INF05010 - Algoritmos avançados
Notas de aula

Marcus Ritt
mrpritt@inf.ufrgs.br

24 de Agosto de 2010

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

Conteúdo

1 Algoritmos em grafos **3**
- 1.1 Filas de prioridade e heaps . . . 3
  - 1.1.1 Heaps binários . . . 6
  - 1.1.2 Heaps binomiais . . . 9
  - 1.1.3 Heaps Fibonacci . . . 13
  - 1.1.4 Rank-pairing heaps . . . 17
  - 1.1.5 Tópicos . . . 25
  - 1.1.6 Exercícios . . . 25
- 1.2 Fluxos em redes . . . 26
  - 1.2.1 Algoritmo de Ford-Fulkerson . . . 27
  - 1.2.2 Algoritmo de Edmonds-Karp . . . 31
  - 1.2.3 Variações do problema . . . 32
  - 1.2.4 Aplicações . . . 35
  - 1.2.5 Outros problemas de fluxo . . . 38
- 1.3 Emparelhamentos . . . 39
  - 1.3.1 Aplicações . . . 42
  - 1.3.2 Grafos bi-partidos . . . 42
  - 1.3.3 Exercícios . . . 50

2 Tabelas hash **51**
- 2.1 Hashing com listas encadeadas . . . 51
- 2.2 Hashing com endereçamento aberto . . . 54
- 2.3 Cuco hashing . . . 56
- 2.4 Filtros de Bloom . . . 57

3 Algoritmos de aproximação **61**
- 3.1 Aproximação para o problema da árvore de Steiner mínima . . . 61
- 3.2 Aproximações para o PCV . . . 63
- 3.3 Algoritmos de aproximação para cortes . . . 64
- 3.4 Exercícios . . . 68

4 Algoritmos randomizados **69**
- 4.1 Corte mínimo . . . 73
- 4.2 Teste de primalidade . . . 77

5 Complexidade e algoritmos parametrizados **83**

A Técnicas para a análise de algoritmos **87**

1 Algoritmos em grafos

1.1 Filas de prioridade e heaps

Uma fila de prioridade é uma estrutura de dados útil em vários aplicações. Exemplos são árvores geradores mínimas, caminhos mais curtos de um vértice para todos outros (algoritmo de Dijkstra) e 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' := {v0} para um v0 ∈ V
2  T := ∅
3  while V' ≠ V do
4     escolhe e = {u,v} com custo mínimo
5        entre V' e V \ V' (com u ∈ V', v ∈ V \ V')
6     V' := V' ∪ {v}
7     T := T ∪ {e}
8  end while
```

Algoritmo 1.2 (Prim refinado)
Implementação mais concreta:

```
1  T := ∅
2  for u ∈ V \ {v} do
3     if u ∈ N(v) then
4        value(u) := c_uv
5        pred(u) := v
```

```
6     else
7       value(u) := ∞
8     end if
9     insert(Q, (value(u), u)) { pares (chave, elemento) }
10  end for
11  while Q ≠ ∅ do
12    v := deletemin(Q)
13    T := T ∪ {pred(v)v}
14    for u ∈ N(v) do
15    if u ∈ Q e c_vu < value(u) then
16          value(u) := c_uv
17          pred(u) := v
18          update(Q, u, c_vu)
19        end if
20    end for
21  end while
```

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

$\Diamond$

Observação 1.1
Implementação com vetor de distâncias: insert $= O(1)$[1], deletemin $= O(n)$, update $= O(1)$, e temos custo $O(n + n^2 + m) = O(n^2 + m)$. Isso é assintoticamente ótimo para grafos densos, i.e. $m = \Omega(n^2)$. $\Diamond$

Observação 1.2
Implementação com lista ordenada: insert $= O(n)$, deletemin $= O(1)$, update $= O(n)$, e temos custo $O(n^2 + n + mn)$. $\Diamond$

Exemplo 1.2
Caminhos mínimos com o algoritmo de Dijsktra

Algoritmo 1.3 (Dijkstra)
Entrada Grafo não-direcionado $G = (V, E)$ com pesos $c_e, e \in E$ nas arestas, e um vértice $s \in V$.

[1] Com chaves compactas $[1, n]$.

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

```
1   d_s := 0; d_v := ∞, ∀v ∈ V \ {s}
2   visited(v) := false, ∀v ∈ V
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 d_u = ∞ then
11          d_u := d_v + d_vu
12          insert(Q, (u, d_u))
13        else
14          d_u := min(d_v + d_vu, d_u)
15          update(Q, (u, d_u))
16        end if
17      end if
18    end for
19  end while
```

A fila de prioridade contém pares de vértices e distâncias.

Proposição 1.1
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 real depende da implementação desses operações. ■

Proposição 1.2
O algoritmo de Dijkstra é correto.

Prova. Provaremos por indução que cada vértice v selecionado na linha 6 do algoritmo d_v é a distância mínima de s para 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

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

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

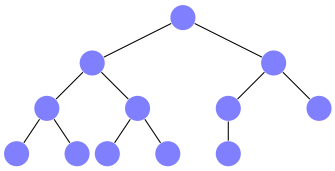
1.1.1 Heaps binários

Teorema 1.1 (Williams (1964))
Uma fila de prioridade pode ser implementado com custo insert $= O(\log n)$, deletemin $= O(\log n)$, 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.

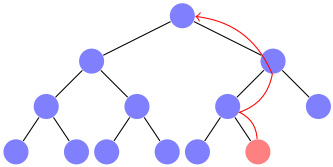
Um *heap* binário é um heap em que cada vértice possui no máximo dois filhos. Implementaremos uma fila de prioridade com um heap binário *completo*. Um heap completo fica organizado de forma que possui folhas somente no último nível, da esquerda para direita. Isso garante uma altura de $O(\log n)$.



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
```

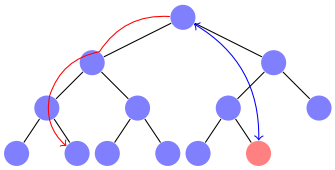


Lema 1.1
Seja T um min-heap. Decremente a chave do nó p. Após heapify-up(T, P) temos novamente um min-heap. A operação custa $O(\log n)$.

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 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,parent(p)) recursivamente. Podemos separar a troca em dois passos: (i) copia d para p. (ii) copia c para parent(p). Após passo (i) temos um min-heap T′ e passo (ii) diminui a chave de parent(p) e como a profundidade de 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 key(left(p))<key(p)) then swap(key(left(p)),key(p))
9     end if
10    { p possui dois filhos }
11    if key(p)>key(left(p)) or key(p)>key(right(p)) then
12      if (key(left(p))<key(right(p)) then
13        swap(key(left(p)),key(p))
14        heapify-down(H,left(p))
15      else
16        swap(key(right(p)),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 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 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 heapify-down não altera ele. Caso $c > d$ heapify-down troca c e d e chama 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 $O(1)$, heapify-up tem complexidade $O(\log n)$. ■

Última operação: atualizar a chave.

```
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

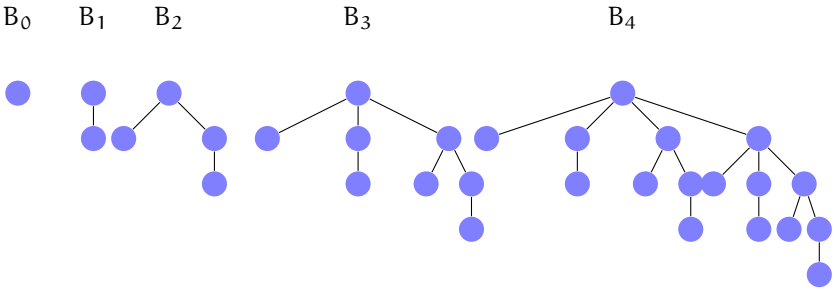
```
1  root(p) := return p = 0
2  pai(p) := return ⌊(p − 1)/2⌋
3  key(p) := return v[p]
4  left(p) := return 2p + 1
5  right(p) := return 2p + 2
6  numchildren(p) := return max(min(n − left(p), 2), 0)
```

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 $\mathsf{pos}[c]$ 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 $u \notin Q$.

1.1.2 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 vértice só. A árvore binomial B_i possui uma raiz com filhos $B_0, \ldots, B_{i-1}$. A *ordem* de B_k é k. Um heap binomial contém no máximo uma árvore binomial de cada ordem.

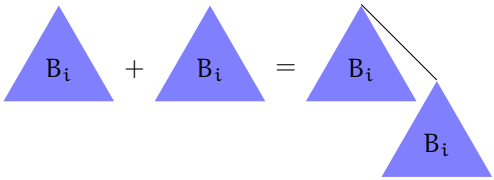


Lema 1.3
Uma árvore binomial tem as seguintes características:

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 1.3
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", e ela tem custo $O(1)$ (veja observações sobre a implementação).



◊

Observação 1.4
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 meld(h_1,h_2).
◊

Com a operação meld, podemos definir as seguintes operações:

- makeheap(c): Retorne um B_0 com chave c. Custo: $O(1)$.
- insert(h,c): meld(h,makeheap(c)). Custo: $O(\log n)$.
- getmin(h): Mantendo um link para a árvore com o menor custo: $O(1)$.
- 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. $B_0, \ldots, B_{k-1}$. Retorne meld(h_1,h_2). Custo: $O(\log n)$.

- updatekey(h,p): Como no caso do heap binário completo com custo $O(\log n)$.

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 séria 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 séria 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 "poupado". Incrementando, usaremos as operações "poupadas" para zerar bits. Desta forma, um incremento custa $O(1)$ e temos custo total $O(2^k)$.
Outra forma de análise amortizada, é usando uma *função potencial* φ, 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 e $c - \varphi(e_1) + \varphi(e_2)$, 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

$$\underbrace{11\cdots1}_{\text{p bits um}}\,0\,\underbrace{\cdots}_{\text{q bits um}}$$

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

$$\underbrace{00\cdots0}_{0}\,1\,\underbrace{\cdots}_{q}$$

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

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

$\Diamond$

Resumindo: Dado um série de operações com custos $c_1, \ldots, c_n$ o custo amortizado dessa operação é $\sum_{1 \le i \le n} c_i/n$. Se 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* φ que atribui cada estrutura de dados antes da operação i um valor não-negativo $\varphi_i \ge 0$ 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

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

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 $a_i \le F$, o custo amortizado desse tipo de operação é F.

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 $O(\log n)$ novas árvores, 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 $O(n \log n + m \log n)$.

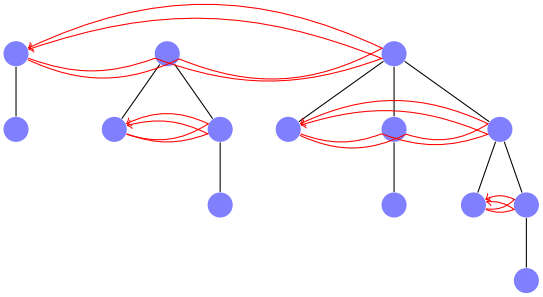
Meld preguiçosa Ao invés de reorganizar os dois heaps em um meld, podemos simplesmente concatená-los em tempo $O(1)$. Isso pode ser implementado sem custo adicional nas outras operações. A única operação que não tem complexidade $O(1)$ é deletemin. Agora temos uma coleção de árvores binomiais não necessariamente de ordem diferente. O deletemin reorganiza o heap, tal que obtemos um heap binomial com árvores de ordem única novamente. Para isso, mantemos um vetor com as árvores de cada ordem, inicialmente vazio. Seqüencialmente, 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 φ 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

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

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



1.1.3 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:

- 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 delete(h,p) como decreasekey(h,p,$-\infty$) e deletemin(h).
- decreasekey(h,p): A ordenação do heap pode ser violada. Corta p é 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 na ordem da árvore. A análise da complexidade de operações como deletemin depende desse fato para garantir que temos $O(\log n)$ árvores no heap. Conseqüência: Temos que garantir, que uma árvore não fica "podado"

demais. 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. Observe também que por isso não mantemos flag na raiz.

Implementações Denotamos com h um heap, c uma chave e p um elemento do heap. minroot(h) é o elemento do heap que correspondo com a raiz da chave mínima, e 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) := if key(minroot(h1))<key(minroot(h2)) h1 else h
14
15 decreasekey(h,p,c) :=
16   key(p):= c
17   if c < key(minRoot(h))
18     minRoot(h) := p
19   if not root(p)
20     if key(parent(p))>key(p)
21       corta p e adiciona na lista de raízes de h
22       cut(p) := false
23       cascading−cut(h,parent(p))
24
25 cascading−cut(h,p) :=
26   { p perdeu um filho }
27   if root(p)
28     return
```

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

Custo amortizado Para análise usaremos um potencial de $c_1 t + c_2 m$ 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, 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

$$O(n) - (c_1 t + c_2 m) + (c_1(t + n) + c_2(m - (n - 1))) = c_0 n - (c_2 - c_1)n + c_2$$

e com $c_2 - c_1 \geq c_0$ temos custo amortizado constante $c_2 = O(1)$.
A operação deletemin tem o custo real $O(M + t)$, com as seguintes contribuições

- Linhas 42–43: $O(M)$.
- Linhas 44–52: $O(M + t)$ 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 53–54: $O(M)$.

Seja m o número de marcas antes do deletemin e o número m' 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 ordem. Portanto, o potencial depois de deletemin e $\varphi' = c_1 t + c_2 m' \leq c_1 M + c_2 m$, e o custo amortizado é

$$\begin{aligned} O(M + t) - (c_1 t + c_2 m) + \varphi' &\leq O(M + t) - (c_1 t + c_2 m) + (c_1 M + c_2 m) \\ &= (c_0 + c_1)M + (c_0 - c_1)t \end{aligned}$$

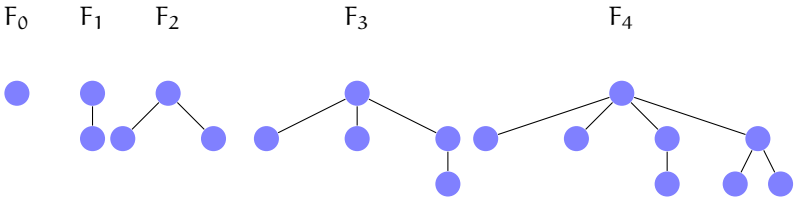
e com $c_1 \geq c_0$ temos custo amortizado $O(M)$.

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 ao menos $i - 2$ filhos.

Prova. No instante em que o filho i foi introduzido, p estava com ao menos $i-1$ filhos. Portanto i estava com ao menos $i-1$ filhos também. Depois filho i perdeu no máximo um filho, e portanto possui ao 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 ao 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 $k \geq 1$

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

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
A ordem máxima de um heap Fibonacci com n elementos é $O(\log n)$.

Sobre a implementação A implementação da árvore é a mesmo que no caso de binomial heaps. Uma vantagem do heap Fibonacci é que podemos usar os nós como ponteiros – lembre que a operação decreasekey precisa isso, 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.

1.1.4 Rank-pairing heaps

Haeupler et al. (2009) propõem um rank-pairing heap com as mesmas garantias de complexidade que um heap Fibonacci e uma implementação simplificada.

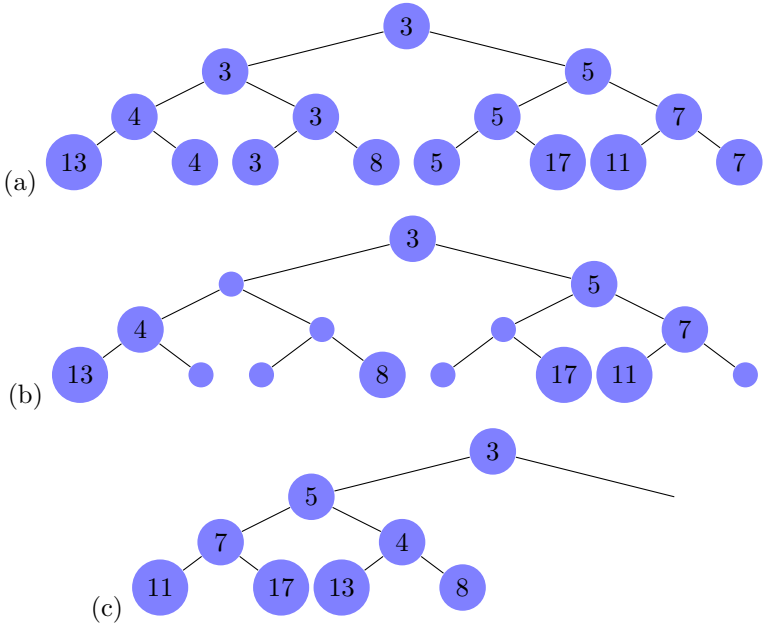


Figura 1.1: Representações de heaps.

Torneios Um *torneio* é uma forma alternativa de representar 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* e um filho *não-ordenado*. 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 rank. O rank de uma folha é 0. Uma comparação *justa* entre dois elementos do mesmo rank resulta num elemento com um rank por um maior no próximo nível . Numa comparação *injusta* entre dois elementos com ranks diferentes, o rank do vencedor é definido pelo maior dois ranks dos participantes (uma alternativa é que o rank não muda).

O rank de um elemento representa um limite inferior do número de elementos que perderam contra-lo:

Lema 1.6
Um torneio com campeão de rank k possui ao menos 2^k elementos.

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

Cada comparação injusta torna o limite inferior dado pelo rank menos preciso. Por isso uma regra na construção de torneios é fazer o maior número de comparações justas possíveis. Podemos implementar as operações de uma fila de prioridade (sem update ou decreasekey) como segue:

```
1  link(t1,t2) :=
2    if t1.c < t2.c then
3      return makechild(t1,t2)
4    else
5      return makechild(t2,t1)
6    end if
7
8  makechild(s,t) :=
9    t.u := s.o
10   s.o := t
11   setrank(t)
12   return s
13
14 setrank(t) :=
15   if t.o.r = t.u.r
16     t.r = t.o.r + 1
17   else
18     t.r = max(t.o.r,t.u.r)
19   end if
20
21 make-heap(c) := return c;
22 insert(h,c) := link(h,make-heap(c))
23 meld(h1,h2) := link(h1,h2)
24 getmin(h) := return h
25 deletemin(h) :=
```

```
26    aloca array r_0 ... r_{h.o.r+1}
27    t = h.o
28    while t not undefined do
29      t' := t.u
30      t.u := undefined
31      insert(t,r)
32      t := t'
33    end while
34    h' := undefined
35    for i = 0,...,h.o.r.+1 do
36      if r_i not undefined
37        h' := link(h',r_i)
38      end if
39    end for
40    return h'
41  end
42
43  insert(t,r) :=
44    if r_{t.o.r+1} is undefined then
45      r_{t.o.r+1} := t
46    else
47      t := link(t, r_{t.o.r+1})
48      r_{t.o.r+1} := undefined
49      insert(t,r)
50    end if
51  end
```

Observação 1.5
"insert" faz somente comparações justas. As comparações injustas ocorrem na construção da árvore final nas linhas 35–39. ◊

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 $O(\log n)$.

Lema 1.8
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 obtemos o torneio com k comparações injustas com r. Além disso r

participou em no máximo $\log n$ comparações justas pelo lema 1.6. Em soma vamos liberar no máximo $k + \log n$ árvores reduzindo o potencial por k, e com no máximo $k + \log n$ comparações 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 Ao invès de reconstruir uma única árvore que representa todo heap, podemos manter uma coleção de $O(\log n)$ árvores só permitir comparações justas. A estrutura de dados resultante é similar com os heaps binomiais: manteremos uma lista de raízes das árvores, junto com um ponteiro para a árvore com a raíz de menor valor:

```
1  insert(h,c) :=
2    insere make–heap(c) na lista de raizes
3    atualize a árvore mínima
4
5  meld(h1,h2) :=
6    concatena as lista 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 ... r_⌈log n⌉
3    remove a árvore mínima da lista de raizes
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 raizes }
18   for i = 0,...,⌈log n⌉ do
19     while |r_i| > 2
20       t := link(r_i.head, r_i.head.next)
21       insere t na lista h
```

```
22          remove r_i.head, r_i.head.next da lista r_i
23        end if
24      end for
25      return h
```

Observação 1.6
Continuando com "link"s justas até sobrar somente uma árvore de cada rank, obteremos um heap binomial. ◊

Lema 1.9
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 executa ao menos $(h - \log n)/2 - 1$ comparações justas, reduzindo o potencial por ao 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 ideía 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 ranks do heap que perde uma sub-árvore. Para tornar o ajuste dos ranks eficiente, vamos permitir uma folga nos ranks. Num heap binomial a diferença do rank de um elemento com o rank 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 rank 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 rank 1 e 1, 1 e 2 ou 0 e ao menos 2. (Figura 1.2.)
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
```

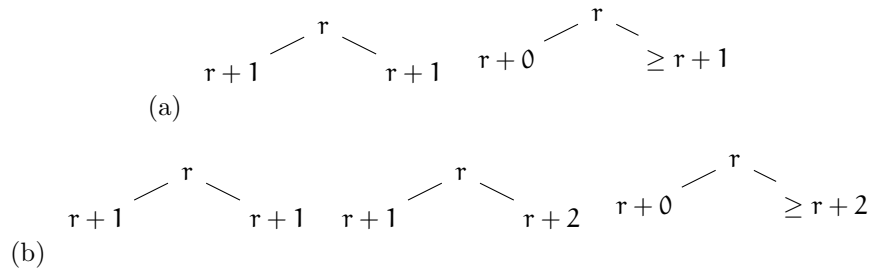


Figura 1.2: Diferenças no rank de rp-heaps do tipo 1 (a) e tipo 2 (b).

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

```
30        k = max(rank(u.o),rank(u.u))+1
31      end if
32      if u.r = k then
33        return
34      else
35        u.r := k
36        decreaserank(parent(u))
37
38  delete(h,e) :=
39    decreasekey(h,e,-∞)
40    deletemin(h)
    ]
```

Observação 1.7
A (suposta) eficiência do rp-heap vem do fato que o decreasekey altera os ranks do heap, e pouco da estrutura dele (como no caso do heap Fibonacci). ◊

Lema 1.10
Uma semi-árvore do tipo 2 com rank k contém ao menos ϕ^k elementos, sendo $\phi = (1+\sqrt{5})/2$ a razão áurea.

Prova. Por indução. Para folhas o lema é válida. Caso a raiz com rank 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. Pelas regras dos ranks de árvores de tipo dois, essas duas árvores possuem ranks $k-1$ 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 rank k satisfaz a recorrência

$$n_k = n_{k-1} + n_{k_2}$$

que é a recorrência dos números Fibonacci. ■

Lema 1.11
As operações "decreasekey" e "delete" possuem custo amortizado $O(1)$ e $O(\log n)$

Prova. Ver (Haeupler et al., 2009). ■

Resumo: Filas de prioridade

	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	$\mathbf{O(1)}$	$O(1)$	$O(\log n)$	$O(\log n)$	(update)	$O(\log n)$
Heap binomial(1)	$O(1)$	$O(1)$	$\mathbf{O(\log n)}$	$O(\log n)$	(update)	$O(\log n)$
Heap Fibonacci	$O(1)$	$O(1)$	$O(\log n)$	-	$\mathbf{O(1)}$	$O(\log n)$
rp-heap	$O(1)$	$O(1)$	$\mathbf{O(\log n)}$	-	$\mathbf{O(1)}$	$\mathbf{O(\log n)}$

Tabela 1.1: Complexidade das operações de uma fila de prioridade. Complexidades em negrito são amortizados. (1): meld preguiçoso.

1.1.5 Tópicos

O algoritmo (assintoticamente) mais rápido para árvores geradoras mínimas usa *soft heaps* é possui complexidade $O(m\alpha(m, n))$, com α a função inversa de Ackermann (Chazelle, 2000; Kaplan and Zwick, 2009).

1.1.6 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.

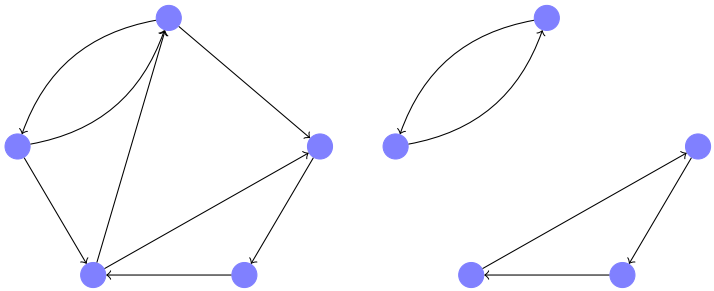


Figura 1.3: Grafo (esquerda) com circulação (direita)

1.2 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 saintes 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 $c : E \to \mathbb{R}$ nos arcos. Uma atribuição de fluxos aos arcos $f : E \to \mathbb{R}$ em G se chama *circulação*, se os fluxos respeitam os limites da capacidade ($f_e \leq c_e$) e satisfazem a conservação do fluxo

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

(ver Fig. 1.3).

Lema 1.12
Qualquer atribuição de fluxos f satisfaz $\sum_{v \in V} f(v) = 0$.

Prova.

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

■

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

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 1.13
Um fluxo s–t satisfaz $f(s) + f(t) = 0$.

Prova. Pelo lema 1.12 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

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

Observação 1.8
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. ◊

1.2.1 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 $P = (v_0 = s, v_1, \ldots, v_{n-1}, v_n = t)$, podemos aumentar o fluxo atual f um valor que corresponde ao "gargalo"

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

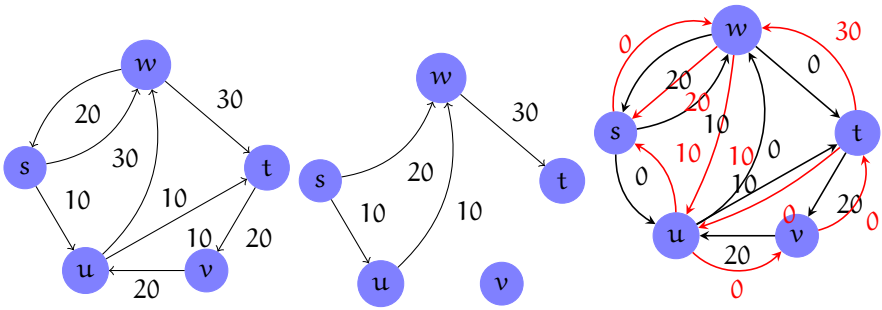


Figura 1.4: Esquerda: Grafo com capacidades. Centro: Fluxo com valor 30. Direita: O grafo residual correspondente.

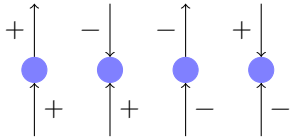


Figura 1.5: Manter a conservação do fluxo.

Observação 1.9
Repetidamente procurar um caminho com gargalo positivo e aumentar nem sempre produz um fluxo máximo. Na Fig. 1.4 o fluxo máximo possível é 40, obtido pelo aumentos 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. ◊

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 sainte, (ii) aumentar o fluxo num arco entrante, e diminuir num outro arco entrante, (iii) diminuir o fluxo num arco entrante e diminuir num arco sainte e (iv) diminuir o fluxo num arco entrante e aumentar num arco entrante (ver Fig. 1.5).
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 $f_e < c_e$.

- 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. 1.4 o grafo residual possui um caminho $P = (s, w, u, t)$ que aumenta o fluxo por 10. O algoritmo de Ford-Fulkerson (Ford and Fulkerson, 1956) 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 ∈ E: f_e := 0
2  while existe um caminho s—t em G_f do
3    Seja P um caminho s—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 1.14
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 1.15
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 1.16
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 *apertado* para um fluxo f, caso $f_e = c_e$.

Lema 1.17
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 1.18
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 1.19
Quando o algoritmo de Ford-Fulkerson termina, o valor do fluxo é máximo.

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 1.18, 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. 1.6). 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. 1.6). Usando uma busca por profundidade para achar caminhos s–t ele termina, mas é ineficiente (Dean et al., 2006).

1.2.2 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. selecione o caminho mais curto entre s e t. Nos temos (sem prova)

Teorema 1.3
O algoritmo de Edmonds-Karp precisa $O(nm)$ iterações, e portanto termina em $O(nm^2)$.

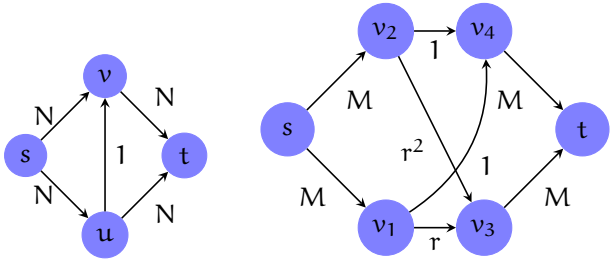


Figura 1.6: Esquerda: Pior caso para o algoritmo de Ford-Fulkerson com pesos inteiros aumentando o fluxo por $2N$ vezes por 1 nos caminhos (s,u,v,t) e (s,v,u,t). Direita: Menor grafo com pesos irracionais em que o algoritmo de Ford-Fulkerson falha (Zwick, 1995). $M \geq 3$, e $r = (1+\sqrt{1-4\lambda})/2$ 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.

1.2.3 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

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

O problema (1.3) pode ser reduzido para um problema de fluxo máximo simples em $G' = (V', E', c')$ (veja Fig. 1.7(a)) com

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

Lema 1.20
Se f' é solução máxima de (1.4), $f = f'|_E$ é uma solução máxima de (1.3).

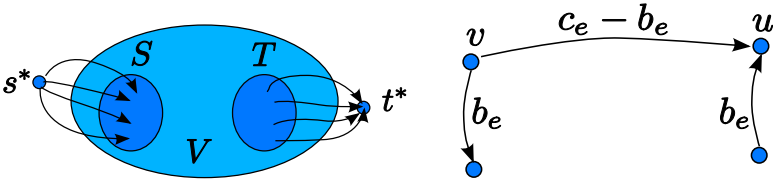


Figura 1.7: Reduções entre variações do problema do fluxo máximo. Esquerda: Fontes e destinos múltiplos. Direita: Limite inferior e superior para a capacidade de arcos.

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. ■

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

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

O problema (1.5) pode ser reduzido para um problema de fluxo máximo sim-

ples em $G' = (V', E', c')$ (veja Fig. 1.7(b)) com

$$\begin{aligned} V' &= V \\ E' &= E \cup \{(v,t) \mid (v,u) \in E\} \cup \{(s,u) \mid (v,u) \in E\} \\ c'_e &= \begin{cases} c_e - b_e & e \in E \\ b_{(v,u)} & e = (v,t) \\ b_{(v,u)} & e = (s,u) \end{cases} \end{aligned} \tag{1.6}$$

Lema 1.21
Problema (1.5) possui uma viável sse (1.6) possui uma solução máxima com todos arcos auxiliares $E' \setminus E$ apertados. Neste caso, se f é um fluxo máximo em (1.5),

$$f'_e = \begin{cases} f_e - b_e & e \in E \\ b_f & e = (v,t) \text{ criado por } f = (v,u) \\ b_f & e = (s,u) \text{ criado por } f = (v,u) \end{cases}$$

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

Prova. (Exercício.) ■

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

$$\begin{aligned} \textbf{existe} \quad & f \\ \textbf{s.a} \quad & f(v) = -d_v && \forall v \in V \\ & f_e \le c_e && e \in E. \end{aligned} \tag{1.7}$$

Evidentemente $\sum_{v \in V} d_v = 0$ é uma condição necessária (lema (1.12)). O problema (1.7) pode ser reduzido para um problema de fluxo máximo em $G' = (V', E')$ com

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

Lema 1.22
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.) ■

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

$$\begin{aligned} \textbf{existe} \quad & f \\ \textbf{s.a} \quad & f(v) = d_v && \forall v \in V \\ & b_e \le f_e \le c_e && e \in E. \end{aligned} \tag{1.9}$$

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

$$V' = V$$
$$E' = E \tag{1.10}$$
$$c_e = c_e - b_e$$
$$d'_v = d_v - \sum_{e\in\delta^-(v)} b_e + \sum_{e\in\delta^+(v)} b_e \tag{1.11}$$

Lema 1.23
O problema (1.9) possui solução sse problema (1.10) possui solução.

Prova. (Exercício.) ■

1.2.4 Aplicações

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

- Cada cliente i recebe ao mesmo 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 um 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. 1.9 (b). A desvantagem dessa abordagem é que a separação ignora o contexto de um pixel. Um pixel, "foreground" com todos pixel 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 partição do conjunto de todos pixels I em $A \dot{\cup} B$ tem um valor de

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

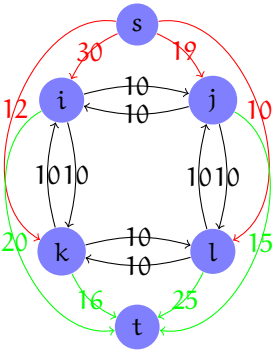
nesse modelo, e o nosso objetivo é achar uma partição que maximiza $q(A, B)$. Isso é equivalente a minimizar

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

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. 1.8).

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



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

Figura 1.8: Exemplo da construção para uma imagem 2×2. Direita: Tabela com valores pele/não-pele. Esquerda: Grafo com penalidade fixa $p_{ij} = 10$.

Figura 1.9: 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 $p = 5000$. A probabilidade de um pixel representar pele foi determinado conforme Jones and Rehg (1998).

4. São Paulo (GRU), 11.00 – Manaus (MAO), 14.00
5. Manaus (MAO), 14.15 – Belem (BEL), 15.15
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 acimas. 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.

1.2.5 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.

1.3 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: $M = \{u_1v_1, u_2v_2, \ldots\}$.) 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 $c : E \to \mathbb{Q}$, seja $c(M) = \sum_{e \in M} c_e$ o *valor* do emparelhamento M.

EMPARELHAMENTO MÁXIMO (EM)

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

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 $G = (V, E, c)$ não-direcionado com pesos $c : E \to \mathbb{Q}$ 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 $G = (V, E, c)$ não-direcionado com pesos $c : E \to \mathbb{Q}$ 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 1.4
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

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

Todo emparelhamento M em G de valor $c(M)$ define um emparelhamento perfeito M' em G' de valor $c'(M') = -c(M)$: M' consiste das arestas em M. Além disso, todo vértice não emparelhado em V será emparelhado com o novo vértice correspondente em M' com uma aresta de custo 0. Similarmente, os restantes novos vértices não emparelhados em V' são emparelhados em M' com arestas de custo 0 entre si. 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

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

A formulação do problema do emparelhamento máximo é

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

Observação 1.10
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. ◊

Observação 1.11
O dual da relaxação linear de (1.12) é

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

e o dual da relaxação linear de (1.13)

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

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 1.10 rende no caso não-ponderado:

Teorema 1.5 (Berge, 1951)
Em grafos bi-partidos o tamanho da menor cobertura por vértices é igual ao tamanho do emparelhamento máximo.

◊

1.3.1 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 trabalhar e da tarefa. Isso pode ser resolvido como problema de emparelhamento perfeito mínimo.

Particionamento de polígonos ortogonais

Teorema 1.6
(Sack and Urrutia, 2000, cap. 11,th. 1) Um polígono ortogonal com n vértices de reflexo (ingl. reflex vertex, i.e., com ângulo interno maior que π), 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 interseção. Um conjunto independente máximo é o complemento de uma cobertura por vértices mínima, o problema dual (1.15) de um emparelhamento máximo. Portanto, o tamanho de um emparelhamento máximo é igual $n - h$. Podemos obter o conjunto independente que procuramos usando "a metade" do emparelhamento (os vértices de uma parte só) e os vértices não emparelhados. 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?).

1.3.2 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.

Redução para o problema do fluxo máximo

Teorema 1.7
Um EM em grafos bi-partidos pode ser obtido em tempo $O(mn)$.

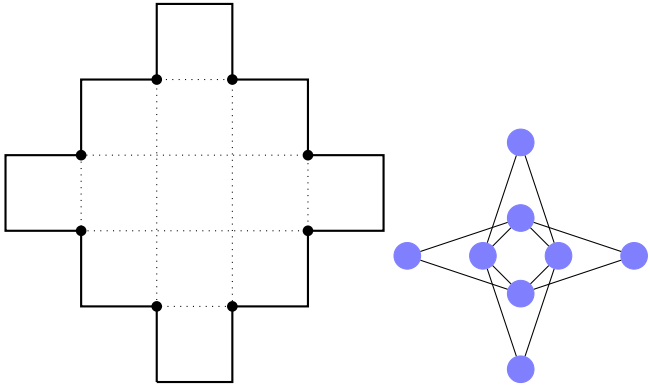


Figura 1.10: Esquerda: Polígono ortogonal com vértices de reflexo (pontos) e cordas (pontilhadas). Direita: grafo de intersecção.

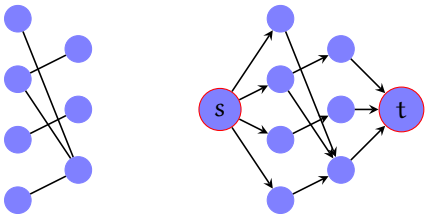


Figura 1.11: Redução do problema de emparelhamento máximo para o problema do fluxo máximo

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 1.8
O valor do fluxo máximo é igual a cardinalidade de um emparelhamento máximo.

Prova. Dado um emparelhamento máximo $M = \{v_{11}v_{21}, \ldots, v_{1n}v_{2n}\}$, 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 (1.14)). 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 $v_iv_{i+1} \in M$ 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 1.12
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 1.9 (Hopcroft and Karp (1973))
Seja M^* um emparelhamento máximo e M um emparelhamento arbitrário. O conjunto $M \oplus M^*$ contém ao menos $k = |M^*| - |M|$ caminhos M-aumentandos distintos. Um deles possui comprimento menor que $|V|/k - 1$.
Prova. Considere os componentes de G em relação aos arcos $M := M \oplus M^*$. Cada vértice possui no máximo grau 2. Portanto, cada componente é ou um vértice livre, ou um caminhos simples ou um ciclo. Os caminhos e ciclos possuem alternadamente arcos de M e M^*. Portanto os ciclos tem comprimento par. Os caminhos de comprimento impar são ou M-aumentandos ou M^*-aumentandos, mas o segundo caso é impossível, porque M^* é máximo. Agora

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

e portanto $M \oplus M^*$ contém k arcos mais de M^* que de M. Isso mostra que existem ao menos $|M^*| - |M|$ caminhos M-aumentandos, porque somente os caminhos de comprimento impar possuem exatamente um arco mais de M^*. Ao 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 1.2 (Berge (1957))
Um emparelhamento é máximo sse não existe um caminho M-aumentando.

Rascunho de um algoritmo:

Algoritmo 1.5 (Emparelhamento máximo)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Um emparelhamento máximo M.

```
1  M = ∅
2  while (existe um caminho M–aumentando P) do
3     M := M ⊕ P
4  end while
5  return M
```

Problema: como achar caminhos M-aumentandos de forma eficiente?

Observação 1.13
Um caminho M-aumentando 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 $O(m)$. ◊

Teorema 1.10
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 $O(n)$. ■

Observação 1.14
O último teorema é o mesmo que teorema (1.7). ◊

Observação 1.15
Pelo teorema (1.9) sabemos que em geral 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 aumentandos disjuntos e aplica-los para obter um novo emparelhamento. Observe que pelo teorema (1.9) 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 aumentando. (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 $O(m)$. ◊

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

Lema 1.24
Após cada fase, o comprimento de um caminho aumentando mínimo aumenta ao menos dois.

Lema 1.25
O algoritmo termina em no máximo $\sqrt{n}$ fases.

Teorema 1.11
O problema do emparelhamento máximo não-ponderado em grafos bi-partidos pode ser resolvido em tempo $O(m\sqrt{n})$.
Prova. Pelas lemas 1.24 e 1.25 e a observação que toda fase pode ser completada em $O(m)$. ■

Usaremos outro lema para provar os dois lemas acima.

Lema 1.26
Seja M um emparelhamento, P um caminho M-aumentando mínimo, e Q um caminho $M \oplus P$-aumentando. 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 náo possuem vértices em comum, Q é M-aumentando, $P \cap Q = \emptyset$ e a desigualdade é conseqüência da minimalidade de P.
Caso contrário: $P \oplus Q$ consiste em dois caminhos, e eventualmente um coleção de ciclos. Os dois caminhos são M-aumentandos, pelas seguintes observações:

1. O início e termino de P é livre em M, porque P é M-aumentando.

2. O início e termino de Q é livre em M: eles não pertencem a P, porque são livres em M'.

3. Nenhum outro vértice de P ou Q é livre em relação a M: P só contém dois vértices livres e Q só contém dois vértices livres em Q mas não em P.

4. Temos dois caminhos M-aumentandos, começando com um vértice livre em Q e terminando com um vértice livre em P. O caminho em $Q \setminus P$ é M-alternante, porque as arestas livres em M' são exatamente as arestas livres em M. O caminho Q entra em P sempre após uma aresta livre em M, porque o primeiro vértice em P já é emparelhado em M e sai de P sempre antes de uma aresta livre em M, porque o último vértice em P já é emparelhado. Portanto os dois caminhos em $P \oplus Q$ são M-aumentandos.

Os dois caminhos M-aumentandos 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 1.24). Seja S o conjunto de caminhos M-aumentandos da fase anterior, e P um caminho aumentando. Caso P é disjunto de todos caminhos em S, ele deve ser mais comprido, porque S é um conjunto máximo de caminhos aumentandos. 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 1.26. ■

Prova. (do lema 1.25). Seja M^* um emparelhamento máximo e M o emparelhamento obtido após de $\sqrt{n}/2$ fases. O comprimento de qualquer caminho M-aumentando é no mínimo $\sqrt{n}$, pelo lema 1.24. Pelo teorema 1.9 existem ao menos $|M^*| - |M|$ caminhos M-aumentandos disjuntos. Mas então $|M^*| - |M| \leq \sqrt{n}$, porque no caso contrário eles possuem mais que n vértices em total. Como o emparelhamento cresce ao menos um em cada fase, o algoritmo executar 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. Para subclasses de grafos bipartidos existem algoritmos melhores. Por exemplo, existe um algoritmo randomizado para grafos bipartidos regulares com complexidade de tempo esperado $O(n \log n)$ (Goel et al., 2010).

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 1.15: ele consiste em uma 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[2] 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
12       for all uv ∈ E, uv ∉ M do
13         if not m_v then
14           H_uv := true
15           U_2 := U_2 ∪ v
16         end if
17       end for
18     end for
19
20     { determina vizinhos em U_1 via arestas emparelhadas }
21     found := false          { ao menos um caminho encontrado? }
22     U_1 := ∅
23     for all u ∈ U_2 do
24       m_u := true
25       if (u livre) then
26         found := true
27       else
```

[2] H, porque o DAG se chama *árvore Hungariano* na literatura.

	Cardinalidade	Ponderado
Bi-partido	$O(n\sqrt{\frac{mn}{\log n}})$ (Alt et al., 1991) $O(m\sqrt{n}\frac{\log(n^2/m)}{\log n})$ (Feder and Motwani, 1995)	$O(nm + n^2 \log n)$ (Kuhn, 1955; Munkres, 1957)
Geral	$O(m\sqrt{n}\frac{\log(n^2/m)}{\log n})$ (Goldberg and Karzanov, 2004; Fremuth-Paeger and Jungnickel, 2003)	$O(n^3)$ (Edmonds, 1965) $O(mn + n^2 \log n)$ (Gabow, 1990)

Tabela 1.2: Resumo emparelhamentos

```
28            v := mate[u]
29            if not m_v then
30              H_uv := true
31              U_1 := U_1 ∪ 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 extraimos um caminho alternante que termina em v como segue:

```
1   extract_path(v) :=
2     P := v
3     while not (v ∈ V_1 and v livre) do
4       if v ∈ V_1
5         v := mate[v]
6       else
7         v := escolhe {u | H_uv, uv ∉ M}
8       end if
9       P := vP
10    end while
11
12    remove o caminho e todos vértices sem predecessor
13    end while
14  end
```

1.3.3 Exercícios

Exercício 1.5
É possível somar uma constante $c \in \mathbb{R}$ para todos custos de uma instância do EPM ou EPPM, mantendo a otimalidade da solução?

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 $c \in U$: insert(c,H)
- Deleção de uma chave $c \in U$: delete(c,H)
- Teste da pertinência: Chave $c \in H$? 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 um alternativa para outros 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.
- Invento outro método de *resolução de colisões*.

2.1 Hashing com listas encadeadas

Define uma função hash $h : U \to [m]$. Mantemos uma coleção de m listas $l_0, \ldots, l_{m-1}$ 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*:

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

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 $n_i = \sum_{1 \le j \le n} c_{ji}$ e com isso

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

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

```
1  insert(c,H) :=
2    insert(c,l_h(c))
3
4  lookup(c,H) :=
5    lookup(c,l_h(c))
6
7  delete(c,H) :=
8    delete(c,l_h(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 α. 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 $c_1, \ldots, c_n$ a seqüencia em que a chaves foram inseridas. Supondo que inserimos a chaves no inicio da lista, $E[p]$ é um mais 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. $E[X_{ij}] = \Pr[h(c_i) = h(c_j)] = \sum_{1 \le k \le m} \Pr[h(c_i) = k] \Pr[h(c_j) = k] = 1/m$. Para a chave c_i, seja p_i a posição dela na sua lista. Temos

$$E[p_i] = E[1 + \sum_{j<i} X_{ij}] = 1 + \sum_{j<i} E[X_{ij}] = 1 + (i-1)/m$$

e para uma chave aleatória c

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

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, $h(i) = i \mod m$ 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 impares é impar, 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 tem 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 $U \to [m]$ é *universal* se

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

ou equivalente

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

para qualquer par de chaves c_1, c_2.

Teorema 2.3
Se escolhemos uma função hash $h \in \mathcal{H}$ uniformemente, para uma chave c arbitrário o tamanho esperado de $l_{h(c)}$ é

- α, 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 $c = (c_0, \ldots, c_r)_m$ uma chave na base m, escolhe $a = (a_0, \ldots, a_r)_m$ randomicamente e define

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

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 $h(c, 0), \ldots, h(c, m-1)$ 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)] = textfree
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

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

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

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

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
A função lookup precisa no máximo $1/\alpha \ln 1/(1-\alpha)$ testes caso a chave está na tabela com $\alpha < 1$, e cada chave tem a mesma probabilidade de ser procurada.

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

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

e com $H_m - H_{m-n} \le \ln(m) - \ln(m-n)$ 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 nenhumas ou poucas deleções.

Funções hash para endereçamento aberto

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

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
```

Para inserir uma chave, temos que resolver o problema de que as duas posições candidatas sejam ocupadas. A solução do cuco hashing é comportar-se como um cuco com ovos de outras aves: jogá-los fora do seu "ninho": insert ocupa a posição de uma das duas chaves. A chave "jogada fora" tem que ser 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 entrar 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
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)
```

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 $h_1, \ldots, h_k$.

```
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
```

Apos de inserir todas n chaves, a probabilidade que um dado bit é ainda 0 é

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

que é igual ao valor esperado da fração de bits não setados. Sendo ρ 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 ρ é com alta probabilidade perto do seu valor esperado (Broder and Mitzenmacher, 2003). Broder and Mitzenmacher (2003) 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 filtro 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 mando os resultados para A e A filtra os elementos errados.

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)	$O(1/\alpha \ln 1/(1-\alpha))$	$O(1/\alpha \ln 1/(1-\alpha))$	-
Cucko	$\Theta(\mathbf{1})$	$\Theta(1)$	$\Theta(1)$

3 Algoritmos de aproximação

(As notas seguem Vazirani (2001).)
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 ótimo e $\mathrm{OPT}(x) = \varphi(x, y^*)$ para o valor da solução ótima. Lembramos que uma *aproximação absoluta* garante que $D(x, y) = |\mathrm{OPT}(x) - \varphi(x, y)| \leq D$ para uma constante D e todo x, enquanto uma *aproximação relativa* garante que o *erro relativo* $E(x, y) = D(x, y)/\max\{\mathrm{OPT}(x), \varphi(x, y)\} \leq E$ para uma constante E e todos x.

Definição 3.1
Uma *redução preservando a aproximação* entre dois problemas de minimização Π_1 e Π_2 consiste em um par de funções f e g (computáveis em tempo polinomial) tal que para instância x_1 de Π_1, $x_2 := f(x_1)$ é instância de Π_2 com

$$\mathrm{OPT}_{\Pi_2}(x_2) \leq \mathrm{OPT}_{\Pi_1}(x_1) \tag{3.1}$$

e para uma solução y_2 de Π_2 temos uma solução $y_1 := g(x_1, y_2)$ de Π_1 com

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

Uma redução preservando a aproximação fornece uma α-aproximação para Π_1 dada uma α-aproximação para Π_2, porque

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

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

3.1 Aproximação para 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

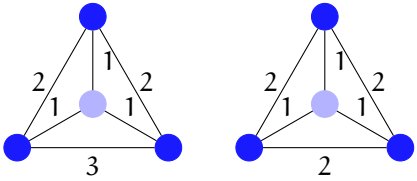


Figura 3.1: Grafo com fecho métrico.

subgrafo conexo mínimo que inclui um dado conjunto de *vértices necessários* $R \subseteq V$. 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 3.1
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. ◊

Definição 3.2
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 3.1
Existe um 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 $c'_{ij} \leq c_{ij}$ é 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'. ■
Consequência: Para o problema do ASM é suficiente considerar o caso métrico.

Teorema 3.2
O AGM sobre R é uma 2-aproximação para o problema do ASM.

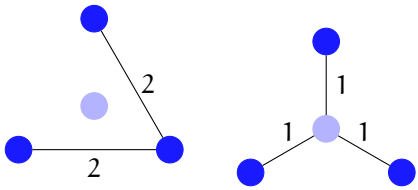


Figura 3.2: AGM sobre R e melhor solução. ●: vértice em R, ●: vértice Steiner.

Prova. Considere a solução ótima S^* de ASM. Duplica todas arestas[1] 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. ■

3.2 Aproximações para o PCV

Teorema 3.3
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

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

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.

[1] Esse transformação torna G em um multigrafo.

2. Duplica todas arestas de A.
3. Acha um caminho Euleriano nesse grafo.
4. Remove vértices duplicados.

Teorema 3.4
O algoritmo acima define uma 2-aproximação.

Prova. A melhor solução do PCV menos uma aresta é uma árvore geradora de G. Portanto $c(A) \leq \mathrm{OPT}$. A solução S obtida pelo algoritmo acima satisfaz $c(S) \leq 2c(A)$ e portanto $c(S) \leq 2\mathrm{OPT}$, pelo mesmo argumento da prova do teorema 3.2. ■

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 impar (por quê?), 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 3.5
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 $\mathrm{OPT}/2$: 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 impares no ciclo. Com E_1 o emparelhamento de menor custo, temos

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

e portanto

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

■

3.3 Algoritmos de aproximação para cortes

Seja $G = (V, A, c)$ um grafo conectado com com pesos c nas arestas. Lembramos que um corte C é um conjunto de arestas que separa o grafo em duas partições $S \dot{\cup} V \setminus S$. 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:

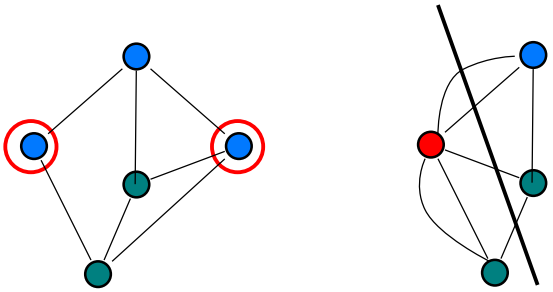


Figura 3.3: Identificação de dois terminais e um corte no grafo reduzido. Vértices em verde, terminais em azul. A grafo reduzido possui múltiplas arestas entre vértices.

- Corte múltiplo mínimo (CMM): Dado terminais $s_1, \ldots, s_k$ determine o menor corte C que separa todos terminas.
- 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 3.1 (CI)
Entrada Grafo $G = (V, A, c)$ e terminais $s_1, \ldots, s_k$.

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.

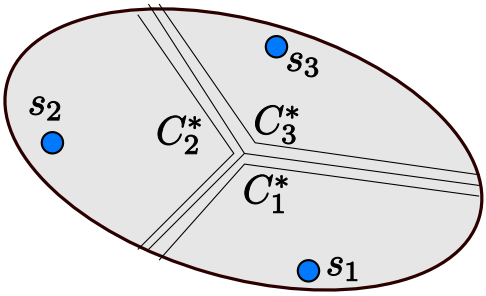


Figura 3.4: Corte múltiplo e decomposição em cortes isolantes.

2. Remove o maior desses cortes e retorne a união dos restantes.

Teorema 3.6
Algoritmo 3.1 é uma $2-2/k$-aproximação.

Prova. Considere o corte mínimo C^*. Ele pode ser representado com a união de k cortes que separam os k componentes individualmente:

$$C^* = \bigcup_{1 \le i \le k} C_i^*.$$

(Veja fig. 3.4.) Cada aresta de C^* faz parte das cortes das duas componentes adjacentes, e portanto

$$\sum_{1 \le i \le k} w(C_i^*) = 2w(C^*)$$

e ainda $w(C_i) \le w(C_i^*)$ para os cortes C_i do algoritmo 3.1, porque nos usamos o corte isolante mínimo de cada componente. Logo para o corte C retornado pelo algoritmo temos

$$w(C) \le (1-1/k) \sum_{1 \le i \le k} w(C_i) \le (1-1/k) \sum_{1 \le i \le k} w(C_i^*) \le 2(1-1/k)w(C^*).$$

■

A análise do algoritmo é ótimo, como o seguinte 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.

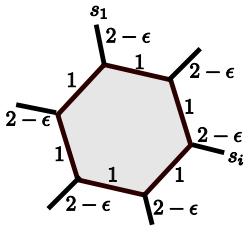


Figura 3.5: Exemplo de um grafo em que o algoritmo 3.1 retorne uma $2-2/k$-aproximação.

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.

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 com $n-1$ cortes s–t mínimos.

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

Algoritmo 3.2 (KCM)
Entrada Grafo $G = (V, A, c)$.

Saida 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 3.7
Algoritmo 3.2 é uma $2-2/k$-aproximação.

Prova. Seja $C^* = \bigcup_{1 \leq i \leq k} C_i^*$ uma corte mínimo, decomposto igual à prova anterior. O nosso objetivo e 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 $V_1, \ldots, V_k$: 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 $V_1, \ldots, V_{k-1}$ 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

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

■

3.4 Exercícios

Exercício 3.1
Por que um subgrafo de menor custo sempre é uma árvore?

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.

- Probabilidade morrer caindo da cama: $1/2 \times 10^6$ (Roach and Pieper, 2007).
- 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. 6×10^{-7} erros por segundo num memória de 256 MB.[1]
- 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 $100\,m^2$).

Portanto, um algoritmo que retorna uma resposta falsa com baixa probabilidade é aceitável. Em retorno um algoritmo randomizado em geral é

- mais simples;
- mais eficiente: para alguns problemas, o algoritmos randômica é o mais eficiente conhecido;
- maios robusto: algoritmos randômicos podem ser menos dependente da distribuição das entradas.
- a única alternativa: para alguns problemas, conhecemos só algoritmos randômicos.

Classes de complexidade

Definição 4.1
Seja Σ 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

[1] 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 (Semiconductor, 2004).

- $x \in L \Rightarrow \Pr[A(x) = \text{sim}] \geq \alpha$.
- $x \notin L \Rightarrow \Pr[A(x) = \text{não}] \geq \beta$.

(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 $co - RP = R(1, 1/2)$ consiste dos problemas com erro no lado de "não";
- a classe $ZPP := RP \cap co - RP$ (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 polinomios)
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 $O(n)$. 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) ≠ 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 $\mathsf{co-RP}$. ◊

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 RP, $\mathsf{co-RP}$ e BPP. Isso não se aplica a PP: é possível que ϵ diminui exponencialmente com o tamanho da instância. Um exemplo de amplificação de probabilidade encontra-se na prova do teorema 4.4.

Relação entre as classes

Teorema 4.1
$\mathsf{RP} \subseteq \mathsf{NP}$.

Prova. Supõe que temos um algoritmo em RP para algum problema L. Podemos, não-deterministicamente, gerar todas seqüencias 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". ■

Teorema 4.2
Uma caraterização alternativa da classe ZPP é como classe de problemas tal que existe um algoritmo A

- que responde ou "sim", ou "não" ou "não sei",
- com $\Pr[A(x) = \text{não sei}] \leq 1/2$, e
- caso ele responde, ele não erra, i.e., para x tal que $A(x) \neq$ não sei temos $A(x) = 1 \iff x \in L$.

Prova. Para $L \in \mathsf{ZPP}$ temos dois algoritmos $A_1 \in \mathsf{RP}$ e $A_2 \in \mathsf{co-RP}$. Vamos construir um algoritmo

```
1  if A1(x) = não e A2(x) = não then
2    return ''não''
3  else if A1(x) = não e A2(x) = sim then
```

```
4    return ''não sei''
5  else if A1(x) = sim e A2(x) = não then
6    { caso impossível }
7  else if A1(x) = sim e A2(x) = sim then
8    return ''sim''
9  end if
```

O algoritmo responde corretamente "sim" e "não", porque um dos dois algoritmos não erra. Qual a probabilidade do segundo caso? Para $x \in L$, $\Pr[A_1(x)\text{não} \wedge A_2(x) = \text{sim}] \leq 1/2 \times 1 = 1/2$. Similarmente, para $x \notin L$, $\Pr[A_1(x)\text{não} \wedge A_2(x) = \text{sim}] \leq 1 \times 1/2 = 1/2$. ■

Teorema 4.3
$\mathsf{ZPP} \subseteq \mathsf{RP}$ e $\mathsf{ZPP} \subseteq \mathsf{co}-\mathsf{RP}$.

Prova. Seja A um algoritmo para $L \in \mathsf{ZPP}$. Constrói outro algoritmo que sempre responde "não" caso A responde "não sei", e senão responde igual. No caso de $\mathsf{co}-\mathsf{RP}$ analogamente constrói um algoritmos que responde "sim" nos casos "não sei" de A. ■

Teorema 4.4
$\mathsf{RP} \subseteq \mathsf{BPP}$ e $\mathsf{co}-\mathsf{RP} \subseteq \mathsf{BPP}$.

Prova. Seja A um algoritmo para $L \in \mathsf{RP}$. Constrói um algoritmo A'

```
1  if A(x) = não e A(x) = não then
2    return ''não''
3  else
4    return ''sim''
5  end if
```

Caso $x \notin L$, $\Pr[A'(x) = \text{não}] = \Pr[A(x) = \text{não} \wedge A(x) = \text{não}] = 1 \times 1 = 1$. 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 $\mathsf{co}-\mathsf{RP}$ 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-determinísticamente podemos "chutar" uma solução

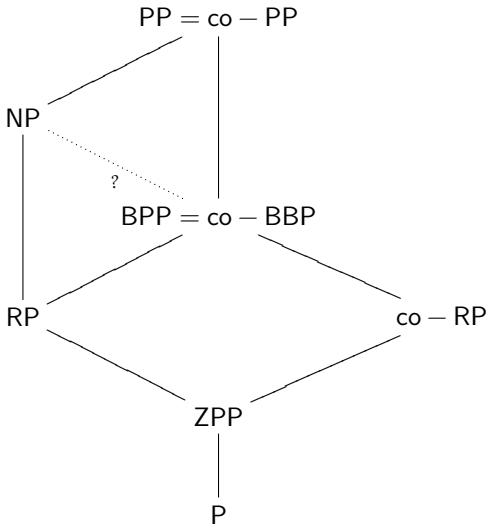


Figura 4.1: Relações entre classes de complexidade para algoritmos randomizados.

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 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.

4.1 Corte mínimo

Corte mínimo

Entrada Grafo não-direcionado $G = (V, A)$ com pesos $c : A \to \mathbb{Z}_+$ 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 contraímo-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     escolhe 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
TBD ◊

Dizemos que uma aresta "sobrevive" uma contração, caso ele não foi contraído.

Lema 4.1
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 ao 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

$$\prod_{0\le i<n-t} 1-\frac{2}{n-i} = \prod_{0\le i<n-t} \frac{n-i-2}{n-i}$$
$$= \frac{(n-2)(n-3)\cdots t-1}{n(n-1)\cdots t+1} = \binom{t}{2}/\binom{n}{2} = \Omega((t/n)^2).$$

■

Teorema 4.5
Para um dado corte mínimo de tamanho k, a probabilidade do algoritmo acima retornar esse corte 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 4.1
O que acontece se repetirmos o algoritmo algumas vezes? Seja C_i a variável indicador que na repetição i o corte mínimo foi encontrado. Temos $P[C_i = 1] \ge 2n^{-2}$ e portanto $P[C_i = 0] \le 1 - 2n^{-2}$. Para kn^2 repetições, vamos encontrar $C = \sum C_i$ cortes mínimos com probabilidade

$$P[C \ge 1] = 1 - P[C = 0] \ge 1 - (1-2n^{-2})^{kn^2} \ge 1 - e^{-2k}.$$

Para $k = \log n$ obtemos $P[C \ge 1] \ge 1 - n^{-2}$. ◊

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.

```
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 4.2
A probabilidade de um corte mínimo sobreviver $t = \lceil 1 + n/\sqrt{2} \rceil$ contrações é no mínimo $1/2$.

Prova. Pelo lema 4.1 a probabilidade é

$$\frac{\lceil 1 + n/sqrt2 \rceil \, (\lceil 1 + n/sqrt2 \rceil - 1)}{n(n-1)} \geq \frac{(1 + n/\sqrt{2})(n/\sqrt{2})}{n(n-1)} = \frac{\sqrt{2} + n}{2(n-1)} \geq \frac{n}{2n} = \frac{1}{2}.$$

■

Seja $P(t)$ a probabilidade que um corte com k arestas sobrevive caso o grafo possui t vértices. Temos

$$P[\text{o corte sobrevive em } H_1] \geq 1/2P(\lceil 1 + t/\sqrt{2} \rceil)$$

$$P[\text{o corte sobrevive em } H_2] \geq 1/2P(\lceil 1 + t/\sqrt{2} \rceil)$$

$$P[\text{o corte não sobrevive em } H_1 \text{ e } H_2] \leq (1 - 1/2P(\lceil 1 + t/\sqrt{2} \rceil))^2$$

$$P(t) = P[\text{o corte sobrevive em } H_1 \text{ ou } H_2] \geq 1 - (1 - 1/2P(\lceil 1 + t/\sqrt{2} \rceil))^2$$

$$= P(\lceil 1 + t\sqrt{2} \rceil) - 1/4P(\lceil 1 + t/\sqrt{2} \rceil)^2$$

Para resolver essa recorrência, define $Q(k) = P(\sqrt{2}^k)$ com base $Q(0) = 1$ para obter a recorrência simplificada

$$\begin{aligned} Q(k+1) = P(\sqrt{2}^{k+1}) &= P(\left\lceil 1 + \sqrt{2}^k \right\rceil) - 1/4 P(\left\lceil 1 + \sqrt{2} \right\rceil^2)^2 \\ &\approx P(\sqrt{2}^k) - P(\sqrt{2}^k)^2/4 = Q(k) - Q(k)^2/4 \end{aligned}$$

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 de cmr2 observe que temos $O(\log n)$ níveis recursivos e cada contração pode ser feito em tempo $O(n^2)$, portanto

$$T_n = 2T(\left\lceil 1 + n/\sqrt{2} \right\rceil) + O(n^2).$$

Aplicando o teorema de Akra-Bazzi obtemos a equação característica $2(1/\sqrt{2})^p = 1$ com solução $p = 2$ e

$$T_n \in \Theta(n^2(1 + \int_1^n \frac{cu^2}{u^3} du)) = \Theta(n^2 \log n).$$

4.2 Teste de primalidade

Um problema importante na criptografia é achar números primos grandes (p.ex. RSA). Escolhendo um número n randômico, qual a probabilidade de n ser primo?

Teorema 4.6 (Hadamard (1896), de la Vallée Poussin (1896))
(Teorema dos números primos.)
Para $\pi(n) = |\{p \leq n \mid p \text{ primo}\}|$ temos

$$\lim_{n\to\infty} \frac{\pi(n)}{n/\ln n} = 1.$$

(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 achar 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  Prim1(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 4.7 (Fermat, Euler)
Para p primo e $a \geq 0$ temos

$$a^p \equiv a \mod p.$$

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 $(a+1)^p \equiv a^p + 1$ e

$$(a+1)^p - (a+1) \equiv a^p + 1 - (a+1) = a^p - a \equiv 0.$$

(A última identidade é a hipótese da indução.) ■

Definição 4.2
Para $a, b \in \mathbb{Z}$ denotamos com (a, b) o máximo divisor em comum (MDC) de a e b. No caso $(a, b) = 1$, a e b são *coprimo*.

Teorema 4.8 (Divisão modulo p)
Se p primo e $(b, p) = 1$

$$ab \equiv cb \mod p \Rightarrow a \equiv c \mod p.$$

(Em palavras: Numa identidade modulo p podemos dividir por números coprimos com p.)

Prova.

$$\begin{aligned} ab \equiv cd &\iff \exists k\, ab + kp = cb \\ &\iff \exists k\, a + kp/b = c \end{aligned}$$

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 $1 \leq a < p$)

$$a^{p-1} \equiv 1 \mod p. \tag{4.1}$$

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^(p−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 4.3
Um número composto n que satisfaz $a^{n-1} \equiv 1 \mod a$ é um *número pseudo-primo com base* a. Um *número Carmichael* é um número pseudo-primo para qualquer base a com $(a, n) = 1$.

Os primeiros números Carmichael são $561 = 3 \times 11 \times 17$, 1105 e 1729 (veja OEIS A002997). Existe um número infinito deles:

Teorema 4.9 (Alford et al. (1994))
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
$\lceil (10^n)^{2/7} \rceil$	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 4.10
Para p primo temos

$$x^2 \equiv 1 \mod p \Rightarrow x \equiv \pm 1 \mod p.$$

O teste de Miller-Rabin usa essa característica para melhorar o teste acima. Podemos escrever $n - 1 = 2^t u$ para um u impar. Temos $a^{n-1} = (a^u)^{2^t} \equiv 1$. Portanto, se $a^{n-1} \equiv 1$,

$$\text{Ou } a^u \equiv 1 \mod p \text{ ou existe um menor } i \in [0, t] \text{ tal que } (a^u)^{2^i} \equiv 1$$

Caso p é primo, $\sqrt{(a^u)^{2^i}} = (a^u)^{2^{i-1}} \equiv -1$ pelo teorema (4.10). Por isso:

Definição 4.4
Um número n é um *pseudo-primo forte com base* a caso

$$\text{Ou } a^u \equiv 1 \mod p \text{ ou existe um menor } i \in [0, t-1] \text{ tal que } (a^u)^{2^i} \equiv -1 \tag{4.2}$$

```
1  Primo3(n) :=
2    seleciona a ∈ [1, n−1] randômicamente
3    if (a, n) ≠ 1 return ''Não''
4    seja n − 1 = 2^t u
5    if a^u ≡ 1 return ''Sim''
6    if (a^u)^{2^i} ≡ −1 para um i ∈ [0, t−1] return ''Sim''
7    return ''Não''
```

Teorema 4.11 (Monier (1980),Rabin (1980))
Caso n e composto e impar, 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 4.12 (Damgård et al. (1993))
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

$$499^2 \times 4^{2-\sqrt{499}} \approx 2^{-22}.$$

$\Diamond$

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

$$\text{Se o HGR é verdade } w(n) < 2\log^2 n \tag{4.3}$$

com HGR a hipótese generalizada de Riemann (uma conjetura aberta). Supondo HGR, obtemos um algoritmo determinístico com complexidade $O(\log^5 n)$. Em 2002, Agrawal et al. (2004) 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.

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. ◊

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 $\forall a \in A : a \cap C \neq \emptyset$.

Parâmetro O tamanho k da cobertura.

Objetivo Minimizar $|C|$.

Abordagem com força bruta:

```
1  mvc(G = (V, A)) :=
2     if A = ∅ return ∅
3     seleciona aresta {u, v} ∈ A não coberta
4     C1 := {u} ∪ mvc(G \ {u})
5     C2 := {v} ∪ mvc(G \ {v})
6     return a menor entre as coberturas C1 e C2
```

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 poder 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] Introduzimos k na entrada, porque k mede uma característica da solução. Para evitar conmplexidades artificias, entende-se que k nestes casos é codificado em *unário*.

```
1  BT–SAT(φ, α) :=
2     if α é atribuição completa: return φ(α)
3     if alguma claúsula não é satisfeita: return false
4     if BT–SAT(φ, α1) return true
5     return BT–SAT(φ, α0)
```

($\alpha 0$ e $\alpha 1$ denotam extensões de uma atribuição parcial das variáveis.)
Aplicado para 3SAT, das 8 atribuições por claúsula 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 3SAT possui complexidade $O(1.324^n)$. ◊

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 Δ 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 ao 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 Δ(G) ≤ 2 then
3        determina a cobertura mínima C em tempo O(n)
4        return C
5     end if
6     seleciona um vértice v com grau δ_v ≥ 3
7     C1 := {v} ∪ mvc'(G \ {v})
8     C2 := N(v) ∪ mvc'(G \ N(v))
9     return a menor cobertura entre C1 e C2
```

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 parameterizado para k-cobertura por vértices. O número de vértices nessa árvore é

$$V_i = V_{i-1} + V_{i-3} + 1.$$

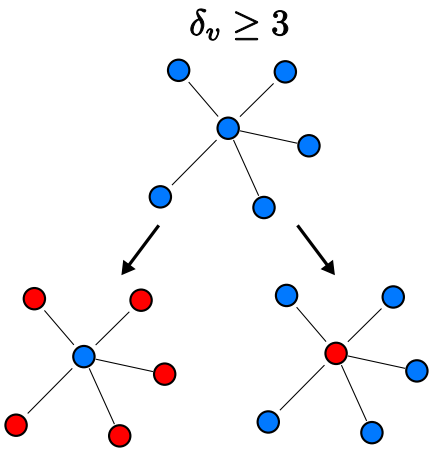


Figura 5.1: Subproblemas geradas pela decisão da inclusão de um vértice v. Vermelho: vértices selecionadas para a cobertura.

Lema 5.1
A solução dessa recorrência é $V_i = O(1.4656^i)$.

Teorema 5.2
O problema k-cobertura por vértices é tratável por parâmetro fixo em $O(1.4656^k n)$.

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 $i \geq i_0$

$$\begin{aligned} & V_i = V_{i-1} + V_{i-3} + 1 \leq c^{i-1} + c^{i-3} + 1 \leq c^i \\ \iff \quad & c^{i-3}(c^3 - c^2 - 1) \geq 1 \\ \iff \quad & c^3 - c^2 - 1 \geq 0 \end{aligned}$$

(O último passo é justificado porque para $c > 1$ e i_0 suficientemente grande o produto vai ser ≥ 1.) $c^3 - c^2 - 1$ possui uma única raiz positiva ≈ 1.4656 e para $c \geq 1.4656$ temos $c^3 - c^2 - 1 \geq 0$. ■

A Técnicas para a análise de algoritmos

Análise de recorrências

Teorema A.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

$$|g'(x)| \in O(x^c); \qquad |h_i(x)| \leq x/\log^{1+\epsilon} x$$

para um $\epsilon > 0$ e a constante x_0 e 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 A.2 (Graham et al. (1988))
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 α a raiz com a maior valor absoluto com multiplicidade l do *polinômio característico*

$$z^d - \alpha_1 z^{d-d_1} - \cdots - \alpha_k z^{d-d_k}$$

com $d = \max_k d_k$. Então

$$T(n) = \Theta(n^l \alpha^n) = \Theta^*(\alpha^n).$$

Bibliografia

Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

W. R. Alford, A. Granville, and C. Pomerance. There are infinitely many Carmichael numbers. *Annals Math.*, 140, 1994.

H. Alt, N. Blum, K. Mehlhorn, and M. Paul. Computing a maximum cardinality matching in a bipartite graph in time $o(n^{1.5}\sqrt{m \log n}$. *Information Processing Letters*, 37:237–240, 1991.

Claude Berge. Two theorems in graph theory. *Proc. National Acad. Science*, 43:842–844, 1957.

Andrei Broder and Michael Mitzenmacher. Network applications of bloom filter: A survey. *Internet Mathematics*, 1(4):485–509, 2003.

Bernhard Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal ACM*, 47:1028–1047, 2000.

Ivan Damgård, Peter Landrock, and Carl Pomerance. Average case error estimates for the strong probable prime test. *Mathematics of computation*, 61(203):177–194, 1993.

C.-J. de la Vallée Poussin. Recherches analytiques la thÃ©orie des nombres premiers. *Ann. Soc. scient. Bruxelles*, 20:183–256, 1896.

Brian C. Dean, Michel X. Goemans, and Nicole Immorlica. Finite termination of "augmenting path"algorithms in the presence of irrational problem data. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 268–279, London, UK, 2006. Springer-Verlag. doi: http://dx.doi.org/10.1007/11841036_26.

J. Edmonds. Paths, trees, and flowers. *Canad. J. Math*, 17:449–467, 1965.

T. Feder and R. Motwani. Clique partitions, graph compression and speeding-up algorithms. *Journal of Computer and System Sciences*, 51:261–272, 1995.

L. R. Ford and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

C. Fremuth-Paeger and 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. *Networks*, 41:137–142, 2003.

H. N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 434–443, 1990.

Ashish Goel, Michael Kapralov, and Sanjeev Khanna. Perfect matchings in $o(n \log n)$ time in regular bipartite graphs. In *STOC 2010*, 2010.

A. V. Goldberg and A. V. Karzanov. Maximum skew-symmetric flows and matchings. *Mathematical Programming A*, 100:537–568, 2004.

Ronald Lewis Graham, Donald Ervin Knuth, and Oren Patashnik. *Concrete Mathematics: a foundation for computer science.* Addison-Wesley, 1988.

J. Hadamard. Sur la distribution des zéros de la fonction zeta(s) et ses conséquences arithmétiques. *Bull. Soc. math. France*, 24:199–220, 1896.

Bernhard Haeupler, Siddharta Sen, and Robert E. Tarjan. Heaps simplified. *(Preprint)*, 2009. arXiv:0903.0116.

J. E. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Comput.*, 2:225–231, 1973.

Michael J. Jones and James M. Rehg. Statistical color models with application to skin detection. Technical Report CRL 98/11, Cambridge Research Laboratory, 1998.

Haim Kaplan and Uri Zwick. A simpler implementation and analysis of Chazelle's soft heaps. In *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*, pages 477–485, Philadelphia, PA, USA, 2009. Society for Industrial and Applied Mathematics.

H. W. Kuhn. The hungarian method for the assignment problem. *Naval Re4search Logistic Quarterly*, 2:83–97, 1955.

L. Monier. Evaluation and comparison of two efficient probabilistic primality testing algorithms. *Theoret. Comp. Sci.*, 12:97–108, 1980.

J. Munkres. Algorithms for the assignment and transporation problems. *J. Soc. Indust. Appl. Math*, 5(1):32–38, 1957.

Michael O. Rabin. Probabilistic algorithm for primality testing. *J. Number Theory*, 12:128–138, 1980.

Emma Roach and Vivien Pieper. Die Welt in Zahlen. *Brand eins*, 3, 2007.

J.R. Sack and J. Urrutia, editors. *Handbook of computational geometry*. Elsevier, 2000.

Terrazon Semiconductor. Soft errors in electronic memory. Whitepaper, 2004.

Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.

J. W. J. Williams. Algorithm 232: Heapsort. *Comm. ACM*, 7(6):347–348, 1964.

Uri Zwick. The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate. *Theoretical Computer Science*, 148(1):165 – 170, 1995. doi: DOI:10.1016/0304-3975(95)00022-O.

Índice

alternante, 33

cuco hashing, 44

dicionário, 38

emparelhado, 33
emparelhamento, 29
    perfeito, 29
endereçamento aberto, 42

função hash, 38

grafo residual, 20

livre, 33

método de divisão, 40
método de multiplicação, 40

perfeito, 29
permutação, 42

uniforme, 42

valor hash, 39