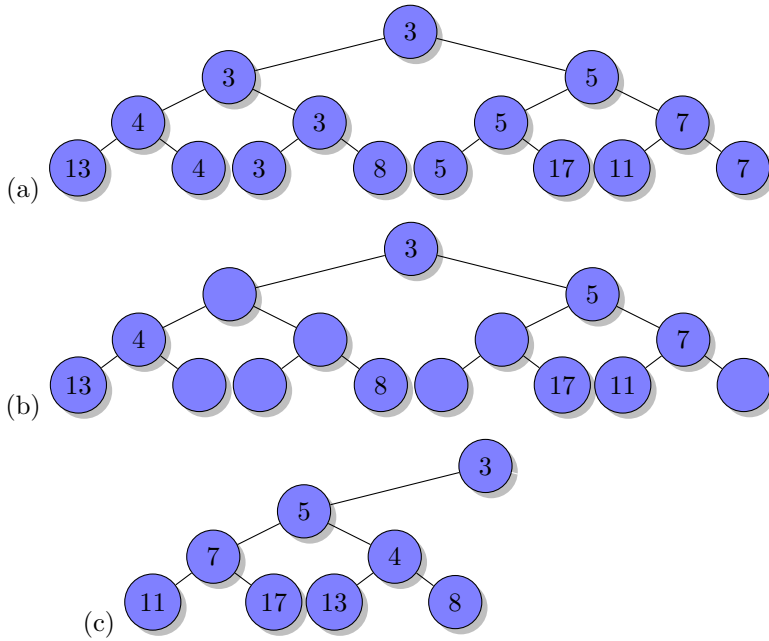


Figura 1.1.: Representações de heaps.

## 1. Algoritmos em grafos

1 def Node(c, r, o, u)

Podemos implementar as operações de uma fila de prioridade (sem update ou decreasekey) como segue:

```
1  { compara duas árvores }
2  link(t1, t2) :=
3      if t1.c < t2.c then
4          return makechild(t1, t2)
5      else
6          return makechild(t2, t1)
7      end if
8
9  makechild(s, t) :=
10     t.u := s.o
11     s.o := t
12     setrank(t)
13     s.r := s.r + 1
14     return s
15
16  setrank(t) :=
17     if t.o.r = t.u.r
18         t.r = t.o.r + 1
19     else
20         t.r = max(t.o.r, t.u.r)
21     end if
22
23  { cria um heap com um único elemento com chave c }
24  make heap(c) := return Node(c, 0, undefined, undefined)
25
26  { inserte chave c no heap }
27  insert(h, c) := link(h, make heap(c))
28
29  { união de dois heaps }
30  meld(h1, h2) := link(h1, h2)
31
32  { elemento mínimo do heap }
33  getmin(h) := return h
34
35  { deleção do elemento mínimo do heap }
36  deletemin(h) :=
37     aloca array r0...rh.o.r+1
```

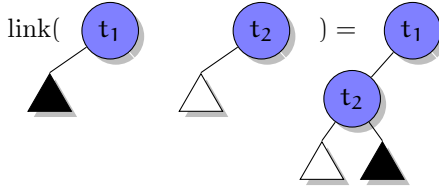


Figura 1.2.: A operação “link” para semi-árvores no caso  $t_1.c < t_2.c$ .

```

38  t = h.o
39  while t not undefined do
40      t' := t.u
41      t.u := undefined
42      register(t, r)
43      t := t'
44  end while
45  h' := undefined
46  for i = 0, ..., h.o.r + 1 do
47      if ri not undefined
48          h' := link(h', ri)
49      end if
50  end for
51  return h'
52 end
53
54 register(t, r) :=
55     if rt.o.r+1 is undefined then
56         rt.o.r+1 := t
57     else
58         t := link(t, rt.o.r+1)
59         rt.o.r+1 := undefined
60         register(t, r)
61     end if
62 end

```

(A figura 1.2 visualiza a operação “link”).

### Observação 1.7

Todas comparações de “register” são justas. As comparações injustas ocorrem na construção da árvore final nas linhas 35–39.  $\diamond$

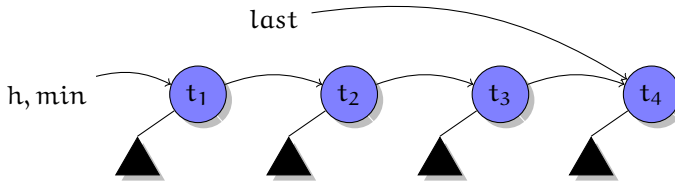


Figura 1.3.: Representação de um heap binomial.

### Lema 1.7

Num torneio balanceado o custo amortizado de “make-heap”, “insert”, “meld” e “getmin” é  $O(1)$ , o custo amortizado de “deletemin” é  $O(\log n)$ .

**Prova.** Usaremos o número de comparações injustas no torneio como potencial. “make-heap” e “getmin” não alteram o potencial, “insert” e “meld” aumentam o potencial por no máximo um. Portanto a complexidade amortizada dessas operações é  $O(1)$ . Para analisar “deletemin” da raiz  $r$  do torneio vamos supor que houve  $k$  comparações injustas com  $r$ . Além dessas comparações injustas,  $r$  participou em no máximo  $\log n$  comparações justas pelo lema 1.6. Em soma vamos liberar no máximo  $k + \log n$  árvores, que reduz o potencial por  $k$ , e com no máximo  $k + \log n$  comparações podemos produzir um novo torneio. Dessas  $k + \log n$  comparações no máximo  $\log n$  são comparações injustas. Portanto o custo amortizado é  $k + \log n - k + \log n = 2 \log n = O(\log n)$ .

■

**Heaps binomiais com varredura única** O custo de representar o heap numa árvore única é permitir comparações injustas. Uma alternativa é permitir somente comparações justas, que implica em manter uma coleção de  $O(\log n)$  árvores. A estrutura de dados resultante é similar com os heaps binomiais: manteremos uma lista (simples) de raízes das árvores, junto com um ponteiro para a árvore com a raiz de menor valor. O heap é representado pela raiz de menor valor, ver Fig. 1.3.

```

1  insert(h, c) :=
2    insere make heap(c) na lista de raízes
3    atualize a árvore mínima
4
5  meld(h1, h2) :=
6    concatena as listas de h1 e h2
7    atualize a árvore mínima

```

Somente “deletemin” opera diferente agora:

```

1 deletemin(h) :=
2   aloca um array de listas  $r_0 \dots r_{\lceil \log n \rceil}$ 
3   remove a árvore mínima da lista de raízes
4   distribui as restantes árvores sobre  $r$ 
5
6    $t := h.o$ 
7   while  $t$  not undefined do
8      $t' := t.u$ 
9      $t.u :=$  undefined
10    insere  $t$  na lista  $r_{t.o.r+1}$ 
11     $t := t'$ 
12  end while
13
14  { executa o maior número possível }
15  { de comparações justas num único passo }
16
17   $h :=$  undefined { lista final de raízes }
18  for  $i = 0, \dots, \lceil \log n \rceil$  do
19    while  $|r_i| \geq 2$ 
20       $t := \text{link}(r_i.\text{head}, r_i.\text{head}.\text{next})$ 
21      insere  $t$  na lista  $h$ 
22      remove  $r_i.\text{head}, r_i.\text{head}.\text{next}$  da lista  $r_i$ 
23    end if
24    if  $|r_i| = 1$  insere  $r_i.\text{head}$  na lista  $h$ 
25  end for
26  return  $h$ 

```

**Observação 1.8**

Continuando com comparações justas até sobrar somente uma árvore de cada posto, obteremos um heap binomial.  $\diamond$

**Lema 1.8**

Num heap binomial com varredura única o custo amortizado de “make-heap”, “insert”, “meld”, “getmin” é  $O(1)$ , o custo amortizado de “deletemin” é  $O(\log n)$ .

**Prova.** Usaremos o dobro do número de árvores como potencial. “getmin” não altera o potencial. “make-heap”, “insert” e “meld” aumentam o potencial por no máximo dois (uma árvore), e portanto possuem custo amortizado  $O(1)$ . “deletemin” libera no máximo  $\log n$  árvores, porque todas comparações foram justas. Com um número total de  $h$  árvores, o custo de deletemin é  $O(h)$ . Sem perda de generalidade vamos supor que o custo é  $h$ . A varredura final

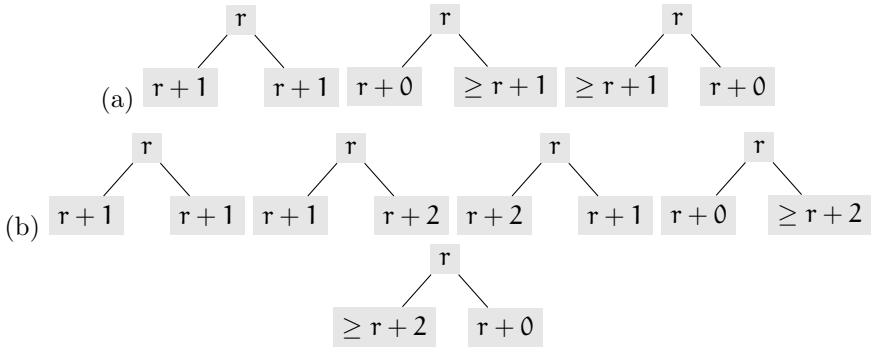


Figura 1.4.: Diferenças no posto de rp-heaps do tipo 1 (a) e tipo 2 (b).

executa pelo menos  $(h - \log n)/2 - 1$  comparações justas, reduzindo o potencial por pelo menos  $h - \log n - 2$ . Portanto o custo amortizado de “deletemin” é  $h - (h - \log n - 2) = \log n + 2 = O(\log n)$ . ■

**rp-heaps** O objetivo do rp-heap é adicionar ao heap binomial de varredura única uma operação “decreasekey” com custo amortizado  $O(1)$ . A ideia e os problemas são os mesmos do heap Fibonacci: (i) para tornar a operação eficiente, vamos cortar a sub-árvore do elemento cuja chave foi diminuída. (ii) o heap Fibonacci usava cortes em cascata para manter um número suficiente de elementos na árvore; no rp-heap ajustaremos os postos do heap que perde uma sub-árvore. Para poder cortar sub-árvores temos que permitir uma folga nos postos. Num heap binomial a diferença do posto de um elemento com o posto do seu pai (caso existe) sempre é um. Num rp-heap do tipo 1, exigimos somente que os dois filhos de um elemento possuem diferença do posto 1 e 1, ou 0 e ao menos 1. Num rp-heap do tipo 2, exigimos que os dois filhos de um elemento possuem diferença do posto 1 e 1, 1 e 2 ou 0 e pelo menos 2. (Figura 1.4.)

Com isso podemos implementar o “decreasekey” (para rp-heaps do tipo 2) como segue:

```

1 decreasekey(h, e, Δ) :=
2   e.c := e.c - Δ
3   if root(e)
4     return
5   if parent(e).o = e then
6     parent(e).o := e.u
```



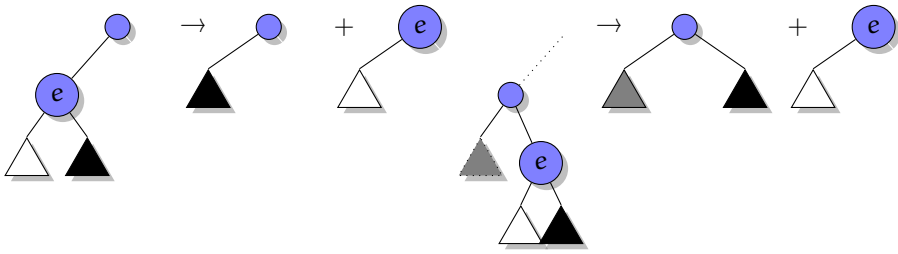


Figura 1.5.: A operação “decreasekey”.

```

7  else
8      parent(e).u := e.u
9  end if
10 parent(e).u := parent(e)
11 e.u := undefined
12 u := parent(e)
13 parent(e) := undefined
14 insere e na lista de raízes de h
15 decreaserank(u)
16
17 rank(e) :=
18     if e is undefined
19         return -1
20     else
21         return e.r
22
23 decreaserank(u) :=
24     if root(u)
25         return
26     if rank(u.o) > rank(u.u)+1 then
27         k := rank(u.o)
28     else if rank(u.u) > rank(u.o)+1 then
29         k := rank(u.u)
30     else
31         k = max(rank(u.o), rank(u.u))+1
32     end if
33     if u.r = k then
34         return
35     else

```

## 1. Algoritmos em grafos

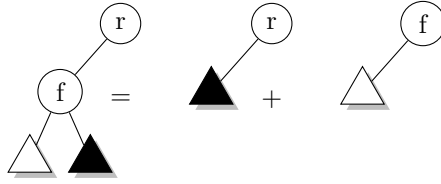


Figura 1.6.: Separar uma semi-árvore de posto  $k$  em duas.

```

36      u.r := k
37      decreaserank(parent(u))
38
39 delete(h, e) :=
40     decreasekey(h, e,  $-\infty$ )
41     deletemin(h)

```

### Observação 1.9

Para implementar o rp-heap precisamos além dos ponteiros para o filho ordenado e não-ordenado um ponteiro para o pai do elemento. A (suposta) eficiência do rp-heap vem do fato que o decreasekey altera os postos do heap, e pouco da estrutura dele e do fato que ele usa somente três ponteiros por elemento, e não quatro como o heap Fibonacci.  $\diamond$

### Lema 1.9

Uma semi-árvore do tipo 2 com posto  $k$  contém pelo menos  $\phi^k$  elementos, sendo  $\phi = (1 + \sqrt{5})/2$  a razão áurea.

**Prova.** Por indução. Para folhas o lema é válido. Caso a raiz com posto  $k$  não é folha podemos obter duas semi-árvores: a primeira é o filho da raiz sem o seu filho não-ordenado, e a segunda é a raiz com o filho não ordenado do seu filho ordenado (ver Fig. 1.6). Pelas regras dos postos de árvores de tipo dois, essas duas árvores possuem postos  $k-1$  e  $k-1$ , ou  $k-1$  e  $k-2$  ou  $k$  e no máximo  $k-2$ . Portanto, o menor número de elementos  $n_k$  contido numa semi-árvore de posto  $k$  satisfaz a recorrência

$$n_k = n_{k-1} + n_{k-2}$$

que é a recorrência dos números Fibonacci.  $\blacksquare$

### Lema 1.10

As operações “decreasekey” e “delete” possuem custo amortizado  $O(1)$  e  $O(\log n)$

**Prova.** Ver [26].  $\blacksquare$

Tabela 1.2.: Complexidade das operações de uma fila de prioridade. Complexidades em **negrito** são amortizados. (1): meld preguiçoso.

	insert	getmin	deletemin	update	decreasekey	delete
Vetor	$O(1)$	$O(1)$	$O(n)$	$O(1)$	(update)	$O(1)$
Lista ordenada	$O(n)$	$O(1)$	$O(1)$	$O(n)$	(update)	$O(1)$
Heap binário	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	(update)	$O(\log n)$
Heap binomial	<b><math>O(1)</math></b>	$O(1)$	$O(\log n)$	$O(\log n)$	(update)	$O(\log n)$
Heap binomial(1)	$O(1)$	$O(1)$	<b><math>O(\log n)</math></b>	$O(\log n)$	(update)	$O(\log n)$
Heap Fibonacci	$O(1)$	$O(1)$	$O(\log n)$	-	<b><math>O(1)</math></b>	$O(\log n)$
rp-heap	$O(1)$	$O(1)$	<b><math>O(\log n)</math></b>	-	<b><math>O(1)</math></b>	<b><math>O(\log n)</math></b>

**Resumo: Filas de prioridade** A tabela 1.2 resume a complexidade das operações para diferentes implementações de uma fila de prioridade.

### 1.3.5. Árvores de van Emde Boas

Pela observação 1.4 é impossível implementar uma fila de prioridade baseado em comparação de chaves com todas operações em  $O(\log n)$ . Porém existem algoritmos que ordenam  $n$  números em  $O(n \log n)$ , aproveitando o fato que as chaves são números com  $k$  bits, como por exemplo o radix sort que ordena em tempo  $O(kn)$ , ou aproveitando que as chaves possuem um domínio limitado, como por exemplo o counting sort que ordena  $n$  números em  $[k]$  em tempo  $O(n + k)$ .

Uma *árvore de van Emde Boas* (árvore vEB)  $T$  realiza as operações

- $\text{member}(T, e)$ : elemento  $e$  pertence a  $T$ ?
- $\text{insert}(T, e)$ : insere  $e$  em  $T$
- $\text{delete}(T, e)$ : remove  $e$  de  $T$
- $\text{min}(T)$  e  $\text{max}(T)$ : elemento mínimo e máximo de  $T$ , ou “undefined” caso não existe
- $\text{succ}(T, e)$  e  $\text{pred}(T, e)$ : successor e predecessor de  $e$  em  $T$ ;  $e$  não precisa pertencer a  $T$

no universo de chaves  $[0, u - 1]$  em tempo  $O(\log \log u)$  e espaço  $O(u)$ .

Outras operações compostas podem ser implementados, por exemplo

```

1  deletemin(T) :=
2      e := min(T); delete(e); return e
3  deletemax(T) :=
4      e := max(T); delete(e); return e

```

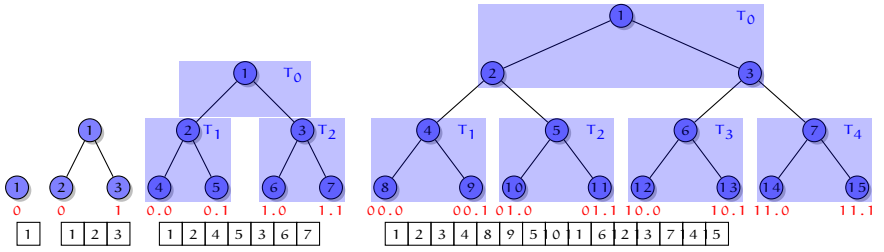


Figura 1.7.: Organização de árvores binárias em ordem de van Emde Boas para  $h \in [4]$ . As folhas são rotuladas por “cluster.subíndice”. Abaixo da árvore a ordem do armazenamento do vértices é dado. Os  $T_i$  correspondem com as subárvores do primeiro nível de recursão.

**Árvores binárias em ordem vEB** Na discussão da implementação de árvores binárias na página 12 discutimos uma representação em ordem da busca por profundidade (BFS order). A ideia da ordem vEB é “cortar” a altura (número de níveis)  $h$  de uma árvore binária (que possui  $n = 2^h - 1$  nodos e  $2^{h-1}$  folhas) pela metade. Com isso obtemos

- uma árvore superior  $T_0$  de altura  $\lfloor h/2 \rfloor$
- e  $b = 2^{\lfloor h/2 \rfloor} = \Theta(2^{h/2}) = \Theta(\sqrt{n})$  árvores inferiores  $T_1, \dots, T_b$  de altura  $\lceil h/2 \rceil$  e com  $2^{\lceil h/2 \rceil} - 1 = \Theta(\sqrt{n})$  nodos.

Os nodos dessa árvore são armazenados em ordem  $T_0, T_1, \dots, T_b$  e toda árvore  $T_i$  é ordenado recursivamente da mesma maneira, até chegar numa árvore de altura  $h = 1$ , como a Figura 1.7 mostra.

Armazenar uma árvore binária em ordem de vEB não altera a complexidade das operações. Uma busca, por exemplo, continua com complexidade  $O(h)$ . Porém, armazenado em ordem da busca por profundidade, uma busca pode gerar  $\Theta(h)$  falhas no *cache*, no pior caso. Na ordem de vEB, a busca sempre atravessa  $\Omega(\log_2 B)$  níveis, com  $B$  o tamanho de uma linha de *cache*, antes de gerar uma nova falha no *cache*. Logo uma busca gera somente  $O(\log_2 n / \log_2 B) = O(\log_B n)$  falhas no *cache*. O layout se chama *cache oblivious* porque funciona sem conhecer o tamanho de uma linha de *cache*  $B$ .

**Árvores vEB** A estrutura básica de uma árvore de vEB é

1. Usar uma árvore binária de altura  $h$  representar  $2^{h-1}$  elementos nas folhas.

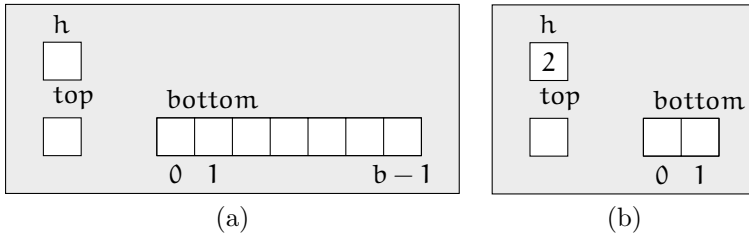


Figura 1.8.: Representação da primeira versão de uma árvore vEB. (a) Forma geral. (b) Caso base.

2. Cada folha armazena um bit, que é 1 caso o elemento correspondente pertence ao conjunto representado.
3. Os bits internos servem como *resumo* da sub-árvore: eles representam a conjunção dos bits dos filhos, i.e. um bit interno é um, caso na sua sub-árvore existe pelo menos uma folha que pertence ao conjunto representado.

Todas as operações da estrutura acima podem ser implementadas em tempo  $O(h) = O(\log u)$ . Para melhorar isso, vamos aplicar a mesma ideia da ordem de van Emde Boas: a árvore é separada em uma árvore superior, e uma série de árvores inferiores, cada uma com altura  $\approx h/2$ . As folhas da árvore superior contém o resumo das raízes das árvores inferiores: por isso a árvore superior possui altura  $\lceil h/2 \rceil + 1$ , uma a mais comparado com a ordem de vEB.

Fig. 1.8 mostra essa representação. A altura da árvore está armazenada no campo  $h$ . Além disso temos um ponteiro “top” para a árvore superior, e um vetor de ponteiros “bottom” de tamanho  $b = 2^{\lceil h/2 \rceil}$  para as raízes das árvores inferiores. No caso base com  $h = 2$ , abusaremos os campos “top” e “bottom” para armazenar os bits da raiz e dos dois filhos: um ponteiro arbitrário diferente de undefined representa um bit 1, o ponteiro undefined o bit 0. Para isso servem as funções auxiliares

```

1  set(p) := p := 1
2  clear(p) := p := undefined
3  bit(p) := return p ≠ undefined

```

Observe que as folhas  $0, 1, \dots, 2^{h-1} - 1$  podem ser representadas com  $h-1$  bits. Os primeiros  $\lceil h/2 \rceil$  bits representam o número da sub-árvore que contém a folha, e os últimos  $\lceil h/2 \rceil - 1$  bits o índice (relativo) da folha na sua sub-árvore. Isso explica a definição das funções auxiliares

```

1  subtree(e) := e  $\gg$   $\lceil h/2 \rceil - 1$ 

```

## 1. Algoritmos em grafos

```
2  subindex( $e$ ) :=  $e \& (1 \ll \lceil h/2 \rceil - 1) - 1$ 
3  element( $s, i$ ) :=  $(s \ll \lceil h/2 \rceil - 1) \mid i$ 
```

para extrair de um elemento o número da sub-árvore correspondente, ou o seu índice nesta sub-árvore, e para determinar o índice na árvore atual do  $i$ -ésimo elemento da sub-árvore  $s$ .

Com isso podemos implementar as operações como segue.

```
1  member( $T, e$ ) :=
2    if  $T.h = 2$ 
3      return bit( $T.bottom[e]$ )
4    return member( $T.bottom[subtree(e)], subindex(e)$ )
5
6  min( $T, e$ ) :=
7    if  $T.h = 2$ 
8      if bit( $T.bottom[0]$ )
9        return 0
10     if bit( $T.bottom[1]$ )
11       return 1
12     return undefined
13
14  c := min( $T.top$ )
15  if  $c = \text{undefined}$ 
16    return  $c$ 
17  return element( $c, \min(T.bottom[c])$ )
18
19  succ( $T, e$ ) :=
20    if  $T.h = 2$ 
21      if  $e = 0$  and bit( $T.bottom[1]$ ) = 1
22        return 1
23      return 0
24
25  s := succ( $T.bottom[subtree(e)], subindex(e)$ )
26  if  $s \neq \text{undefined}$ 
27    return element( $subtree(e), s$ )
28
29  c := succ( $T.top, subtree(e)$ )
30  if  $c = \text{undefined}$ 
31    return  $c$ 
32  return element( $c, \min(T.bottom[c])$ )
33
34  insert( $T, e$ ) :=
```

```

35  if T.h = 2
36      set (T.bottom[e])
37      set (T.top)
38  else
39      insert (T.bottom[subtree(e)], subindex(e))
40      insert (T.top, subtree(e))
41
42  delete (T, e) :=
43      if T.h = 2
44          clear (T.bottom[e])
45          if (bit (T.bottom[1 - e]) = 0)
46              clear (T.top)
47      else
48          delete (T.bottom[subtree(e)], subindex(e))
49          s := min (T.bottom[subtree(e)])
50          if s = undefined
51              delete (T.top, subtree(e))

```

As complexidades das operações implementadas no caso pessimista são (ver as chamadas recursivas acima em vermelho):

**member**  $T(h) = T(\lceil h/2 \rceil) + O(1) = \Theta(\log h) = \Theta(\log u)$ .

**min**  $T(h) = T(\lfloor h/2 \rfloor + 1) + T(\lceil h/2 \rceil) + O(1) = 2T(h/2) + O(1) = \Theta(h) = \Theta(\log u)$ .

**insert**  $T(h) = T(\lceil h/2 \rceil) + T(\lfloor h/2 \rfloor + 1) + O(1) = \Theta(h) = \Theta(\log u)$ .

**succ/delete**  $T(h) = T(\lceil h/2 \rceil) + T(\lfloor h/2 \rfloor + 1) + O(h) = 2T(h/2) + O(h) = \Theta(h \log h) = \Theta(\log u \log \log u)$  (com um trabalho extra de  $O(h)$  para chamar “min”).

Logo todas operações com mais que uma chamada recursiva não possuem a complexidade desejada  $O(\log \log u)$ . A introdução de dois campos “min” e “max” que armazenam o elemento mínimo e máximo, junto com algumas modificações resolvem este problema.

1. Armazenar somente o mínimo, a operação “min” custa somente  $O(1)$  é “insert”, “succ” e “delete” consequentemente somente  $O(h)$ .
2. Armazenado também o máximo, sabemos na operação “succ” se o sucessor está na árvore atual sem buscar, logo a operação “succ” pode ser implementada em  $O(\log \log u)$ .

## 1. Algoritmos em grafos

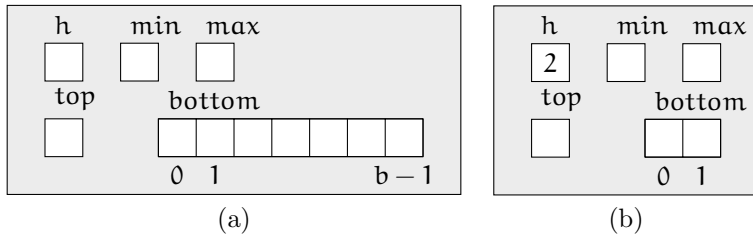


Figura 1.9.: Representação uma árvore vEB. (a) Forma geral. (b) Caso base.

3. A última modificação é *não armazenar* o elemento mínimo na sub-árvore correspondente. Com isso a primeira inserção somente modifica a árvore de resumo (top) e a segunda e as demais operações modificam somente a sub-árvore correspondente. A deleção funciona similarmente: ela remove ou um elemento na sub-árvore, ou o último elemento, modificando somente a árvore de resumo (top). Com isso todas operações podem ser implementadas em  $O(\log \log u)$ .

Na base armazenaremos os elementos somente nos campos “min” e “max”. Por convenção setamos “min” maior que “max” numa árvore vazia. As seguintes funções auxiliares permitem remover os elementos de uma árvore base e determinar se uma árvore possui nenhum, um ou mais elementos.

```

1  clear(T) :=
2    T.min:=1; T.max:=0; // convenção
3
4  empty(T) :=
5    return T.min>T.max
6
7  singleton(T) :=
8    return T.min=T.max
9
10 full(T) :=
11    return T.min<T.max

1 member(T,e) :=
2   if empty(T)
3     return false
4   if T.min = e or T.max = e
5     return true
6
```



```

7   { não é ‘‘min’’ nem ‘‘max’’? a base não contém o elemento }
8   if T.h = 2
9       return false
10
11   return member (T.bottom[subtree(e)], subindex(e))
12
13 min(T) :=
14     if empty(T)
15         return undefined
16     return T.min
17
18 max(T) :=
19     if empty(T)
20         return undefined
21     return T.max
22
23 succ(T, e) :=
24     if T.h=2
25         if e = 0 and T.max = 1
26             return 1
27         return undefined
28
29     if not empty(T) and e < T.min
30         return T.min
31
32     { sucessor na árvore atual }
33     m:=max(T.bottom[subtree(e)])
34     if m ≠ undefined and subindex(e)<m
35         return element(subtree(e),
36                         succ(T.bottom[subtree(e)], subindex(e)))
37
38     { mínimo na árvore sucessora }
39     c := succ(T.top, subtree(e))
40     if c = undefined
41         return c
42     return element(c, min(T.bottom[c]))
43
44 pred(T, e) :=
45     if T.h=2
46         if e = 1 and T.min=0

```

## 1. Algoritmos em grafos

```
47         return 0
48     return undefined
49
50     if not empty(T) and T.max < e
51         return T.max
52
53     { predecessor na árvore atual }
54     m:=min(T.bottom[subtree(e)])
55     if m ≠ undefined and m < subindex(e)
56         return element(subtree(e),
57                         pred(T.bottom[subtree(e)], subindex(e)))
58
59     { máximo na árvore predecessora }
60     c:= pred(T.top, subtree(e))
61     if c = undefined
62         if not empty(T) and T.min < e
63             return T.min
64         else
65             return undefined
66
67     return element(c, max(T.bottom[c]))
68
69 insert(T, e) :=
70     if empty(T)
71         T.min := T.max := e
72     return
73
74     { novo mínimo: setar min, insere min anterior }
75     if e < T.min
76         swap(T.min, e)
77
78     { insere recursivamente }
79     if T.h > 2
80         if empty(T.bottom[subtree(e)])
81             insert(T.top, subtree(e))
82             insert(T.bottom[subtree(e)], subindex(e))
83
84     { novo máximo: atualiza }
85     if T.max < e
```

```

86     T.max := e
87
88 delete(T,e) :=
89     if empty(T)
90         return
91
92     if singleton(T)
93         if T.min = e
94             clear(T)
95         return
96
97     { novo mínimo? }
98     if e = T.min
99         T.min := element(min(T.top), min(T.bottom[min(T.top)]))
100         e := T.min
101
102     { remove e da árvore }
103     delete(T.bottom[subtree(e)], subindex(e))
104
105     if empty(T.bottom[subtree(e)])
106         delete(T.top, subtree(e))
107         if e = T.max
108             c := max(T.top)
109             if c = undefined
110                 T.max := T.min
111             else
112                 T.max := element(c, max(T.bottom[c]))
113     else
114         T.max := element(subtree(e), max(T.bottom[subtree(e)]))

```

Com essas implementações cada função executa uma chamada recursiva e um trabalho constante a mais e logo precisa tempo  $O(\log h)$ . Em particular, na função “insert” caso a sub-árvore do elemento é vazia na linha 80 a segunda chamada “insert” na linha 82 precisa tempo constante. Similarmente, ou a deleção recursiva na linha 103 não remove o último elemento, e talvez custa  $O(\log h)$ , e logo a deleção da linha 106 não é executada, ou ela remove o último elemento e custo somente  $O(1)$ .

### 1.3.6. Tópicos

O algoritmo (assintoticamente) mais rápido para árvores geradoras mínimas usa *soft heaps* e possui complexidade  $O(m\alpha(m, n))$ , com  $\alpha$  a função inversa de Ackermann [8, 30].

Armazenar e atravessar árvores em ordem de van Emde Boas usando índices, similar ao ordem por busca em largura é possível [6]. O consumo de memória das árvores de van Emde Boas pode ser reduzido para  $O(n)$  [12, 9].

### 1.3.7. Exercícios

#### Exercício 1.1

Prove lema 1.3. Dica: Use indução sobre  $n$ .

#### Exercício 1.2

Prove que um heap binomial com  $n$  vértices possui  $O(\log n)$  árvores. Dica: Por contradição.

#### Exercício 1.3 (Laboratório 1)

1. Implementa um heap binário. Escolhe casos de teste adequados e verifica o desempenho experimentalmente.
2. Implementa o algoritmo de Prim usando o heap binário. Novamente verifica o desempenho experimentalmente.

#### Exercício 1.4 (Laboratório 2)

1. Implementa um heap binomial.
2. Verifica o desempenho dele experimentalmente.
3. Verifica o desempenho do algoritmo de Prim com um heap Fibonacci experimentalmente.

#### Exercício 1.5

A proposição 1.3 continua ser correto para grafos com pesos negativos? Justifique.

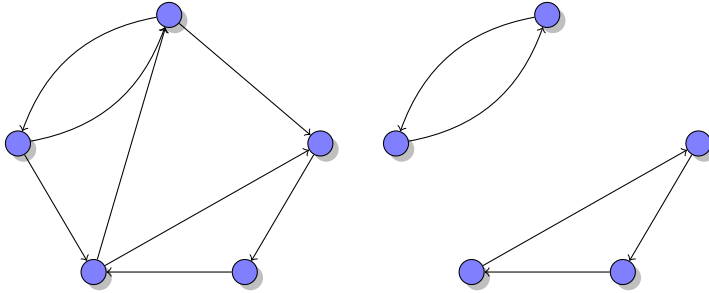


Figura 1.10.: Grafo (esquerda) com circulação (direita)

## 1.4. Fluxos em redes

### Definição 1.1

Para um grafo direcionado  $G = (V, E)$  ( $E \subseteq V \times V$ ) escrevemos  $\delta^+(v) = \{(v, u) \mid (v, u) \in E\}$  para os arcos saíntes de  $v$  e  $\delta^-(v) = \{(u, v) \mid (u, v) \in E\}$  para os arcos entrantes em  $v$ .

Seja  $G = (V, E, c)$  um grafo direcionado e capacitado com capacidades  $c : E \rightarrow \mathbb{R}$  nos arcos. Uma atribuição de fluxos aos arcos  $f : E \rightarrow \mathbb{R}$  em  $G$  se chama *circulação*, se os fluxos respeitam os limites da capacidade ( $f_e \leq c_e$ ) e satisfazem a conservação do fluxo

$$f(v) := \sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e = 0 \quad (1.1)$$

(ver Fig. 1.10).

### Lema 1.11

Qualquer atribuição de fluxos  $f$  satisfaz  $\sum_{v \in V} f(v) = 0$ .

**Prova.**

$$\begin{aligned} \sum_{v \in V} f(v) &= \sum_{v \in V} \sum_{e \in \delta^+(v)} f_e - \sum_{v \in V} \sum_{e \in \delta^-(v)} f_e \\ &= \sum_{(v, u) \in E} f_{(v, u)} - \sum_{(u, v) \in E} f_{(u, v)} = 0 \end{aligned}$$

■

A circulação vira um *fluxo*, se o grafo possui alguns vértices que são fontes ou destinos de fluxo, e portanto não satisfazem a conservação de fluxo. Um

## 1. Algoritmos em grafos

fluxo  $s$ - $t$  possui um único fonte  $s$  e um único destino  $t$ . Um objetivo comum (transporte, etc.) é achar um fluxo  $s$ - $t$  máximo.

### FLUXO $s$ - $t$ MÁXIMO

**Instância** Grafo direcionado  $G = (V, E, c)$  com capacidades  $c$  nos arcos, um vértice origem  $s \in V$  e um vértice destino  $t \in V$ .

**Solução** Um fluxo  $f$ , com  $f(v) = 0, \forall v \in V \setminus \{s, t\}$ .

**Objetivo** Maximizar o fluxo  $f(s)$ .

### Lema 1.12

Um fluxo  $s$ - $t$  satisfaz  $f(s) + f(t) = 0$ .

**Prova.** Pelo lema 1.11 temos  $\sum_{v \in V} f(v) = 0$ . Mas  $\sum_{v \in V} f(v) = f(s) + f(t)$  pela conservação de fluxo nos vértices em  $V \setminus \{s, t\}$ . ■

Uma formulação como programa linear é

$$\begin{array}{ll} \text{maximiza} & f(s) \\ \text{sujeito a} & f(v) = 0 \quad \forall v \in V \setminus \{s, t\} \\ & 0 \leq f_e \leq c_e \quad \forall e \in E. \end{array} \quad (1.2)$$

### Observação 1.10

O programa (1.2) possui uma solução, porque  $f_e = 0$  é uma solução viável. O sistema não é ilimitado, porque todas variáveis são limitadas, e por isso possui uma solução ótima. O problema de encontrar um fluxo  $s$ - $t$  máximo pode ser resolvido em tempo polinomial via programação linear. ◇

#### 1.4.1. Algoritmo de Ford-Fulkerson

Nosso objetivo: Achar um algoritmo *combinatorial* mais eficiente. Idéia básica: Começar com um fluxo viável  $f_e = 0$  e aumentar ele gradualmente. Observação: Se temos um  $s$ - $t$ -caminho  $P = (v_0 = s, v_1, \dots, v_{n-1}, v_n = t)$ , podemos aumentar o fluxo atual  $f$  um valor que corresponde ao “gargalo”

$$g(f, P) := \min_{\substack{e=(v_i, v_{i+1}) \\ 0 \leq i < n}} c_e - f_e.$$

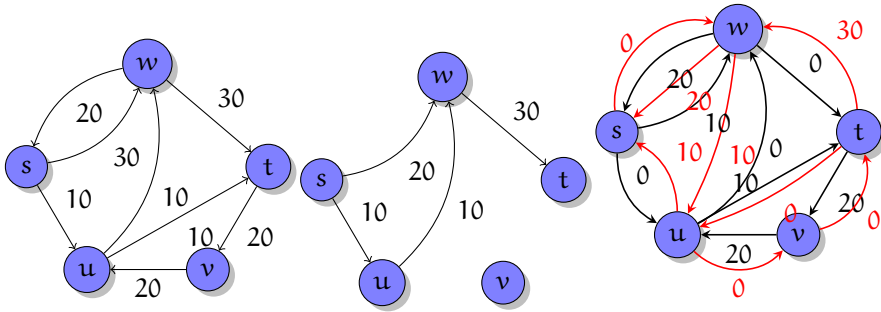


Figura 1.11.: Esquerda: Grafo com capacidades. Centro: Fluxo com valor 30. Direita: O grafo residual correspondente.

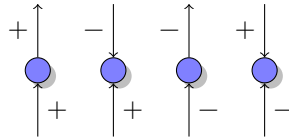


Figura 1.12.: Manter a conservação do fluxo.

### Observação 1.11

Repetidamente procurar um caminho com gargalo positivo e aumentar nem sempre produz um fluxo máximo. Na Fig. 1.11 o fluxo máximo possível é 40, obtido pelo aumento de 10 no caminho  $P_1 = (s, u, t)$  e 30 no caminho  $P_2 = (s, w, t)$ . Mas, se aumentamos 10 no caminho  $P_1 = (s, u, w, t)$  e depois 20 no caminho  $P_2 = (s, w, t)$  obtemos um fluxo de 30 e o grafo não possui mais caminho que aumenta o fluxo.  $\diamond$

Problema no caso acima: para aumentar o fluxo e manter a conservação do fluxo num vértice interno  $v$  temos quatro possibilidades: (i) aumentar o fluxo num arco entrante e saínte, (ii) aumentar o fluxo num arco entrante, e diminuir num outro arco entrante, (iii) diminuir o fluxo num arco entrante e diminuir num arco saínte e (iv) diminuir o fluxo num arco entrante e aumentar num arco entrante (ver Fig. 1.12).

Isso é a motivação para definir para um dado fluxo  $f$  o *grafo residual*  $G_f$  com

- Vértices  $V$
- Arcos para frente (“forward”)  $E$  com capacidade  $c_e - f_e$ , caso  $f_e < c_e$ .

## 1. Algoritmos em grafos

- Arcos para atras (“backward”)  $E' = \{(v, u) \mid (u, v) \in E\}$  com capacidade  $c_{(v,u)} = f_{(u,v)}$ , caso  $f_{(u,v)} > 0$ .

Observe que na Fig. 1.11 o grafo residual possui um caminho  $P = (s, w, u, t)$  que aumenta o fluxo por 10. O algoritmo de Ford-Fulkerson [18] consiste em, repetidamente, aumentar o fluxo num caminho  $s$ – $t$  no grafo residual.

### Algoritmo 1.4 (Ford-Fulkerson)

**Entrada** Grafo  $G = (V, E, c)$  com capacidades  $c_e$  no arcos.

**Saída** Um fluxo  $f$ .

```
1  for all  $e \in E$ :  $f_e := 0$ 
2  while existe um caminho  $s \rightarrow t$  em  $G_f$  do
3    Seja  $P$  um caminho  $s \rightarrow t$  simples
4    Aumenta o fluxo  $f$  um valor  $g(f, P)$ 
5  end while
6  return  $f$ 
```

**Análise de complexidade** Na análise da complexidade, consideraremos somente capacidades em  $\mathbb{N}$  (ou equivalente em  $\mathbb{Q}$ : todas capacidades podem ser multiplicadas pelo menor múltiplo em comum das denominadores das capacidades.)

### Lema 1.13

Para capacidades inteiras, todo fluxo intermediário e as capacidades residuais são inteiros.

**Prova.** Por indução sobre o número de iterações. Inicialmente  $f_e = 0$ . Em cada iteração, o “gargalo”  $g(f, P)$  é inteiro, porque as capacidades e fluxos são inteiros. Portanto, o fluxo e as capacidades residuais após do aumento são novamente inteiros. ■

### Lema 1.14

Em cada iteração, o fluxo aumenta ao menos 1.

**Prova.** O caminho  $s$ – $t$  possui por definição do grafo residual uma capacidade “gargalo”  $g(f, P) > 0$ . O fluxo  $f(s)$  aumenta exatamente  $g(f, P)$ . ■

### Lema 1.15

O número de iterações do algoritmo Ford-Fulkerson é limitado por  $C = \sum_{e \in \delta^+(s)} c_e$ . Portanto ele tem complexidade  $O((n + m)C)$ .



**Prova.**  $C$  é um limite superior do fluxo máximo. Como o fluxo inicialmente possui valor 0 e aumenta ao menos 1 por iteração, o algoritmo de Ford-Fulkerson termina em no máximo  $C$  iterações. Em cada iteração temos que achar um caminho  $s$ - $t$  em  $G_f$ . Representando  $G$  por listas de adjacência, isso é possível em tempo  $O(n+m)$  usando uma busca por profundidade. O aumento do fluxo precisa tempo  $O(n)$  e a atualização do grafo residual é possível em  $O(m)$ , visitando todos arcos. ■

### Corretude do algoritmo de Ford-Fulkerson

#### Definição 1.2

Seja  $\bar{X} := V \setminus X$ . Escrevemos  $F(X, Y) := \{(x, y) \mid x \in X, y \in Y\}$  para os arcos passando do conjunto  $X$  para  $Y$ . O fluxo de  $X$  para  $Y$  é  $f(X, Y) := \sum_{e \in F(X, Y)} f_e$ . Ainda estendemos a notação do fluxo total de um vértice (1.1) para conjuntos:  $f(X) := f(X, \bar{X}) - f(\bar{X}, X)$  é o fluxo neto do saindo do conjunto  $X$ .

Analogamente, escrevemos para as capacidades  $c(X, Y) := \sum_{e \in F(X, Y)} c_e$ . Uma partição  $(X, \bar{X})$  é um *corte*  $s$ - $t$ , se  $s \in X$  e  $t \in \bar{X}$ .

Um arco  $e$  se chama *saturado* para um fluxo  $f$ , caso  $f_e = c_e$ .

#### Lema 1.16

Para qualquer corte  $(X, \bar{X})$  temos  $f(X) = f(s)$ .

**Prova.**

$$f(X) = f(X, \bar{X}) - f(\bar{X}, X) = \sum_{v \in X} f(v) = f(s).$$

(O último passo é correto, porque para todo  $v \in X, v \neq s$ , temos  $f(v) = 0$  pela conservação do fluxo.) ■

#### Lema 1.17

O valor  $c(X, \bar{X})$  de um corte  $s$ - $t$  é um limite superior para um fluxo  $s$ - $t$ .

**Prova.** Seja  $f$  um fluxo  $s$ - $t$ . Temos

$$f(s) = f(X) = f(X, \bar{X}) - f(\bar{X}, X) \leq f(X, \bar{X}) \leq c(X, \bar{X}).$$

Consequência: O fluxo máximo é menor ou igual a o corte mínimo. De fato, a relação entre o fluxo máximo e o corte mínimo é mais forte:

### Teorema 1.2 (Fluxo máximo – corte mínimo)

O valor do fluxo máximo entre dois vértices  $s$  e  $t$  é igual a do corte mínimo.

#### Lema 1.18

Quando o algoritmo de Ford-Fulkerson termina, o valor do fluxo é máximo.

## 1. Algoritmos em grafos

**Prova.** O algoritmo termina se não existe um caminho entre  $s$  e  $t$  em  $G_f$ . Podemos definir um corte  $(X, \bar{X})$ , tal que  $X$  é o conjunto de vértices alcançáveis em  $G_f$  a partir de  $s$ . Qual o valor do fluxo nos arcos entre  $X$  e  $\bar{X}$ ? Para um arco  $e \in F(X, \bar{X})$  temos  $f_e = c_e$ , senão  $G_f$  terá um arco “forward”  $e$ , uma contradição. Para um arco  $e = (u, v) \in F(\bar{X}, X)$  temos  $f_e = 0$ , senão  $G_f$  terá um arco “backward”  $e' = (v, u)$ , uma contradição. Logo

$$f(s) = f(X) = f(X, \bar{X}) - f(\bar{X}, X) = f(X, \bar{X}) = c(X, \bar{X}).$$

Pelo lema 1.17, o valor de um fluxo arbitrário é menor ou igual que  $c(X, \bar{X})$ , portanto  $f$  é um fluxo máximo. ■

**Prova.** (Do teorema 1.2) Pela análise do algoritmo de Ford-Fulkerson. ■

**Desvantagens do algoritmo de Ford-Fulkerson** O algoritmo de Ford-Fulkerson tem duas desvantagens:

- (1) O número de iterações  $C$  pode ser alto, e existem grafos em que  $C$  iterações são necessárias (veja Fig. 1.13). Além disso, o algoritmo com complexidade  $O((n + m)C)$  é somente pseudo-polinomial.
- (2) É possível que o algoritmo não termina para capacidades reais (veja Fig. 1.13). Usando uma busca por profundidade para achar caminhos  $s$ - $t$  ele termina, mas é ineficiente [11].

### 1.4.2. O algoritmo de Edmonds-Karp

O algoritmo de Edmonds-Karp elimina esses problemas. O princípio dele é simples: Para achar um caminho  $s$ - $t$  simples, usa busca por largura, i.e. seleccione o caminho mais curto entre  $s$  e  $t$ . Nos temos

#### **Teorema 1.3**

O algoritmo de Edmonds-Karp precisa  $O(nm)$  iterações, e portanto termina em  $O(nm^2)$ .

#### **Lema 1.19**

Seja  $\delta_f(v)$  a distância entre  $s$  e  $v$  em  $G_f$ . Durante a execução do algoritmo de Edmonds-Karp  $\delta_f(v)$  cresce monotonicamente para todos vértices em  $V$ .

**Prova.** Para  $v = s$  o lema é evidente. Supõe que uma iteração modificando o fluxo  $f$  para  $f'$  diminuirá o valor de um vértice  $v \in V \setminus \{s\}$ , i.e.,  $\delta_f(v) > \delta_{f'}(v)$ . Supõe ainda que  $v$  é o vértice de menor distância  $\delta_{f'}(v)$  em  $G_{f'}$  com essa característica. Seja  $P = (s, \dots, u, v)$  um caminho mais curto de  $s$  para  $v$

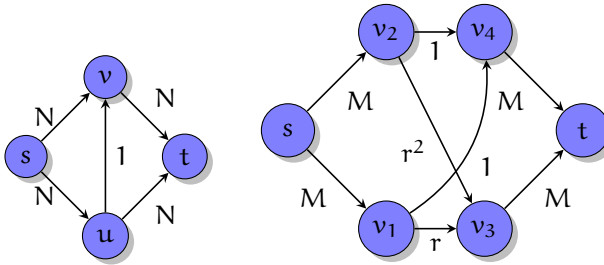


Figura 1.13.: Esquerda: Pior caso para o algoritmo de Ford-Fulkerson com pesos inteiros aumentando o fluxo por  $2N$  vezes por 1 nos caminhos  $(s, u, v, t)$  e  $(s, v, u, t)$ . Direita: Menor grafo com pesos irracionais em que o algoritmo de Ford-Fulkerson falha [43].  $M \geq 3$ , e  $r = (1 + \sqrt{1 - 4\lambda})/2$  com  $\lambda \approx 0.217$  a única raiz real de  $1 - 5x + 2x^2 - x^3$ . Aumentar  $(s, v_1, v_4, t)$  e depois repetidamente  $(s, v_2, v_4, v_1, v_3, t)$ ,  $(s, v_2, v_3, v_1, v_4, t)$ ,  $(s, v_1, v_3, v_2, v_4, t)$ , e  $(s, v_1, v_4, v_2, v_3, t)$  converge para o fluxo máximo  $2 + r + r^2$  sem terminar.

em  $G_{f'}$ . O valor de  $u$  não diminuiu nessa iteração (pela escolha de  $v$ ), i.e.,  $\delta_f(u) \leq \delta_{f'}(u)$  (\*).

O arco  $(u, v)$  não existe in  $G_f$ , senão a distância do  $v$  in  $G_f$  é no máximo a distância do  $v$  in  $G_{f'}$ : Supondo  $(u, v) \in E(G_f)$  temos

$$\begin{aligned} \delta_f(v) &\leq \delta_f(u) + 1 && \text{pela desigualdade triangular} \\ &\leq \delta_{f'}(u) + 1 && (*) \\ &\leq \delta_{f'}(v) && \text{porque } uv \text{ está num caminho mínimo em } G_{f'}, \end{aligned}$$

uma contradição com a hipótese que a distância de  $v$  diminuiu. Portanto,  $(u, v) \notin E(G_f)$  mas  $(u, v) \in E(G_{f'})$ . Isso só é possível se o fluxo de  $v$  para  $u$  aumentou nessa iteração. Em particular,  $vu$  foi parte de um caminho mínimo de  $s$  para  $u$ . Para  $v = t$  isso é uma contradição imediata. Caso  $v \neq t$ , temos

$$\begin{aligned} \delta_f(v) &= \delta_f(u) - 1 \\ &\leq \delta_{f'}(u) - 1 && (*) \\ &= \delta_{f'}(v) - 2 && \text{porque } uv \text{ está num caminho mínimo em } G_{f'}, \end{aligned}$$

novamente uma contradição com a hipótese que a distância de  $v$  diminuiu. Logo, o vértice  $v$  não existe. ■

**Prova.** (do teorema 1.3)

## 1. Algoritmos em grafos

Chama um arco num caminho que aumenta o fluxo com capacidade igual ao gargalo *crítico*. Em cada iteração existe ao menos um arco crítico que desaparece do grafo residual. Provaremos que cada arco pode ser crítico no máximo  $n/2 - 1$  vezes, que implica em no máximo  $m(n/2 - 1) = O(mn)$  iterações.

No grafo  $G_f$  em que um arco  $uv \in E$  é crítico pela primeira vez temos  $\delta_f(u) = \delta_f(v) - 1$ . O arco só aparece novamente no grafo residual caso alguma iteração diminui o fluxo em  $uv$ , i.e., aumenta o fluxo  $vu$ . Nessa iteração, com fluxo  $f'$ ,  $\delta_{f'}(v) = \delta_{f'}(u) - 1$ . Em soma temos

$$\begin{aligned}\delta_{f'}(u) &= \delta_{f'}(v) + 1 \\ &\geq \delta_f(v) + 1 && \text{pelo lema 1.19} \\ &= \delta_f(u) + 2,\end{aligned}$$

i.e., a distância do  $u$  entre dois instantes em que  $uv$  é crítico aumenta por pelo menos dois. Enquanto  $u$  é alcançável por  $s$ , a sua distância é no máximo  $n - 2$ , porque o caminho não contém  $s$  nem  $t$ , e por isso a aresta  $uv$  pode ser crítico por no máximo  $(n - 2)/2 = n/2 - 1$  vezes. ■

Zadeh [42] apresenta instâncias em que o algoritmo de Edmonds-Karp precisa  $\Theta(n^3)$  iterações, logo o resultado do teorema 1.3 é o melhor possível para grafos densos.

### 1.4.3. O algoritmo “caminho mais gordo” (“fattest path”)

Idéia [14]: usar o caminho de maior gargalo para aumentar o fluxo. (Exercício 1.6 pede provar que isso é possível com uma modificação do algoritmo de Dijkstra em tempo  $O(n \log n + m)$ .)

#### Lema 1.20

Um fluxo  $f$  pode ser decomposto em no máximo  $m$  fluxos  $f_1, \dots, f_k$  tal que o fluxo  $f_i$  é positivo somente num caminho  $p_i$  entre  $s$  e  $t$ .

**Prova.** Dado um fluxo  $f$ , encontra um caminho  $p$  de  $s$  para  $t$  usando somente arcos com fluxo positivo. Define um fluxo no caminho cujo valor é o valor do menor fluxo de algum arco em  $p$ . Subtraindo esse fluxo do fluxo  $f$  obtemos um novo fluxo reduzido. Repete até o valor do fluxo  $f$  é zero.

Em cada iteração pelo menos um arco com fluxo positivo tem fluxo zero depois da subtração do caminho  $p$ . Logo o algoritmo termina em no máximo  $m$  iterações. O conjunto de caminhos encontrados com os fluxos correspondentes satisfaz as condições do lema. ■

#### Teorema 1.4

O caminho com o maior gargalo aumenta o fluxo por, pelo menos,  $\text{OPT}/m$ .

**Prova.** Considera o fluxo máximo. Pelo lema 1.20 existe uma decomposição do fluxo em no máximo  $m$  fluxos em caminhos  $s$ - $t$ . Logo um dos caminhos possui valor pelo menos  $\text{OPT}/m$ . ■

### Teorema 1.5

A complexidade do algoritmo de Ford-Fulkerson usando o caminho de maior gargalo é  $O((n \log n + m)m \log C)$  para um limitante superior  $C$  do fluxo máximo.

**Prova.** Seja  $f_i$  o valor do caminho encontrado na  $i$ -ésima iteração,  $G_i$  o grafo residual após do aumento e  $\text{OPT}_i$  o fluxo máximo em  $G_i$ . Observe que  $G_0$  é o grafo de entrada e  $\text{OPT}_0 = \text{OPT}$  o fluxo máximo. Temos

$$\text{OPT}_{i+1} = \text{OPT}_i - f_i \leq \text{OPT}_i - \text{OPT}_i/(2m) = (1 - 1/(2m))\text{OPT}_i.$$

A segunda desigualdade é válida pelo teorema 1.4, considerando que o grafo residual possui no máximo  $2m$  arcos. Logo

$$\text{OPT}_i \leq (1 - 1/(2m))^i \text{OPT} \leq e^{-i/(2m)} \text{OPT}.$$

O algoritmo termina caso  $\text{OPT}_i < 1$ , por isso número de iterações é no máximo  $2m \ln \text{OPT} + 1$ . Cada iteração custo  $O(m + n \log n)$ . ■

### Corolário 1.2

Caso  $U$  é um limite superior da capacidade de um arco, o algoritmo termina em no máximo  $O(m \log mU)$  passos.

#### 1.4.4. O algoritmo push-relabel

O algoritmo push-relabel é um representante de classe de algoritmos, que não trabalha com um fluxo e caminhos aumentantes, mas mantém um *pré-fluxo*  $f$  que satisfaz

- os limites de capacidade ( $0 \leq f_e \leq c_e$ )
- e requer somente que o *excesso*  $e(v) = -f(v)$  de um vértice  $v \neq s$  é não-negativo.

Um vértice  $v \neq t$  com  $e(v) > 0$  é chamado *ativo*. A ideia do algoritmo é que vértices possuem uma “altura” e o fluxo passa para vértices de altura mais baixa (“operação push”) ou, caso isso não é possível a altura de um vértice ativo aumenta (“operação relabel”). Concretamente, manteremos um *potencial* (“altura”)  $p_v$  para cada  $v \in V$ , tal que,

$$\begin{aligned} p_s &= n; & p_t &= 0; \\ p_v &\geq p_u - 1 & (u, v) &\in A(G_f). \end{aligned} \tag{*}$$

## 1. Algoritmos em grafos

### Observação 1.12

Pela condição (\*), para um caminho  $v_0, v_1, \dots, v_k$  em  $G_f$  temos  $p_{v_0} \leq p_{v_1} + 1 \leq p_{v_2} + 2 \leq \dots \leq p_{v_k} + k$ .  $\diamond$

### Lema 1.21

A condição (\*) é satisfeita sse  $G_f$  não possui caminho  $s$ - $t$ .

**Prova.** “ $\Rightarrow$ ”: Supõe existe um caminho  $s$ - $t$  simples  $v_0 = s, v_1, \dots, v_k = t$ . Pela observação (1.12)

$$p_s = p_{v_0} \leq p_{v_k} + k = p_t + k = k < n - 1,$$

uma contradição. “ $\Leftarrow$ ”: Sejam  $X$  os vértices alcançáveis em  $G_f$  a partir de  $s$  (incluindo  $s$ ). Define  $p_v = n$  para  $v \in X$  e  $p_v = 0$  para  $v \in \bar{X}$ .  $\blacksquare$

O lema mostra que enquanto algoritmos de caminho aumentante são algoritmos primais, mantendo uma solução factível, até encontrar o ótimo, algoritmos da classe push-relabel podem ser vistos como algoritmos duais: eles mantêm o critério de otimalidade, até encontrar uma solução factível.

Podemos realizar as operações “push” e “relabel” como segue. A operação “push( $u, v$ )” num arco  $(u, v) \in A(G_f)$  manda o fluxo  $\min\{c_a, e(v)\}$  de  $u$  para  $v$ . A operação “relabel( $v$ )” aumenta a altura  $p_v$  do vértice  $v$  por uma unidade.

```

1  push( $u, v$ ) :=
2    { pré condição:  $u$  é ativo }
3    { pré condição:  $p_v = p_u - 1$  }
4    { pré condição:  $(u, v) \in A(G_f)$  }
5    aumenta o fluxo em  $(u, v)$  por  $\min\{c_{(u,v)}, e(u)\}$ 
6    { atualiza  $G_f$  de acordo }
7  end
8
9  relabel( $v$ ) :=
10   { pré condição:  $v$  é ativo }
11   { pré condição: não existe  $(u, v) \in A(G_f)$  com  $p_v = p_u - 1$  }
12    $p_v := p_v + 1$ 
13 end
```

Observe que as duas operações mantêm a condição (\*).

### Algoritmo 1.5 (Push-relabel)

**Entrada** Grafo  $G = (V, A, c)$  com capacidades  $c_a$  no arcos.

**Saída** Um fluxo  $f$ .

1  $p_s := n$ ;  $p_v := 0$ ,  $\forall v \in V \setminus \{s\}$

```

2   $f_a := c_a$ ,  $\forall a \in \delta^+(s)$  senão  $f_a := 0$ 
3  while existe vértice ativo do
4    escolhe o vértice ativo  $u$  de maior  $p_u$ 
5    repete até  $u$  é inativo
6      if existe arco  $(u, v) \in G_f$  com  $p_v = p_u - 1$  then
7        push( $u, v$ )
8      else
9        relabel( $u$ )
10     end if
11   end
12 end while
13 return  $f$ 

```

**Lema 1.22**

O algoritmo push-relabel é parcialmente correto (i.e. correto caso termina).

**Prova.** Ao terminar não existe vértice ativo. Logo  $f$  é um fluxo. Pelo lema 1.21 não existe caminho  $s$ – $t$  em  $G_f$ . Logo pelo teorema 1.2 o fluxo é ótimo. ■  
A terminação é garantido por

**Teorema 1.6**

O algoritmo push-relabel executa  $O(n^3)$  operações push e  $O(n^2)$  operações relabel.

**Prova.** Um vértice ativo  $v$  tem excesso de fluxo, logo existe um caminho  $v$ – $s$  em  $G_f$ . Por (1.12)  $p_v \leq p_s + (n-1) < 2n$ , e o número de operações relabel é no  $O(n^2)$ . Supõe que uma operação push satura um arco  $a = (u, v)$  (i.e. manda fluxo  $c_a$ ). Para mandar fluxo novamente, temos que mandar primeiramente fluxo de  $v$  para  $u$ ; isso só pode ser feito depois de pelo menos duas operações relabel em  $v$ . Logo o número de operações push saturantes é  $O(mn)$ . Para operações push não-saturantes, podemos observar que entre duas operações relabel temos no máximo  $n$  desses operações, porque cada uma torna o vértice de maior  $p_v$  inativo (talvez ativando vértices de menor potencial), logo tem no máximo  $O(n^3)$  deles. ■

Para garantir uma complexidade de  $O(n^3)$  temos que implementar um “push” em  $O(1)$  e um “relabel” em  $O(n)$ . Para este fim, manteremos uma lista dos vértices em ordem do potencial. Para cada vértice manteremos uma lista de arcos candidatos para operações push, i.e. arcos para vizinhos com potencial um a menos com capacidade residual positiva.

Uma busca linear na lista de vértices encontra o vértice de maior potencial. Entre dois operações relabel a busca pode continuar no último ponto e precisa tempo  $O(n)$  em total, logo a busca custa no máximo  $O(n^3)$  sobre toda

Tabela 1.3.: Complexidade para diversos algoritmos de fluxo máximo [38].

Ano	Referência	Complexidade	Obs
1951	Dantzig	$O(n^2 m C)$	Simplex
1955	Ford & Fulkerson	$O(mC) = O(mnU)$	Cam. aument.
1970	Dinitz	$O(nm^2)$	Cam. min. aument.
1972	Edmonds & Karp	$O(m^2 \log C)$	Escalonamento
1973	Dinitz	$O(nm \log C)$	Escalonamento
1974	Karzanov	$O(n^3)$	Preflow-Push
1977	Cherkassky	$O(n^2 m^{1/2})$	Preflow-Push
1986	Goldberg & Tarjan	$O(nm \log(n^2/m))$	Push-Relabel
1987	Ahuja & Orlin	$O(nm + n^2 \log C)$	Push-Relabel & Esc.
1990	Cheriyani et al.	$O(n^3 / \log n)$	
1990	Alon	$O(nm + n^{8/3} \log n)$	
1992	King et al.	$O(nm + n^{2+\epsilon})$	
1997	Goldberg & Rao	$O(m^{3/2} \log(n^2/m) \log C)$ $O(n^{2/3} m \log(n^2/m) \log C)$	
2012	Orlin	$O(nm)$	

execução do algoritmo. Para a operação push podemos simplesmente consultar a lista de candidatos. Para um push saturante, o candidato será removido. Isso custa  $O(1)$ . Finalmente no caso de um relabel temos que encontrar em  $O(n)$  a nova posição do vértice na lista, e reconstruir a lista de candidatos, que também precisa tempo  $O(n)$ . Logo todas operações relabel custam não mais que  $O(n^3)$ .

#### 1.4.5. Variações do problema

**Fontes e destinos múltiplos** Para  $G = (V, E, c)$  define um conjunto de fontes  $S \subseteq V$  e um conjunto de destinos  $T \subseteq V$ , com  $S \cap T = \emptyset$ , e considera

$$\begin{array}{lll}
 \text{maximiza} & f(S) & \\
 \text{sujeito a} & f(v) = 0 & \forall v \in V \setminus (S \cup T) \\
 & f_e \leq c_e & \forall e \in E.
 \end{array} \tag{1.3}$$



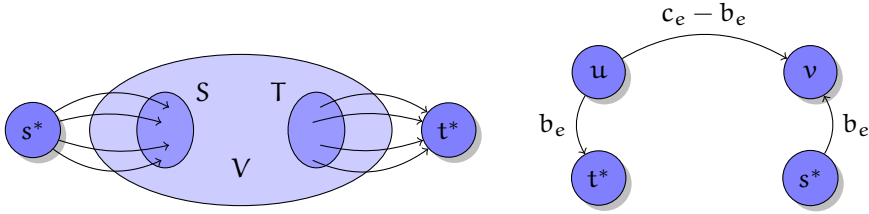


Figura 1.14.: Reduções entre variações do problema do fluxo máximo. Esquerda: Fontes e destinos múltiplos. Direita: Limite inferior e superior para a capacidade de arcos.

O problema (1.3) pode ser reduzido para um problema de fluxo máximo simples em  $G' = (V', E', c')$  (veja Fig. 1.14(a)) com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ E' &= E \cup \{(s^*, s) \mid s \in S\} \cup \{(t, t^*) \mid t \in T\} \\ c'_e &= \begin{cases} c_e & e \in E \\ c(S, \bar{S}) & e = (s^*, s) \\ c(\bar{T}, T) & e = (t, t^*) \end{cases} \end{aligned} \quad (1.4)$$

### Lema 1.23

Se  $f'$  é solução máxima de (1.4),  $f = f'|_E$  é uma solução máxima de (1.3). Conversamente, se  $f$  é uma solução máxima de (1.3),

$$f'_e = \begin{cases} f_e & e \in E \\ f(s) & e = (s^*, s) \\ -f(t) & e = (t, t^*) \end{cases}$$

é uma solução máxima de (1.4).

**Prova.** Supõe  $f$  é solução máxima de (1.3). Seja  $f'$  uma solução de (1.4) com valor  $f'(s^*)$  maior. Então  $f'|_E$  é um fluxo válido para (1.3) com solução  $f'|_E(S) = f'(s^*)$  maior, uma contradição.

Conversamente, para cada fluxo válido  $f$  em  $G$ , a extensão  $f'$  definida acima é um fluxo válido em  $G'$  com o mesmo valor. Portanto o valor do maior fluxo em  $G'$  é maior ou igual ao valor do maior fluxo em  $G$ . ■

## 1. Algoritmos em grafos

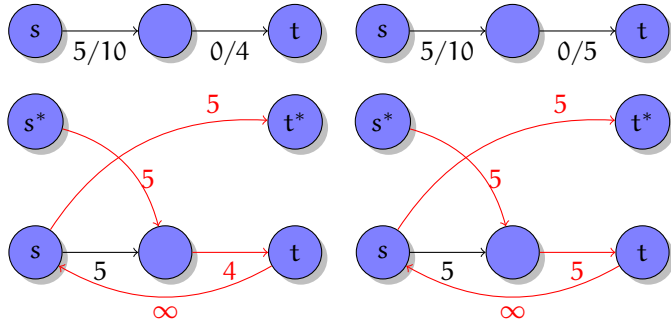


Figura 1.15.: Dois exemplos da transformação do lema 1.24. Esquerda: Grafo sem solução viável e grafo transformado com fluxo máximo 4. Direita: Grafo com solução viável e grafo transformado com fluxo máximo 5.

**Limites inferiores** Para  $G = (V, E, b, c)$  com limites inferiores  $b : E \rightarrow \mathbb{R}$  considere o problema

$$\begin{aligned} &\text{maximiza} && f(s) \\ &\text{sujeito a} && f(v) = 0 && \forall v \in V \setminus \{s, t\} \\ & && b_e \leq f_e \leq c_e && e \in E. \end{aligned} \quad (1.5)$$

O problema (1.5) pode ser reduzido para um problema de fluxo máximo simples em  $G' = (V', E', c')$  (veja Fig. 1.14(b)) com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ E' &= E \cup \{(u, t^*) \mid (u, v) \in E\} \cup \{(s^*, v) \mid (u, v) \in E\} \cup \{(t^*, s^*)\} \\ c'_e &= \begin{cases} c_e - b_e & e \in E \\ \sum_{v \in N^+(u)} b_{(u,v)} & e = (u, t^*) \\ \sum_{u \in N^-(v)} b_{(u,v)} & e = (s^*, v) \\ \infty & e = (t, s) \end{cases} \end{aligned} \quad (1.6)$$

### Lema 1.24

Problema (1.5) possui uma solução viável sse (1.6) possui uma solução máxima com todos arcos auxiliares  $E' \setminus E$  saturados. Neste caso, se  $f$  é um fluxo máximo

em (1.5),

$$f'_e = \begin{cases} f_e - b_e & e \in E \\ \sum_{u \in N^+(v)} b_{(v,u)} & e = (v, t^*) \\ \sum_{u \in N^-(v)} b_{(u,v)} & e = (s^*, u) \\ f(s) & e = (t, s) \end{cases}$$

é um fluxo máximo de (1.6) com arcos auxiliares saturados. Conversamente, se  $f'$  é um fluxo máximo para (1.6) com arcos auxiliares saturados,  $f_e = f'_e + b_e$  é um fluxo máximo em (1.5).

**Prova.** (Exercício.) ■

Para obter um fluxo máximo de (1.5) podemos maximizar o fluxo a partir da solução viável obtida, com qualquer variante do algoritmo de Ford-Fulkerson. Uma alternativa para obter um fluxo máximo com limites inferiores nos arcos é primeiro mandar o limite inferior de cada arco, que torna o problema num problema de encontrar o fluxo  $s$ - $t$  máximo num grafo com demandas.

**Existência de uma circulação com demandas** Para  $G = (V, E, c)$  com demandas  $d_v$ , com  $d_v > 0$  para destinos e  $d_v < 0$  para fontes, considere

$$\begin{array}{lll} \text{existe} & f & \\ \text{s.a} & f(v) = -d_v & \forall v \in V \\ & f_e \leq c_e & e \in E. \end{array} \quad (1.7)$$

Evidentemente  $\sum_{v \in V} d_v = 0$  é uma condição necessária (lema (1.11)). O problema (1.7) pode ser reduzido para um problema de fluxo máximo em  $G' = (V', E')$  com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ E' &= E \cup \{(s^*, v) \mid v \in V, d_v < 0\} \cup \{(v, t^*) \mid v \in V, d_v > 0\} \\ c_e &= \begin{cases} c_e & e \in E \\ -d_v & e = (s^*, v) \\ d_v & e = (v, t^*) \end{cases} \end{aligned} \quad (1.8)$$

### Lema 1.25

Problema (1.7) possui uma solução sse problema (1.8) possui uma solução com fluxo máximo  $D = \sum_{v: d_v > 0} d_v$ .

**Prova.** (Exercício.) ■

## 1. Algoritmos em grafos

**Circulações com limites inferiores** Para  $G = (V, E, b, c)$  com limites inferiores e superiores, considere

$$\begin{array}{ll} \text{existe} & f \\ \text{s.a} & f(v) = d_v \quad \forall v \in V \\ & b_e \leq f_e \leq c_e \quad e \in E. \end{array} \quad (1.9)$$

O problema pode ser reduzido para a existência de uma circulação com somente limites superiores em  $G' = (V', E', c', d')$  com

$$\begin{array}{ll} V' = V \\ E' = E \end{array} \quad (1.10)$$

$$\begin{array}{ll} c_e = c_e - b_e \\ d'_v = d_v - \sum_{e \in \delta^-(v)} b_e + \sum_{e \in \delta^+(v)} b_e \end{array} \quad (1.11)$$

### Lema 1.26

O problema (1.9) possui solução sse problema (1.10) possui solução.

**Prova.** (Exercício.) ■

### 1.4.6. Aplicações

**Projeto de pesquisa de opinião** O objetivo é projetar uma pesquisa de opinião, com as restrições

- Cada cliente  $i$  recebe ao menos  $c_i$  perguntas (para obter informação suficiente) mas no máximo  $c'_i$  perguntas (para não cansar ele). As perguntas podem ser feitas somente sobre produtos que o cliente já comprou.
- Para obter informações suficientes sobre um produto, entre  $p_i$  e  $p'_i$  clientes tem que ser interrogados sobre ele.

Um modelo é um grafo bi-partido entre clientes e produtos, com aresta  $(c_i, p_j)$  caso cliente  $i$  já comprou produto  $j$ . O fluxo de cada aresta possui limite inferior 0 e limite superior 1. Para representar os limites de perguntas por produto e por cliente, introduziremos ainda dois vértices  $s$ , e  $t$ , com arestas  $(s, c_i)$  com fluxo entre  $c_i$  e  $c'_i$  e arestas  $(p_j, t)$  com fluxo entre  $p_j$  e  $p'_j$  e uma aresta  $(t, s)$ .

**Segmentação de imagens** O objetivo é segmentar uma imagem em duas partes, por exemplo “foreground” e “background”. Supondo que temos uma “probabilidade”  $a_i$  de pertencer ao “foreground” e outra “probabilidade” de pertencer ao “background”  $b_i$  para cada pixel  $i$ , uma abordagem direta é definir que pixels com  $a_i > b_i$  são “foreground” e os outros “background”. Um exemplo pode ser visto na Fig. 1.17 (b). A desvantagem dessa abordagem é que a separação ignora o contexto de um pixel. Um pixel, “foreground” com todos os pixels adjacentes em “background” provavelmente pertence ao “background” também. Portanto obtemos um modelo melhor introduzindo penalidades  $p_{ij}$  para separar (atribuir à categorias diferentes) pixel adjacentes  $i$  e  $j$ . Um particionamento do conjunto de todos os pixels  $I$  em  $A \dot{\cup} B$  tem um valor de

$$q(A, B) = \sum_{i \in A} a_i + \sum_{i \in B} b_i - \sum_{(i,j) \in A \times B} p_{ij}$$

nesse modelo, e o nosso objetivo é achar uma partição que maximiza  $q(A, B)$ . Isso é equivalente a minimizar

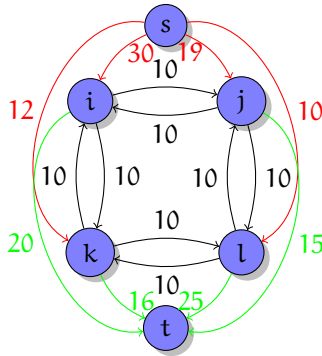
$$\begin{aligned} Q(A, B) &= \sum_{i \in I} a_i + b_i - \sum_{i \in A} a_i - \sum_{i \in B} b_i + \sum_{(i,j) \in A \times B} p_{ij} \\ &= \sum_{i \in B} a_i + \sum_{i \in A} b_i + \sum_{(i,j) \in A \times B} p_{ij}. \end{aligned}$$

A solução mínima de  $Q(A, B)$  pode ser visto como corte mínimo num grafo. O grafo possui um vértice para cada pixel e uma aresta com capacidade  $p_{ij}$  entre dois pixels adjacentes  $i$  e  $j$ . Ele possui ainda dois vértices adicionais  $s$  e  $t$ , arestas  $(s, i)$  com capacidade  $a_i$  para cada pixel  $i$  e arestas  $(i, t)$  com capacidade  $b_i$  para cada pixel  $i$  (ver Fig. 1.16).

**Seqüenciamento** O objetivo é programar um transporte com um número  $k$  de veículos disponíveis, dado pares de origem-destino com tempo de saída e chegada. Um exemplo é um conjunto de vôos é

1. Porto Alegre (POA), 6.00 – Florianopolis (FLN), 7.00
2. Florianopolis (FLN), 8.00 – Rio de Janeiro (GIG), 9.00
3. Fortaleza (FOR), 7.00 – João Pessoa (JPA), 8.00
4. São Paulo (GRU), 11.00 – Manaus (MAO), 14.00
5. Manaus (MAO), 14.15 – Belem (BEL), 15.15

## 1. Algoritmos em grafos



	i	j	k	l
a	30	19	12	10
b	20	15	16	25

Figura 1.16.: Exemplo da construção para uma imagem  $2 \times 2$ . Direita: Tabela com valores pele/não-pele. Esquerda: Grafo com penalidade fixa  $p_{ij} = 10$ .



Figura 1.17.: Segmentação de imagens com diferentes penalidades  $p$ . Acima: (a) Imagem original (b) Segmentação somente com probabilidades ( $p = 0$ ) (c)  $p = 1000$  (d)  $p = 10000$ . Abaixo: (a) Walter Gramatté, Selbstbildnis mit rotem Mond, 1926 (b) Segmentação com  $p = 10000$ . A probabilidade de um pixel representar pele foi determinado conforme Jones e Rehg [28].

## 6. Salvador (SSA), 17.00 – Recife (REC), 18.00

O mesmo avião pode ser usado para mais que um par de origem e destino, se o destino do primeiro é o origem do segundo, em tem tempo suficiente entre a chegada e saída (para manutenção, limpeza, etc.) ou tem tempo suficiente para deslocar o avião do destino para o origem.

Podemos representar o problema como grafo direcionado acíclico. Dado pares de origem destino, ainda adicionamos pares de destino-origem que são compatíveis com as regras acima. A idéia é representar aviões como fluxo: cada aresta origem-destino é obrigatório, e portanto recebe limites inferiores e superiores de 1, enquanto uma aresta destino-origem é facultativa e recebe limite inferior de 0 e superior de 1. Além disso, introduzimos dois vértices  $s$  e  $t$ , com arcos facultativos de  $s$  para qualquer origem e de qualquer destino para  $t$ , que representam os começos e finais da viagem completa de um avião. Para decidir se existe um solução com  $k$  aviões, finalmente colocamos um arco  $(t, s)$  com limite inferior de 0 e superior de  $k$  e decidir se existe uma circulação nesse grafo.

**O problema**  $P \mid \text{pmtn}, r_i \mid L_{\max}$  Primeiramente resolveremos um problema mais simples: será que existe um sequenciamento tal que toda tarefa  $i$  executa dentro do seu intervalo  $[r_i, d_i]$ ? Equivalentemente, será que existe uma solução com  $L_{\max} = 0$ ?

Seja  $\{t_1, t_2, \dots, t_k\} = \{r_1, r_2, \dots, r_n\} \cup \{d_1, d_2, \dots, d_n\}$ , com  $t_1 \leq t_2 \leq \dots \leq t_k$ . (Observe que  $k \leq 2n$ , e  $k < 2n$  no caso de tempos repetidos.) Podemos ver os  $t_i$  como eventos em que uma tarefa fica disponível ou tem que terminar o seu processamento. Os  $t_i$  definem  $k-1$  intervalos  $I_i = [t_i, t_{i+1}]$  para  $i \in [k-1]$  com duração  $S_i = t_{i+1} - t_i$  correspondente. Cada tarefa  $j$  pode ser executada no intervalo  $T_i$  caso  $I_i \subseteq [r_j, d_j]$ . Logo podemos modelar o problema via um grafo direcionado bipartido com vértices  $T \dot{\cup} I$ , sendo  $T = [n]$  o conjunto de tarefas e  $I = \{I_i \mid i \in [k-1]\}$  o conjunto de intervalos, e com arcos  $(j, i)$  caso tarefa  $j$  pode ser executada no intervalo  $i$ . Para completar o grafo adicionaremos um arco  $(s, j)$  de um vértice origem  $s$  para cada tarefa  $j$ , e um arco  $(i, t)$  de cada intervalo para um vértice destino  $t$ . Um fluxo nesse grafo representa tempo, e teremos capacidades  $p_j$  entre  $s$  e tarefa  $j$ ,  $S_i$  entre tarefa  $j$  e intervalo  $i$ , e  $mS_i$  entre  $T_i$  e  $t$ , sendo  $mS_i$  o tempo total disponível durante o intervalo  $i$ . A figura 1.18 mostra a construção completa.

Logo  $P \mid \text{pmtn}, r_i \mid L_{\max}$  pode ser resolvido em tempo  $O(mn \log \bar{L})$ .

Com essa abordagem podemos resolver o problema original por busca binária: para cada valor do  $L_{\max}$  entre 0 e  $\bar{L}$  testaremos se existe uma solução tal que cada tarefa executa no intervalo  $[r_i, d_i + L_{\max}]$ . Um limite superior simples é

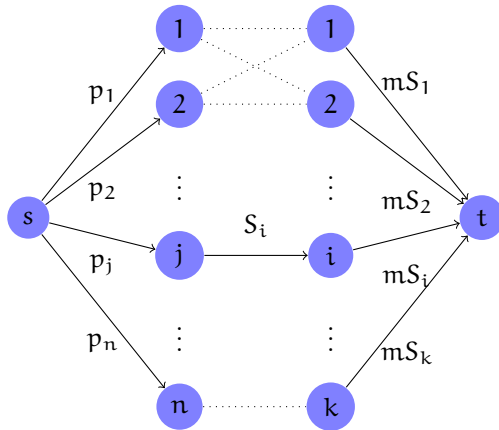


Figura 1.18.: Problema de fluxo para resolver a versão de decisão do problema  $P \mid pmt_n, r_i \mid L_{\max}$ .

$\bar{L} = \max_i r_i + \sum_i p_i - \min_i d_i$  executando todas tarefas após a liberação da última numa única máquina em ordem arbitrária.

#### 1.4.7. Outros problemas de fluxo

Obtemos um outro problema de fluxo em redes introduzindo *custos* de transporte por unidade de fluxo:

##### FLUXO DE MENOR CUSTO

**Entrada** Grafo direcionado  $G = (V, E)$  com capacidades  $c \in \mathbb{R}_+^{|E|}$  e custos  $r \in \mathbb{R}_+^{|E|}$  nos arcos, um vértice origem  $s \in V$ , um vértice destino  $t \in V$ , e valor  $v \in \mathbb{R}_+$ .

**Solução** Um fluxo  $s$ - $t$   $f$  com valor  $v$ .

**Objetivo** Minimizar o *custo*  $\sum_{e \in E} c_e f_e$  do fluxo.

Diferente do problema de menor fluxo, o valor do fluxo é fixo.

#### 1.4.8. Exercícios

##### Exercício 1.6

Mostra como podemos modificar o algoritmo de Dijkstra para encontrar o caminho mais curto entre dois vértices num um grafo para encontrar o cami-



nho com o maior gargalo entre dois vértices. (Dica: Enquanto o algoritmo de Dijkstra procura o caminho com a menor soma de distâncias, estamos procurando o caminho com o maior capacidade mínimo.)

## 1.5. Emparelhamentos

Dado um grafo não-direcionado  $G = (V, E)$ , um *emparelhamento* é uma seleção de arestas  $M \subseteq E$  tal que todo vértice tem no máximo grau 1 em  $G' = (V, M)$ . (Notação:  $M = \{u_1v_1, u_2v_2, \dots\}$ .) O nosso interesse em emparelhamentos é maximizar o número de arestas selecionados ou, no caso as arestas possuem pesos, maximizar o peso total das arestas selecionados.

Para um grafo com pesos  $c : E \rightarrow \mathbb{Q}$ , seja  $c(M) = \sum_{e \in M} c_e$  o *valor* do emparelhamento  $M$ .

### EMPARELHAMENTO MÁXIMO (EM)

**Entrada** Um grafo não-direcionado  $G = (V, E)$ .

**Solução** Um emparelhamento  $M \subseteq E$ , i.e. um conjunto de arcos, tal que para todos vértices  $v$  temos  $|N(v) \cap M| \leq 1$ .

**Objetivo** Maximiza  $|M|$ .

### EMPARELHAMENTO DE PESO MÁXIMO (EPM)

**Entrada** Um grafo não-direcionado  $G = (V, E, c)$  com pesos  $c : E \rightarrow \mathbb{Q}$  nas arestas.

**Solução** Um emparelhamento  $M \subseteq E$ .

**Objetivo** Maximiza o valor  $c(M)$  de  $M$ .

Um emparelhamento se chama *perfeito* se todo vértice possui vizinho em  $M$ . Uma variação comum do problema é

### EMPARELHAMENTO PERFEITO DE PESO MÍNIMO (EPPM)

**Entrada** Um grafo não-direcionado  $G = (V, E, c)$  com pesos  $c : E \rightarrow \mathbb{Q}$  nas arestas.

**Solução** Um emparelhamento perfeito  $M \subseteq E$ , i.e. um conjunto de arcos, tal que para todos vértices  $v$  temos  $|N(v) \cap M| = 1$ .

**Objetivo** Minimiza o valor  $c(M)$  de  $M$ .

Observe que os pesos em todos problemas podem ser negativos. O problema de encontrar um emparelhamento de peso mínimo em  $G = (V, E, c)$  é equivalente

com EPM em  $-G := (V, E, -c)$  (por quê?). Até EPPM pode ser reduzido para EPM.

**Teorema 1.7**

EPM e EPPM são problemas equivalentes.

**Prova.** Seja  $G = (V, E, c)$  uma instância de EPM. Define um conjunto de vértices  $V'$  que contém  $V$  e mais  $|V|$  novos vértices e um grafo completo  $G' = (V', V' \times V', c')$  com

$$c'_e = \begin{cases} -c_e & \text{caso } e \in E \\ 0 & \text{caso contrário} \end{cases}.$$

Dado um emparelhamento  $M$  em  $G$  podemos definir um emparelhamento perfeito  $M'$  em  $G'$ :  $M'$  inclui todas arestas em  $M$ . Além disso, um vértice em  $V$  não emparelhado em  $M$  será emparelhado com o novo vértice correspondente em  $V'$  com uma aresta de custo 0 em  $M'$ . Similarmente, os restantes vértices não emparelhados em  $V'$  são emparelhados em  $M'$  com arestas de custo 0 entre si. Pela construção, o valor de  $M'$  é  $c'(M') = -c(M)$ . Dado um emparelhamento  $M'$  em  $G'$  podemos obter um emparelhamento  $M$  em  $G$  com valor  $-c(M')$  removendo as arestas que não pertencem a  $G$ . Portanto, um EPPM em  $G'$  é um EPM em  $G$ .

Conversamente, seja  $G = (V, E, c)$  uma instância de EPPM. Define  $C := 1 + \sum_{e \in E} |c_e|$ , novos pesos  $c'_e = C - c_e$  e um grafo  $G' = (V, E, c')$ . Para emparelhamentos  $M_1$  e  $M_2$  arbitrários temos

$$c(M_2) - c(M_1) \leq \sum_{\substack{e \in E \\ c_e > 0}} c_e - \sum_{\substack{e \in E \\ c_e < 0}} c_e = \sum_{e \in E} |c_e| < C.$$

Portanto, um emparelhamento de peso máximo em  $G'$  também é um emparelhamento de cardinalidade máxima: Para  $|M_1| < |M_2|$  temos

$$c'(M_1) = C|M_1| - c(M_1) < C|M_1| + C - c(M_2) \leq C|M_2| - c(M_2) = c'(M_2).$$

Se existe um emparelhamento perfeito no grafo original  $G$ , então o EPM em  $G'$  é perfeito e as arestas do EPM em  $G'$  definem um EPPM em  $G$ . ■

**Formulações com programação inteira** A formulação do problema do emparelhamento perfeito mínimo para  $G = (V, E, c)$  é

$$\begin{aligned} \text{minimiza} \quad & \sum_{e \in E} c_e x_e \\ \text{sujeito a} \quad & \sum_{u \in N(v)} x_{uv} = 1, \quad \forall v \in V \\ & x_e \in \mathbb{B}. \end{aligned} \tag{1.12}$$

A formulação do problema do emparelhamento máximo é

$$\begin{aligned} \text{maximiza} \quad & \sum_{e \in E} c_e x_e \\ \text{sujeito a} \quad & \sum_{u \in N(v)} x_{uv} \leq 1, \quad \forall v \in V \\ & x_e \in \mathbb{B}. \end{aligned} \tag{1.13}$$

**Observação 1.13**

A matriz de coeficientes de (1.12) e (1.13) é totalmente unimodular no caso bipartido (pelo teorema de Hoffman-Kruskal). Portanto: a solução da relaxação linear é inteira. (No caso geral isso não é verdadeiro,  $K_3$  é um contra-exemplo, com solução ótima  $3/2$ .) Observe que isso resolve o caso ponderado sem custo adicional.  $\diamond$

**Observação 1.14**

O dual da relaxação linear de (1.12) é

$$\begin{aligned} \text{CIM: maximiza} \quad & \sum_{v \in V} y_v \\ \text{sujeito a} \quad & y_u + y_v \leq c_{uv}, \quad \forall uv \in E \\ & y_v \in \mathbb{R}. \end{aligned} \tag{1.14}$$

e o dual da relaxação linear de (1.13)

$$\begin{aligned} \text{MVC: minimiza} \quad & \sum_{v \in V} y_v \\ \text{sujeito a} \quad & y_u + y_v \geq c_{uv}, \quad \forall uv \in E \\ & y_v \in \mathbb{R}_+. \end{aligned} \tag{1.15}$$

Com pesos unitários  $c_{uv} = 1$  e restringindo  $y_v \in \mathbb{B}$  o primeiro dual é a formulação do conjunto independente máximo e o segundo da cobertura por vértices mínima. Portanto, a observação 1.13 rende no caso não-ponderado:

**Teorema 1.8 (Berge, 1951)**

Em grafos bi-partidos o tamanho da menor cobertura por vértices é igual ao tamanho do emparelhamento máximo.

**Proposição 1.4**

Um subconjunto de vértices  $I \subseteq V$  de um grafo não-direcionado  $G = (V, A)$  é um conjunto independente sse  $V \setminus I$  é uma cobertura por vértices. Em particular um conjunto independente máximo  $I$  corresponde com uma cobertura por vértices mínima  $V \setminus I$ .

**Prova.** (Exercício 1.8.)

**1.5.1. Aplicações**

**Alocação de tarefas** Queremos alocar  $n$  tarefas a  $n$  trabalhadores, tal que cada tarefa é executada, e cada trabalhador executa uma tarefa. O custos de execução dependem do trabalhador e da tarefa. Isso pode ser resolvido como problema de emparelhamento perfeito mínimo.

**Particionamento de polígonos ortogonais****Teorema 1.9 (Sack e Urrutia [37, cap. 11, th. 1])**

Um polígono ortogonal com  $n$  vértices de reflexo (ingl. reflex vertex, i.e., com ângulo interno maior que  $\pi$ ),  $h$  buracos (ingl. holes) pode ser minimalmente particionado em  $n - l - h + 1$  retângulos. A variável  $l$  é o número máximo de cordas (diagonais) horizontais ou verticais entre vértices de reflexo sem intersecção.

O número  $l$  é o tamanho do conjunto independente máximo no grafo de intersecção das cordas: cada corda é representada por um vértice, e uma aresta representa a duas cordas com intersecção. Pela proposição 1.6 podemos obter uma cobertura mínima via um emparelhamento máximo, que é o complemento de um conjunto independente máximo. Podemos achar o emparelhamento em tempo  $O(n^{5/2})$  usando o algoritmo de Hopcroft-Karp, porque o grafo de intersecção é bi-partido (por quê?).

**1.5.2. Grafos bi-partidos**

Na formulação como programa inteira a solução do caso bi-partido é mais fácil. Isso também é o caso para algoritmos combinatoriais, e portanto começamos estudar grafos bi-partidos.

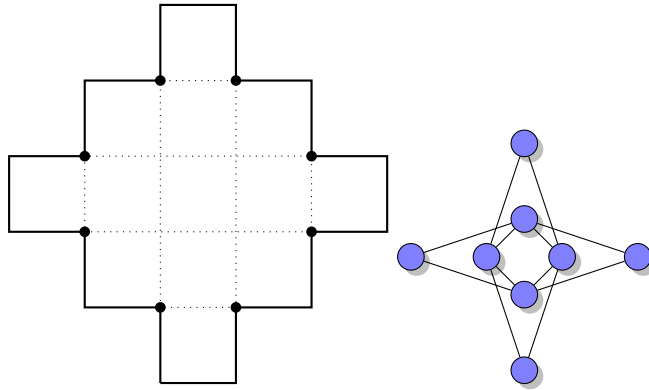


Figura 1.19.: Esquerda: Polígono ortogonal com vértices de reflexo (pontos) e cordas (pontilhadas). Direita: grafo de intersecção.

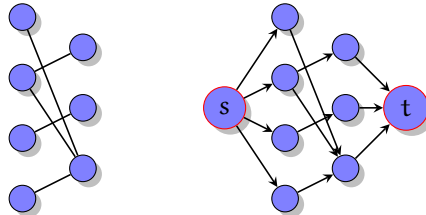


Figura 1.20.: Redução do problema de emparelhamento máximo para o problema do fluxo máximo

### Redução para o problema do fluxo máximo

#### Teorema 1.10

Um EM em grafos bi-partidos pode ser obtido em tempo  $O(mn)$ .

**Prova.** Introduz dois vértices  $s$ ,  $t$ , liga  $s$  para todos vértices em  $V_1$ , os vértices em  $V_1$  com vértices em  $V_2$  e os vértices em  $V_2$  com  $t$ , com todos os pesos unitários. Aplica o algoritmo de Ford-Fulkerson para obter um fluxo máximo. O número de aumentos é limitado por  $n$ , cada busca tem complexidade  $O(m)$ , portanto o algoritmo de Ford-Fulkerson termina em tempo  $O(mn)$ . ■

#### Teorema 1.11

O valor do fluxo máximo é igual a cardinalidade de um emparelhamento máximo.

**Prova.** Dado um emparelhamento máximo  $M = \{v_{11}v_{21}, \dots, v_{1n}v_{2n}\}$ , podemos construir um fluxo com arcos  $sv_{1i}$ ,  $v_{1i}v_{2i}$  e  $v_{2i}t$  com valor  $|M|$ .

Dado um fluxo máximo, existe um fluxo integral equivalente (veja lema (1.13)). Na construção acima os arcos possuem fluxo 0 ou 1. Escolhe todos arcos entre  $V_1$  e  $V_2$  com fluxo 1. Não existe vértice com grau 2, pela conservação de fluxo. Portanto, os arcos formam um emparelhamento cuja cardinalidade é o valor do fluxo. ■

**Solução não-ponderado combinatorial** Um caminho  $P = v_1v_2v_3 \dots v_k$  é *alternante* em relação a  $M$  (ou  $M$ -alternante) se  $v_i v_{i+1} \in M$  sse  $v_{i+1} v_{i+2} \notin M$  para todos  $1 \leq i \leq k-2$ . Um vértice  $v \in V$  é *livre* em relação a  $M$  se ele tem grau 0 em  $M$ , e *emparelhado* caso contrário. Um arco  $e \in E$  é *livre* em relação a  $M$ , se  $e \notin M$ , e *emparelhado* caso contrário. Escrevemos  $|P| = k-1$  pelo *comprimento* do caminho  $P$ .

### Observação 1.15

Caso temos um caminho  $P = v_1v_2v_3 \dots v_{2k+1}$  que é  $M$ -alternante com  $v_1$  é  $v_{2k+1}$  livre, podemos obter um emparelhamento  $M \setminus (P \cap M) \cup (P \setminus M)$  de tamanho  $|M| - k + (k-1) = |M| + 1$ . Notação: Diferença simétrica  $M \oplus P = (M \setminus P) \cup (P \setminus M)$ . A operação  $M \oplus P$  é um *aumento* do emparelhamento  $M$ . ◇

### Teorema 1.12 (Hopcroft e Karp [27])

Seja  $M^*$  um emparelhamento máximo e  $M$  um emparelhamento arbitrário. O conjunto  $M \oplus M^*$  contém pelo menos  $k = |M^*| - |M|$  caminhos  $M$ -aumentantes disjuntos (de vértices). Um deles possui comprimento menor que  $|V|/k - 1$ .

**Prova.** Considere os componentes de  $G$  em relação aos arcos  $M \oplus M^*$ . Cada vértice possui no máximo grau 2. Portanto, os componentes são vértices livres, caminhos simples ou ciclo. Os caminhos e ciclos possuem alternadamente arestas de  $M$  e  $M^*$ , logo os ciclos tem comprimento par. Os caminhos de comprimento ímpar são ou  $M$ -aumentantes ou  $M^*$ -aumentantes, mas o segundo caso é impossível, porque  $M^*$  é máximo. Logo

$$|M^* \setminus M| = |M^*| - |M^* \cap M| = |M| - |M^* \cap M| + k = |M \setminus M^*| + k$$

e portanto  $M \oplus M^*$  contém  $k$  arcos mais de  $M^*$  que de  $M$ . Isso mostra que existem pelo menos  $|M^*| - |M|$  caminhos  $M$ -aumentantes, porque somente os caminhos de comprimento ímpar possuem exatamente um arco mais de  $M^*$ . Pelo menos um desses caminhos tem que ter um comprimento menor ou igual que  $|V|/k - 1$ , porque no caso contrário eles contém em total mais que  $|V|$  vértices. ■

**Corolário 1.3 (Berge [5])**

Um emparelhamento é máximo sse não existe um caminho  $M$ -aumentante.

Rascunho de um algoritmo:

**Algoritmo 1.6 (Emparelhamento máximo)**

**Entrada** Grafo não-direcionado  $G = (V, E)$ .

**Saída** Um emparelhamento máximo  $M$ .

```

1   $M = \emptyset$ 
2  while (existe um caminho  $M$  aumentante  $P$ ) do
3     $M := M \oplus P$ 
4  end while
5  return  $M$ 
```

Problema: como achar caminhos  $M$ -aumentantes de forma eficiente?

**Observação 1.16**

Um caminho  $M$ -aumentante começa num vértice livre em  $V_1$  e termina num vértice livre em  $V_2$ . Idéia: Começa uma busca por largura com todos vértices livres em  $V_1$ . Segue alternadamente arcos livres em  $M$  para encontrar vizinhos em  $V_2$  e arcos em  $M$ , para encontrar vizinhos em  $V_1$ . A busca para ao encontrar um vértice livre em  $V_2$  ou após de visitar todos vértices. Ela tem complexidade  $O(m + n)$ .  $\diamond$

**Teorema 1.13**

O problema do emparelhamento máximo não-ponderado em grafos bi-partidos pode ser resolvido em tempo  $O(mn)$ .

**Prova.** Última observação e o fato que o emparelhamento máximo tem tamanho  $O(n)$ .  $\blacksquare$

**Observação 1.17**

O último teorema é o mesmo que teorema (1.10).  $\diamond$

**Observação 1.18**

Pelo teorema (1.12) sabemos que existem vários caminhos  $M$ -alternantes disjuntos (de vértices) e nos podemos aumentar  $M$  com todos eles em paralelo. Portanto, estruturamos o algoritmo em fases: cada fase procura um conjunto de caminhos aumentantes disjuntos e aplicá-los para obter um novo emparelhamento. Observe que pelo teorema (1.12) um aumento com o maior conjunto de caminhos  $M$ -alternantes disjuntos resolve o problema imediatamente, mas não sabemos como achar esse conjunto de forma eficiente. Portanto, procuramos somente um conjunto máximo de caminhos  $M$ -alternantes disjuntos de menor comprimento.



Podemos achar um conjunto desse tipo após uma busca por profundidade da seguinte maneira usando o DAG (grafo direcionado acíclico) definido pela busca por profundidade. (i) Escolhe um vértice livre em  $V_2$ . (ii) Segue os predecessores para achar um caminho aumentante. (iii) Coloca todos vértices em uma fila de deleção. (iv) Processa a fila de deleção: Até a fila é vazia, remove um vértice dela. Remove todos arcos adjacentes no DAG. Caso um vértice sucessor após de remoção de um arco possui grau de entrada 0, coloca ele na fila. (v) Repete o procedimento no DAG restante, para achar outro caminho, até não existem mais vértices livres em  $V_2$ . A nova busca ainda possui complexidade  $O(m)$ .  $\diamond$

O que ganhamos com essa nova busca? Os seguintes dois lemas dão a resposta:

**Lema 1.27**

Em cada fase o comprimento de um caminho aumentante mínimo aumenta por pelo menos dois.

**Lema 1.28**

O algoritmo termina em no máximo  $\sqrt{n}$  fases.

**Teorema 1.14**

O problema do emparelhamento máximo não-ponderado em grafos bi-partidos pode ser resolvido em tempo  $O(m\sqrt{n})$ .

**Prova.** Pelas lemas 1.27 e 1.28 e a observação que toda fase pode ser completada em  $O(m)$ .  $\blacksquare$

Usaremos outro lema para provar os dois lemas acima.

**Lema 1.29**

Seja  $M$  um emparelhamento,  $P$  um caminho  $M$ -aumentante mínimo, e  $Q$  um caminho  $M \oplus P$ -aumentante. Então  $|Q| \geq |P| + 2|P \cap Q|$ . ( $P \cap Q$  denota as arestas em comum entre  $P$  e  $Q$ .)

**Prova.** Caso  $P$  e  $Q$  não possuem vértices em comum,  $Q$  é  $M$ -aumentante,  $P \cap Q = \emptyset$  e a desigualdade é consequência da minimalidade de  $P$ .

Caso contrário,  $P$  e  $Q$  possuem um vértice em comum, e logo também uma aresta, senão após da aplicação de  $Q$  um vértice tem grau dois.  $P \oplus Q$  consiste em dois caminhos, e eventualmente um coleção de ciclos. Os dois caminhos são  $M$ -aumentantes, pelas seguintes observações:

1. O início e termino de  $P$  é livre em  $M$ , porque  $P$  é  $M$ -aumentante.
2. O início e termino de  $Q$  é livre em  $M$ : eles não pertencem a  $P$ , porque são livres em  $M \oplus P$ .

## 1. Algoritmos em grafos

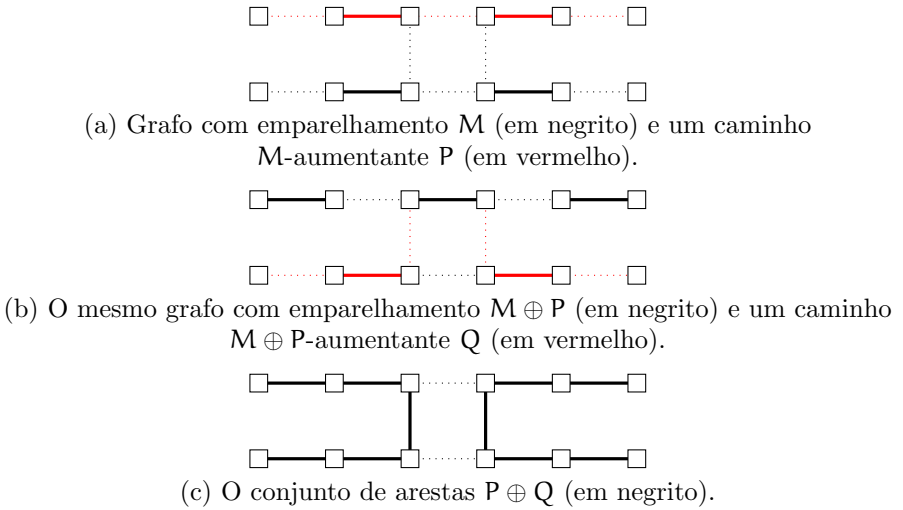


Figura 1.21.: Ilustração do lema 1.29.

3. Nenhum outro vértice de  $P \oplus Q$  é livre em relação a  $M$ :  $P$  só contém dois vértices livres e  $Q$  só contém dois vértices livres em  $Q \setminus P$ .
4. Temos dois caminhos  $M$ -aumentantes, começando com um vértice livre em  $Q$  e terminando com um vértice livre em  $P$ . O parte do caminho  $Q$  em  $Q \setminus P$  é  $M$ -alternante, porque as arestas livres em  $M \oplus P$  são exatamente as arestas livres em  $M$ . O caminho  $Q$  entra em  $P$  e sai de  $P$  com arestas livres, porque todo vértice em  $P$  está emparelhado em  $M \oplus P$ . Portanto os dois caminhos em  $P \oplus Q$  são  $M$ -aumentantes.

Os dois caminhos  $M$ -aumentantes em  $P \oplus Q$  tem que ser maiores que  $|P|$ . Com isso temos  $|P \oplus Q| \geq 2|P|$  e

$$|Q| = |P \oplus Q| + 2|P \cap Q| - |P| \geq |P| + 2|P \cap Q|.$$

**Prova.** (do lema 1.27). Seja  $S$  o conjunto de caminhos  $M$ -aumentantes da fase anterior, e  $P$  um caminho aumentante. Caso  $P$  é disjuncto de todos caminhos em  $S$ , ele deve ser mais comprido, porque  $S$  é um conjunto máximo de caminhos aumentantes. Caso  $P$  possui um vértice em comum com algum caminho em  $S$ , ele possui também um arco em comum (por quê?) e podemos aplicar lema 1.29. ■

**Prova.** (do lema 1.28). Seja  $M^*$  um emparelhamento máximo e  $M$  o emparelhamento obtido após de  $\sqrt{n}/2$  fases. O comprimento de qualquer caminho  $M$ -aumentante é no mínimo  $\sqrt{n}$ , pelo lema 1.27. Pelo teorema 1.12 existem pelo menos  $|M^*| - |M|$  caminhos  $M$ -aumentantes disjuntos de vértices. Mas então  $|M^*| - |M| \leq \sqrt{n}$ , porque no caso contrário eles possuem mais que  $n$  vértices em total. Como o emparelhamento cresce pelo menos um em cada fase, o algoritmo executa no máximo mais  $\sqrt{n}$  fases. Portanto, o número total de fases é  $O(\sqrt{n})$ . ■

O algoritmo de Hopcroft-Karp é o melhor algoritmo conhecido para encontrar emparelhamentos máximos em grafos bipartidos não-ponderados esparsos<sup>4</sup>. Para subclasses de grafos bipartidos existem algoritmos melhores. Por exemplo, existe um algoritmo randomizado para grafos bipartidos regulares com complexidade de tempo esperado  $O(n \log n)$  [22].

**Sobre a implementação** A seguir supomos que o conjunto de vértices é  $V = [1, n]$  e um grafo  $G = (V, E)$  bi-partido com partição  $V_1 \dot{\cup} V_2$ . Podemos representar um emparelhamento usando um vetor **mate**, que contém, para cada vértice emparelhado, o índice do vértice vizinho, e 0 caso o vértice é livre.

O núcleo de uma implementação do algoritmo de Hopcroft e Karp é descrito na observação 1.18: ele consiste numa busca por largura até encontrar um ou mais caminhos  $M$ -alternantes mínimos e depois uma fase que extrai do DAG definido pela busca um conjunto máximo de caminhos disjuntos (de vértices). A busca por largura começa com todos vértices livres em  $V_1$ . Usamos um vetor  $H$  para marcar os arcos que fazem parte do DAG definido pela busca por largura<sup>5</sup> e um vetor  $m$  para marcar os vértices visitados.

```

1  search_paths(M) :=
2    for all v ∈ V do m_v := false
3    for all e ∈ E do H_e := false
4
5    U_1 := {v ∈ V_1 | v livre}
6
7    do
8      { determina vizinhos em U_2 via arestas livres }
9      U_2 := ∅
10     for all u ∈ U_1 do
11       m_u := true
```

<sup>4</sup>Feder e Motwani [16, 17] propuseram um algoritmo em  $O(\sqrt{n}m(2 - \log_n m))$  que é melhor em grafos densos.

<sup>5</sup>H, porque o DAG se chama *árvore húngara* na literatura.

## 1. Algoritmos em grafos

```

12      for all  $uv \in E, uv \notin M$  do
13          if not  $m_v$  then
14               $H_{uv} := \text{true}$ 
15               $U_2 := U_2 \cup v$ 
16          end if
17      end for
18  end for
19
20  { determina vizinhos em  $U_1$  via arestas emparelhadas }
21  found := false      { pelo menos um caminho encontrado? }
22   $U_1 := \emptyset$ 
23  for all  $u \in U_2$  do
24       $m_u := \text{true}$ 
25      if (u livre) then
26          found := true
27      else
28           $v := \text{mate}[u]$ 
29          if not  $m_v$  then
30               $H_{uv} := \text{true}$ 
31               $U_1 := U_1 \cup v$ 
32          end if
33      end for
34  end for
35  while (not found)
36  end
```

Após da busca, podemos extrair um conjunto máximo de caminhos  $M$ -alternantes mínimos disjuntos. Enquanto existe um vértice livre em  $V_2$ , nos extraímos um caminho alternante que termina em  $v$  como segue:

```

1  extract_path( $v$ ) :=
2       $P := v$ 
3      while not ( $v \in V_1$  and  $v$  livre) do
4          if  $v \in V_1$ 
5               $v := \text{mate}[v]$ 
6          else
7               $v := \text{escolhe } \{u \mid H_{uv}, uv \notin M\}$ 
8              (ou remove  $P$  e falha caso não existe predecessor em  $H$ )
9          end if
10          $P := vP$ 
11     end while
12
```

```

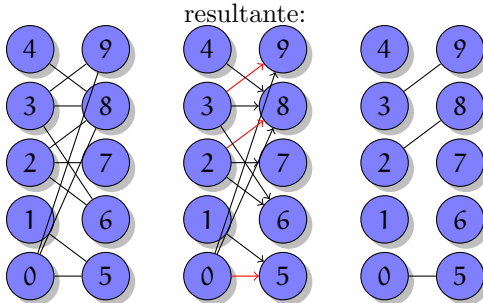
13 remove o caminho e todos vértices sem predecessor
14 end while
15 end

```

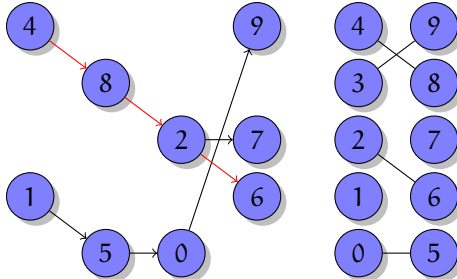
**Exemplo 1.5**

Segue um exemplo de aplicação do algoritmo de Hopcroft-Karp.

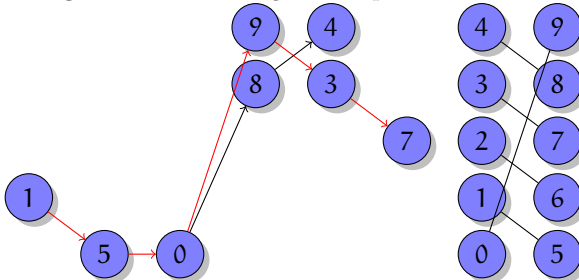
Grafo original, árvore Húngara primeira iteração e emparelhamento



Árvore Húngara segunda iteração e emparelhamento resultante:



Árvore Húngara terceira iteração e emparelhamento resultante:



◇

**Emparelhamentos e coberturas**

### Proposição 1.5

Seja  $G = (S \dot{\cup} T, E)$  um grafo bipartido e  $M \subseteq E$  um emparelhamento em  $G$ . Seja  $R$  o conjunto de todos vértices livres em  $S$  e todos vértices alcançáveis por uma busca na árvore Húngara (i.e. via arestas livres de  $S$  para  $T$  e arestas do emparelhamento de  $T$  para  $S$ ). Então  $(S \setminus R) \cup (T \cap R)$  é uma cobertura por vértices em  $G$ .

**Prova.** Seja  $u, v \in E$  uma aresta não coberta. Logo  $u \in S \setminus (S \setminus R) = R$  e  $v \in T \setminus (T \cap R) = T \setminus R$ . Caso  $uv \notin M$ ,  $uv$  é parte da árvore Húngara e  $v \in R$ , uma contradição. Mas caso  $uv \in M$ ,  $vu$  é parte da árvore Húngara e  $v$  precede  $u$ , logo  $v \in R$ , novamente uma contradição. ■

A próxima proposição mostra que no caso de um emparelhamento máximo obtemos uma cobertura mínima.

### Proposição 1.6

Seja  $G = (S \dot{\cup} T, E)$ . Caso  $M$  é um emparelhamento máximo o conjunto  $(S \setminus R) \cup (T \cap R)$  é uma cobertura mínima.

**Prova.** Como  $M$  é máximo não existe caminho  $M$ -aumentante, e logo  $T \cap R$  contém somente vértices emparelhados. Caso  $|T \cap R| = v$ ,  $R$  também contém exatamente  $v$  vértices emparelhados em  $S$ . Além disso  $R$  contém  $|S| - |M|$  vértices livres em  $S$ . Logo  $|S \setminus R| = |S| - (|S| - |M|) - v = |M| - v$  e  $|(S \setminus R) \cup (T \cap R)| = |M|$ , i.e. a cobertura possui a mesma cardinalidade que o emparelhamento. Mas o tamanho de qualquer emparelhamento é um limite inferior para a cobertura mínima, porque ela tem que conter pelo menos um vértice da cada aresta emparelhada. Logo  $(S \setminus R) \cup (T \cap R)$  é uma cobertura mínima. ■

**Solução ponderada em grafos bi-partidos** Dado um grafo  $G = (S \dot{\cup} T, E)$  bipartido com pesos  $c : E \rightarrow \mathbb{Q}_+$  queremos achar um emparelhamento de maior peso. Escrevemos  $V = S \cup T$  para o conjunto de todos vértices em  $G$ .

### Observação 1.19

O caso ponderado pode ser restrito para emparelhamentos perfeitos: caso  $S$  e  $T$  possuem cardinalidade diferente, podemos adicionar vértices, e depois completar todo grafo com arestas de custo 0. O problema de encontrar um emparelhamento perfeito máximo (ou mínimo) em grafos ponderados é conhecido pelo nome “problema de alocação” (ingl. assignment problem). ◇

### Observação 1.20

A redução do teorema 1.10 para um problema de fluxo máximo não se aplica no caso ponderado. Mas, com a simplificação da observação 1.19, podemos

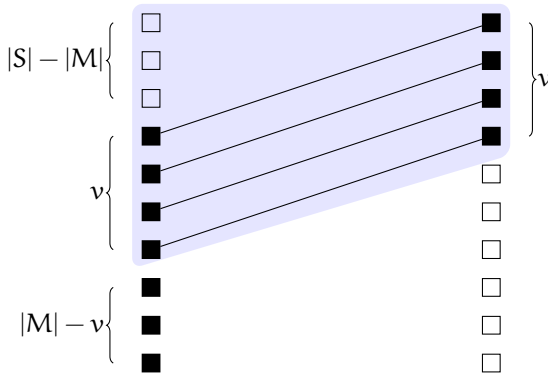


Figura 1.22.: Ilustração da prova da proposição 1.6.

reduzir o problema no caso ponderado para um problema de fluxo de menor custo: a capacidade de todas as arestas é 1, e o custo de transporte são os pesos das arestas. Como o emparelhamento é perfeito, procuramos um fluxo de valor  $|V|/2$ , de menor custo.  $\diamond$

O dual do problema 1.15 é a motivação para

### Definição 1.3

Um *rotulamento* é uma atribuição  $y : V \rightarrow \mathbb{R}_+$ . Ele é *viável* caso  $y_u + y_v \geq c_e$  para todas as arestas  $e = (u, v)$ . (Um rotulamento viável é *c-cobertura por vértices*.) Uma aresta é *apertada* (ingl. tight) caso  $y_u + y_v = c_e$ . O subgrafo de arestas apertadas é  $G_y = (V, E', c)$  com  $E' = \{e \in E \mid e \text{ apertada em } y\}$ .

Pelo teorema forte de dualidade e o fato que a relaxação linear dos sistemas acima possui uma solução integral (ver observação 1.13) temos

### Teorema 1.15 (Egerváry [15])

Para um grafo bi-partido  $G = (S \cup T, E, c)$  com pesos não-negativos  $c : E \rightarrow \mathbb{Q}_+$  nas arestas, o maior peso de um emparelhamento perfeito é igual ao peso da menor *c-cobertura por vértices*.

**O método húngaro** Aplicando um caminho *M*-aumentante  $P = (v_1 v_2 \dots v_{2n+1})$  produz um emparelhamento de peso  $c(M) + \sum_{i \text{ ímpar}} c_{v_i v_{i+1}} - \sum_{i \text{ par}} c_{v_i v_{i+1}}$ . Isso motiva a definição de uma árvore húngara ponderada. Para um emparelhamento  $M$ , seja  $H_M$  o grafo direcionado com as arestas  $e \in M$  orientadas de  $T$  para  $S$  com peso  $l_e := w_e$ , e com as restantes arestas  $e \in E \setminus M$  orientadas de  $S$  para  $T$  com peso  $l_e := -w_e$ . Com isso a aplicação do caminho

## 1. Algoritmos em grafos

$M$ -aumentante  $P$  produz um emparelhamento de peso  $c(M) - l(P)$  em que  $l(P) = \sum_{1 \leq i \leq 2n} l_{v_i v_{i+1}}$  é o comprimento do caminho  $P$ .

Com isso podemos modificar o algoritmo para emparelhamentos máximos para

### Algoritmo 1.7 (Emparelhamento de peso máximo)

**Entrada** Um grafo não-direcionado ponderado  $G = (V, E, c)$ .

**Saída** Um emparelhamento de maior peso  $c(M)$ .

```
1   $M = \emptyset$ 
2  while (existe um caminho  $M$  aumentante  $P$ ) do
3    encontra o caminho  $M$  aumentante mínimo  $P$  em  $H_M$ 
4    caso  $l(P) \geq 0$ : return  $M$ ;
5     $M := M \oplus P$ 
6  end while
7  return  $M$ 
```

Chamaremos um emparelhamento  $M$  *extremo* caso ele possui o maior peso entre todos emparelhamentos de tamanho  $|M|$ .

### Observação 1.21

O grafo  $H_M$  de um emparelhamento extremo  $M$  não possui ciclo (par) negativo. Isso seria uma contradição com a maximalidade de  $M$ . Portanto podemos encontrar o caminho mínimo no passo 3 do algoritmo usando o algoritmo de Bellman-Ford em tempo  $O(mn)$ . Com isso a complexidade do algoritmo é  $O(mn^2)$ .  $\diamond$

### Observação 1.22

Lembrando Bellman-Ford: Seja  $d_k(t)$  a distância mínima entre  $s$  e  $t$  com um caminho usando no máximo  $k$  arcos ou  $\infty$  caso tal caminho não existe. Temos

$$d_{k+1}(t) = \min\{d_k(t), \min_{(u,t) \in A} d_k(u) + l(u,t)\}$$

com  $d_0(t) = 0$  caso  $t$  é um vértice livre em  $S$  e  $d_0(t) = \infty$  caso contrário. O algoritmo se aplica igualmente para as distâncias de um conjunto de vértices, como o conjunto de vértices livres em  $S$ . A atualização de  $k$  para  $k+1$  é possível em  $O(m)$  e como  $k < n$  o algoritmo possui complexidade  $O(nm)$ .  $\diamond$

### Teorema 1.16

Cada emparelhamento encontrado no algoritmo 1.7 é extremo.

**Prova.** Por indução sobre  $|M|$ . Para  $M = \emptyset$  o teorema é correto. Seja  $M$  um emparelhamento extremo,  $P$  o caminho aumentante encontrado pelo algoritmo 1.7 e  $N$  um emparelhamento de tamanho  $|M|+1$  arbitrário. Como  $|N| >$



$|M|$ ,  $M \cup N$  contém uma componente que é um caminho  $Q$   $M$ -aumentante (por um argumento similar com aquele da prova do teorema de Hopcroft-Karp 1.12). Sabemos  $l(Q) \geq l(P)$  pela minimalidade de  $P$ .  $N \oplus Q$  é um emparelhamento de cardinalidade  $|M|$  ( $Q$  é um caminho com arestas em  $N$  e  $M$  com uma aresta em  $N$  a mais), logo  $c(N \oplus Q) \leq c(M)$ . Com isso temos

$$c(N) = c(N \oplus Q) - l(Q) \leq c(M) - l(P) = c(M \oplus P)$$

(observe que o comprimento  $l(Q)$  é definido no emparelhamento  $M$ ). ■

### Proposição 1.7

Caso não existe caminho  $M$ -aumentante com comprimento negativo no algoritmo 1.7,  $M$  é máximo.

**Prova.** Supõe que existe um emparelhamento  $N$  com  $c(N) > c(M)$ . Logo  $|N| > |M|$  porque  $M$  é possui o maior peso entre todos emparelhamentos de cardinalidade no máximo  $|M|$ . Pelo teorema de Hopcroft-Karp, existem  $|N| - |M|$  caminhos  $M$ -aumentantes disjuntos de vértices em  $N \oplus M$ . Nenhum deles tem comprimento negativo, pelo critério de parada do algoritmo. Portanto  $c(N) \leq c(M)$ , uma contradição. ■

### Fato 1.1

É possível encontrar o caminho mínimo no passo 3 em tempo  $O(m + n \log n)$  usando uma transformação para distâncias positivas e aplicando o algoritmo de Dijkstra. Com isso um algoritmo em tempo  $O(n(m + n \log n))$  é possível.

### 1.5.3. Emparelhamentos em grafos não-bipartidos

**O caso não-ponderado** Dado um grafo não-direcionado  $G = (V, E)$  e um emparelhamento  $M$ , podemos simplificar a árvore húngara para um grafo direcionado  $D = (V, A)$  com  $A = \{(u, v) \mid \exists x \in V : ux \in E, xv \in M\}$ . Qualquer passeio  $M$ -alternante entre dois vértices livres em  $G$  corresponde com um caminho  $M$ -alternante em  $D$ .

O problema no caso não-bipartido são laços ímpares. No caso bi-partido, todo laço é par e pode ser eliminado sem consequências: de fato o caminho  $M$ -alternante mais curto não possui laço. No caso não bi-partido não todo caminho no grafo auxiliar corresponde com um caminho  $M$ -alternante no grafo original. O caminho  $v_1v_3v_5v_7v_9$  corresponde com o caminho  $M$ -alternante  $v_1v_2v_3v_4v_5v_6v_7v_8v_9v_{10}$ , mas o caminho  $v_1v_8c_6v_5v_7v_9$  que corresponde com o passeio  $v_1v_9v_8v_7v_6v_4v_5v_6v_7v_8v_{10}$  não é um caminho  $M$ -alternante que aumento o emparelhamento. O problema é que o laço ímpar  $v_6v_4v_5v_6$  não pode ser eliminado sem consequências.

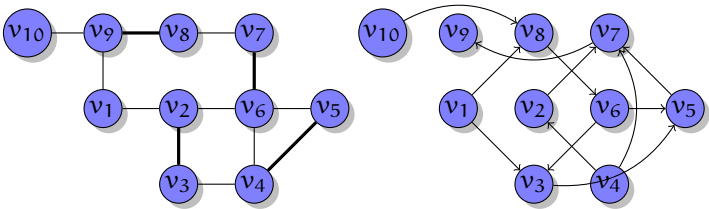


Figura 1.23.: Grafo com emparelhamento e grafo auxiliar.

Tabela 1.4.: Resumo emparelhamentos

	Cardinalidade		Ponderado
Bi-partido	$O(n\sqrt{\frac{mn}{\log n}})$	[3]	$O(nm + n^2 \log n)$ [31, 33]
	$O(m\sqrt{n\frac{\log(n^2/m)}{\log n}})$ [17]		
Geral	$O(m\sqrt{n\frac{\log(n^2/m)}{\log n}})$ [23, 19]		$O(n^3)$ [13] $O(mn + n^2 \log n)$ [21]

1.5.4. Exercícios

Exercício 1.7

É possível somar uma constante  $c \in \mathbb{R}$  para todos custos de uma instância do EPM ou EPPM, mantendo a otimalidade da solução?

Exercício 1.8

Prova a proposição 1.4.

## 2. Tabelas hash

Em *hashing* nosso interesse é uma estrutura de dados  $H$  para gerenciar um conjunto de chaves sobre um universo  $U$  e que oferece as operações de um *dicionário*:

- Inserção de uma chave  $c \in U$ :  $\text{insert}(c, H)$
- Deleção de uma chave  $c \in U$ :  $\text{delete}(c, H)$
- Teste da pertinência: Chave  $c \in H$ ?  $\text{lookup}(c, H)$

Uma característica do problema é que tamanho  $|U|$  do universo de chaves possíveis pode ser grande, por exemplo o conjunto de todos strings ou todos números inteiros. Portanto usar a chave como índice de um vetor de booleano não é uma opção. Uma tabela hash é uma alternativa para outras estruturas de dados de dicionários, p.ex. árvores. O princípio de tabelas hash: aloca uma tabela de tamanho  $m$  e usa uma *função hash* para calcular a posição de uma chave na tabela. Como o tamanho da tabela hash é menor que o número de chaves possíveis, existem chaves com  $h(c_1) = h(c_2)$ , que geram *colisões*. Temos dois métodos para lidar com isso:

- *Hashing perfeito*: Escolhe uma função hash, que para um dado conjunto de chaves não tem colisões. Isso é possível se o conjunto de chaves é conhecido e estático.
- Invente outro método de *resolução de colisões*.

### 2.1. Hashing com listas encadeadas

Define uma função hash  $h : U \rightarrow [m]$ . Mantemos uma coleção de  $m$  listas  $l_0, \dots, l_{m-1}$  e a lista  $l_i$  contém as chaves  $c$  com *valor hash*  $h(c) = i$ . Supondo que a avaliação de  $h$  é possível em  $O(1)$ , a inserção custa  $O(1)$ , e o teste é proporcional ao tamanho da lista.

Para obter uma distribuição razoável das chaves nas listas, supomos que  $h$  é uma função hash *simples* e *uniforme*:

$$\Pr[h(c) = i] = 1/m. \quad (2.1)$$

## 2. Tabelas hash

Seja  $n_i := |l_i|$  o tamanho da lista  $i$  e  $c_{ji} := \Pr[h(i) = j]$  a variável aleatória que indica se chave  $j$  pertence a lista  $i$ . Temos  $n_i = \sum_{1 \leq j \leq n} c_{ji}$  e com isso

$$E[n_i] = E\left[\sum_{1 \leq j \leq n} c_{ji}\right] = \sum_{1 \leq j \leq n} E[c_{ji}] = \sum_{1 \leq j \leq n} \Pr[h(c_j) = i] = n/m.$$

O valor  $\alpha := n/m$  é a *fator de ocupação* da tabela hash.

```

1  insert(c, H) :=
2      insert(c, lh(c))
3
4  lookup(c, H) :=
5      lookup(c, lh(c))
6
7  delete(c, H) :=
8      delete(c, lh(c))

```

### Teorema 2.1

Uma busca sem sucesso precisa tempo esperado de  $\Theta(1 + \alpha)$ .

**Prova.** A chave  $c$  tem a probabilidade  $1/m$  de ter um valor hash  $i$ . O tamanho esperado da lista  $i$  é  $\alpha$ . Uma busca sem sucesso nessa lista precisa tempo  $\Theta(\alpha)$ . Junto com a avaliação da função hash em  $\Theta(1)$ , obtemos tempo esperado total  $\Theta(1 + \alpha)$ . ■

### Teorema 2.2

Uma busca com sucesso precisa tempo esperado de  $\Theta(1 + \alpha)$ .

**Prova.** Supomos que a chave  $c$  é uma das chaves na tabela com probabilidade uniforme. Então, a probabilidade de pertencer a lista  $i$  (ter valor hash  $i$ ) é  $n_i/n$ . Uma busca com sucesso toma tempo  $\Theta(1)$  para avaliação da função hash, e mais um número de operações proporcional à posição  $p$  da chave na sua lista. Com isso obtemos tempo esperado  $\Theta(1 + E[p])$ .

Para determinar a posição esperada na lista,  $E[p]$ , seja  $c_1, \dots, c_n$  a sequência em que as chaves foram inseridas. Supondo que inserimos as chaves no início da lista,  $E[p]$  é um mais que o número de chaves inseridos depois de  $c$  na mesma lista.

Seja  $X_{ij}$  um variável aleatória que indica se chaves  $c_i$  e  $c_j$  tem o mesmo valor hash.  $E[X_{ij}] = \Pr[h(c_i) = h(c_j)] = \sum_{1 \leq k \leq m} \Pr[h(c_i) = k] \Pr[h(c_j) = k] = 1/m$ . Seja  $p_i$  a posição da chave  $c_i$  na sua lista. Temos

$$E[p_i] = E\left[1 + \sum_{j:j>i} X_{ij}\right] = 1 + \sum_{j:j>i} E[X_{ij}] = 1 + (n - i)/m$$

e para uma chave aleatória  $c$

$$\begin{aligned} E[p] &= \sum_{1 \leq i \leq n} 1/n \quad E[p_i] = \sum_{1 \leq i \leq n} 1/n(1 + (n - i)/m) \\ &= 1 + n/m - (n + 1)/(2m) = 1 + \alpha/2 - \alpha/(2n). \end{aligned}$$

Portanto, o tempo esperado de uma busca com sucesso é

$$\Theta(1 + E[p]) = \Theta(2 + \alpha/2 - \alpha/2n) = \Theta(1 + \alpha).$$

■

**Seleção de uma função hash** Para implementar uma tabela hash, temos que escolher uma função hash, que satisfaz (2.1). Para facilitar isso, supomos que o universo de chaves é um conjunto  $U = [u]$  de números inteiros. (Para tratar outro tipos de chave, costuma-se convertê-los para números inteiros.) Se cada chave ocorre com a mesma probabilidade,  $h(i) = i \bmod m$  é uma função hash simples e uniforme. Essa abordagem é conhecida como *método de divisão*. O problema com essa função na prática é que não conhecemos a distribuição de chaves, e ela provavelmente não é uniforme. Por exemplo, se  $m$  é par, o valor hash de chaves pares é par, e de chaves ímpares é ímpar, e se  $m = 2^k$  o valor hash consiste nos primeiros  $k$  bits. Uma escolha que funciona na prática é um número primo “suficientemente” distante de uma potência de 2.

O *método de multiplicação* define

$$h(c) = \lfloor m \{Ac\} \rfloor.$$

O método funciona para qualquer valor de  $m$ , mas depende de uma escolha adequada de  $A \in \mathbb{R}$ . Knuth propôs  $A \approx (\sqrt{5} - 1)/2$ .

**Hashing universal** Outra idéia: Para qualquer função hash  $h$  fixa, sempre existe um conjunto de chaves, tal que essa função hash gera muitas colisões. (Em particular, um “adversário” que conhece a função hash pode escolher chaves  $c$ , tal que  $h(c) = \text{const.}$ . Para evitar isso podemos escolher uma função hash randômica de uma família de funções hash.

Uma família  $\mathcal{H}$  de funções hash  $U \rightarrow [m]$  é *universal* se

$$|\{h \in \mathcal{H} \mid h(c_1) = h(c_2)\}| = |\mathcal{H}|/m$$

ou equivalente

$$\Pr[h(c_1) = h(c_2)] = 1/m$$

para qualquer par de chaves  $c_1, c_2$ .

### Teorema 2.3

Se escolhermos uma função hash  $h \in \mathcal{H}$  uniformemente, para uma chave arbitrária  $c$  o tamanho esperado de  $l_{h(c)}$  é

- $\alpha$ , caso  $c \notin H$ , e
- $1 + \alpha$ , caso  $c \in H$ .

**Prova.** Para chaves  $c_1, c_2$  seja  $X_{ij} = [h(c_1) = h(c_2)]$  e temos

$$E[X_{ij}] = \Pr[X_{ij} = 1] = \Pr[h(c_1) = h(c_2)] = 1/m$$

pela universalidade de  $\mathcal{H}$ . Para uma chave fixa  $c$  seja  $Y_c$  o número de colisões.

$$E[Y_c] = E\left[\sum_{\substack{c' \in H \\ c' \neq c}} X_{cc'}\right] = \sum_{\substack{c' \in H \\ c' \neq c}} E[X_{cc'}] \leq \sum_{\substack{c' \in H \\ c' \neq c}} 1/m.$$

Para uma chave  $c \notin H$ , o tamanho da lista é  $Y_c$ , e portanto o tem tamanho esperado  $E[Y_c] \leq n/m = \alpha$ . Caso  $c \in H$ , o tamanho da lista é  $1 + Y_c$  e com  $E[Y_c] = (n - 1)/m$  esperadamente

$$1 + (n - 1)/m = 1 + \alpha - 1/m < 1 + \alpha.$$

■

Um exemplo de um conjunto de funções hash universais: Seja  $c = (c_0, \dots, c_r)_m$  uma chave na base  $m$ , escolha  $a = (a_0, \dots, a_r)_m$  randomicamente e define

$$h_a = \sum_{0 \leq i \leq r} c_i a_i \mod m.$$

**Hashing perfeito** Hashing é *perfeito* sem colisões. Isso podemos garantir somente caso conhecemos a chaves a serem inseridos na tabela. Para uma função aleatória de uma família universal de funções hash para uma tabela hash de tamanho  $m$ , o número esperado de colisões é  $E[\sum_{i \neq j} X_{ij}] = \sum_{i \neq j} E[X_{ij}] \leq n^2/m$ . Portanto, caso escolhermos uma tabela de tamanho  $m > n^2$  o número esperado de colisões é menos que um. Em particular, para  $m > 2n^2$  a probabilidade de uma colisão é  $P[\sum_{i \neq j} X_{ij} \geq 0] \leq \sum_{i \neq j} P[X_{ij} = 1] = n^2/m < 1/2$ .

## 2.2. Hashing com endereçamento aberto

Uma abordagem para resolução de colisões, chamada *endereçamento aberto*, é escolher outra posição para armazenar uma chave, caso  $h(c)$  é ocupada. Uma estratégia para conseguir isso é procurar uma posição livre numa permutação de todos índices restantes. Assim garantimos que um insert tem sucesso enquanto ainda existe uma posição livre na tabela. Uma função hash  $h(c, i)$  com dois argumentos, tal que  $h(c, 0), \dots, h(c, m-1)$  é uma permutação de  $[m]$ , representa essa estratégia.

```

1  insert(c, H) :=
2      for i in [m]
3          if H[h(c, i)] = free
4              H[h(c, i)] = c
5          return
6
7  lookup(c, H) :=
8      for i in [m]
9          if H[h(c, i)] = free
10             return false
11         if H[h(c, i)] = c
12             return true
13     return false

```

A função  $h(c, i)$  é *uniforme*, se a probabilidade de uma chave randômica ter associada uma dada permutação é  $1/m!$ . A seguir supomos que  $h$  é uniforme.

### Teorema 2.4

As funções lookup e insert precisam no máximo  $1/(1 - \alpha)$  testes caso a chave não está na tabela.

**Prova.** Seja  $X$  o número de testes até achar uma posição livre. Temos

$$E[X] = \sum_{i \geq 1} i \Pr[X = i] = \sum_{i \geq 1} \sum_{j \geq i} \Pr[X = i] = \sum_{i \geq 1} \Pr[X \geq i].$$

Com  $T_i$  o evento que o teste  $i$  ocorre e a posição  $i$  é ocupada, podemos escrever

$$\Pr[X \geq i] = \Pr[T_1 \cap \dots \cap T_{i-1}] = \Pr[T_1] \Pr[T_2|T_1] \Pr[T_3|T_1, T_2] \dots \Pr[T_{i-1}|T_1, \dots, T_{i-2}].$$

Agora  $\Pr[T_1] = n/m$ , e como  $h$  é uniforme  $\Pr[T_2|T_1] = n-1/(m-1)$  e em geral

$$\Pr[T_k|T_1, \dots, T_{k-1}] = (n-k+1)/(m-k+1) \leq n/m = \alpha.$$

## 2. Tabelas hash

Portanto  $\Pr[X \geq i] \leq \alpha^{i-1}$  e

$$E[X] = \sum_{i \geq 1} \Pr[X \geq i] \leq \sum_{i \geq 1} \alpha^{i-1} = \sum_{i \geq 0} \alpha^i = 1/(1 - \alpha).$$

■

### Lema 2.1

Para  $i < j$ , temos  $H_i - H_j \leq \ln(i) - \ln(j)$ .

**Prova.**

$$H_i - H_j \leq \int_{j+1}^{i+1} \frac{1}{x-1} dx = \ln(i) - \ln(j)$$

■

### Teorema 2.5

Caso  $\alpha < 1$  a função lookup precisa esperadamente  $1/\alpha \ln 1/(1 - \alpha)$  testes caso a chave está na tabela, e cada chave tem a mesma probabilidade de ser procurada.

**Prova.** Seja  $c$  a  $i$ -gésima chave inserida. No momento de inserção temos  $\alpha = (i - 1)/m$  e o número esperado de testes  $T$  até encontrar a posição livre foi  $1/(1 - (i - 1)/m) = m/(m - (i - 1))$ , e portanto o número esperado de testes até encontrar uma chave arbitrária é

$$E[T] = 1/n \sum_{1 \leq i \leq n} m/(m - (i - 1)) = 1/\alpha \sum_{0 \leq i < n} 1/(m - i) = 1/\alpha (H_m - H_{m-n})$$

e com  $H_m - H_{m-n} \leq \ln(m) - \ln(m - n)$  temos

$$E[T] = 1/\alpha (H_m - H_{m-n}) < 1/\alpha (\ln(m) - \ln(m - n)) = 1/\alpha \ln(1/(1 - \alpha)).$$

■

Remover elementos de uma tabela hash com endereçamento aberto é mais difícil, porque a busca para um elemento termina ao encontrar uma posição livre. Para garantir a corretude de lookup, temos que marcar posições como “removidas” e continuar a busca nessas posições. Infelizmente, nesse caso, as garantias da complexidade não mantem-se – após uma série de deleções e inserções toda posição livre será marcada como “removida” tal que delete e lookup precisam  $n$  passos. Portanto o endereçamento aberto é favorável só se temos nenhuma ou poucas deleções.



## Funções hash para endereçamento aberto

- Linear:  $h(c, i) = h(c) + i \bmod m$
- Quadrática:  $h(c, i) = h(c) + c_1 i + c_2 i^2 \bmod m$
- Hashing duplo:  $h(c, i) = h_1(c) + i h_2(c) \bmod m$

Nenhuma das funções é uniforme, mas o hashing duplo mostra um bom desempenho na prática.

## 2.3. Cuco hashing

*Cuco hashing* é outra abordagem que procura posições alternativas na tabela em caso de colisões, com o objetivo de garantir um tempo de acesso constante no pior caso. Para conseguir isso, usamos duas funções hash  $h_1$  e  $h_2$ , e inserimos uma chave em uma das duas posições  $h_1(c)$  ou  $h_2(c)$ . Desta forma a busca e a deleção possuem complexidade constante  $O(1)$ :

```

1 lookup(c, H) :=
2   if H[h1(c)] = c or H[h2(c)] = c
3     return true
4   return false
5
6 delete(c, H) :=
7   if H[h1(c)] = c
8     H[h1(c)] := free
9   if H[h2(c)] = c
10    H[h2(c)] := free

```

Inserir uma chave é simples, caso uma das posições alternativas é livre. No caso contrário, a solução do cuco hashing é comportar-se como um cuco com ovos de outras aves que jogá-los fora do seu “ninho”: “insert” ocupa a posição de uma das duas chaves. A chave “jogada fora” será inserida novamente na tabela. Caso a posição alternativa dessa chave é livre, a inserção termina. Caso contrário, o processo se repete. Esse procedimento termina após uma série de reinserções ou entra num laço infinito. Nesse último caso temos que realocar todas chaves com novas funções hash.

```

1 insert(c, H) :=
2   if H[h1(c)] = c or H[h2(c)] = c
3     return
4   p := h1(c)
5   do n vezes

```

## 2. Tabelas hash

```

6      if H[p] = free
7          H[p] := c
8          return
9      swap(c, H[p])
10     { escolhe a outra posição da chave atual }
11     if p = h1(c)
12         p := h2(c)
13     else
14         p := h1(c)
15     rehash(H)
16     insert(c, H)

```

Uma maneira de visualizar uma tabela hash com cuco hashing, é usar o *grafo cuco*: Caso foram inseridos as chaves  $c_1, \dots, c_n$  na tabela nas posições  $p_1, \dots, p_n$ , o grafo é  $G = (V, A)$ , com  $V = [m]$  é  $(p_i, h_2(c_i)) \in A$  caso  $h_1(c_i) = p_i$  e  $(p_i, h_1(c_i)) \in A$  caso  $h_2(c_i) = p_i$ , i.e., os arcos apontam para a posição alternativa. O grafo cuco é um grafo direcionado e eventualmente possui ciclos. Uma característica do grafo cuco é que uma posições  $p$  é eventualmente analisada na inserção de uma chave  $c$  somente se existe um caminho de  $h_1(c)$  ou  $h_2(c)$  para  $p$ . Para a análise é suficiente considerar o grafo cuco não-direcionado.

### Exemplo 2.1

Para chaves de dois dígitos  $c_1c_2$  seja  $h_1(c) = 3c_1 + c_2 \bmod m$  e  $h_2(c) = 4c_1 + c_2$ . Para  $m = 10$  obtemos para uma sequencia aleatória de chaves

c	31	41	59	26	53	58	97
$h_1(c)$	0	3	4	2	8	3	4
$h_2(c)$	3	7	9	4	3	8	3

e a seguinte sequencia de tabelas hash

0	1	2	3	4	5	6	7	8	9	
										Inicial
31										Inserção 31
31			41							Inserção 41
31			41	59						Inserção 59
31		26	41	59						Inserção 26
31		26	41	59				53		Inserção 53
31		26	58	59			41	53		Inserção 58
31		26	58	97			41	53	59	Inserção 59

◇

### Lema 2.2

Para posições  $i$  e  $j$  e um  $c > 1$  tal que  $m \geq 2cn$ , a probabilidade de existir um caminho mínimo de  $i$  para  $j$  de comprimento  $d \geq 1$  é, no máximo,  $c^{-d}/m$ .

**Prova.** Observe que a probabilidade de um item  $c$  ter posições  $i$  e  $j$  como alternativas é no máximo  $P[h_1(c) = i, h_2(c) = j] + P[h_1(c) = j, h_2(c) = i] = 2/m^2$ . Portanto a probabilidade de pelo menos uma das  $n$  chaves ter essa característica é no máximo  $2n/m^2 = c^{-1}/m$ .

A prova do lema é por indução sobre  $d$ . Para  $d = 1$  a afirmação está correto pela observação acima. Para  $d > 1$  existe um caminho mínimo de comprimento  $d - 1$  de  $i$  para um  $k$ . A probabilidade disso é no máximo  $c^{-(d-1)}/m$  e a probabilidade de existir um elemento com posições alternativas  $k$  e  $j$  no máximo  $c^{-1}/m$ . Portanto, para um  $k$  fixo, a probabilidade existir um caminho de comprimento  $d$  é no máximo  $c^{-d}/m^2$  e considerando todas posições  $k$  possíveis no máximo  $c^{-d}/m$ . ■

Com isso a probabilidade de existir um caminho entre duas chaves  $i$  e  $j$ , é igual a probabilidade de existir um caminho começando em  $h_1(i)$  ou  $h_2(i)$  e terminando em  $h_1(j)$  ou  $h_2(j)$ , que é no máximo  $4 \sum_{i \geq 1} c^{-i}/m \leq 4/m(c - 1) = O(1/m)$ . Logo o número esperado de itens visitados numa inserção é  $4n/m(c - 1) = O(1)$ , caso não é necessário reconstruir a tabela hash.

## 2.4. Filtros de Bloom

Um filtro de Bloom armazena um conjunto de  $n$  chaves, com as seguintes restrições:

- Não é mais possível remover elementos.
- É possível que o teste de pertinência tem sucesso, sem o elemento fazer parte do conjunto (“false positive”).

Um filtro de Bloom consiste em  $m$  bits  $B_i$ ,  $1 \leq i \leq m$ , e usa  $k$  funções hash  $h_1, \dots, h_k$ .

```

1 insert(c, B) :=
2   for i in 1...k
3      $b_{h_i(c)} := 1$ 
4   end for
5
6 lookup(c, B) :=
7   for i in 1...k
8     if  $b_{h_i(c)} = 0$ 
9       return false
10  return true
```

## 2. Tabelas hash

Tabela 2.1.: Complexidade das operações em tabelas hash. Complexidades em negrito são amortizados.

	insert	lookup	delete
Listas encadeadas	$\Theta(1)$	$\Theta(1 + \alpha)$	$\Theta(1 + \alpha)$
Endereçamento aberto	$O(1/(1 - \alpha))$	$O(1/(1 - \alpha))$	-
(com/sem sucesso)	$O(1/\alpha \ln 1/(1 - \alpha))$	$O(1/\alpha \ln 1/(1 - \alpha))$	-
Cuco	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Após de inserir  $n$  chaves, um dado bit é ainda 0 com probabilidade

$$p' = \left(1 - \frac{1}{m}\right)^{kn} = \left(1 - \frac{kn/m}{kn}\right)^{kn} \approx e^{-kn/m}$$

que é igual ao valor esperado da fração de bits não setados<sup>1</sup>. Sendo  $\rho$  a fração de bits não setados realmente, a probabilidade de erradamente classificar um elemento como membro do conjunto é

$$(1 - \rho)^k \approx (1 - p')^k \approx \left(1 - e^{-kn/m}\right)^k$$

porque  $\rho$  é com alta probabilidade perto do seu valor esperado [7]. Broder e Mitzenmacher [7] também mostram que o número ótimo  $k$  de funções hash para dados valores de  $n, m$  é  $m/n \ln 2$  e com isso temos um erro de classificação  $\approx (1/2)^k$ .

Aplicações:

1. Hifenação: Manter uma tabela de palavras com hifenação excepcional (que não pode ser determinado pelas regras).
2. Comunicação efetiva de conjuntos, p.ex. seleção em bancos de dados distribuídas. Para calcular um join de dois bancos de dados  $A, B$ , primeiro  $A$  filtra os elementos, mando um filtro de Bloom  $S_A$  para  $B$  e depois  $B$  executa o join baseado em  $S_A$ . Para eliminação de eventuais elementos classificados erradamente,  $B$  manda os resultados para  $A$  e  $A$  filtra os elementos errados.

<sup>1</sup>Lembrando que  $e^x \geq (1 + x/n)^n$  para  $n > 0$ .

### 3. Algoritmos de aproximação

Para vários problemas não conhecemos um algoritmo eficiente. Para problemas NP-completos, em particular, uma solução eficiente é pouco provável. Um *algoritmo de aproximação* calcula uma solução aproximada para um problema de otimização. Diferente de uma heurística, o algoritmo *garante* a qualidade da aproximação no pior caso. Dado um problema e um algoritmo de aproximação  $A$ , escrevemos  $A(x) = y$  para a solução aproximada da instância  $x$ ,  $\varphi(x, y)$  para o valor dessa solução,  $y^*$  para a solução ótima e  $OPT(x) = \varphi(x, y^*)$  para o valor da solução ótima.

#### 3.1. Problemas, classes e reduções

##### Definição 3.1

Um *problema de otimização*  $\Pi = (\mathcal{P}, \varphi, \text{opt})$  é uma relação binária  $\mathcal{P} \subseteq I \times S$  com instâncias  $x \in I$  e soluções  $y \in S$ , junto com

- uma função de otimização (função de objetivo)  $\varphi : \mathcal{P} \rightarrow \mathbb{N}$  (ou  $\mathbb{Q}$ ).
- um objetivo: Encontrar mínimo ou máximo

$$OPT(x) = \text{opt}\{\varphi(x, y) \mid (x, y) \in \mathcal{P}\}$$

junto com uma solução  $y^*$  tal que  $f(x, y^*) = OPT(x)$ .

O par  $(x, y) \in \mathcal{P}$  caso  $y$  é uma solução para  $x$ .

Uma instância  $x$  de um problema de otimização possui soluções  $S(x) = \{y \mid (x, y) \in \mathcal{P}\}$ .

##### Convenção 3.1

Escrevemos um problema de otimização na forma

NOME

**Instância**  $x$

**Solução**  $y$

**Objetivo** Minimiza ou maximiza  $\varphi(x, y)$ .

### 3. Algoritmos de aproximação

Com um dado problema de otimização correspondem três problemas:

- Construção: Dado  $x$ , encontra a solução ótima  $y^*$  e seu valor  $\text{OPT}(x)$ .
- Avaliação: Dado  $x$ , encontra valor ótimo  $\text{OPT}(x)$ .
- Decisão: Dado  $x$  e  $k$ , decide se  $\text{OPT}(x) \geq k$  (maximização) ou  $\text{OPT}(x) \leq k$  (minimização).

#### Definição 3.2

Uma relação binária  $R$  é *polinomialmente limitada* se

$$\exists p \in \text{poly} : \forall (x, y) \in R : |y| \leq p(|x|).$$

#### Definição 3.3 (Classes de complexidade)

A classe PO consiste dos problemas de otimização tal que existe um algoritmo polinomial  $A$  com  $\varphi(x, A(x)) = \text{OPT}(x)$  para  $x \in I$ .

A classe NPO consiste dos problemas de otimização tal que

- (i) As instâncias  $x \in I$  são reconhecíveis em tempo polinomial.
- (ii) A relação  $\mathcal{P}$  é polinomialmente limitada.
- (iii) Para  $y$  arbitrário, polinomialmente limitado:  $(x, y) \in \mathcal{P}$  é decidível em tempo polinomial.
- (iv)  $\varphi$  é computável em tempo polinomial.

#### Definição 3.4

Uma *redução preservando a aproximação* entre dois problemas de minimização  $\Pi_1$  e  $\Pi_2$  consiste num par de funções  $f$  e  $g$  (computáveis em tempo polinomial) tal que para instância  $x_1$  de  $\Pi_1$ ,  $x_2 := f(x_1)$  é instância de  $\Pi_2$  com

$$\text{OPT}_{\Pi_2}(x_2) \leq \text{OPT}_{\Pi_1}(x_1) \quad (3.1)$$

e para uma solução  $y_2$  de  $\Pi_2$  temos uma solução  $y_1 := g(x_1, y_2)$  de  $\Pi_1$  com

$$\varphi_{\Pi_1}(x_1, y_1) \leq \varphi_{\Pi_2}(x_2, y_2) \quad (3.2)$$

Uma redução preservando a aproximação fornece uma  $\alpha$ -aproximação para  $\Pi_1$  dada uma  $\alpha$ -aproximação para  $\Pi_2$ , porque

$$\varphi_{\Pi_1}(x_1, y_1) \leq \varphi_{\Pi_2}(x_2, y_2) \leq \alpha \text{OPT}_{\Pi_2}(x_2) \leq \alpha \text{OPT}_{\Pi_1}(x_1).$$

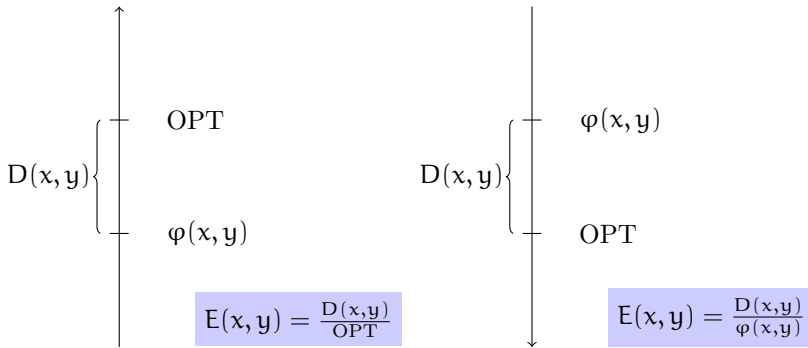
Observe que essa definição é vale somente para problemas de minimização. A definição no caso de maximização é semelhante.

## 3.2. Medidas de qualidade

Uma *aproximação absoluta* garante que  $D(x, y) = |\text{OPT}(x) - \varphi(x, y)| \leq D$  para uma constante  $D$  e todo  $x$ , enquanto uma *aproximação relativa* garante que o *erro relativo*  $E(x, y) = D(x, y) / \max\{\text{OPT}(x), \varphi(x, y)\} \leq \epsilon \leq 1$  todos  $x$ . Um algoritmo que consegue um aproximação com constante  $\epsilon$  também se chama  $\epsilon$ -aproximativo. Tais algoritmos fornecem uma solução que difere no máximo um fator constante da solução ótima. A classe de problemas de otimização que permitem uma  $\epsilon$ -aproximação em tempo polinomial para uma constante  $\epsilon$  se chama APX.

Uma definição alternativa é a *taxa de aproximação*  $R(x, y) = 1/(1 - E(x, y)) \geq 1$ . Um algoritmo com taxa de aproximação  $r$  se chama  $r$ -aproximativo. (Não tem perigo de confusão com o erro relativo, porque  $r \geq 1$ .)

### Aproximação relativa



#### Exemplo 3.1

Coloração de grafos planares e a problema de determinar a árvore geradora e a árvore Steiner de grau mínimo [20] permitem uma aproximação absoluta, mas não o problema da mochila.

Os problemas da mochila e do caixeiro viajante métrico permitem uma aproximação absoluta constante, mas não o problema do caixeiro viajante.  $\diamond$

## 3.3. Técnicas de aproximação

### 3.3.1. Algoritmos gulosos

#### Cobertura por vértices

**Algoritmo 3.1 (Cobertura por vértices)**

**Entrada** Grafo não-direcionado  $G = (V, E)$ .

**Saída** Cobertura por vértices  $C \subseteq V$ .

```

1  VC-GV(G) :=
2    (C, G) := Reduz(G)
3    if  $V = \emptyset$  then
4      return C
5    else
6      escolhe  $v \in V : \deg(v) = \Delta(G)$  { grau máximo }
7      return  $C \cup \{v\} \cup \text{VC-GV}(G - v)$ 
8    end if

```

**Proposição 3.1**

O algoritmo VC-GV é uma  $O(\log |V|)$ -aproximação.

**Prova.** Seja  $G_i$  o grafo depois da iteração  $i$  e  $C^*$  uma cobertura ótima, i.e.,  $|C^*| = \text{OPT}(G)$ .

A cobertura ótima  $C^*$  é uma cobertura para  $G_i$  também. Logo, a soma dos graus dos vértices em  $C^*$  (contando somente arestas em  $G_i$ !) ultrapassa o número de arestas em  $G_i$

$$\sum_{v \in C^*} \delta_{G_i}(v) \geq \|G_i\|$$

e o grau médio dos vértices em  $G_i$  satisfaz

$$\bar{\delta}_{G_i}(G_i) = \frac{\sum_{v \in C^*} \delta_{G_i}(v)}{|C^*|} \geq \frac{\|G_i\|}{|C^*|} = \frac{\|G_i\|}{\text{OPT}(G)}.$$

Como o grau máximo é maior que o grau médio temos também

$$\Delta(G_i) \geq \frac{\|G_i\|}{\text{OPT}(G)}.$$

Com isso podemos estimar

$$\begin{aligned}
 \sum_{0 \leq i < \text{OPT}} \Delta(G_i) &\geq \sum_{0 \leq i < \text{OPT}} \frac{\|G_i\|}{\text{OPT}(G)} \geq \sum_{0 \leq i < \text{OPT}} \frac{\|G_{\text{OPT}}\|}{\text{OPT}(G)} \\
 &= \|G_{\text{OPT}}\| = \|G\| - \sum_{0 \leq i < \text{OPT}} \Delta(G_i)
 \end{aligned}$$



ou

$$\sum_{0 \leq i < \text{OPT}} \Delta(G_i) \geq \|G\|/2,$$

i.e. a metade das arestas foi removido em OPT iterações. Essa estimativa continua a ser válido, logo após

$$\text{OPT} \lceil \lg \|G\| \rceil \leq \text{OPT} \lceil 2 \log |G| \rceil = O(\text{OPT} \log |G|)$$

iterações não tem mais arestas. Como em cada iteração foi escolhido um vértice, a taxa de aproximação é  $\log |G|$ . ■

### Algoritmo 3.2 (Cobertura por vértices)

**Entrada** Grafo não-direcionado  $G = (V, E)$ .

**Saída** Um cobertura por vértices  $C \subseteq V$ .

```

1  VC-GE(G) :=
2    (C, G) := Reduz(G)
3    if E = ∅ then
4      return C
5    else
6      escolhe e = {u, v} ∈ E
7      return C ∪ {u, v} ∪ VC-GE(G - {u, v})
8  end if
```

### Proposição 3.2

Algoritmo VC-GE é uma 2-aproximação para VC.

**Prova.** Cada cobertura contém pelo menos um dos dois vértices escolhidos, logo

$$|C| \geq \phi_{\text{VC-GE}}(G)/2 \Rightarrow 2\text{OPT}(G) \geq \phi_{\text{VC-GE}}(G).$$

■

### Algoritmo 3.3 (Cobertura por vértices)

**Entrada** Grafo não-direcionado  $G = (V, E)$ .

**Saída** Cobertura por vértices  $C \subseteq V$ .

```

1  VC-B(G) :=
2    (C, G) := Reduz(G)
3    if V = ∅ then
4      return C
```

### 3. Algoritmos de aproximação

```
5   else
6       escolha  $v \in V : \deg(v) = \Delta(G)$  { grau máximo }
7        $C_1 := C \cup \{v\} \cup VC-B(G-v)$ 
8        $C_2 := C \cup N(v) \cup VC-B(G-v-N(v))$ 
9       if  $|C_1| < |C_2|$  then
10          return  $C_1$ 
11       else
12          return  $C_2$ 
13       end if
14   end if
```

#### Problema da mochila

##### KNAPSACK

**Instância** Um número  $n$  de itens com valores  $v_i \in \mathbb{N}$  e tamanhos  $t_i \in \mathbb{N}$ , para  $i \in [n]$ , um limite  $M$ , tal que  $t_i \leq M$  (todo item cabe na mochila).

**Solução** Uma seleção  $S \subseteq [n]$  tal que  $\sum_{i \in S} t_i \leq M$ .

**Objetivo** Maximizar o valor total  $\sum_{i \in S} v_i$ .

Observação: O problema da mochila é NP-completo.

#### Como aproximar?

- Idéia: Ordene por  $v_i/t_i$  (“valor médio”) em ordem decrescente e enche o mochila o mais possível nessa ordem.

#### Abordagem

```
1 K-G( $v_i, t_i$ ) :=
2   ordene os itens tal que  $v_i/t_i \geq v_j/t_j, \forall i < j$ .
3   for  $i \in X$  do
4       if  $t_i < M$  then
5            $S := S \cup \{i\}$ 
6            $M := M - t_i$ 
7       end if
8   end for
9   return  $S$ 
```

**Aproximação boa?**

- Considere

$$\begin{aligned} v_1 &= 1, \dots, v_{n-1} = 1, v_n = M - 1 \\ t_1 &= 1, \dots, t_{n-1} = 1, t_n = M = kn \quad k \in \mathbb{N} \text{ arbitrário} \end{aligned}$$

- Então:

$$v_1/t_1 = 1, \dots, v_{n-1}/t_{n-1} = 1, v_n/t_n = (M - 1)/M < 1$$

- K-G acha uma solução com valor  $\varphi(x) = n - 1$ , mas o ótimo é  $\text{OPT}(x) = M - 1$ .
- Taxa de aproximação:

$$\text{OPT}(x)/\varphi(x) = \frac{M - 1}{n - 1} = \frac{kn - 1}{n - 1} \geq \frac{kn - k}{n - 1} = k$$

- K-G não possui taxa de aproximação fixa!
- Problema: Não escolhemos o item com o maior valor.

**Tentativa 2: Modificação**

```

1  K-G' ( vi , ti ) :=
2    S1 := K-G( vi , ti )
3    v1 := ∑i ∈ S1 vi
4    S2 := {argmaxi vi}
5    v2 := ∑i ∈ S2 vi
6    if v1 > v2 then
7      return S1
8    else
9      return S2
10   end if

```

**Aproximação boa?**

- O algoritmo melhorou?
- Surpresa

**Proposição 3.3**

K-G' é uma 2-aproximação, i.e.  $\text{OPT}(x) < 2\varphi_{\text{K-G}'}(x)$ .

### 3. Algoritmos de aproximação

**Prova.** Seja  $j$  o primeiro item que K-G não coloca na mochila. Nesse ponto temos valor e tamanho

$$\bar{v}_j = \sum_{1 \leq i < j} v_i \leq \varphi_{K-G}(x) \quad (3.3)$$

$$\bar{t}_j = \sum_{1 \leq i < j} t_i \leq M \quad (3.4)$$

Afirmção:  $\text{OPT}(x) < \bar{v}_j + v_j$ . Nesse caso

(a) Seja  $v_j \leq \bar{v}_j$ .

$$\text{OPT}(x) < \bar{v}_j + v_j \leq 2\bar{v}_j \leq 2\varphi_{K-G}(x) \leq 2\varphi_{K-G'}$$

(b) Seja  $v_j > \bar{v}_j$

$$\text{OPT}(x) < \bar{v}_j + v_j < 2v_j \leq 2v_{\max} \leq 2\varphi_{K-G'}$$

Prova da afirmação: No momento em que item  $j$  não cabe, temos espaço  $M - \bar{t}_j < t_j$  sobrando. Como os itens são ordenados em ordem de densidade decrescente, obtemos um limite superior para a solução ótima preenchendo esse espaço com a densidade  $v_j/t_j$ :

$$\text{OPT}(x) \leq \bar{v}_j + (M - \bar{t}_j) \frac{v_j}{t_j} < \bar{v}_j + v_j.$$

■

#### 3.3.2. Aproximações com randomização

##### Randomização

- Idéia: Permite escolhas randômicas (“joga uma moeda”)
- Objetivo: Algoritmos que decidem correta com probabilidade alta.
- Objetivo: Aproximações com *valor esperado* garantido.
- Minimização:  $E[\varphi_A(x)] \leq 2\text{OPT}(x)$
- Maximização:  $2E[\varphi_A(x)] \geq \text{OPT}(x)$

##### Randomização: Exemplo

**SATISFATIBILIDADE MÁXIMA, MAXIMUM SAT**

**Instância** Uma fórmula  $\varphi \in \mathcal{L}(V)$  sobre variáveis  $V = \{v_1, \dots, v_m\}$ ,  $\varphi = C_1 \wedge C_2 \wedge \dots \wedge C_n$  em FNC.

**Solução** Uma atribuição de valores de verdade  $\alpha : V \rightarrow \mathbb{B}$ .

**Objetivo** Maximiza o número de cláusulas satisfeitas

$$|\{C_i \mid \llbracket C_i \rrbracket_\alpha = 1\}|.$$

**Nossa solução**

```

1 SAT-R( $\varphi$ ) :=
2   seja  $\varphi = \varphi(v_1, \dots, v_k)$ 
3   for all  $i \in [1, k]$  do
4     escolha  $v_i = 1$  com probabilidade  $1/2$ 
5   end for
```

**Observação 3.1**

A quantidade  $\llbracket C \rrbracket_\alpha$  é o valor da cláusula  $C$  na atribuição  $\alpha$ . ◇

**Aproximação?**

- Surpresa: Algoritmo é 2-aproximação.

**Prova.** O valor esperado de uma cláusula  $C$  com  $l$  variáveis é  $E[\llbracket C \rrbracket] = P[\llbracket C \rrbracket = 1] = 1 - 2^{-l} \geq 1/2$ . Logo o valor esperado do número total  $T = \sum_{i \in [n]} \llbracket C_i \rrbracket$  de cláusulas satisfeitas é

$$E[T] = E\left[\sum_{i \in [n]} \llbracket C_i \rrbracket\right] = \sum_{i \in [n]} E[\llbracket C_i \rrbracket] \geq n/2 \geq \text{OPT}/2$$

pela linearidade do valor esperado. ■

**Outro exemplo**

Cobertura por vértices guloso e randomizado.

```

1 VC-RG( $G$ ) :=
2   seja  $\bar{w} := \sum_{v \in V} \deg(v)$ 
3    $C := \emptyset$ 
4   while  $E \neq \emptyset$  do
```

### 3. Algoritmos de aproximação

```
5     escolha  $v \in V$  com probabilidade  $\deg(v)/\bar{w}$ 
6      $C := C \cup \{v\}$ 
7      $G := G - v$ 
8 end while
9 return  $C \cup V$ 
```

Resultado:  $E[\phi_{VC-RG}(x)] \leq 2OPT(x)$ .

## 3.4. Esquemas de aproximação

### Novas considerações

- Frequentemente uma  $r$ -aproximação não é suficiente.  $r = 2$ : 100% de erro!
- Existem aproximações melhores? p.ex. para SAT? problema do mochila?
- Desejável: Esquema de aproximação em tempo polinomial (EATP); polynomial time approximation scheme (PTAS)
  - Para cada entrada e taxa de aproximação  $r$ :
  - Retorne  $r$ -aproximação em tempo polinomial.

### Um exemplo: Mochila máxima (Knapsack)

- Problema da mochila (veja página 94):
- Algoritmo MM-PD com programação dinâmica (pág. 141): tempo  $O(n \sum_i v_i)$ .
- Desvantagem: Pseudo-polinomial.

Denotamos uma instância do problema da mochila com  $I = (\{v_i\}, \{t_i\})$ .

```
1 MM-PTAS( $I, r$ ) :=
2    $v_{\max} := \max_i \{v_i\}$ 
3    $t := \lfloor \log \frac{r-1}{r} \frac{v_{\max}}{n} \rfloor$ 
4    $v'_i := \lfloor v_i / 2^t \rfloor$  para  $i = 1, \dots, n$ 
5   Define a nova instância  $I' = (\{v'_i\}, \{t_i\})$ 
6   return MM PD( $I'$ )
```

### Teorema 3.1

MM-PTAS é uma  $r$ -aproximação em tempo  $O(rn^3/(r-1))$ .

**Prova.** A complexidade da preparação nas linhas 1–3 é  $O(n)$ . A chamada para MM-PD custa

$$\begin{aligned} O\left(n \sum_i v'_i\right) &= O\left(n \sum_i \frac{v_i}{((r-1)/r)(v_{\max}/n)}\right) \\ &= O\left(\frac{r}{r-1} n^2 \sum_i v_i/v_{\max}\right) = O\left(\frac{r}{r-1} n^3\right). \end{aligned}$$

Seja  $S = \text{MM-PTAS}(I)$  a solução obtida pelo algoritmo e  $S^*$  uma solução ótima.

$$\begin{aligned} \varphi_{\text{MM-PTAS}}(I, S) &= \sum_{i \in S} v_i \geq \sum_{i \in S} 2^t \lfloor v_i/2^t \rfloor && \text{definição de } \lfloor \cdot \rfloor \\ &\geq \sum_{i \in S^*} 2^t \lfloor v_i/2^t \rfloor && \text{otimalidade de MM-PD sobre } v'_i \\ &\geq \sum_{i \in S^*} v_i - 2^t && (\text{A.2}) \\ &= \left( \sum_{i \in S^*} v_i \right) - 2^t |S^*| \\ &\geq \text{OPT}(I) - 2^t n \end{aligned}$$

Portanto

$$\begin{aligned} \text{OPT}(I) &\leq \varphi_{\text{MM-PTAS}}(I, S) + 2^t n \leq \varphi_{\text{MM-PTAS}}(I, S) + \frac{\text{OPT}(x)}{v_{\max}} 2^t n \\ &\iff \text{OPT}(I) \left(1 - \frac{2^t n}{v_{\max}}\right) \leq \varphi_{\text{MM-PTAS}}(I, S) \end{aligned}$$

e com  $2^t n/v_{\max} \leq (r-1)/r$

$$\iff \text{OPT}(I) \leq r \varphi_{\text{MM-PTAS}}(I, S).$$

Um EATP frequentemente não é suficiente para resolver um problema adequadamente. Por exemplo temos um EATP para

- o problema do caixeiro viajante euclidiano com complexidade  $O(n^{3000/\epsilon})$  (Arora, 1996);
- o problema do mochila múltiplo com complexidade  $O(n^{12(\log 1/\epsilon)/e^8})$  (Chekuri, Kanna, 2000);

### 3. Algoritmos de aproximação

- o problema do conjunto independente máximo em grafos com complexidade  $O(n^{(4/\pi)(1/\epsilon^2+1)^2(1/\epsilon^2+2)^2})$  (Erlebach, 2001).

Para obter uma aproximação com 20% de erro, i.e.  $\epsilon = 0.2$  obtemos algoritmos com complexidade  $O(n^{15000})$ ,  $O(n^{375000})$  e  $O(n^{523804})$ , respectivamente!

### 3.5. Aproximando o problema da árvore de Steiner mínima

Seja  $G = (V, A)$  um grafo completo, não-direcionado com custos  $c_a \geq 0$  nos arcos. O problema da árvore Steiner mínima (ASM) consiste em achar o subgrafo conexo mínimo que inclui um dado conjunto de *vértices necessários* ou *terminais*  $R \subseteq V$ . Esse subgrafo sempre é uma árvore (ex. 3.1). O conjunto  $V \setminus R$  forma os *vértices Steiner*. Para um conjunto de arcos  $A$ , define o custo  $c(A) = \sum_{a \in A} c_a$ .

#### Observação 3.2

ASM é NP-completo. Para um conjunto fixo de vértices Steiner  $V' \subseteq V \setminus R$ , a melhor solução é a árvore geradora mínima sobre  $R \cup V'$ . Portanto a dificuldade é a seleção dos vértices Steiner da solução ótima.  $\diamond$

#### Definição 3.5

Os custos são *métricos* se eles satisfazem a desigualdade triangular, i.e.

$$c_{ij} \leq c_{ik} + c_{kj}$$

para qualquer tripla de vértices  $i, j, k$ .

#### Teorema 3.2

Existe uma redução preservando a aproximação de ASM para a versão métrica do problema.

**Prova.** O “fecho métrico” de  $G = (V, A)$  é um grafo  $G'$  completo sobre vértices e com custos  $c'_{ij} := d_{ij}$ , sendo  $d_{ij}$  o comprimento do menor caminho entre  $i$  e  $j$  em  $G$ . Evidentemente  $c'_{ij} \leq c_{ij}$  e portanto (3.1) é satisfeita. Para ver que (3.2) é satisfeita, seja  $T'$  uma solução de ASM em  $G'$ . Define  $T$  como união de todos caminhos definidos pelos arcos em  $T'$ , menos um conjunto de arcos para remover eventuais ciclos. O custo de  $T$  é no máximo  $c(T')$  porque o custo de todo caminho é no máximo o custo da aresta correspondente em  $T'$ .  $\blacksquare$

Consequência: Para o problema do ASM é suficiente considerar o caso métrico.

#### Teorema 3.3

O AGM sobre  $R$  é uma 2-aproximação para o problema do ASM.



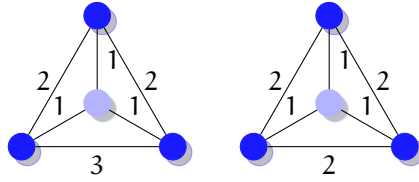


Figura 3.1.: Grafo com fecho métrico.

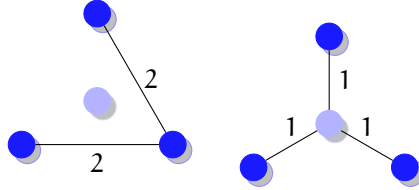


Figura 3.2.: AGM sobre  $R$  e melhor solução. ●: vértice em  $R$ , ●: vértice Steiner.

**Prova.** Considere a solução ótima  $S^*$  de ASM. Duplica todas arestas<sup>1</sup> tal que todo vértice possui grau par. Acha um caminho Euleriano nesse grafo. Remove vértices duplicados nesse caminho. O custo do caminho  $C$  obtido dessa forma não é mais que o dobro do custo original: o grafo com todas arestas custa  $2c(S^*)$  e a remoção de vértices duplicados não aumenta esse custo, pela metricidade. Como esse caminho é uma árvore geradora, temos  $c(A) \leq c(C) \leq 2c(S^*)$  para AGM  $A$ . ■

### 3.6. Aproximando o PCV

#### Teorema 3.4

Para função polinomial  $\alpha(n)$  o PCV não possui  $\alpha(n)$ -aproximação em tempo polinomial, caso  $P \neq NP$ .

**Prova.** Via redução de HC para PCV. Para uma instância  $G = (V, A)$  de HC define um grafo completo  $G'$  com

$$c_a = \begin{cases} 1 & a \in A \\ \alpha(n)n & \text{caso contrário} \end{cases}$$

<sup>1</sup>Isso transforma  $G$  num multigrafo.

### 3. Algoritmos de aproximação

Se  $G$  possui um ciclo Hamiltoniano, então o custo da menor rota é  $n$ . Caso contrário qualquer rota usa ao menos uma aresta de custo  $\alpha(n)n$  e portanto o custo total é  $\geq \alpha(n)n$ . Portanto, dado uma  $\alpha(n)$ -aproximação de PCV podemos decidir HC em tempo polinomial. ■

**Caso métrico** No caso métrico podemos obter uma aproximação melhor. Determina uma rota como segue:

1. Determina uma AGM  $A$  de  $G$ .
2. Duplica todas arestas de  $A$ .
3. Acha um caminho Euleriano nesse grafo.
4. Remove vértices duplicados.

#### Teorema 3.5

O algoritmo acima define uma 2-aproximação.

**Prova.** A melhor solução do PCV menos uma aresta é uma árvore geradora de  $G$ . Portanto  $c(A) \leq \text{OPT}$ . A solução  $S$  obtida pelo algoritmo acima satisfaz  $c(S) \leq 2c(A)$  e portanto  $c(S) \leq 2\text{OPT}$ , pelo mesmo argumento da prova do teorema 3.3. ■

O fator 2 dessa aproximação é resultado do passo 2 que duplica todas arestas para garantir a existência de um caminho Euleriano. Isso pode ser garantido mais barato: A AGM  $A$  possui um número par de vértices com grau ímpar (ver exercício 3.2), e portanto podemos calcular um emparelhamento perfeito mínimo  $E$  entre esse vértices. O grafo com arestas  $A \cup E$  possui somente vértices com grau par e portanto podemos aplicar os restantes passos nesse grafo.

#### Teorema 3.6

A algoritmo usando um emparelhamento perfeito mínimo no passo 2 é uma 3/2-aproximação.

**Prova.** O valor do emparelhamento  $E$  não é mais que  $\text{OPT}/2$ : remove vértices não emparelhados em  $E$  da solução ótima do PCV. O ciclo obtido dessa forma é a união dois emparelhamentos perfeitos  $E_1$  e  $E_2$  formados pelas arestas pares ou ímpares no ciclo. Com  $E_1$  o emparelhamento de menor custo, temos

$$c(E) \leq c(E_1) \leq (c(E_1) + c(E_2))/2 = \text{OPT}/2$$

e portanto

$$c(S) = c(A) + c(E) \leq \text{OPT} + \text{OPT}/2 = 3/2\text{OPT}. \quad \blacksquare$$

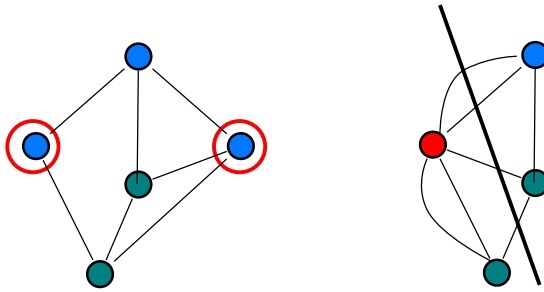


Figura 3.3.: Identificação de dois terminais e um corte no grafo reduzido. Vértices em verde, terminais em azul. A grafo reduzido possui múltiplas arestas entre vértices.

### 3.7. Aproximando problemas de cortes

Seja  $G = (V, A, c)$  um grafo conectado com pesos  $c$  nas arestas. Lembramos que um corte  $C$  é um conjunto de arestas que separa o grafo em duas partes  $S \dot{\cup} V \setminus S$ . Dado dois vértices  $s, t \in V$ , o problema de achar um corte mínimo que separa  $s$  e  $t$  pode ser resolvido via fluxo máximo em tempo polinomial. Generalizações desse problema são:

- Corte múltiplo mínimo (CMM): Dado terminais  $s_1, \dots, s_k$  determine o menor corte  $C$  que separa todos.
- $k$ -corte mínimo ( $k$ -CM): Mesmo problema, sem terminais definidos. (Observe que todos  $k$  componentes devem ser não vazios).

#### Fato 3.1

CMM é NP-difícil para qualquer  $k \geq 3$ .  $k$ -CM possui uma solução polinomial em tempo  $O(n^{k^2})$  para qualquer  $k$ , mas é NP-difícil, caso  $k$  faz parte da entrada.

**Solução de CMM** Chamamos um corte que separa um vértice dos outros um *corte isolante*. Idéia: A união de cortes isolantes para todo  $s_i$  é um corte múltiplo. Para calcular o corte isolante para um dado terminal  $s_i$ , identificamos os restantes terminais em um único vértice  $S$  e calculamos um corte mínimo entre  $s_i$  e  $S$ . (Na identificação de vértices temos que remover self-loops, e somar os pesos de múltiplas arestas.)

Isso leva ao algoritmo

**Algoritmo 3.4 (CI)**

**Entrada** Grafo  $G = (V, A, c)$  e terminais  $s_1, \dots, s_k$ .

**Saída** Um corte múltiplo que separa os  $s_i$ .

- 1 Para cada  $i \in [1, k]$ : Calcula o corte isolante  $C_i$  de  $s_i$ .
- 2 Remove o maior desses cortes e retorne a união dos restantes.

**Teorema 3.7**

Algoritmo 3.4 é uma  $2 - 2/k$ -aproximação.

**Prova.** Considere o corte mínimo  $C^*$ . De acordo com a Fig. 3.4 ele pode ser representado pela união de  $k$  cortes que separam os  $k$  componentes individualmente:

$$C^* = \bigcup_{i \in [k]} C_i^*.$$

Cada aresta de  $C^*$  faz parte das cortes das duas componentes adjacentes, e portanto

$$\sum_{i \in [k]} w(C_i^*) = 2w(C^*)$$

e ainda  $w(C_i) \leq w(C_i^*)$  para os cortes  $C_i$  do algoritmo 3.4, porque usamos o corte isolante mínimo de cada componente. Logo, para o corte  $C$  retornado pelo algoritmo temos

$$w(C) \leq (1 - 1/k) \sum_{i \in [k]} w(C_i) \leq (1 - 1/k) \sum_{i \in [k]} w(C_i^*) \leq 2(1 - 1/k)w(C^*).$$

■

A análise do algoritmo é ótimo, como o exemplo da Fig. 3.5 mostra. O menor corte que separa  $s_i$  tem peso  $2 - \epsilon$ , portanto o algoritmo retorne um corte de peso  $(2 - \epsilon)k - (2 - \epsilon) = (k - 1)(2 - \epsilon)$ , enquanto o menor corte que separa todos terminais é o ciclo interno de peso  $k$ .

**Solução de  $k$ -CM** Problema: Como saber a onde cortar?

**Fato 3.2**

Existem somente  $n-1$  cortes diferentes num grafo. Eles podem ser organizados numa árvore de *Gomory-Hu* (AGH)  $T = (V, T)$ . Cada aresta dessa árvore define um corte associado em  $G$  pelos dois componentes após a sua remoção.

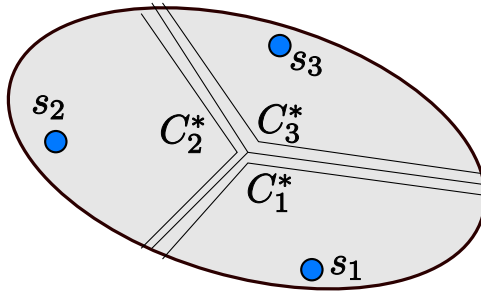


Figura 3.4.: Corte múltiplo e decomposição em cortes isolantes.

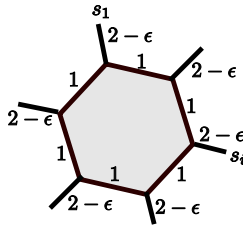


Figura 3.5.: Exemplo de um grafo em que o algoritmo 3.4 retorne uma  $2 - 2/k$ -aproximação.

### 3. Algoritmos de aproximação

1. Para cada  $u, v \in V$  o menor corte  $u-v$  em  $G$  é igual a o menor corte  $u-v$  em  $T$  (i.e. a aresta de menor peso no caminho único entre  $u$  e  $v$  em  $T$ ).
2. Para cada aresta  $a \in T$ ,  $w'(a)$  é igual a valor do corte associado.

Por conseqüência, a AGH codifica o valor de todos cortes em  $G$ .

Ele pode ser calculado determinando  $n - 1$  cortes  $s-t$  mínimos:

1. Define um grafo com um único vértice que representa todos vértices do grafo original. Chama um vértice que representa mais que um vértice do grafo original *gordo*.
2. Enquanto existem vértices gordos:
  - a) Escolhe um vértice gordo e dois vértices do grafo original que ele representa.
  - b) Calcula um corte mínimo entre esses vértices.
  - c) Separa o vértice gordo de acordo com o corte mínimo encontrado.

Observação: A união dos cortes definidos por  $k - 1$  arestas na AGH separa  $G$  em pelo menos  $k$  componentes. Isso leva ao seguinte algoritmo.

#### **Algoritmo 3.5 (KCM)**

**Entrada** Grafo  $G = (V, A, c)$ .

**Saída** Um  $k$ -corte.

- 1 Calcula uma AGH  $T$  em  $G$ .
- 2 Forma a união dos  $k - 1$  cortes mais leves definidos por  $k - 1$  arestas em  $T$ .

#### **Teorema 3.8**

Algoritmo 3.5 é uma  $2 - 2/k$ -aproximação.

**Prova.** Seja  $C^* = \bigcup_{i \in [k]} C_i^*$  um corte mínimo, decomposto igual à prova anterior. O nosso objetivo é demonstrar que existem  $k - 1$  cortes definidos por uma aresta em  $T$  que são mais leves que os  $C_i^*$ .

Removendo  $C^*$  de  $G$  gera componentes  $V_1, \dots, V_k$ : Define um grafo sobre esses componentes identificando vértices de uma componente com arcos da AGH  $T$  entre os componentes, e eventualmente removendo arcos até obter uma nova árvore  $T'$ . Seja  $C_k^*$  o corte de maior peso, e define  $V_k$  como raiz da árvore.

Desta forma, cada componente  $V_1, \dots, V_{k-1}$  possui uma aresta associada na direção da raiz. Para cada dessas arestas  $(u, v)$  temos

$$w(C_i^*) \geq w'(u, v)$$

porque  $C_i^*$  isola o componente  $V_i$  do resto do grafo (particularmente separa  $u$  e  $v$ ), e  $w'(u, v)$  é o peso do menor corte que separa  $u$  e  $v$ . Logo

$$w(C) \leq \sum_{a \in T'} w'(a) \leq \sum_{1 \leq i < k} w(C_i^*) \leq (1-1/k) \sum_{i \in [k]} w(C_i^*) = 2(1-1/k)w(C^*).$$

■

### 3.8. Aproximando empacotamento unidimensional

Dado  $n$  itens com tamanhos  $s_i \in \mathbb{Z}_+$ ,  $i \in [n]$  e contêineres de capacidade  $S \in \mathbb{Z}_+$  o problema do *empacotamento unidimensional* é encontrar o menor número de contêineres em que os itens podem ser empacotados.

#### EMPACOTAMENTO UNIDIMENSIONAL (MIN-EU) (BIN PACKING)

**Entrada** Um conjunto de  $n$  itens com tamanhos  $s_i \in \mathbb{Z}_+$ ,  $i \in [n]$  e o tamanho de um contêiner  $S$ .

**Solução** Uma partição de  $[n] = C_1 \cup \dots \cup C_m$  tal que  $\sum_{i \in C_k} s_i \leq S$  para  $k \in [m]$ .

**Objetivo** Minimiza o número de partes (“contêineres”)  $m$ .

A versão de decisão do empacotamento unidimensional (EU) pede decidir se o itens cabem em  $m$  contêineres.

#### Fato 3.3

EU é fortemente NP-completo.

#### Proposição 3.4

Para um tamanho  $S$  fixo EU pode ser resolvido em tempo  $O(n^{S^S})$ .

**Prova.** Podemos supor, sem perda de generalidade, que os itens possuem tamanhos  $1, 2, \dots, S-1$ . Um padrão de alocação de um contêiner pode ser descrito por uma tupla  $(t_1, \dots, t_{S-1})$  sendo  $t_i$  o número de itens de tamanho  $i$ . Seja  $T$  o conjunto de todas padrões que cabem num contêiner. Como  $0 \leq t_i \leq S$  o número total de padrões  $T$  é menor que  $(S+1)^{S-1} = O(S^S)$ .

### 3. Algoritmos de aproximação

Uma ocupação de  $m$  contêineres pode ser descrito por uma tupla  $(n_1, \dots, n_T)$  com  $n_i$  sendo o número de contêineres que usam padrão  $i$ . O número de contêineres é no máximo  $n$ , logo  $0 \leq n_i \leq n$  e o número de alocações diferentes é no máximo  $(n+1)^T = O(n^T)$ . Logo podemos enumerar todas possibilidades em tempo polinomial. ■

#### Proposição 3.5

Para um  $m$  fixo, EU pode ser resolvido em tempo pseudo-polinomial.

**Prova.** Seja  $B(S_1, \dots, S_m, i) \in \{\text{falso}, \text{verdadeiro}\}$  a resposta se itens  $i, i+1, \dots, n$  cabem em  $m$  contêineres com capacidades  $S_1, \dots, S_m$ .  $B$  satisfaz

$$B(S_1, \dots, S_m, i) = \begin{cases} \bigvee_{\substack{1 \leq j \leq m \\ s_i \leq S_j}} B(S_1, \dots, S_j - s_j, \dots, S_m, i+1) & i \leq n \\ \text{verdadeiro} & i > n \end{cases}$$

e  $B(S, \dots, S, 1)$  é a solução do EU<sup>2</sup>. A tabela  $B$  possui no máximo  $n(S+1)^m$  entradas, cada uma computável em tempo  $O(m)$ , logo o tempo total é no máximo  $O(mn(S+1)^m)$ . ■

#### Observação 3.3

Em tempo adicional  $O(S)$  podemos resolver também MIN-EU, procurando o menor  $i$  tal que  $B(\underbrace{S, \dots, S}_{i \text{ vezes}}, 0, \dots, 0, n)$  é verdadeiro. ◇

A proposição 3.4 pode ser melhorada usando programação dinâmica.

#### Proposição 3.6

Para um número fixo  $k$  de tamanhos diferentes, min-EU pode ser resolvido em tempo  $O(n^{2k})$ .

**Prova.** Seja  $B(i_1, \dots, i_k)$  o menor número de contêineres necessário para empacotar  $i_j$  itens do  $j$ -ésimo tamanho e  $T$  o conjunto de todas padrões de alocação de um contêiner.  $B$  satisfaz

$$B(i_1, \dots, i_k) = \begin{cases} 1 + \min_{\substack{t \in T \\ t \leq i}} B(i_1 - t_1, \dots, i_k - t_k) & \text{caso } (i_1, \dots, i_k) \notin T \\ 1 & \text{caso contrário} \end{cases}$$

e  $B(n_1, \dots, n_k)$  é a solução do EU, com  $n_i$  o número de itens de tamanho  $i$  na entrada. A tabela  $B$  tem no máximo  $n^k$  entradas. Como o número de itens em cada padrão de alocação é no máximo  $n$ , temos  $|T| \leq n^k$  e logo o tempo total para preencher  $B$  é no máximo  $O(n^{2k})$ . ■

#### Corolário 3.1

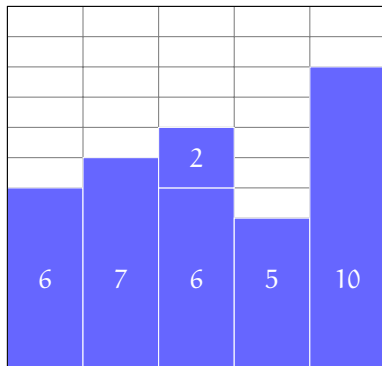
Para um tamanho  $S$  fixo min-EU pode ser resolvido em tempo  $O(n^{2S})$ .

<sup>2</sup>Observe que a disjunção vazia é falso.



**Abordagem prática?**

- Idéia simples: Próximo que cabe (PrC).
- Por exemplo: Itens 6, 7, 6, 2, 5, 10 com limite 12.

**Aproximação?**

- Interessante: PrC é 2-aproximação.
- Observação: PrC é um algoritmo on-line.

**Prova.** Seja  $B$  o número de contêineres usadas,  $V = \sum_{i \in [n]} s_i$ .  $B \leq 2 \lceil V \rceil$  porque dois contêineres consecutivos contém uma soma  $> 1$ . Mas precisamos pelo menos  $\lceil V \rceil$  contêineres, logo  $\text{OPT}(x) \geq \lceil V \rceil$ . Portanto,  $\varphi_{\text{PrC}}(x) \leq 2 \lceil V \rceil \leq 2 \text{OPT}(x)$ . ■

**Aproximação melhor?**

- Isso é a melhor estimativa possível para este algoritmo!
- Considere os  $4n$  itens

$$\underbrace{1/2, 1/2n, 1/2, 1/2n, \dots, 1/2, 1/2n}_{2n \text{ vezes}}$$

- O que faz PrC?  $\varphi_{\text{PrC}}(x) = 2n$ : contêineres com

### 3. Algoritmos de aproximação

$1/2n$	$1/2n$	$1/2n$	$1/2n$	$\dots$		$1/2n$	$1/2n$
$1/2$	$1/2$	$1/2$	$1/2$			$1/2$	$1/2$

- Ótimo:  $n$  contêineres com dois elementos de  $1/2$  + um com  $2n$  elementos de  $1/2n$ .  $\text{OPT}(x) = n = 1$ .

$1/2$	$1/2$	$1/2$	$1/2$	$\dots$		$1/2$	$1/2$	$1/2n$
$1/2$	$1/2$	$1/2$	$1/2$			$1/2$	$1/2$	$1/2n$
								$\vdots$
								$1/2n$
								$1/2n$
								$1/2n$

- Portanto: Assintoticamente a taxa de aproximação 2 é estrito.

#### Melhores estratégias

- Primeiro que cabe (PiC), on-line, com “estoque” na memória
- Primeiro que cabe em ordem decrescente: PiCD, off-line.
- Taxa de aproximação?

$$\varphi_{\text{PiC}}(x) \leq \lceil 1.7\text{OPT}(x) \rceil$$

$$\varphi_{\text{PiCD}}(x) \leq 1.5\text{OPT}(x) + 1$$

**Prova.** (Da segunda taxa de aproximação.) Considere a partição  $A \cup B \cup C \cup D = \{v_1, \dots, v_n\}$  com

$$A = \{v_i \mid v_i > 2/3\}$$

$$B = \{v_i \mid 2/3 \geq v_i > 1/2\}$$

$$C = \{v_i \mid 1/2 \geq v_i > 1/3\}$$

$$D = \{v_i \mid 1/3 \geq v_i\}$$

PiCD primeiro vai abrir  $|A|$  contêineres com os itens do tipo  $A$  e depois  $|B|$  contêineres com os itens do tipo  $B$ . Temos que analisar o que acontece com os itens em  $C$  e  $D$ .

Supondo que um contêiner contém somente itens do tipo D, os outros contêineres tem espaço livre menos que  $1/3$ , senão seria possível distribuir os itens do tipo D para outros contêineres. Portanto, nesse caso

$$B \leq \left\lceil \frac{V}{2/3} \right\rceil \leq 3/2V + 1 \leq 3/2\text{OPT}(x) + 1.$$

Caso contrário (nenhum contêiner contém somente itens tipo D), PiCD encontra a solução ótima. Isso pode ser justificado pelos seguintes observações:

- 1) O número de contêineres sem itens tipo D é o mesmo (eles são os últimos distribuídos em não abrem um novo contêiner). Logo é suficiente mostrar

$$\varphi_{\text{PiCD}}(x \setminus D) = \text{OPT}(x \setminus D).$$

- 2) Os itens tipo A não importam: Sem itens D, nenhum outro item cabe junto com um item do tipo A. Logo:

$$\varphi_{\text{PiCD}}(x \setminus D) = |A| + \varphi_{\text{PiCD}}(x \setminus (A \cup D)).$$

- 3) O melhor caso para os restantes itens são *pares* de elementos em B e C: Nesse situação, PiCD acha a solução ótima.

■

### Aproximação melhor?

- Tese doutorado D. S. Johnson, 1973, 70 pág

$$\varphi_{\text{PiCD}}(x) \leq \frac{11}{9}\text{OPT}(x) + 4$$

- Baker, 1985

$$\varphi_{\text{PiCD}}(x) \leq \frac{11}{9}\text{OPT}(x) + 3$$

#### 3.8.1. Um esquema de aproximação assintótico para min-EU

Duas ideias permitem aproximar min-EU em  $(1+\epsilon)\text{OPT}(I)+1$  para  $\epsilon \in (0, 1]$ .

### 3. Algoritmos de aproximação

**Ideia 1: Arredondamento** Para uma instância  $I$ , define uma instância  $R$  arredondada como segue:

1. Ordene os itens de forma não-decrescente e forma grupos de  $k$  itens.
2. Substitui o tamanho de cada item pelo tamanho do maior elemento no seu grupo.

#### Lema 3.1

Para uma instância  $I$  e a instância  $R$  arredondada temos

$$\text{OPT}(R) \leq \text{OPT}(I) + k$$

**Prova.** Supõe que temos uma solução ótima para  $I$ . Os itens da  $i$ -ésima grupo de  $R$  cabem nos lugares dos itens da  $i + 1$ -ésima grupo dessa solução. Para o último grupo de  $R$  temos que abrir no máximo  $k$  contêineres. ■

#### Ideia 2: Descartando itens menores

#### Lema 3.2

Supõe temos um empacotamento para itens de tamanho maior que  $s_0$  em  $B$  contêineres. Então existe um empacotamento de todos itens com no máximo

$$\max\left\{B, \sum_{i \in [n]} s_i / (S - s_0) + 1\right\}$$

contêineres.

**Prova.** Empacota os itens menores gulosamente no primeiro contêiner com espaço suficiente. Sem abrir um novo contêiner o limite é obviamente correto. Caso contrário, supõe que precisamos  $B'$  contêineres.  $B' - 1$  contêineres contém itens de tamanho total mais que  $S - s_0$ . A ocupação total  $W$  deles tem que ser menor que o tamanho total dos itens, logo

$$(B' - 1)(S - s_0) \leq W \leq \sum_{i \in [n]} s_i.$$

■

#### Juntando as ideias

#### Teorema 3.9

Para  $\epsilon \in (0, 1]$  podemos encontrar um empacotamento usando no máximo  $(1 + \epsilon)\text{OPT}(I) + 1$  contêineres.

**Prova.** O algoritmo tem dois passos:

1. Empacota todos itens de tamanho maior que  $s_0 = \lceil \epsilon/2 S \rceil$  usando arredondamento.
2. Empacota os itens menores depois.

Seja  $I'$  a instância com os  $n' \leq n$  itens maiores. No primeiro passo, formamos grupos com  $\lfloor n'\epsilon^2/4 \rfloor$  itens. Isso resulta em no máximo

$$\frac{n'}{\lfloor n'\epsilon^2/4 \rfloor} \leq \frac{2n'}{n'\epsilon^2/4} = \frac{8}{\epsilon^2}$$

grupos. (A primeira desigualdade usa  $\lfloor x \rfloor \geq x/2$  para  $x \geq 1$ . Podemos supor que  $n'\epsilon^2/4 \geq 1$ , i.e.  $n' \geq 4/\epsilon^2$ . Caso contrário podemos empacotar os itens em tempo constante usando a proposição 3.6.)

Arredondando essa instância de acordo com lema 3.1 podemos obter uma solução em tempo  $O(n^{16/\epsilon^2})$  pela proposição 3.6. Sabemos que  $\text{OPT}(I') \geq n' \lceil \epsilon/2 S \rceil / S \geq n'\epsilon/2$ . Logo temos uma solução com no máximo

$$\text{OPT}(I') + \lfloor n\epsilon^2/4 \rfloor \leq \text{OPT}(I') + n'\epsilon^2/4 \leq (1 + \epsilon/2)\text{OPT}(I') \leq (1 + \epsilon/2)\text{OPT}(I)$$

contêineres.

O segundo passo, pelo lema 3.2, produz um empacotamento com no máximo

$$\max \left\{ (1 + \epsilon/2)\text{OPT}(I), \sum_{i \in [n]} s_i / (S - s_0) + 1 \right\}$$

contêineres, mas

$$\frac{\sum_{i \in [n]} s_i}{S - s_0} \leq \frac{\sum_{i \in [n]} s_i}{S(1 - \epsilon/2)} \leq \frac{\text{OPT}(I)}{1 - \epsilon/2} \leq (1 + \epsilon)\text{OPT}(I).$$

■

### 3.9. Aproximando problemas de sequenciamento

Problemas de sequenciamento recebem nomes da forma

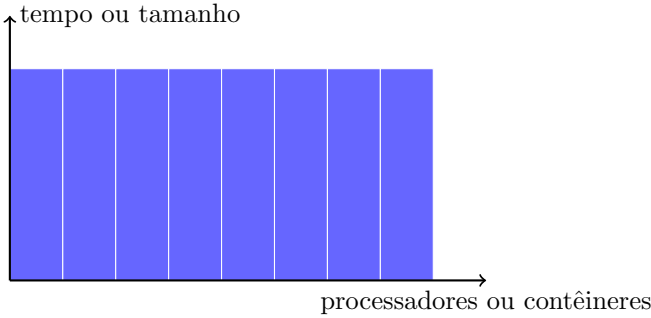
$$\alpha \mid \beta \mid \gamma$$

com campos

3. Algoritmos de aproximação

Máquina $\alpha$	
1	Um processador
P	processadores paralelos
Q	processadores relacionados
R	processadores arbitrários
Restrições $\beta$	
$d_i$	Deadlines
$r_i$	Release times
$p_i = p$	Tempo uniforme $p$
prec	Precedências
Função objetivo $\gamma$	
$C_{\max}$	maximum completion (duração máxima)
$\sum_i C_i$	duração média
$L_i$	lateness $C_i - d_i$
$T_i$	tardiness $\max(L_i, 0)$

Relação com empacotamento unidimensional:



- Empacotamento unidimensional: Dado  $C_{\max}$  minimize o número de processadores.
- $P \parallel C_{\max}$ : Dado um número de contêineres, minimiza o tamanho dos contêineres.

SEQUENCIAMENTO EM PROCESSORES PARALELOS ( $P \parallel C_{\max}$ )

**Entrada** O número  $m$  de processadores e  $n$  tarefas com tempo de execução  $p_i, i \in [n]$ .

**Solução** Um *sequenciamento*, definido por uma alocação  $M_1 \dot{\cup} \dots \dot{\cup} M_m = [n]$  das tarefas às máquinas.

**Objetivo** Minimizar o *makespan* (tempo de término)  $C_{\max} = \max_{j \in [m]} C_j$ , com  $C_j = \sum_{i \in M_j} p_i$  o tempo de término da máquina  $j$ .

**Fato 3.4**

O problema  $P \parallel C_{\max}$  é fortemente NP-completo.

Um limite inferior para  $C_{\max}^* = \text{OPT}$  é

$$LB = \max\left\{\max_{i \in [n]} p_i, \sum_{i \in [n]} p_i / m\right\}.$$

Uma classe de algoritmos gulosos para este problema são os algoritmos de *sequenciamento em lista* (inglês: list scheduling). Eles processam as tarefas em alguma ordem, e alocam a tarefa atual sempre à máquina de menor tempo de término atual.

**Proposição 3.7**

Sequenciamento em lista com ordem arbitrária permite uma  $2 - 1/m$ -aproximação em tempo  $O(n \log n)$ .

**Prova.** Seja  $C_{\max}$  o resultado do sequenciamento em lista. Considera uma máquina com tempo de término  $C_{\max}$ . Seja  $j$  a última tarefa alocada nessa máquina e  $C$  o término da máquina antes de alocar tarefa  $j$ . Logo,

$$\begin{aligned} C_{\max} = C + p_j &\leq \sum_{i \in [j-1]} p_i / m + p_j \leq \sum_{i \in [n]} p_i / m - p_j / m + p_j \\ &\leq LB + (1 - 1/m)LB = (2 - 1/m)LB \leq (2 - 1/m)C_{\max}^*. \end{aligned}$$

A primeira desigualdade é correto, porque alocando tarefa  $j$  a máquina tem tempo de término mínimo. Usando uma fila de prioridade a máquina com o menor tempo de término pode ser encontrada em tempo  $O(\log n)$ . ■

**Observação 3.4**

Pela prova da proposição 3.7 temos

$$LB \leq C_{\max}^* \leq 2LB.$$

◇

O que podemos ganhar com algoritmos off-line? Uma abordagem é ordenar as tarefas por tempo execução não-crescente e aplicar o algoritmo guloso. Essa abordagem é chamada LPT (largest processing time).

### Proposição 3.8

LPT é uma  $4/3 - m/3$ -aproximação em tempo  $O(n \log n)$ .

**Prova.** Seja  $p_1 \geq p_2 \geq \dots p_n$  e supõe que isso é o menor contra-exemplo em que o algoritmo retorne  $C_{\max} > (4/3 - m/3)C_{\max}^*$ . Não é possível que a alocação do item  $j < n$  resulta numa máquina com tempo de término  $C_{\max}$ , porque  $p_1, \dots, p_j$  seria um contra-exemplo menor (mesmo  $C_{\max}$ , menor  $C_{\max}^*$ ). Logo a alocação de  $p_n$  define o resultado  $C_{\max}$ .

Caso  $p_n \leq C_{\max}^*/3$  pela prova da proposição 3.7 temos  $C_{\max} \leq (4/3 - m/3)C_{\max}^*$ , uma contradição. Mas caso  $p_n > C_{\max}^*/3$  todas tarefas possuem tempo de execução pelo menos  $C_{\max}^*/3$  e no máximo duas podem ser executadas em cada máquina. Logo  $C_{\max} \leq 2/3 C_{\max}^*$ , outra contradição. ■

#### 3.9.1. Um esquema de aproximação para $P \parallel C_{\max}$

Pela observação 3.4 podemos reduzir o  $P \parallel C_{\max}$  para o empacotamento unidimensional via uma busca binária no intervalo  $[LB, 2LB]$ . Pela proposição 3.5 isso é possível em tempo  $O(\log LB \cdot mn(2LB + 1)^m)$ .

Com mais cuidado a observação permite um esquema de aproximação em tempo polinomial assintótico: similar com o esquema de aproximação para empacotamento unidimensional, vamos *remover elementos menores e arredondar* a instância.

#### Algoritmo 3.6 (Sequencia)

**Entrada** Uma instância  $I$  de  $P \parallel C_{\max}$ , um término máximo  $C$  e um parâmetro de qualidade  $\epsilon$ .

- 1 Sequencia( $I, C, \epsilon$ ):=
- 2   remove as tarefas menores com  $p_j < \epsilon C$ ,  $j \in [n]$
- 3   arredonda cada  $p_j \in [\epsilon C(1 + \epsilon)^i, \epsilon C(1 + \epsilon)^{i+1})$  para algum  $i$   
                   para  $p'_j = \epsilon C(1 + \epsilon)^i$
- 4   resolve a instância arredondada com programação  
           dinâmica (proposição 3.6)
- 5   empacota os itens menores gulosamente, usando  
           novas máquinas para manter o término  $(1 + \epsilon)C$

#### Lema 3.3

O algoritmo Sequencia gera um sequenciamento que termina em no máximo  $(1 + \epsilon)C$  em tempo  $O(n^{2 \lceil \log_{1+\epsilon} 1/\epsilon \rceil})$ . Ele não usa mais máquinas que o mínimo necessário para executar as tarefas com término  $C$

**Prova.** Para cada intervalo válido temos  $\epsilon C(1 + \epsilon)^i \leq C$ , logo o número de intervalos é no máximo  $k = \lceil \log_{1+\epsilon} 1/\epsilon \rceil$ . O valor  $k$  também é um limite



para o número de valores  $p'_j$  distintos e pela proposição 3.6 o terceiro passo resolve a instância arredondada em tempo  $O(n^{2k})$ . Essa solução com os itens de tamanho original termina em no máximo  $(1 + \epsilon)C$ , porque  $p_j/p'_j < 1 + \epsilon$ . O número mínimo de máquinas para executar as tarefas em tempo  $C$  é o valor  $m := \min\text{-EU}(C, (p_j)_{j \in [n]})$  do problema de empacotamento unidimensional correspondente. Caso o último passo do algoritmo não usa novas máquinas ele precisa  $\leq m$  máquinas, porque a instância arredondada foi resolvida exatamente. Caso contrário, uma tarefa com tempo de execução menor que  $\epsilon C$  não cabe nenhuma máquina, e todas máquinas usadas tem tempo de término mais que  $C$ . Logo o empacotamento ótimo com término  $C$  tem que usar pelo menos o mesmo número de máquinas. ■

### Proposição 3.9

O resultado da busca binária usando o algoritmo Sequencia  $C_{\max} = \min\{C \in [LB, 2LB] \mid \text{Sequencia}(I, C, \epsilon) \leq m\}$  é no máximo  $C_{\max}^*$ .

**Prova.** Com  $\text{Sequencia}(I, C, \epsilon) \leq \min\text{-EU}(C, (p_i)_{i \in [n]})$  temos

$$\begin{aligned} C_{\max} &= \min\{C \in [LB, 2LB] \mid \text{Sequencia}(I, C, \epsilon) \leq m\} \\ &\leq \min\{C \in [LB, 2LB] \mid \min\text{-EU}(C, (p_i)_{i \in [n]}) \leq m\} \\ &= C_{\max}^* \end{aligned}$$

■

### Teorema 3.10

A busca binária usando o algoritmo Sequencia para determinar determina um sequenciamento em tempo  $O(n^{2\lceil \log_{1+\epsilon} 1/\epsilon \rceil} \log LB)$  de término máximo  $(1 + \epsilon)C_{\max}^*$ .

**Prova.** Pelo lema 3.3 e a proposição 3.9. ■

## 3.10. Exercícios

### Exercício 3.1

Por que um subgrafo conexo de menor custo sempre é uma árvore?

### Exercício 3.2

Mostra que o número de vértices com grau ímpar num grafo sempre é par.

### Exercício 3.3

Um aluno propõe a seguinte heurística para o empacotamento unidimensional: Ordene os itens em ordem crescente, coloca o item com peso máximo junto com quantas itens de peso mínimo que é possível, e depois continua com o segundo maior item, até todos itens foram colocados em bins. Temos o algoritmo

### 3. Algoritmos de aproximação

```
1  ordene itens em ordem crescente
2  m := 1; M := n
3  while (m < M) do
4      abre novo contêiner, coloca  $v_M$ , M := M - 1
5      while ( $v_m$  cabe e m < M) do
6          coloca  $v_m$  no contêiner atual
7          m := m + 1
8      end while
9  end while
```

Qual a qualidade desse algoritmo? É um algoritmo de aproximação? Caso sim, qual a taxa de aproximação dele? Caso não, por quê?

#### Exercício 3.4

Prof. Rapidez propõe o seguinte pré-processamento para o algoritmo SAT-R de aproximação para MAX-SAT (página 97): Caso a instância contém cláusulas com um único literal, vamos escolher uma delas, definir uma atribuição parcial que satisfazê-la, e eliminar a variável correspondente. Repetindo esse procedimento, obtemos uma instância cujas cláusulas tem 2 ou mais literais. Assim, obtemos  $l \geq 2$  na análise do algoritmo, o podemos garantir que  $E[X] \geq 3n/4$ , i.e. obtemos uma 4/3-aproximação.

Este análise é correto ou não?

## 4. Algoritmos randomizados

Um algoritmo randomizado usa *eventos randômicos* na sua execução. Modelos computacionais adequadas são máquinas de Turing randômicas – mais usadas na área de complexidade – ou máquinas RAM com um comando do tipo `random(S)` que retorne um elemento randômico do conjunto  $S$ .

Veja alguns exemplos de probabilidades:

- Probabilidade morrer caindo da cama:  $1/2 \times 10^6$  [36].
- Probabilidade acertar 6 números de 60 na mega-sena:  $1/50063860$ .
- Probabilidade que a memória falha: em memória moderna temos 1000 FIT/MBit, i.e.  $6 \times 10^{-7}$  erros por segundo num memória de 256 MB.<sup>1</sup>
- Probabilidade que um meteorito destrói um computador em cada milissegundo:  $\geq 2^{-100}$  (supondo que cada milênio ao menos um meteorito destrói uma área de  $100 \text{ m}^2$ ).

Portanto, um algoritmo que retorna uma resposta falsa com baixa probabilidade é aceitável. Em retorno um algoritmo randomizado frequentemente é

- mais simples;
- mais eficiente: para alguns problemas, o algoritmos randômica é o mais eficiente conhecido;
- mais robusto: algoritmos randômicos podem ser menos dependente da distribuição das entradas.
- a única alternativa: para alguns problemas, conhecemos só algoritmos randômicos.

### 4.1. Teoria de complexidade

#### Classes de complexidade

---

<sup>1</sup>FIT é uma abreviação de “failure-in-time” e é o número de erros cada  $10^9$  segundos. Para saber mais sobre erros em memória veja [39].

### Definição 4.1

Seja  $\Sigma$  algum alfabeto e  $R(\alpha, \beta)$  a classe de linguagens  $L \subseteq \Sigma^*$  tal que existe um algoritmo de decisão em tempo polinomial  $A$  que satisfaz

- $x \in L \Rightarrow \Pr[A(x) = \text{sim}] \geq \alpha$ .
- $x \notin L \Rightarrow \Pr[A(x) = \text{não}] \geq \beta$ .

(A probabilidade é sobre todas sequências de bits randômicos  $r$ . Como o algoritmo executa em tempo polinomial no tamanho da entrada  $|x|$ , o número de bits randômicas  $|r|$  é polinomial em  $|x|$  também.)

Com isso podemos definir

- a classe  $RP := R(1/2, 1)$  (randomized polynomial), dos problemas que possuem um algoritmo com erro unilateral (no lado do “sim”); a classe  $\text{co-RP} = R(1, 1/2)$  consiste dos problemas com erro no lado de “não”;
- a classe  $ZPP := RP \cap \text{co-RP}$  (zero-error probabilistic polynomial) dos problemas que possuem algoritmo randomizado sem erro;
- a classe  $PP := R(1/2 + \epsilon, 1/2 + \epsilon)$  (probabilistic polynomial), dos problemas com erro  $1/2 + \epsilon$  nos dois lados; e
- a classe  $BPP := R(2/3, 2/3)$  (bounded-error probabilistic polynomial), dos problemas com erro  $1/3$  nos dois lados.

Algoritmos que respondem corretamente somente com uma certa probabilidade também são chamados do tipo *Monte Carlo*, enquanto algoritmos que usam randomização somente internamente, mas respondem sempre corretamente são do tipo *Las Vegas*.

### Exemplo 4.1 (Teste de identidade de polinômios)

Dado dois polinômios  $p(x)$  e  $q(x)$  de grau máximo  $d$ , como saber se  $p(x) \equiv q(x)$ ? Caso temos os dois na forma canônica  $p(x) = \sum_{0 \leq i \leq d} p_i x^i$  ou na forma fatorada  $p(x) = \prod_{1 \leq i \leq d} (x - r_i)$  isso é simples responder por comparação de coeficientes em tempo  $\tilde{O}(n)$ . E caso contrário? Uma conversão para a forma canônica pode custar  $\Theta(d^2)$  multiplicações. Uma abordagem randomizada é vantajosa, se podemos avaliar o polinômio mais rápido (por exemplo em  $O(d)$ ):

- 1 `identico(p, q) :=`
- 2   Seleciona um número randômico  $r$  no intervalo  $[1, 100d]$ .
- 3   Caso  $p(r) = q(r)$  retorne “sim”.
- 4   Caso  $p(r) \neq q(r)$  retorne “não”.

Caso  $p(x) \equiv q(x)$ , o algoritmo responde “sim” com certeza. Caso contrário a resposta pode ser errada, se  $p(r) = q(r)$  por acaso. Qual a probabilidade disso?  $p(x) - q(x)$  é um polinômio de grau  $d$  e possui no máximo  $d$  raízes. Portanto, a probabilidade de encontrar um  $r$  tal que  $p(r) = q(r)$ , caso  $p \not\equiv q$  é  $d/100d = 1/100$ . Isso demonstra que o teste de identidade pertence à classe  $\text{co-RP}$ .  $\diamond$

#### Observação 4.1

É uma pergunta em aberto se o teste de identidade pertence a  $P$ .  $\diamond$

#### 4.1.1. Amplificação de probabilidades

Caso não estamos satisfeitos com a probabilidade de  $1/100$  no exemplo acima, podemos repetir o algoritmo  $k$  vezes, e responder “sim” somente se todas  $k$  repetições responderam “sim”. A probabilidade erradamente responder “não” para polinômios idênticos agora é  $(1/100)^k$ , i.e. ela diminui exponencialmente com o número de repetições.

Essa técnica é uma *amplificação* da probabilidade de obter a solução correta. Ela pode ser aplicada para melhorar a qualidade de algoritmos em todas classes “Monte Carlo”. Com um número constante de repetições, obtemos uma probabilidade baixa nas classes  $\text{RP}$ ,  $\text{co-RP}$  e  $\text{BPP}$ . Isso não se aplica a  $\text{PP}$ : é possível que  $\epsilon$  diminui exponencialmente com o tamanho da instância. Um exemplo de amplificação de probabilidade encontra-se na prova do teorema 4.6.

#### Teorema 4.1

$R(\alpha, 1) = R(\beta, 1)$  para  $0 < \alpha, \beta < 1$ .

**Prova.** Sem perda de generalidade seja  $\alpha < \beta$ . Claramente  $R(\beta, 1) \subseteq R(\alpha, 1)$ . Supõe que  $A$  é um algoritmo que testemunha  $L \in R(\alpha, 1)$ . Execute  $A$  no máximo  $k$  vezes, respondendo “sim” caso  $A$  responde “sim” em alguma iteração e “não” caso contrário. Chama esse algoritmo  $A'$ . Caso  $x \notin L$  temos  $P[A'(x) = \text{“não”}] = 1$ . Caso  $x \in L$  temos  $P[A'(x) = \text{“sim”}] \geq 1 - (1 - \alpha)^k$ , logo para  $k \geq \ln 1 - \beta / \ln 1 - \alpha$ ,  $P[A'(x) = \text{“sim”}] \geq \beta$ . ■

#### Corolário 4.1

$\text{RP} = R(\alpha, 1)$  para  $0 < \alpha < 1$ .

#### Teorema 4.2

$R(\alpha, \alpha) = R(\beta, \beta)$  para  $1/2 < \alpha, \beta$ .

**Prova.** Sem perda de generalidade seja  $\alpha < \beta$ . Claramente  $R(\beta, \beta) \subseteq R(\alpha, \alpha)$ .

#### 4. Algoritmos randomizados

Supõe que  $A$  é um algoritmo que testemunha  $L \in R(\alpha, \alpha)$ . Executa  $A$   $k$  vezes, responde “sim” caso a maioria de respostas obtidas foi “sim”, e “não” caso contrário. Chama esse algoritmo  $A'$ . Para  $x \in L$  temos

$$P[A'(x) = \text{“sim”}] = P[A(x) = \text{“sim”} \geq \lfloor k/2 \rfloor + 1 \text{ vezes}] \geq 1 - e^{-2k(\alpha-1/2)^2}$$

e para  $k \geq \ln(\beta-1)/2(\alpha-1/2)^2$  temos  $P[A'(x) = \text{“sim”}] \geq \beta$ . Similarmente, para  $x \notin L$  temos  $P[A'(x) = \text{“não”}] \geq \beta$ . Logo  $L \in R(\beta, \beta)$ . ■

#### Corolário 4.2

$BPP = R(\alpha, \alpha)$  para  $1/2 < \alpha$ .

#### Observação 4.2

Os resultados acima são válidos ainda caso o erro diminue polinomialmente com o tamanho da instância, i.e.  $\alpha, \beta \geq n^{-c}$  no caso do teorema 4.1 e  $\alpha, \beta \geq 1/2 + n^{-c}$  no caso do teorema 4.2 para um constante  $c$  (ver por exemplo Arora e Barak [4]). ◇

#### 4.1.2. Relação entre as classes

##### Duas caracterizações alternativas de ZPP

#### Definição 4.2

Um algoritmo  $A$  é *honesto* se

- i) ele responde ou “sim”, ou “não” ou “não sei”,
- ii)  $\Pr[A(x) = \text{“não sei”}] \leq 1/2$ , e
- iii) no caso ele responde, ele não erra, i.e., para  $x$  tal que  $A(x) \neq \text{“não sei”}$  temos  $A(x) = \text{“sim”} \iff x \in L$ .

Uma linguagem é honesta caso ela possui um algoritmo honesto. Com isso também podemos falar da classe das linguagens honestas.

#### Teorema 4.3

ZPP é a classe das linguagens honestas.

#### Lema 4.1

Caso  $L \in ZPP$  existe um algoritmo honesto para  $L$ .

**Prova.** Para um  $L \in ZPP$  existem dois algoritmos  $A_1 \in RP$  e  $A_2 \in co-RP$ . Vamos construir um algoritmo

```

1  if  $A_1(x) = \text{"n\~ao"}$  e  $A_2(x) = \text{"n\~ao"}$  then
2    return ‘ ‘n\~ao’ ’
3  else if  $A_1(x) = \text{"n\~ao"}$  e  $A_2(x) = \text{"sim"}$  then
4    return ‘ ‘n\~ao sei’ ’
5  else if  $A_1(x) = \text{"sim"}$  e  $A_2(x) = \text{"n\~ao"}$  then
6    { caso imposs\~ivel }
7  else if  $A_1(x) = \text{"sim"}$  e  $A_2(x) = \text{"sim"}$  then
8    return ‘ ‘sim’ ’
9  end if

```

O algoritmo responde corretamente “sim” e “n\~ao”, porque um dos dois algoritmos n\~ao erra. Qual a probabilidade do segundo caso? Para  $x \in L$ ,  $\Pr[A_1(x) = \text{"n\~ao"} \wedge A_2(x) = \text{"sim"}] \leq 1/2 \times 1 = 1/2$ . Similarmente, para  $x \notin L$ ,  $\Pr[A_1(x) = \text{"n\~ao"} \wedge A_2(x) = \text{"sim"}] \leq 1 \times 1/2 = 1/2$ . ■

#### Lema 4.2

Caso  $L$  possui um algoritmo honesto  $L \in RP$  e  $L \in co - RP$ .

**Prova.** Seja  $A$  um algoritmo honesto. Constrói outro algoritmo que sempre responde “n\~ao” caso  $A$  responde “n\~ao sei”, e sen\~ao responde igual. No caso de  $co - RP$  analogamente constrói um algoritmos que responde “sim” nos casos “n\~ao sei” de  $A$ . ■

#### Definição 4.3

Um algoritmo  $A$  é *sem falha* se ele sempre responde “sim” ou “n\~ao” corretamente em *tempo polinomial esperado*. Com isso podemos também falar de linguagens sem falha e a classe das linguagens sem falha.

#### Teorema 4.4

ZPP é a classe das linguagens sem falha.

#### Lema 4.3

Caso  $L \in ZPP$  existe um algoritmo sem falha para  $L$ .

**Prova.** Sabemos que existe um algoritmo honesto para  $L$ . Repete o algoritmo honesto até encontrar um “sim” ou “n\~ao”. Como o algoritmo honesto executa em tempo polinomial  $p(n)$ , o tempo esperado desse algoritmo ainda é polinomial:

$$\sum_{k>0} k2^{-k}p(n) \leq 2p(n)$$

■

#### Lema 4.4

Caso  $L$  possui um algoritmo  $A$  sem falha,  $L \in RP$  e  $L \in co - RP$ .

#### 4. Algoritmos randomizados

**Prova.** Caso  $A$  tem tempo esperado  $p(n)$  executa ele para um tempo  $2p(n)$ . Caso o algoritmo responde, temos a resposta certa. Caso contrário, responde “sim”. Pela desigualdade de Markov temos uma resposta com probabilidade  $P[T \geq 2p(n)] \leq p(n)/2p(n) = 1/2$ . Isso mostra que existe um algoritmo honesto para  $L$ , e pelo lema 4.2  $L \in RP$ . O argumento para  $L \in co - RP$  é similar. ■

#### Mais relações

##### Teorema 4.5

$RP \subseteq NP$  e  $co - RP \subseteq co - NP$

**Prova.** Supõe que temos um algoritmo em  $RP$  para algum problema  $L$ . Podemos, não-deterministicamente, gerar todas sequências  $r$  de bits randômicos e responder “sim” caso alguma execução encontra “sim”. O algoritmo é correto, porque caso para um  $x \notin L$ , não existe uma sequência randômica  $r$  tal que o algoritmo responde “sim”. A prova do segundo caso é similar. ■

##### Teorema 4.6

$RP \subseteq BPP$  e  $co - RP \subseteq BPP$ .

**Prova.** Seja  $A$  um algoritmo para  $L \in RP$ . Constrói um algoritmo  $A'$

```
1  if  $A(x) = \text{“não”}$  e  $A(x) = \text{“não”}$  then
2    return ‘ ‘ não ’ ’
3  else
4    return ‘ ‘ sim ’ ’
5  end if
```

Caso  $x \notin L$ ,  $\Pr[A'(x) = \text{“não”}] = \Pr[A(x) = \text{“não”} \wedge A(x) = \text{“não”}] = 1 \times 1 = 1$ .  
1. Caso  $x \in L$ ,

$$\begin{aligned} \Pr[A'(x) = \text{“sim”}] &= 1 - \Pr[A'(x) = \text{“não”}] = 1 - \Pr[A(x) = \text{“não”} \wedge A(x) = \text{“não”}] \\ &\geq 1 - 1/2 \times 1/2 = 3/4 > 2/3. \end{aligned}$$

(Observe que para  $k$  repetições de  $A$  obtemos  $\Pr[A'(x) = \text{“sim”}] \geq 1 - 1/2^k$ , i.e., o erro diminui exponencialmente com o número de repetições.) O argumento para  $co - RP$  é similar. ■

**Relação com a classe NP e abundância de testemunhas** Lembramos que a classe  $NP$  contém problemas que permitem uma verificação de uma solução em tempo polinomial. Não-deterministicamente podemos “chutar” uma solução e verificá-la. Se o número de soluções positivas de cada instância é mais que a metade do número total de soluções, o problema pertence a  $RP$ : podemos



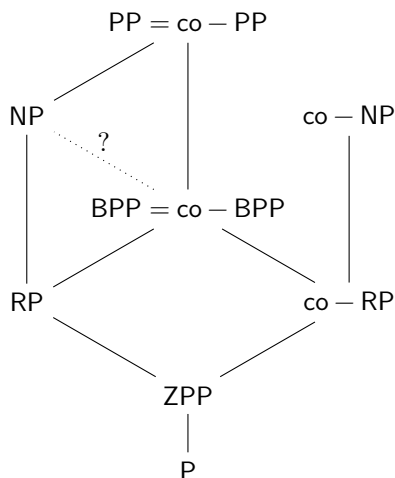


Figura 4.1.: Relações entre classes de complexidade para algoritmos randomizados.

gerar uma solução randômica e testar se ela possui a característica desejada. Uma problema desse tipo possui uma *abundância de testemunhas*. Isso demonstra a importância de algoritmos randomizados. O teste de equivalência de polinômios acima é um exemplo de abundância de testemunhas.

## 4.2. Seleção

O algoritmo determinístico para selecionar o  $k$ -ésimo elemento de uma sequência não ordenada  $x_1, \dots, x_n$  discutido na seção A.1 (página 142) pode ser simplificado usando randomização: escolheremos, um elemento pivô  $m = x_i$  aleatório. Com isso o algoritmo A.1 fica mais simples:

### Algoritmo 4.1 (Seleção randomizada)

**Entrada** Números  $x_1, \dots, x_n$ , posição  $k$ .

**Saída** O  $k$ -ésimo maior número.

```

1   $S(k, \{x_1, \dots, x_n\}) :=$ 
2    if  $n \leq 1$ 
3      calcula e retorne o  $k$  ésimo elemento
  
```

#### 4. Algoritmos randomizados

```

4   end if
5    $m := x_i$  para um  $i \in [n]$  aleatória
6    $L := \{x_i \mid x_i < m, 1 \leq i \leq n\}$ 
7    $R := \{x_i \mid x_i \geq m, 1 \leq i \leq n\}$ 
8    $i := |L| + 1$ 
9   if  $i = k$  then
10    return  $m$ 
11  else if  $i > k$  then
12    return  $S(k, L)$ 
13  else
14    return  $S(k - i, R)$ 
15  end if

```

Para determinar a complexidade podemos observar que com probabilidade  $1/n$  temos  $|L| = i$  e  $|R| = n - i$  e o caso pessimista é uma chamada recursiva com  $\max\{i, n - i\}$  elementos. Logo, com custo  $cn$  para particionar o conjunto e os testes temos

$$\begin{aligned}
T(n) &\leq \sum_{i \in [0, n]} 1/n T(\max\{n - i, i\}) + cn \\
&= 1/n \left( \sum_{i \in [0, \lfloor n/2 \rfloor]} T(n - i) + \sum_{i \in [\lceil n/2 \rceil, n]} T(i) \right) + cn \\
&2/n \sum_{i \in [0, \lfloor n/2 \rfloor]} T(n - i) + cn
\end{aligned}$$

Separando o termo  $T(n)$  do lado direito obtemos

$$\begin{aligned}
(1 - 2/n)T(n) &\leq 2/n \sum_{i \in [1, \lfloor n/2 \rfloor]} T(n - i) + cn \\
\iff T(n) &\leq \frac{2}{n-2} \left( \sum_{i \in [1, \lfloor n/2 \rfloor]} T(n - i) + cn^2/2 \right).
\end{aligned}$$

Provaremos por indução que  $T(n) \leq c'n$  para uma constante  $c'$ . Para um  $n \leq n_0$  o problema pode ser claramente resolvido em tempo constante (por exemplo em  $O(n_0 \log n_0)$  via ordenação). Logo, supõe que  $T(i) \leq c'i$  para

$i < n$ . Demonstraremos que  $T(n) \leq c'n$ . Temos

$$\begin{aligned} T(n) &\leq \frac{2}{n-2} \left( \sum_{i \in [1, \lfloor n/2 \rfloor]} T(n-i) + cn^2/2 \right) \\ &\leq \frac{2c'}{n-2} \left( \sum_{i \in [1, \lfloor n/2 \rfloor]} n-i + cn^2/2c' \right) \\ &= \frac{2c'}{n-2} ((2n - \lfloor n/2 \rfloor - 1) \lfloor n/2 \rfloor / 2 + cn^2/2c') \end{aligned}$$

e com  $2n - \lfloor n/2 \rfloor - 1 \leq 3/2n$

$$\begin{aligned} &\leq \frac{c'}{n-2} (3/4n^2 + cn^2/c') \\ &= c'n \frac{(3/4 + c/c')n}{n-2} \end{aligned}$$

Logo para  $n \geq n_0 := 14$  temos  $n/(n-2) \leq 7/6$  e temos

$$T(n) \leq c'n(7/8 + c/c') \leq c'n$$

para um  $c' > 8c$ .

### 4.3. Corte mínimo

#### CORTE MÍNIMO

**Entrada** Grafo não-direcionado  $G = (V, A)$  com pesos  $c : A \rightarrow \mathbb{Z}_+$  nas arestas.

**Solução** Uma partição  $V = S \cup (V \setminus S)$ .

**Objetivo** Minimizar o peso do corte  $\sum_{\substack{\{u,v\} \in A \\ u \in S, v \in V \setminus S}} c_{\{u,v\}}$ .

Soluções determinísticas:

- Calcular a árvore de Gomory-Hu: a aresta de menor peso define o corte mínimo.
- Calcular o corte mínimo (via fluxo máximo) entre um vértice fixo  $s \in V$  e todos outros vértices: o menor corte encontrado é o corte mínimo.

Custo em ambos casos:  $O(n)$  aplicações de um algoritmo de fluxo máximo, i.e.  $O(mn^2 \log(n/m))$  no caso do algoritmo de Goldberg-Tarjan.

**Solução randomizada para pesos unitários** No que segue supomos que os pesos são unitários, i.e.  $c_a = 1$  para  $a \in A$ . Uma abordagem simples é baseada na seguinte observação: se escolhemos uma aresta que não faz parte de um corte mínimo, e contraímos-la (i.e. identificamos os vértices adjacentes), obtemos um grafo menor, que ainda contém o corte mínimo. Se escolhemos uma aresta randomicamente, a probabilidade de por acaso escolher uma aresta de um corte mínimo é baixa.

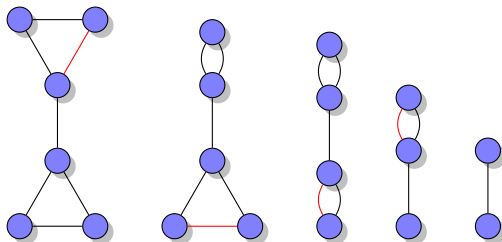
```

1  cmr(G) :=
2    while G possui mais que dois vértices
3      escolha uma aresta {u,v} randômicamente
4      identifica u e v em G
5    end while
6    return o corte definido pelos dois vértices em G

```

#### Exemplo 4.2

Uma sequencia de contrações (das arestas vermelhas).



◇

Dizemos que uma aresta “sobrevive” uma contração, caso ele não foi contraído.

#### Lema 4.5

A probabilidade que os  $k$  arestas de um corte mínimo sobrevivem  $n - t$  contrações (de  $n$  para  $t$  vértices) é  $\Omega((t/n)^2)$ .

**Prova.** Como o corte mínimo é  $k$ , cada vértice possui grau pelo menos  $k$ , e portanto o número de arestas após iteração  $0 \leq i < n - t$  e maior ou igual a  $k(n - i)/2$  (com a convenção que a “iteração 0” produz o grafo inicial). Supondo que as  $k$  arestas do corte mínimo sobreviveram a iteração  $i$ , a probabilidade de não sobreviver a próxima iteração é  $k/(k(n - i)/2) = 2/(n - i)$ .

Logo, a probabilidade do corte sobreviver todas iterações é ao menos

$$\begin{aligned} \prod_{0 \leq i < n-t} 1 - \frac{2}{n-i} &= \prod_{0 \leq i < n-t} \frac{n-i-2}{n-i} \\ &= \frac{(n-2)(n-3) \cdots t-1}{n(n-1) \cdots t+1} = \binom{t}{2} / \binom{n}{2} = \Omega((t/n)^2). \end{aligned}$$

■

#### Teorema 4.7

Dado um corte mínimo  $C$  de tamanho  $k$ , a probabilidade do algoritmo  $\text{cmr}$  retornar  $C$  é  $\Omega(n^{-2})$ .

**Prova.** Caso o grafo possui  $n$  vértices, o algoritmo termina em  $n-2$  iterações: podemos aplicar o lema acima com  $t = 2$ . ■

#### Observação 4.3

O que acontece se repetirmos o algoritmo algumas vezes? Seja  $C_i$  uma variável que indica se o corte mínimo foi encontrado repetição  $i$ . Temos  $P[C_i = 1] \geq 2n^{-2}$  e portanto  $P[C_i = 0] \leq 1 - 2n^{-2}$ . Para  $kn^2$  repetições, vamos encontrar  $C = \sum C_i$  cortes mínimos com probabilidade

$$P[C \geq 1] = 1 - P[C = 0] \geq 1 - (1 - 2n^{-2})^{kn^2} \geq 1 - e^{-2k}.$$

Para  $k = \log n$  obtemos  $P[C \geq 1] \geq 1 - n^{-2}$ . ◇

Logo, se repetimos esse algoritmo  $n^2 \log n$  vezes e retornamos o menor corte encontrado, achamos o corte mínimo com probabilidade razoável. Se a implementação realiza uma contração em  $O(n)$  o algoritmo possui complexidade  $O(n^2)$  e com as repetições em total  $O(n^4 \log n)$ .

**Implementação de contrações** Para garantir a complexidade acima, uma contração tem que ser implementada em  $O(n)$ . Isso é possível tanto na representação por uma matriz de adjacência, quanto na representação pela listas de adjacência. A contração de dois vértices adjacentes resulta em um novo vértice, que é adjacente aos vizinhos dos dois. Na contração arestas de um vértice com si mesmo são removidas. Múltiplas arestas entre dois vértices tem que ser mantidas para garantir a corretude do algoritmo.

**Um algoritmo melhor** O problema principal com o algoritmo acima é que nas últimas iterações, a probabilidade de contrair uma aresta do corte mínimo é grande. Para resolver esse problema, executaremos o algoritmo duas vezes para instâncias menores, para aumentar a probabilidade de não contrair o corte mínimo.

#### 4. Algoritmos randomizados

```

1  cmr2(G) :=
2    if (G possui menos que 6 vértices)
3      determina o corte mínimo C por exaustão
4      return C
5    else
6      t := ⌈1 + n/√2⌉
7      seja G1 o resultado de n - t contrações em G
8      seja G2 o resultado de n - t contrações em G
9      C1 := cmr2(G1)
10     C2 := cmr2(G2)
11     return o menor dos dois cortes C1 e C2
12  end if

```

Esse algoritmo possui complexidade de tempo  $O(n^2 \log n)$  e encontra um corte mínimo com probabilidade  $\Omega(1/\log n)$ .

##### Lema 4.6

A probabilidade de um corte mínimo sobreviver  $t = \lceil 1 + n/\sqrt{2} \rceil$  contrações é no mínimo  $1/2$ .

**Prova.** Pelo lema 4.5 a probabilidade é

$$\frac{\lceil 1 + n/\sqrt{2} \rceil (\lceil 1 + n/\sqrt{2} \rceil - 1)}{n(n-1)} \geq \frac{(1 + n/\sqrt{2})(n/\sqrt{2})}{n(n-1)} = \frac{\sqrt{2} + n}{2(n-1)} \geq \frac{n}{2n} = \frac{1}{2}.$$

■

Seja  $P(t)$  a probabilidade que um corte com  $k$  arestas sobrevive caso o grafo possui  $t$  vértices. Temos

$$P[\text{o corte sobrevive em } H_1] \geq 1/2P(\lceil 1 + t/\sqrt{2} \rceil)$$

$$P[\text{o corte sobrevive em } H_2] \geq 1/2P(\lceil 1 + t/\sqrt{2} \rceil)$$

$$P[\text{o corte não sobrevive em } H_1 \text{ e } H_2] \leq (1 - 1/2P(\lceil 1 + t/\sqrt{2} \rceil))^2$$

$$\begin{aligned}
 P(t) = P[\text{o corte sobrevive em } H_1 \text{ ou } H_2] &\geq 1 - (1 - 1/2P(\lceil 1 + t/\sqrt{2} \rceil))^2 \\
 &= P(\lceil 1 + t/\sqrt{2} \rceil) - 1/4P(\lceil 1 + t/\sqrt{2} \rceil)^2
 \end{aligned}$$

Para resolver essa recorrência, define  $Q(k) = P(\sqrt{2}^k)$  com base  $Q(0) = 1$  para obter a recorrência simplificada

$$\begin{aligned} Q(k+1) &= P(\sqrt{2}^{k+1}) = P(\lceil 1 + \sqrt{2}^k \rceil) - 1/4P(\lceil 1 + \sqrt{2}^k \rceil)^2 \\ &\approx P(\sqrt{2}^k) - P(\sqrt{2}^k)^2/4 = Q(k) - Q(k)^2/4 \end{aligned}$$

e depois  $R(k) = 4/Q(k) - 1$  com base  $R(0) = 3$  para obter

$$\frac{4}{R(k+1)+1} = \frac{4}{R(k)+1} - \frac{4}{(R(k)+1)^2} \iff R(k+1) = R(k) + 1 + 1/R(k).$$

$R(k)$  satisfaz

$$k < R(k) < k + H_{k-1} + 3$$

**Prova.** Por indução. Para  $k = 1$  temos  $1 < R(1) = 13/3 < 1 + H_0 + 3 = 5$ . Caso a HI está satisfeito, temos

$$\begin{aligned} R(k+1) &= R(k) + 1 + 1/R(k) > R(k) + 1 > k + 1 \\ R(k+1) &= R(k) + 1 + 1/R(k) < k + H_{k-1} + 3 + 1 + 1/k = (k+1) + H_k + 3 \end{aligned}$$

■

Logo,  $R(k) = k + \Theta(\log k)$ , e com isso  $Q(k) = \Theta(1/k)$  e finalmente  $P(t) = \Theta(1/\log t)$ .

Para determinar a complexidade do algoritmo `cmr2` observe que temos  $O(\log n)$  níveis recursivos e cada contração pode ser feito em tempo  $O(n^2)$ , portanto

$$T_n = 2T(\lceil 1 + n/\sqrt{2} \rceil) + O(n^2).$$

Aplicando o teorema de Akra-Bazzi obtemos a equação característica  $2(1/\sqrt{2})^p = 1$  com solução  $p = 2$  e

$$T_n \in \Theta(n^2(1 + \int_1^n \frac{cu^2}{u^3} du)) = \Theta(n^2 \log n).$$

## 4.4. Teste de primalidade

Um problema importante na criptografia é encontrar números primos grandes (p.ex. RSA). Escolhendo um número  $n$  randômico, qual a probabilidade de  $n$  ser primo?

**Teorema 4.8 (Hadamard [25], Vallée Poussin [40])**

(Teorema dos números primos.)

Para  $\pi(n) = |\{p \leq n \mid p \text{ primo}\}|$  temos

$$\lim_{n \rightarrow \infty} \frac{\pi(n)}{n/\ln n} = 1.$$

(Em particular  $\pi(n) = \Theta(n/\ln n)$ .)

Portanto, a probabilidade de um número randômico no intervalo  $[2, n]$  ser primo assintoticamente é somente  $1/\ln n$ . Então para encontrar um número primo, temos que testar se  $n$  é primo mesmo. Observe que isso não é igual a fatoração de  $n$ . De fato, temos testes randomizados (e determinísticos) em tempo polinomial, enquanto não sabemos fatorar nesse tempo. Uma abordagem simples é testar todos os divisores:

```

1  Prim01(n) :=
2    for i = 2, 3, 5, 7, ..., ⌊√n⌋ do
3      if i|n return ‘‘Não’’
4    end for
5    return ‘‘Sim’’
```

O tamanho da entrada  $n$  é  $t = \log n$  bits, portanto o número de iterações é  $\Theta(\sqrt{n}) = \Theta(2^{t/2})$  e a complexidade  $\Omega(2^{t/2})$  (mesmo contando o teste de divisão com  $O(1)$ ) desse algoritmo é exponencial. Para testar a primalidade mais eficiente, usaremos uma característica particular dos números primos.

**Teorema 4.9 (Fermat, Euler)**

Para  $p$  primo e  $a \geq 0$  temos

$$a^p \equiv a \pmod{p}.$$

**Prova.** Por indução sobre  $a$ . Base: evidente. Seja  $a^p \equiv a$ . Temos

$$(a+1)^p = \sum_{0 \leq i \leq p} \binom{p}{i} a^i$$

e para  $0 < i < p$

$$p \mid \binom{p}{i} = \frac{p(p-1) \cdots (p-i+1)}{i(i-1) \cdots 1}$$

porque  $p$  é primo. Portanto  $(a+1)^p \equiv a^p + 1 \pmod{p}$

$$(a+1)^p - (a+1) \equiv a^p + 1 - (a+1) = a^p - a \equiv 0.$$

(A última identidade é a hipótese da indução.) ■



**Definição 4.4**

Para  $a, b \in \mathbb{Z}$  denotamos com  $(a, b)$  o maior divisor em comum (MDC) de  $a$  e  $b$ . No caso  $(a, b) = 1$ ,  $a$  e  $b$  são *coprimo*.

**Teorema 4.10 (Divisão modulo  $p$ )**

Se  $p$  primo e  $(b, p) = 1$

$$ab \equiv cb \pmod{p} \Rightarrow a \equiv c \pmod{p}.$$

(Em palavras: Numa identidade modulo  $p$  podemos dividir por números coprimos com  $p$ .)

**Prova.**

$$\begin{aligned} ab \equiv cb &\iff \exists k \, ab + kp = cb \\ &\iff \exists k \, a + kp/b = c \end{aligned}$$

Como  $a, c \in \mathbb{Z}$ , temos  $kp/b \in \mathbb{Z}$  e  $b|k$  ou  $b|p$ . Mas  $(b, p) = 1$ , então  $b|k$ . Definindo  $k' := k/b$  temos  $\exists k' \, a + k'p = c$ , i.e.  $a \equiv c$ . ■

Logo, para  $p$  primo e  $(a, p) = 1$  (em particular se  $1 \leq a < p$ )

$$a^{p-1} \equiv 1 \pmod{p}. \quad (4.1)$$

Um teste melhor então é

```

1 Primo2(n) :=
2   seleciona a ∈ [1, n - 1] randômicamente
3   if (a, n) ≠ 1 return ‘‘Não’’
4   if a^{n-1} ≡ 1 return ‘‘Sim’’
5   return ‘‘Não’’
```

Complexidade: Uma multiplicação e divisão com  $\log n$  dígitos é possível em tempo  $O(\log^2 n)$ . Portanto, o primeiro teste (o algoritmo de Euclides em  $\log n$  passos) pode ser feito em tempo  $O(\log^3 n)$  e o segundo teste (exponenciação modular) é possível implementar com  $O(\log n)$  multiplicações (exercício!).

Corretude: O caso de uma resposta “Não” é certo, porque  $n$  não pode ser primo. Qual a probabilidade de falhar, i.e. do algoritmo responder “Sim”, com  $n$  composto? O problema é que o algoritmo falha no caso de *números Carmichael*.

**Definição 4.5**

Um número composto  $n$  que satisfaz  $a^{n-1} \equiv 1 \pmod{a}$  é um *número pseudo-primo com base  $a$* . Um *número Carmichael* é um número pseudo-primo para qualquer base  $a$  com  $(a, n) = 1$ .

#### 4. Algoritmos randomizados

Os primeiros números Carmichael são  $561 = 3 \times 11 \times 17$ , 1105 e 1729 (veja OEIS A002997). Existe um número infinito deles:

**Teorema 4.11 (Alford, Granville e Pomerance [2])**

Seja  $C(n)$  o número de números Carmichael até  $n$ . Assintoticamente temos  $C(n) > n^{2/7}$ .

**Exemplo 4.3**

$C(n)$  até  $10^{10}$  (OEIS A055553):

$n$	1	2	3	4	5	6	7	8	9	10	
$C(10^n)$	0	0	1	7	16	43	105	255	646	1547	$\diamond$
$\lceil (10^n)^{2/7} \rceil$	2	4	8	14	27	52	100	194	373	720	

Caso um número  $n$  não é primo, nem número de Carmichael, mais que  $n/2$  dos  $a \in [1, n-1]$  com  $(a, n) = 1$  não satisfazem (4.1) ou seja, com probabilidade  $> 1/2$  acharemos um testemunha que  $n$  é composto. O problema é que no caso de números Carmichael não temos garantia.

**Teorema 4.12 (Raiz modular)**

Para  $p$  primo temos

$$x^2 \equiv 1 \pmod{p} \Rightarrow x \equiv \pm 1 \pmod{p}.$$

O teste de Miller-Rabin usa essa característica para melhorar o teste acima. Podemos escrever  $n-1 = 2^t u$  para um  $u$  ímpar. Temos  $a^{n-1} = (a^u)^{2^t} \equiv 1$ . Portanto, se  $a^{n-1} \equiv 1$ ,

$$\text{Ou } a^u \equiv 1 \pmod{p} \text{ ou existe um menor } i \in [0, t] \text{ tal que } (a^u)^{2^i} \equiv 1$$

Caso  $p$  é primo,  $\sqrt{(a^u)^{2^i}} = (a^u)^{2^{i-1}} \equiv -1$  pelo teorema (4.12) e a minimalidade de  $i$  (que exclui os caso  $\equiv 1$ ). Por isso:

**Definição 4.6**

Um número  $n$  é um *pseudo-primo forte com base a* caso

$$\text{Ou } a^u \equiv 1 \pmod{p} \text{ ou existe um menor } i \in [0, t-1] \text{ tal que } (a^u)^{2^i} \equiv -1 \quad (4.2)$$

```

1  Primo3(n) :=
2    seleciona a ∈ [1, n-1] randômicamente
3    if (a, n) ≠ 1 return ‘‘Não’’
4    seja n-1 = 2tu
5    if au ≡ 1 return ‘‘Sim’’
6    if (au)2i ≡ -1 para um i ∈ [0, t-1] return ‘‘Sim’’
7    return ‘‘Não’’
```

**Teorema 4.13 (Monier [32], Rabin [35])**

Caso  $n$  é composto e ímpar, mais que  $3/4$  dos  $a \in [1, n-1]$  com  $(a, n) = 1$  não satisfazem o critério (4.2) acima.

Portanto com  $k$  testes randômicos, a probabilidade de falhar  $P[\text{Sim} \mid n \text{ composto}] \leq (1/4)^k = 2^{-2k}$ . Na prática a probabilidade é menor:

**Teorema 4.14 ([10])**

A probabilidade de um único teste falhar para um número com  $k$  bits e  $\leq k^2 4^{2-\sqrt{k}}$ .

**Exemplo 4.4**

Para  $n \in [2^{499}, 2^{500} - 1]$  a probabilidade de não detectar um  $n$  composto com um único teste é menor que

$$499^2 \times 4^{2-\sqrt{499}} \approx 2^{-22}.$$

◇

**Teste determinístico** O algoritmo pode ser convertido em um algoritmo determinístico, testando ao menos  $1/4$  dos  $a$  com  $(a, n) = 1$ . De fato, temos para menor o testemunha  $w(n)$  de um número  $n$  ser composto

$$\text{Se o HGR é verdade } w(n) < 2 \log^2 n \quad (4.3)$$

com HGR a hipótese generalizada de Riemann (uma conjectura aberta). Supondo HGR, obtemos um algoritmo determinístico com complexidade  $O(\log^5 n)$ . Em 2002, Agrawal, Kayal e Saxena [1] descobriram um algoritmo determinístico (sem a necessidade da HGR) em tempo  $\tilde{O}(\log^{12} n)$  que depois foi melhorado para  $\tilde{O}(\log^6 n)$ .

Para testar: [http://www.jjam.de/Java/Applets/Primzahlen/Miller\\_Rabin.html](http://www.jjam.de/Java/Applets/Primzahlen/Miller_Rabin.html).

## 4.5. Exercícios

**Exercício 4.1**

Encontre um primo  $p$  e um valor  $b$  tal que a identidade do teorema 4.10 não é correta.

**Exercício 4.2**

Encontre um número  $p$  não primo tal que a identidade do teorema 4.12 não é correta.



## 5. Complexidade e algoritmos parametrizados

A complexidade de um problema geralmente é resultado de diversos elementos. Um *algoritmo parameterizado* separa explicitamente os elementos que tornam um problema difícil, dos que são simples de tratar. A análise da *complexidade parameterizada* quantifica essas partes separadamente. Por isso, a complexidade parameterizada é chamada uma complexidade “de duas dimensões”.

### Exemplo 5.1

O problema de satisfatibilidade (SAT) é NP-completo, i.e. não conhecemos um algoritmo cuja complexidade cresce somente polinomialmente com o tamanho da entrada. Porém, a complexidade deste problema cresce principalmente com o número de variáveis, e não com o tamanho da entrada: com  $k$  variáveis e entrada de tamanho  $n$  solução trivial resolve o problema em tempo  $O(2^k n)$ . Em outras palavras, para *parâmetro*  $k$  fixo, a complexidade é linear no tamanho da entrada.  $\diamond$

### Definição 5.1

Um problema que possui um parâmetro  $k \in \mathbb{N}$  (que depende da instância) e permite um algoritmo de complexidade  $f(k)|x|^{O(1)}$  para entrada  $x$  e com  $f$  uma função arbitrária, se chama tratável por parâmetro fixo (ingl. fixed-parameter tractable, fpt). A classe de complexidade correspondente é FPT.

Um problema tratável por parâmetro fixo se torna tratável na prática, se o nosso interesse são instâncias com parâmetro pequeno. É importante observar que um problema permite diferentes parameterizações. O objetivo de projeto de algoritmos parameterizados consiste em descobrir para quais parâmetros que são pequenos ná prática o problema possui um algoritmo parameterizado. Neste sentido, o algoritmo parameterizado para SAT não é interessante, porque o número de variáveis na prática é grande.

A seguir consideramos o problema NP-complete de *cobertura por vértices*. Uma versão parameterizada é

#### K-COBERTURA POR VÉRTICES

**Instância** Um grafo não-direcionado  $G = (V, A)$  e um número  $k^1$ .

**Solução** Uma cobertura  $C$ , i.e. um conjunto  $C \subseteq V$  tal que  $\forall a \in A :$

## 5. Complexidade e algoritmos parametrizados

$$a \cap C \neq \emptyset.$$

**Parâmetro** O tamanho  $k$  da cobertura.

**Objetivo** Minimizar  $|C|$ .

Abordagem com força bruta:

```

1 mvc( $G = (V, A)$ ) :=
2   if  $A = \emptyset$  return  $\emptyset$ 
3   seleciona aresta  $\{u, v\} \in A$  não coberta
4    $C_1 := \{u\} \cup \text{mvc}(G \setminus \{u\})$ 
5    $C_2 := \{v\} \cup \text{mvc}(G \setminus \{v\})$ 
6   return a menor entre as coberturas  $C_1$  e  $C_2$ 

```

Supondo que a seleção de uma aresta e a redução dos grafos é possível em  $O(n)$ , a complexidade deste abordagem é dado pela recorrência

$$T_n = 2T_{n-1} + O(n)$$

com solução  $T_n = O(2^n)$ . Para achar uma solução com no máximo  $k$  vértices, podemos podar a árvore de busca definido pelo algoritmo mvc na profundidade  $k$ . Isso resulta em

### Teorema 5.1

O problema  $k$ -cobertura por vértices é tratável por parâmetro fixo em  $O(2^k n)$ .

**Prova.** Até o nível  $k$  vamos visitar  $O(2^k)$  vértices na árvore de busca, cada um com complexidade  $O(n)$ . ■

O projeto de algoritmos parameterizados frequentemente consiste em

- achar uma parameterização tal que o parte super-polinomial da complexidade é limitada para um parte do problema que depende de um parâmetro  $k$  que é pequeno na prática;
- encontrar o melhor algoritmo possível para o parte super-polinomial.

### Exemplo 5.2

Considere o algoritmo direto (via uma árvore de busca, ou backtracking) para SAT.

```

1 BT-SAT( $\varphi, \alpha$ ) :=
2   if  $\alpha$  é atribuição completa: return  $\varphi(\alpha)$ 
3   if alguma cláusula não é satisfeita: return false
4   if BT-SAT( $\varphi, \alpha 1$ ) return true
5   return BT SAT( $\varphi, \alpha 0$ )

```

---

<sup>1</sup>Introduzimos  $k$  na entrada, porque  $k$  mede uma característica da solução. Para evitar complexidades artificiais, entende-se que  $k$  nestes casos é codificado em *unário*.

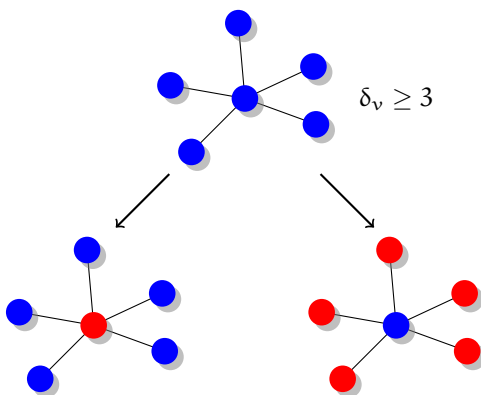


Figura 5.1.: Subproblemas geradas pela decisão da inclusão de um vértice  $v$ .  
Vermelho: vértices selecionadas para a cobertura.

( $\alpha 0$  e  $\alpha 1$  denotam extensões de uma atribuição parcial das variáveis.)

Aplicado para 3SAT, das 8 atribuições por cláusula podemos excluir uma que não a satisfaz. Portanto a complexidade de BT-SAT é  $O(7^{n/3}) = O(\sqrt[3]{7^n}) = O(1.9129^n)$ . (Exagerando – mas não mentindo – podemos dizer que isso é uma aceleração exponencial sobre a abordagem trivial que testa todas  $2^n$  atribuições.)

O melhor algoritmo para 3-SAT possui complexidade  $O(1.324^n)$ .  $\diamond$

**Um algoritmo melhor para cobertura por vértices** Consequência: O projeto cuidadoso de uma árvore de busca pode melhorar a complexidade. Vamos aplicar isso para o problema de cobertura por vértices.

Um melhor algoritmo para a  $k$ -cobertura por vértices pode ser obtido pelas seguintes observações

- Caso o grau máximo  $\Delta$  de  $G$  é 2, o problema pode ser resolvido em tempo  $O(n)$ , porque  $G$  é uma coleção de caminhos simples e ciclos.
- Caso contrário, temos pelo menos um vértice  $v$  de grau  $\delta_v \geq 3$ . Ou esse vértice faz parte da cobertura mínima, ou todos seus vizinhos  $N(v)$  (veja figura 5.1).

```

1  mvc'(G) :=
2    if  $\Delta(G) \leq 2$  then
3      determina a cobertura mínima C em tempo  $O(n)$ 
```

## 5. Complexidade e algoritmos parametrizados

```

4      return C
5  end if
6  seleciona um vértice  $v$  com grau  $\delta_v \geq 3$ 
7   $C_1 := \{v\} \cup \text{mvc}'(G \setminus \{v\})$ 
8   $C_2 := N(v) \cup \text{mvc}'(G \setminus N(v))$ 
9  return a menor cobertura entre  $C_1$  e  $C_2$ 

```

O algoritmo resolve o problema de cobertura por vértices mínima de forma exata. Se podamos a árvore de busca após selecionar  $k$  vértices obtemos um algoritmo parametrizado para  $k$ -cobertura por vértices. O número de vértices nessa árvore é

$$V_i \leq V_{i-1} + V_{i-4} + 1.$$

### Lema 5.1

A solução dessa recorrência é  $V_i = O(1.3803^i)$ .

### Teorema 5.2

O problema  $k$ -cobertura por vértices é tratável por parâmetro fixo em  $O(1.3803^k n)$ .

**Prova.** Considerações acima com trabalho limitado por  $O(n)$  por vértice na árvore de busca. ■

**Prova.** (Do lema acima.) Com o ansatz  $V_i \leq c^i$  obtemos uma prova por indução se para um  $i \geq i_0$

$$\begin{aligned}
 V_i &\leq V_{i-1} + V_{i-4} + 1 \leq c^{i-1} + c^{i-4} + 1 \leq c^i \\
 \iff c^{i-4}(c^4 - c^3 - 1) &\geq 1 \\
 \iff c^4 - c^3 - 1 &\geq 0
 \end{aligned}$$

(O último passo é justificado porque para  $c > 1$  e  $i_0$  suficientemente grande o produto vai ser  $\geq 1$ .)  $c^4 - c^3 - 1$  possui uma única raiz positiva  $\approx 1.32028$  e para  $c \geq 1.3803$  temos  $c^3 - c^2 - 1 \geq 0$ . ■



## A. Material auxiliar

### Definições

#### Definição A.1

Uma relação binária  $R$  é *polinomialmente limitada* se

$$\exists p \in \text{poly} : \forall (x, y) \in R : |y| \leq p(|x|)$$

#### Definição A.2 (Pisos e tetos)

Para  $x \in \mathbb{R}$  o *piso*  $\lfloor x \rfloor$  é o maior número inteiro menor que  $x$  e o *teto*  $\lceil x \rceil$  é o menor número inteiro maior que  $x$ . Formalmente

$$\lfloor x \rfloor = \max\{y \in \mathbb{Z} | y \leq x\}$$

$$\lceil x \rceil = \min\{y \in \mathbb{Z} | y \geq x\}$$

O *parte fracionário* de  $x$  é  $\{x\} = x - \lfloor x \rfloor$ .

Observe que o parte fracionário sempre é positivo, por exemplo  $\{-0.3\} = 0.7$ .

#### Proposição A.1 (Regras para pisos e tetos)

Pisos e tetos satisfazem

$$x \leq \lceil x \rceil < x + 1 \tag{A.1}$$

$$x - 1 < \lfloor x \rfloor \leq x \tag{A.2}$$

### A.1. Algoritmos

#### Soluções do problema da mochila com Programação Dinâmica

##### Mochila máxima (Knapsack)

- Seja  $S^*(k, v)$  a solução de tamanho menor entre todas soluções que
  - usam somente itens  $S \subseteq [1, k]$  e
  - tem valor exatamente  $v$ .
- Temos

$$S^*(k, 0) = \emptyset$$

$$S^*(1, v_1) = \{1\}$$

$$S^*(1, v) = \text{undef} \quad \text{para } v \neq v_1$$

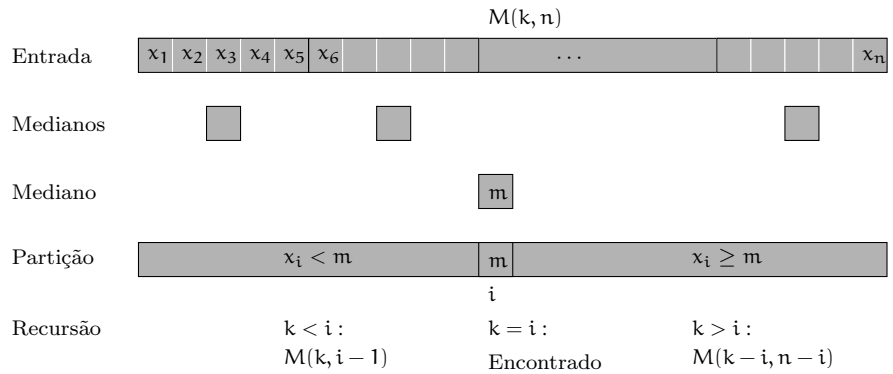


Figura A.1.: Funcionamento do algoritmo recursivo para seleção.

### Mochila máxima (Knapsack)

- $S^*$  obedece a recorrência

$$S^*(k, v) = \min_{\text{tamanho}} \begin{cases} S^*(k-1, v-v_k) \cup \{k\} & \text{se } v_k \leq v \text{ e } S^*(k-1, v-v_k) \text{ definido} \\ S^*(k-1, v) \end{cases}$$

- Menor tamanho entre os dois

$$\sum_{i \in S^*(k-1, v-v_k)} t_i + t_k \leq \sum_{i \in S^*(k-1, v)} t_i.$$

- Melhor valor: Escolhe  $S^*(n, v)$  com o valor máximo de  $v$  definido.
- Tempo e espaço:  $O(n \sum_i v_i)$ .

**Seleção** Dado um conjunto de números, o problema da seleção consiste em encontrar o  $k$ -ésimo maior elemento. Com ordenação o problema possui solução em tempo  $O(n \log n)$ . Mas existe um outro algoritmo mais eficiente. Podemos determinar o mediano de grupos de cinco elementos, e depois o recursivamente o mediano  $m$  desses medianos. Com isso, o algoritmo particiona o conjunto de números em um conjunto  $L$  de números menores que  $m$  e um conjunto  $R$  de números maiores que  $m$ . O mediano  $m$  é na posição  $i := |L| + 1$  desta sequência. Logo, caso  $i = k$   $m$  é o  $k$ -ésimo elemento. Caso  $i > k$  temos que procurar o  $k$ -ésimo elemento em  $L$ , caso  $i < k$  temos que procurar o  $k - i$ -ésimo elemento em  $R$  (ver figura A.1).

O algoritmo é eficiente, porque a seleção do elemento particionador  $m$  garante que o subproblema resolvido na segunda recursão é no máximo um fator  $7/10$  do problema original. Mais preciso, o número de medianos é maior que  $n/5$ , logo o número de medianos antes de  $m$  é maior que  $n/10 - 1$ , o número de elementos antes de  $m$  é maior que  $3n/10 - 3$  e com isso o número de elementos depois de  $m$  é menor que  $7n/10 + 3$ . Por um argumento similar, o número de elementos antes de  $m$  é também menor que  $7n/10 + 3$ . Portanto temos um custo no caso pessimista de

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 5 \\ T(\lceil n/5 \rceil) + \Theta(7n/10 + 3) + \Theta(n) & \text{caso contrário} \end{cases}$$

e com  $5^{-p} + (7/10)^p = 1$  temos  $p = \log_2 7 \approx 0.84$  e

$$\begin{aligned} T(n) &= \Theta\left(n^p \left(1 + \int_1^n u^{-p} du\right)\right) \\ &= \Theta(n^p (1 + (n^{1-p}/(1-p) - 1/(1-p)))) \\ &= \Theta(c_1 n^p + c_2 n) = \Theta(n). \end{aligned}$$

#### Algoritmo A.1 (Seleção)

**Entrada** Números  $x_1, \dots, x_n$ , posição  $k$ .

**Saída** O  $k$ -ésimo maior número.

```

1  S(k, {x1, ..., xn}) :=
2    if n ≤ 5
3      calcula e retorne o k-ésimo elemento
4    end if
5    mi := median(x5i+1, ..., xmin(5i+5, n)) para 0 ≤ i < ⌈n/5⌉.
6    m := S(⌈⌈n/5⌉/2⌋, m1, ..., m⌈n/5⌉-1)
7    L := {xi | xi < m, 1 ≤ i ≤ n}
8    R := {xi | xi ≥ m, 1 ≤ i ≤ n}
9    i := |L| + 1
10   if i = k then
11     return m
12   else if i > k then
13     return S(k, L)
14   else
15     return S(k - i, R)
16   end if
```

A. *Material auxiliar*

└──

## B. Técnicas para a análise de algoritmos

### Análise de recorrências

#### Teorema B.1 (Akra-Bazzi e Leighton)

Dado a recorrência

$$T(x) = \begin{cases} \Theta(1) & \text{se } x \leq x_0 \\ \sum_{1 \leq i \leq k} a_i T(b_i x + h_i(x)) + g(x) & \text{caso contrário} \end{cases}$$

com constantes  $a_i > 0$ ,  $0 < b_i < 1$  e funções  $g$ ,  $h$ , tal que

$$|g'(x)| \in O(x^c); \quad |h_i(x)| \leq x / \log^{1+\epsilon} x$$

para um  $\epsilon > 0$  e a constante  $x_0$  é suficientemente grande

$$T(x) \in \Theta \left( x^p \left( 1 + \int_1^x \frac{g(u)}{u^{p+1}} du \right) \right)$$

com  $p$  tal que  $\sum_{1 \leq i \leq k} a_i b_i^p = 1$ .

#### Teorema B.2 (Graham, Knuth e Patashnik [24])

Dado a recorrência

$$T(n) = \begin{cases} \Theta(1) & n \leq \max_{1 \leq i \leq k} d_i \\ \sum_i \alpha_i T(n - d_i) & \text{caso contrário} \end{cases}$$

seja  $\alpha$  a raiz com a maior valor absoluto com multiplicidade  $l$  do *polinômio característico*

$$z^d - \alpha_1 z^{d-d_1} - \dots - \alpha_k z^{d-d_k}$$

com  $d = \max_k d_k$ . Então

$$T(n) = \Theta(n^l \alpha^n) = \Theta^*(\alpha^n).$$



## Bibliografia

- [1] Manindra Agrawal, Neeraj Kayal e Nitin Saxena. “PRIMES is in P”. Em: *Annals of Mathematics* 160.2 (2004), pp. 781–793.
- [2] W. R. Alford, A. Granville e C. Pomerance. “There are infinitely many Carmichael numbers”. Em: *Annals Math.* 140 (1994).
- [3] H. Alt et al. “Computing a maximum cardinality matching in a bipartite graph in time  $O(n^{1.5}\sqrt{m \log n})$ ”. Em: *Information Processing Letters* 37 (1991), pp. 237–240.
- [4] Sanjeev Arora e Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [5] Claude Berge. “Two theorems in graph theory”. Em: *Proc. National Acad. Science* 43 (1957), pp. 842–844.
- [6] G. S. Brodal, R. Fagerberg e R. Jacob. *Cache Oblivious Search Trees via Binary Trees of Small Height*. Rel. téc. RS-01-36. BRICS, 2001.
- [7] Andrei Broder e Michael Mitzenmacher. “Network applications of Bloom filter: A survey”. Em: *Internet Mathematics* 1.4 (2003), pp. 485–509.
- [8] Bernhard Chazelle. “A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity”. Em: *Journal ACM* 47 (2000), pp. 1028–1047.
- [9] Thomas H. Cormen et al. *Introduction to Algorithms*. 3rd. The MIT Press, 2009.
- [10] Ivan Damgård, Peter Landrock e Carl Pomerance. “Average case error estimates for the strong probable prime test”. Em: *Mathematics of computation* 61.203 (1993), pp. 177–194.
- [11] Brian C. Dean, Michel X. Goemans e Nicole Immorlica. “Finite termination of ”augmenting path” algorithms in the presence of irrational problem data”. Em: *ESA ’06: Proceedings of the 14th conference on Annual European Symposium*. Zurich, Switzerland: Springer-Verlag, 2006, pp. 268–279. DOI: [http://dx.doi.org/10.1007/11841036\\_26](http://dx.doi.org/10.1007/11841036_26).
- [12] R. Dementiev et al. “Engineering a Sorted List Data Structure for 32 Bit Keys”. Em: *Workshop on Algorithm Engineering & Experiments*. 2004, pp. 142–151.

- [13] J. Edmonds. “Paths, Trees, and Flowers”. Em: *Canad. J. Math* 17 (1965), pp. 449–467.
- [14] J. Edmonds e R. Karp. “Theoretical improvements in algorithmic efficiency for network flow problems”. Em: *JACM* 19.2 (1972), pp. 248–264.
- [15] Jenő Egerváry. “Matrixok kombinatorius tulajdonságairól (On combinatorial properties of matrices)”. Em: *Matematikai és Fizikai Lapok* 38 (1931), pp. 16–28.
- [16] T. Feder e R. Motwani. “Clique partitions, graph compression and speeding-up algorithms”. Em: *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing (23rd STOC)*. 1991, pp. 123–133.
- [17] T. Feder e R. Motwani. “Clique partitions, graph compression and speeding-up algorithms”. Em: *Journal of Computer and System Sciences* 51 (1995), pp. 261–272.
- [18] L. R. Ford e D. R. Fulkerson. “Maximal flow through a network”. Em: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.
- [19] C. Fremuth-Paeger e D. Jungnickel. “Balanced network flows VIII: a revised theory of phase-ordered algorithms and the  $O(\sqrt{n}m \log(n^2/m)/\log n)$  bound for the nonbipartite cardinality matching problem”. Em: *Networks* 41 (2003), pp. 137–142.
- [20] Martin Fürer e Balaji Raghavachari. “Approximating the minimum-degree steiner tree to within one of optimal”. Em: *Journal of Algorithms* (1994).
- [21] H. N. Gabow. “Data structures for weighted matching and nearest common ancestors with linking”. Em: *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 434–443.
- [22] Ashish Goel, Michael Kapralov e Sanjeev Khanna. “Perfect Matchings in  $O(n \log n)$  Time in Regular Bipartite Graphs”. Em: *STOC 2010*. 2010.
- [23] A. V. Goldberg e A. V. Karzanov. “Maximum skew-symmetric flows and matchings”. Em: *Mathematical Programming A* 100 (2004), pp. 537–568.
- [24] Ronald Lewis Graham, Donald Ervin Knuth e Oren Patashnik. *Concrete Mathematics: a foundation for computer science*. Addison-Wesley, 1988.
- [25] J. Hadamard. “Sur la distribution des zéros de la fonction zeta(s) et ses conséquences arithmétiques”. Em: *Bull. Soc. math. France* 24 (1896), pp. 199–220.



- [26] Bernhard Haeupler, Siddharta Sen e Robert E. Tarjan. “Heaps simplified”. Em: (*Preprint*) (2009). arXiv:0903.0116.
- [27] J. E. Hopcroft e R. Karp. “An  $n^{5/2}$  algorithm for maximum matching in bipartite graphs”. Em: *SIAM J. Comput.* 2 (1973), pp. 225–231.
- [28] Michael J. Jones e James M. Rehg. *Statistical Color Models with Application to Skin Detection*. Rel. téc. CRL 98/11. Cambridge Research Laboratory, 1998.
- [29] John R. Black Jr. e Charles U. Martel. *Designing Fast Graph Data Structures: An Experimental Approach*. Rel. téc. University of California, Davis, 1998.
- [30] Haim Kaplan e Uri Zwick. “A simpler implementation and analysis of Chazelle’s soft heaps”. Em: *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*. New York, New York: Society for Industrial e Applied Mathematics, 2009, pp. 477–485.
- [31] H. W. Kuhn. “The Hungarian Method for the assignment problem”. Em: *Naval Research Logistic Quarterly* 2 (1955), pp. 83–97.
- [32] L. Monier. “Evaluation and comparison of two efficient probabilistic primality testing algorithms”. Em: *Theoret. Comp. Sci.* 12 (1980), pp. 97–108.
- [33] J. Munkres. “Algorithms for the assignment and transporation problems”. Em: *J. Soc. Indust. Appl. Math* 5.1 (1957), pp. 32–38.
- [34] Joon-Sang Park, Michael Penner e Viktor K Prasanna. “Optimizing Graph Algorithms for Improved Cache Performance”. Em: *IEEE Trans. on Parallel and Distributed Systems* 15.9 (2004).
- [35] Michael O. Rabin. “Probabilistic algorithm for primality testing”. Em: *J. Number Theory* 12 (1980), pp. 128–138.
- [36] Emma Roach e Vivien Pieper. “Die Welt in Zahlen”. Em: *Brand eins* 3 (2007).
- [37] J.R. Sack e J. Urrutia, eds. *Handbook of computational geometry*. Elsevier, 2000.
- [38] Alexander Schrijver. *Combinatorial optimization. Polyhedra and efficiency*. Vol. A. Springer, 2003.
- [39] Terrazon. *Soft Errors in Electronic Memory – A White Paper*. Rel. téc. Terrazon Semiconductor, 2004.
- [40] C.-J. de la Vallée Poussin. “Recherches analytiques la théorie des nombres premiers”. Em: *Ann. Soc. scient. Bruxelles* 20 (1896), pp. 183–256.

- [41] J. W. J. Williams. “Algorithm 232: Heapsort”. Em: *Comm. ACM* 7.6 (1964), pp. 347–348.
- [42] Norman Zadeh. “Theoretical Efficiency of the Edmonds-Karp Algorithm for Computing Maximal Flows”. Em: *J. ACM* 19.1 (1972), pp. 184–192.
- [43] Uri Zwick. “The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate”. Em: *Theoretical Computer Science* 148.1 (1995), pp. 165 –170. DOI: [DOI : 10 . 1016 / 0304 - 3975 \(95\) 00022 - 0](https://doi.org/10.1016/0304-3975(95)00022-0).

# Índice

$P \parallel C_{\max}$ , 108

árvore

    binomial, 10

    van Emde Boas, 28–37

árvore geradora mínima, 4

    algoritmo de Prim, 4

APX, 85

NPO, 84

PO, 84

Akra, Louay, 139

Akra-Bazzi

    método de, 139

algoritmo

$\epsilon$ -aproximativo, 85

$r$ -aproximativo, 85

    de aproximação, 83

    guloso, 85

    parameterizado, 131

    randomizado, 113

aproximação

    absoluta, 85

    relativa, 85

Bazzi, Mohamad, 139

bin packing

    empacotamento unidimensional,  
    101

Bloom, Burton Howard, 81

caminho

    alternante, 61

    mais curto, 4, 37

        algoritmo de Dijkstra, 4, 37

caminho mais gordo

    algoritmo de, 45–46

circulação, 38

cobertura por vértices, 86

    aproximação, 86

complexidade

    parameterizada, 131

cuco hashing, 79

dicionário, 73

Dijkstra

    algoritmo de, 4, 37

Dijkstra, Edsger Wybe, 4

Edmonds, Jack R., 43

Edmonds-Karp

    algoritmo de, 43–45

empacotamento unidimensional, 101

emparelhamento, 56

    de peso máximo, 56

    máximo, 56

    perfeito, 56

    de peso mínimo, 56

endereçamento aberto, 77

fila de prioridade, 4–37

    com lista ordenada, 5

    com vetor, 5

filtro de Bloom, 81

fluxo, 39

$s$ - $t$  máximo, 39

    com fontes e destinos múltiplos,  
    46

    de menor custo, 54

- formulação linear, 39
- Ford, Lester Randolph, 39
- Ford-Fulkerson
  - algoritmo de, 39–43
- Fulkerson, Delbert Ray, 39
- função de otimização, 83
- função hash, 73
  - com divisão, 75
  - com multiplicação, 75
  - universal, 76
- função objetivo, 83
- grafo residual, 41
- hashing
  - com endereçamento aberto, 77
  - com listas encadeadas, 73
  - cuco, 79
  - perfeito, 76
  - universal, 75
- heap, 4–37
  - binário, 7, 37
  - implementação, 10
  - binomial, 10, 23, 37
  - custo arnotizado, 14
  - Fibonacci, 15
  - rank-pairing, 19, 25
- Karp, Richard Manning, 43
- Knapsack, 88
- método de divisão, 75
- método de multiplicação, 75
- ordem
  - van Emde Boas, 29
- permutação, 77
- piso, 135
- Prim
  - algoritmo de, 4
- Prim, Robert Clay, 4
- problema
  - da mochila, 135
  - de avaliação, 84
  - de construção, 84
  - de decisão, 84
- problema da mochila, 88, 135
- problema de otimização, 83
- relação
  - polinomialmente limitada, 84
- SAT, 131
- satisfabilidade
  - de fórmulas booleanas, 131
- semi-árvore, 20
- sequenciamento
  - em processadores paralelos, 108
- teto, 135
- torneio, 19
- tratável por parâmetro fixo, 131
- uniforme, 77
- vértice
  - emparelhado, 61
  - livre, 61
- valor hash, 73
- van Emde Boas, Peter, 30
- vertex cover, 86
  - aproximação, 86
- Williams, J. W. J., 7