Algoritmos e complexidade
Notas de aula

Marcus Ritt

com contribuições de
Edson Prestes e Luciana Buriol

19 de Setembro de 2019

Conteúdo

Essas notas servem como suplemento à material do livro "Complexidade de algoritmos" de Toscani/Veloso e o material didático da disciplina "Complexidade de algoritmos" da UFRGS.

Versão 10619 do 2019-09-19, compilada em 19 de Setembro de 2019.

Parte I.

Análise de algoritmos

1. Introdução e conceitos básicos

A teoria da computação começou com a pergunta "Quais problemas são *efetivamente* computáveis?" e foi estudada por matemáticos como Post, Church, Kleene e Turing. Intuitivamente, computadores diferentes, por exemplo um PC ou um Mac, possuem o mesmo poder computacional. Mas é possível que um outro tipo de máquina é mais poderosa que as conhecidas? Uma máquina, cujos programas nem podem ser implementadas num PC ou Mac? Não é fácil responder essa pergunta, porque a resposta depende das possibilidades computacionais do nosso universo, e logo do nosso conhecimento da física. Matemáticos definiram diversos modelos de computação, entre eles o cálculo lambda, as funções parcialmente recursivas, a máquina de Turing e a máquina de RAM, e provaram que todos são (polinomialmente) equivalentes em poder computacional, e são considerados como máquinas universais.
Nossa pergunta é mais específica: "Quais problemas são *eficientemente* computáveis?". Essa pergunta é motivada pela observação de que alguns problemas que, mesmo sendo efetivamente computáveis, são tão complicados, que a solução deles para instâncias do nosso interesse é impraticável.

Exemplo 1.1
Não existe um algoritmo que decide o seguinte: Dado um programa arbitrário (que podemos imaginar escrito em qualquer linguagem de programação como C ou Miranda) e as entradas desse programa. Ele termina? Esse problema é conhecido como "problema de parada". ♢

Visão geral

- Objetivo: Estudar a análise e o projeto de algoritmos.
- Parte 1: Análise de algoritmos, i.e. o estudo teórico do desempenho e uso de recursos.
- Ela é pré-requisito para o projeto de algoritmos.
- Parte 2: As principais técnicas para projetar algoritmos.

Introdução

- Um algoritmo é um procedimento que consiste em um conjunto de regras não ambíguas as quais especificam, para cada entrada, uma sequência finita de operações, terminando com uma saída correspondente.
- Um algoritmo resolve um problema quando, para qualquer entrada, produz uma resposta correta, se forem concedidos tempo e memória suficientes para a sua execução.

Motivação

- Na teoria da computação perguntamos "Quais problemas são efetivamente computáveis?"
- No projeto de algoritmos, a pergunta é mais específica: "Quais problemas são eficientemente computáveis?"
- Para responder, temos que saber o que "eficiente" significa.
- Uma definição razoável é considerar algoritmos em tempo polinomial como eficiente (tese de Cobham-Edmonds).

Custos de algoritmos

- Também temos que definir qual tipo de custo interessa.
- Uma execução tem vários custos associados:

  Tempo de execução, uso de espaço (cache, memória, disco), energia consumida, energia dissipada, ...
- Existem características e medidas que são importantes em contextos diferentes

  Linhas de código fonte (LOC), legibilidade, manutenabilidade, corretude, custo de implementação, robustez, extensibilidade,...
- A medida mais importante: tempo de execução.
- A complexidade pode ser vista como uma propriedade do problema

Mesmo um problema sendo computável, não significa que existe um algoritmo que vale a pena aplicar. O problema

---

EXPRESSÕES REGULARES COM $\cdot^2$

Instância Uma expressão regular e com operações $\cup$ (união), $\cdot^*$ (fecho de Kleene), $\cdot$ (concatenação) e $\cdot^2$ (quadratura) sobre o alfabeto $\Sigma = \{0, 1\}$.

Decisão $L(e) \neq \Sigma^*$?

---

que parece razoavelmente simples é, de fato, EXPSPACE-completo Meyer e Stockmeyer (1972, Corolário 2.1) (no momento é suficiente saber que isso significa que o tempo para resolver o problema cresce pelo menos exponencialmente com o tamanho da entrada).

Exemplo 1.2
Com $e = 0 \cup 1^2$ temos $L(e) = \{0, 11\}$.
Com $e = (0 \cup 1)^2 \cdot 0^*$ temos

$$L(e) = \{00, 01, 10, 11, 000, 010, 100, 110, 0000, 0100, 1000, 1100, \ldots\}.$$

$\Diamond$

Existem exemplos de outros problemas que são decidíveis, mas têm uma complexidade tão grande que praticamente todas instâncias precisam mais recursos que o universo possui (por exemplo a decisão da validade na lógica monádica fraca de segunda ordem com sucessor).

O universo do ponto da vista da ciência da computação Falando sobre os recursos, é de interesse saber quantos recursos nosso universo disponibiliza aproximadamente. A seguinte tabela contém alguns dados básicos:

Idade	$13.75 \pm 0.11 \times 10^9$ anos $\approx 43.39 \times 10^{16}$ s
Tamanho	$\geq 78 \times 10^9$ anos-luz
Densidade	$9.9 \times 10^{-30} g/cm^3$
Número de átomos	10^{80}
Número de bits	10^{120}
Número de operações lógicas elementares até hoje	10^{120}
Operações/s	$\approx 2 \times 10^{102}$

(Os dados correspondem ao consenso científico no momento; obviamente novos descobrimentos podem os mudar *Wilkinson Microwave Anisotropy Probe* (2010) e Lloyd (2002))

Métodos para resolver um sistema de equações lineares Como resolver um sistema quadrático de equações lineares

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &= b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &= b_2 \\ &\cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &= b_n \end{aligned}$$

ou $Ax = b$? Podemos calcular a inversa da matriz A para chegar em $x = bA^{-1}$. O *método de Cramer* nos fornece as equações

$$x_i = \frac{\det(A_i)}{\det(A)}$$

seja A_i a matriz resultante da substituição de b pela i-gésima coluna de A. (A prova dessa fato é bastante simples. Seja U_i a matriz identidade com a i-gésima coluna substituído por x: é simples verificar que $AU_i = A_i$. Com $\det(U_i) = x_i$ e $\det(A)\det(U_i) = \det(A_i)$ temos o resultado.) Portanto, se o trabalho de calcular o determinante de uma matriz de tamanho $n \times n$ é T_n, essa abordagem custa $(n+1)T_n$. Um método direto usa a fórmula de Leibniz

$$\det(A) = \sum_{\sigma \in S_n} \left(\operatorname{sgn}(\sigma) \prod_{i \in [n]} a_{i,\sigma(i)} \right).$$

Isso precisa $n!$ adições (A) e $n!n$ multiplicações (M), com custo total

$$(n+1)(n!A + n!nM) \geq n!(A+M) \approx \sqrt{2\pi n}(n/e)^n(A+M),$$

um número formidável! Mas talvez a fórmula de Leibniz não é o melhor jeito de calcular o determinante! Vamos tentar a fórmula de expansão de Laplace

$$\det(A) = \sum_{i \in [n]} (-1)^{i+j} a_{ij} \det(A_{ij})$$

(sendo A_{ij} a matriz A sem linha a i e sem a coluna j). O trabalho T_n nesse caso é dado pelo recorrência

$$T_n = n(A + M + T_{n-1}); \qquad T_1 = 1$$

cuja solução é

$$T_n = n!\left(1 + (A+M)\sum_{1\le i<n} 1/i!\right)^{1}$$

e como $\sum_{1\le i<n} 1/i!$ aproxima e temos $n! \le T_n \le n!(1+(A+M)e)$ e logo T_n novamente é mais que $n!$. Mas qual é o método mais eficiente para calcular o determinante? Caso for possível em tempo proporcional ao tamanho da entrada n^2, tivermos um algoritmo em tempo aproximadamente n^3.
Antes de responder essa pergunta, vamos estudar uma abordagem diferente da pergunta original, o método de Gauss para resolver um sistema de equações lineares[2]. Em n passos, o matriz é transformada em forma triangular e cada passo não precisa mais que n^2 operações (nesse caso inclusive divisões).

Algoritmo 1.1 (Eliminação Gaussiana)
Entrada Uma matriz $A = (a_{ij}) \in \mathbb{R}^{n\times n}$

Saída A em forma triangular superior.

```
eliminação-gauss(a ∈ ℝ^{n×n})=
for i := 1,...,n do { elimina coluna i }
  for j := i+1,...,n do { elimina linha j }
    for k := n,...,i do
      a_jk := a_jk − a_ik a_ji / a_ii
    end for
  end for
end for
```

Exemplo 1.3
Para resolver

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 7 \\ 7 & 8 & 9 \end{pmatrix}\begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$$

vamos aplicar a eliminação de Gauss à matriz aumentada

$$\begin{pmatrix} 1 & 2 & 3 & 2 \\ 4 & 5 & 7 & 4 \\ 7 & 8 & 9 & 6 \end{pmatrix}$$

[1] $n!\sum_{1\le i<n} 1/i! = \lfloor n!(e-1)\rfloor$.
[2] Ou melhor a eliminação "ordinária", conhecido bem antes do Gauss (Grcar 2011).

obtendo

$$\begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & -3 & -5 & -4 \\ 0 & -6 & -12 & -8 \end{pmatrix}; \qquad \begin{pmatrix} 1 & 2 & 3 & 2 \\ 0 & -3 & -5 & -4 \\ 0 & 0 & -2 & 0 \end{pmatrix}$$

e logo $x_3 = 0$, $x_2 = 3/4$, $x_1 = 1/2$ é uma solução. ◊

Logo temos um algoritmo que determina a solução com

$$\sum_{i \in [n]} 3(n-i+1)(n-i) = n^3 - n$$

operações de ponto flutuante, que é (exceto valores de n bem pequenos) consideravelmente melhor que os resultados com $n!$ operações acima[3].
Observe que esse método também fornece o determinante do matriz: ela é o produto dos elementos na diagonal! De fato, o método é um dos melhores para calcular o determinante. Observe também que ela não serve para melhorar o método de Cramer, porque a solução do problema original já vem junto.

Qual o melhor algoritmo?

- Para um dado problema, existem diversos algoritmos com desempenhos diferentes.
- Queremos resolver um sistema de equações lineares de tamanho n.
- O método de Cramer precisa $\approx 6n!$ operações de ponto flutuante (OPF).
- O método de Gauss precisa $\approx n^3 - n$ OPF.
- Usando um computador de 3 GHz que é capaz de executar um OPF por ciclo temos

n	Cramer	Gauss
2	4 ns	2 ns
3	12 ns	8 ns
4	48 ns	20 ns
5	240ns	40 ns
10	7.3ms	330 ns
20	152 anos	2.7 μs

[3] O resultado pode ser melhorado considerando que a_{ji}/a_{ii} não depende do k

Motivação para algoritmos eficientes

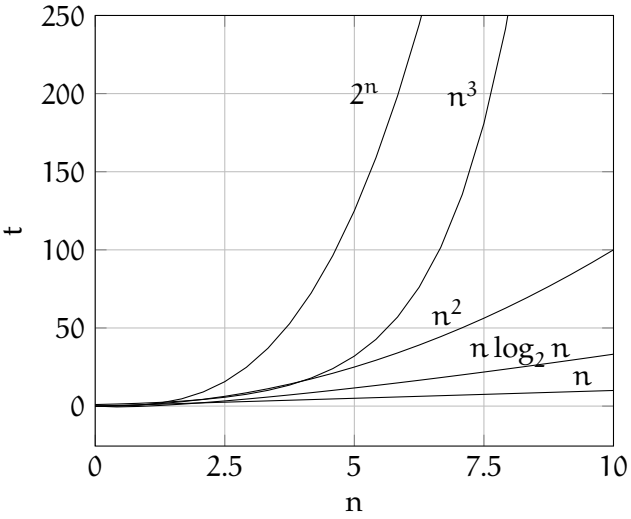
- Com um algoritmo ineficiente, um computador rápido não ajuda!
- Suponha que uma máquina resolva um problema de tamanho N em um dado tempo.
- Qual tamanho de problema uma máquina 10 vezes mais rápida resolve no mesmo tempo?

Número de operações	Máquina rápida
$\log_2 n$	N^{10}
n	$10N$
$n \log_2 n$	$10N$ (N grande)
n^2	$\sqrt{10}N \approx 3.2N$
n^3	$\sqrt[3]{10}N \approx 2.2N$
2^n	$N + \log_2 10 \approx N + 3.3$
3^n	$N + \log_3 10 \approx N + 2.1$

Exemplo 1.4
Esse exemplo mostra como calcular os dados da tabela acima. Dado um algoritmo que precisa $f(n)$ passos de execução numa determinada máquina. Qual o tamanho de problema n' que podemos resolver numa máquina c vezes mais rápido?

A quantidade n' satisfaz $f(n') = cf(n)$. Para funções que possuam uma inversa (por exemplo funções monotônicas) obtemos $n' = f^{-1}(cf(n))$. Por exemplo para $f(n) = \log_2 n$ e $c = 10$ (exemplo acima), temos $\log_2 n' = 10 \log_2 n \iff n' = n^{10}$. $\Diamond$

Crescimento de funções



Crescimento de funções

$n =$	10^1	10^2	10^3	10^4	10^5	10^6
$\log_2 n$	3	7	10	13	17	20
n	10^1	10^2	10^3	10^4	10^5	10^6
$n \log_2 n$	33	6.6×10^2	10^4	1.3×10^5	1.7×10^6	2×10^7
n^2	10^2	10^4	10^6	10^8	10^{10}	10^{12}
n^3	10^3	10^6	10^9	10^{12}	10^{15}	10^{18}
2^n	10^3	1.3×10^{30}	1.1×10^{301}	2×10^{3010}	10^{30103}	10^{301030}

$$1\,\text{ano} \approx 365.2425\text{d} \approx 3.2 \times 10^7\text{s}$$
$$1\,\text{século} \approx 3.2 \times 10^9\text{s}$$
$$1\,\text{milênio} \approx 3.2 \times 10^{10}\text{s}$$

Comparar eficiências

- Como comparar eficiências? Uma medida concreta do tempo depende
  - do tipo da máquina usada (arquitetura, cache, memória, ...)
  - da qualidade e das opções do compilador ou ambiente de execução
  - do tamanho do problema (da entrada)

- Portanto, foram inventadas *máquinas abstratas*.
- A *análise* da complexidade de um algoritmo consiste em determinar o número de operações básicas (atribuição, soma, comparação, ...) em relação ao tamanho da entrada.

Observe que nessa medida o tempo é "discreto".

Análise assintótica

- Em geral, o número de operações fornece um nível de detalhamento grande.
- Portanto, analisamos somente a taxa ou ordem de crescimento, substituindo funções exatas com cotas mais simples.
- Duas medidas são de interesse particular: A complexidade
  - pessimista e
  - média

Também podemos pensar em considerar a complexidade otimista (no caso melhor): mas essa medida faz pouco sentido, porque sempre é possível enganar com um algoritmo que é rápido para alguma entrada.

Exemplo

- Imagine um algoritmo com número de operações

$$an^2 + bn + c$$

- Para análise assintótica não interessam
  - os termos de baixa ordem, e
  - os coeficientes constantes.
- Logo o tempo da execução tem cota n^2, denotado com $O(n^2)$.

Observe que essas simplificações não devem ser esquecidas na escolha de um algoritmo na prática. Existem vários exemplos de algoritmos com desempenho bom assintoticamente, mas não são viáveis na prática em comparação com algoritmos "menos eficientes": A taxa de crescimento esconde fatores constantes e o tamanho mínimo de problema tal que um algoritmo é mais rápido que um outro.

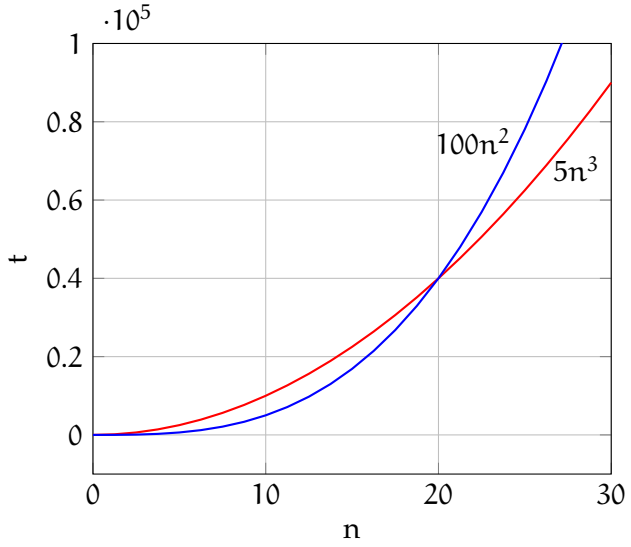
Complexidade de algoritmos

- Considere dois algoritmos A e B com tempo de execução $O(n^2)$ e $O(n^3)$, respectivamente. Qual deles é o mais eficiente ?
- Considere dois programas A e B com tempos de execução $100n^2$ milisegundos,e $5n^3$ milisegundos, respectivamente, qual é o mais eficiente?

Exemplo 1.5
Considerando dois algoritmos com tempo de execução $O(n^2)$ e $O(n^3)$ esperamos que o primeiro seja mais eficiente que o segundo. Para n grande, isso é verdadeiro, mas o tempo de execução atual pode ser $100n^2$ no primeiro e $5n^3$ no segundo caso. Logo para $n < 20$ o segundo algoritmo é mais rápido. ◊

Comparação de tempo de execução

- Assintoticamente consideramos um algoritmo com complexidade $O(n^2)$ melhor que um algoritmo com $O(n^3)$.
- De fato, para n suficientemente grande $O(n^2)$ sempre é melhor.
- Mas na prática, não podemos esquecer o tamanho do problema real.



Exemplo 1.6
Considere dois computadores C_1 e C_2 que executam 10^7 e 10^9 operações por segundo (OP/s) e dois algoritmos de ordenação A e B que necessitam $2n^2$ e

$50n \log_{10} n$ operações com entrada de tamanho n, respectivamente. Qual o tempo de execução de cada combinação para ordenar 10^6 elementos?

Algoritmo	Comp. C_1	Comp. C_2
A	$\frac{2\times(10^6)^2 OP}{10^7 OP/s} = 2 \times 10^5 s$	$\frac{2\times(10^6)^2 OP}{10^9 OP/s} = 2 \times 10^3 s$
B	$\frac{50\times10^6 \log 10^6 OP}{10^7 OP/s} = 30s$	$\frac{50\times10^6 \log 10^6 OP}{10^9 OP/s} = 0.3s$

$\Diamond$

Um panorama de tempo de execução

- Tempo constante: $O(1)$ (raro).
- Tempo sublinear ($\log(n)$, $\log(\log(n))$, etc): Rápido. Observe que o algoritmo não pode ler toda entrada.
- Tempo linear: Número de operações proporcional à entrada.
- Tempo $n \log n$: Comum em algoritmos de divisão e conquista.
- Tempo polinomial n^k: Frequentemente de baixa ordem ($k \leq 10$), considerado eficiente.
- Tempo exponencial: 2^n, $n!$, n^n considerado intratável.

Exemplo 1.7
Exemplos de algoritmos para as complexidades acima:

- Tempo constante: Determinar se uma sequência de números começa com 1.
- Tempo sublinear: Busca binária.
- Tempo linear: Buscar o máximo de uma sequência.
- Tempo $n \log n$: Mergesort.
- Tempo polinomial: Multiplicação de matrizes.
- Tempo exponencial: Busca exaustiva de todos subconjuntos de um conjunto, de todas permutações de uma sequência, etc.

$\Diamond$

Problemas super-polinomiais?

- Consideramos a classe P de problemas com solução em tempo polinomial tratável.
- NP é outra classe importante que contém muitos problemas práticos (e a classe P).
- Não se sabe se todos possuem algoritmo eficiente.
- Problemas NP-completos são os mais complexos do NP: Se um deles tem uma solução eficiente, toda classe tem.
- Vários problemas NP-completos são parecidos com problemas que têm algoritmos eficientes.

Solução eficiente conhecida	Solução eficiente improvável
Ciclo euleriano	Ciclo hamiltoniano
Caminho mais curto	Caminho mais longo
Satisfatibilidade 2-CNF	Satisfatibilidade 3-CNF

Ciclo euleriano

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

Decisão Existe um ciclo euleriano, i.e. um caminho $v_1, v_2, \ldots, v_n$ tal que $v_1 = v_n$ que usa todos arcos exatamente uma vez?

Comentário É decidível em tempo linear usando o teorema de Euler: um grafo conexo contém um ciclo euleriano sse o grau de cada nó é par (Diestel 2005, Teorema 1.8.1)). No caso de um grafo direcionado tem um teorema correspondente: um grafo fortemente conexo contém um ciclo euleriano sse cada nó tem o mesmo número de arcos entrantes e saintes.

Ciclo hamiltoniano

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

Decisão Existe um ciclo hamiltanio, i.e. um caminho $v_1, v_2, \ldots, v_n$ tal que $v_1 = v_n$ que usa todos nós exatamente uma única vez?

1.1. Notação assintótica

O análise de algoritmos considera principalmente recursos como tempo e espaço. Analisando o comportamento de um algoritmo em termos do tamanho da entrada significa achar uma função $c : \mathbb{N} \to \mathbb{R}^+$, que associa com todos entradas de um tamanho n um custo (médio,máximo) $c(n)$. Observe, que é suficiente trabalhar com funções positivas (com co-domínio $\mathbb{R}^+$), porque os recursos de nosso interesse são positivos. A seguir, supomos que todas funções são dessa forma.

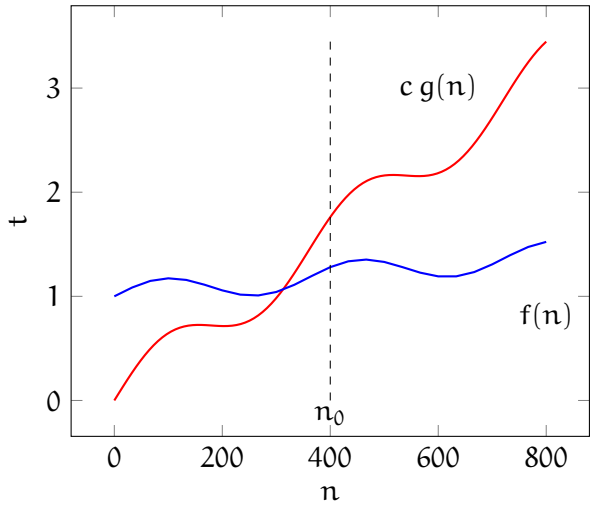
Notação assintótica: O

- Frequentemente nosso interesse é o comportamento *assintótico* de uma função $f(n)$ para $n \to \infty$.
- Por isso, vamos introduzir *classes de crescimento*.
- O primeiro exemplo é *a classe de funções que crescem menos ou igual que* $g(n)$

$$O(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid (\exists c > 0)\exists n_0(\forall n > n_0) : \ f(n) \leq cg(n)\}$$

A definição do O (e as outras definições em seguido) podem ser generalizadas para qualquer função com domínio $\mathbb{R}$.

Notação assintótica: O



Notação assintótica

- Com essas classes podemos escrever por exemplo
$$4n^2 + 2n + 5 \in O(n^2)$$
- Outra notação comum que usa a identidade é
$$4n^2 + 2n + 5 = O(n^2)$$
- Observe que essa notação é uma "equação sem volta" (inglês: one-way equation);
$$O(n^2) = 4n^2 + 2n + 5$$
não é definido.

Para $f \in O(g)$ leia: "f é do ordem de g"; para $f = O(g)$ leiamos as vezes simplesmente "f é O de g". Observe que numa equação como $4n^2 = O(n^2)$, as expressões $4n^2$ e n^2 denotam *funções*, não valores[4].
Caso $f \in O(g)$ com constante $c = 1$, digamos que g é uma *cota assintótica superior* de f Toscani e Veloso (2005, p. 15). Em outras palavras, O define uma cota assintótica superior a menos de constantes.

O: Exemplos

$$5n^2 + n/2 \in O(n^3)$$
$$5n^2 + n/2 \in O(n^2)$$
$$\sin(n) \in O(1)$$

Exemplo 1.8
Mais exemplos

$$n^2 \in O(n^3 \log_2 n) \qquad c = 1; n_0 = 2$$
$$32n \in O(n^3) \qquad c = 32; n_0 = 1$$
$$10^n n^2 \notin O(n2^n) \qquad \text{porque } 10^n n^2 \leq cn2^n \iff 5^n n \leq c$$
$$n \log_2 n \in O(n \log_{10} n) \qquad c = 4; n_0 = 1$$

$\Diamond$

[4]Mais correto (mas menos confortável) seria escrever $\lambda n.4n^2 = O(\lambda n.n^2)$

O: Exemplos

Proposição 1.1
Para um polinômio $p(n) = \sum_{0 \le i \le m} a_i n^i$ temos

$$|p(n)| \in O(n^m) \tag{1.1}$$

Prova.

$$\begin{aligned} |p(n)| &= \left| \sum_{0 \le i \le m} a_i n^i \right| \\ &\le \sum_{0 \le i \le m} |a_i| n^i \qquad \text{Corolário A.1} \\ &\le \sum_{0 \le i \le m} |a_i| n^m = n^m \sum_{0 \le i \le m} |a_i| \end{aligned}$$

■

Notação assintótica: Outras classes

- Funções que crescem (estritamente) menos que $g(n)$

$$o(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid (\forall c > 0)\exists n_0(\forall n > n_0) : \ f(n) \le cg(n)\} \tag{1.2}$$

- Funções que crescem mais ou igual à $g(n)$

$$\Omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid (\exists c > 0)\exists n_0(\forall n > n_0) : \ f(n) \ge cg(n)\} \tag{1.3}$$

- Funções que crescem (estritamente) mais que $g(n)$

$$\omega(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid (\forall c > 0)\exists n_0(\forall n > n_0) : \ f(n) \ge cg(n)\} \tag{1.4}$$

- Funções que crescem igual à $g(n)$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \tag{1.5}$$

Observe que a nossa notação somente é definida "ao redor do ∞", que é suficiente para a análise de algoritmos. Equações como $e^x = 1 + x + O(x^2)$, usadas no cálculo, possuem uma definição de O diferente.
As definições ficam equivalente, substituindo $<$ para $\le$ e $>$ para $\ge$ (veja exercício 1.10).

Convenção 1.1
Se o contexto permite, escrevemos $f \in O(g)$ ao invés de $f(n) \in O(g(n))$, $f \leq cg$ ao invés de $f(n) \leq cg(n)$ etc.

Proposição 1.2 (Caracterização alternativa)
Caracterizações alternativas de O, o, Ω e ω são

$$f(n) \in O(g(n)) \iff \limsup_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{1.6}$$

$$f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \tag{1.7}$$

$$f(n) \in \Omega(g(n)) \iff \liminf_{n\to\infty} \frac{f(n)}{g(n)} > 0 \tag{1.8}$$

$$f(n) \in \omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty \tag{1.9}$$

Prova. Exercício. ■

Convenção 1.2
Escrevemos $f, g, \ldots$ para funções $f(n), g(n), \ldots$ caso não tem ambiguedade no contexto.

Operações

- As notações assintóticas denotam conjuntos de funções.
- Se um conjunto ocorre em uma fórmula, resulta o conjunto de todas combinações das partes.
- Exemplos: $n^{O(1)}$, $\log O(n^2)$, $n^{1+o(1)}$, $(1 - o(1)) \ln n$
- Em uma equação o lado esquerdo é (implicitamente) quantificado universal, e o lado direto existencial.
- Exemplo: $n^2 + O(n) = O(n^4)$ Para todo $f \in O(n)$, existe um $g \in O(n^4)$ tal que $n^2 + f = g$.

Exemplo 1.9
$n^{O(1)}$ denota

$$\{n^{f(n)} \mid \exists c\, f(n) \leq c\} \subseteq \{f(n) \mid \exists c \exists n_0\, \forall n > n_0\, f(n) \leq n^c\}$$

o conjunto das funções que crescem menos que um polinômio. ◊

Uma notação assintótica menos comum é $f = \tilde{O}(g)$ que é uma abreviação para $f = O(g \log_k g)$ para algum k. $\tilde{O}$ é usado se fatores logarítmicos não importam. Similarmente, $f = O^*(g)$ ignora fatores polinomiais, i.e. $f = O(gp)$ para um polinômio $p(n)$.

Características

$$f = O(f) \tag{1.10}$$
$$cO(f) = O(f) \tag{1.11}$$
$$O(f) + O(f) = O(f) \tag{1.12}$$
$$O(O(f)) = O(f) \tag{1.13}$$
$$O(f)O(g) = O(fg) \tag{1.14}$$
$$O(fg) = fO(g) \tag{1.15}$$

Prova. Exercício. ■

Exemplo 1.10
Por exemplo, (1.12) implica que para $f = O(h)$ e $g = O(h)$ temos $f + g = O(h)$. ♢

As mesmas características são verdadeiras para Ω (prova? veja exercício 1.6). E para o, ω e Θ?

Características: "Princípio de absorção" Toscani e Veloso (2005, p. 35)

$$g = O(f) \Rightarrow f + g = \Theta(f)$$

Relações de crescimento Uma vantagem da notação O é que podemos usá-la em fórmulas como $m + O(n)$. Em casos em que isso não for necessário, e queremos simplesmente comparar funções, podemos introduzir relações de crescimento entre funções, obtendo uma notação mais comum. Uma definição natural é

Relações de crescimento

Definição 1.1 (Relações de crescimento)

$$f \prec g \iff f \in o(g) \tag{1.16}$$
$$f \preceq g \iff f \in O(g) \tag{1.17}$$
$$f \succ g \iff f \in \omega(g) \tag{1.18}$$
$$f \succeq g \iff f \in \Omega(g) \tag{1.19}$$
$$f \asymp g \iff f \in \Theta(g) \tag{1.20}$$

Essas relações são chamadas "notação de Vinogradov"[5].
Caso $f \preceq g$ digamos as vezes "f é absorvido pela g". Essas relações satisfazem as características básicas esperadas.

Características das relações de crescimento

Proposição 1.3 (Características das relações de crescimento)
Sobre o conjunto de funções $[\mathbb{N} \to \mathbb{R}^+]$

(a) $f \preceq g \iff g \succeq f$,

(b) $\preceq$ e $\succeq$ são ordenações parciais (reflexivas, transitivas e anti-simétricas em relação de $\asymp$),

(c) $f \prec g \iff g \succ f$,

(d) $\prec$ e $\succ$ são transitivas,

(e) $\asymp$ é uma relação de equivalência.

Prova. Exercício. ■
Observe que esses resultados podem ser traduzidos para a notação O. Por exemplo, como $\asymp$ é uma relação de equivalência, sabemos que Θ também satisfaz

$$f \in \Theta(f)$$
$$f \in \Theta(g) \Rightarrow g \in \Theta(f)$$
$$f \in \Theta(g) \wedge g \in \Theta(h) \Rightarrow f \in \Theta(h)$$

A notação com relações é sugestiva e frequentemente mais fácil de usar, mas nem todas as identidades que ela sugere são válidas, como a seguinte proposição mostra.

[5] Uma notação alternativa é $\ll$ para $\preceq$ e $\gg$ para $\succeq$. Infelizmente a notação não é padronizada.

Identidades falsas das relações de crescimento

Proposição 1.4 (Identidades falsas das relações de crescimento)
É verdadeiro que

$$f \succ g \Rightarrow f \not\preceq g \tag{1.21}$$
$$f \prec g \Rightarrow f \not\succeq g \tag{1.22}$$

mas as seguintes afirmações *não* são verdadeiras:

$$f \not\preceq g \Rightarrow f \succ g$$
$$f \not\succeq g \Rightarrow f \prec g$$
$$f \prec g \bigvee f \asymp g \bigvee f \succ g \qquad \text{(Tricotomia)}$$

Prova. Exercício. ■

Considerando essas características, a notação tem que ser usada com cuidado. Uma outra abordagem é definir O etc. diferente, tal que outras relações acima são verdadeiras. Mas parece que isso não é possível, sem perder outras (Vitányi e Meertens 1985).

1.2. Notas

Alan Turing provou em 1936 que o "problema de parada" não é decidível. O estudo da complexidade de algoritmos começou com o artigo seminal de Hartmanis e Stearns (1965).

O estudo da complexidade de calcular a determinante tem muito mais aspectos interessantes. Um deles é que o método de Gauss pode produzir resultados intermediários cuja representação precisa um número exponencial de bits em função do tamanho da entrada. Portanto, o método de Gauss formalmente não tem complexidade $O(n^3)$. Resultados atuais mostram que uma complexidade de operações de bits $n^{3.2} \log \|A\|^{1+o(1)}$ é possível (Kaltofen e Villard 2004).

Nossa discussão da regra de Cramer usa dois métodos naivos para calcular determinantes. Habgood e Arel (2010) mostram que existe um algoritmo que resolve um sistema de equações lineares usando a regra de Cramer em tempo $O(n^3)$.

1.3. Exercícios

(Soluções a partir da página 431.)

Exercício 1.1
Quais funções são contidos nos conjuntos $O(-1)$, $o(-1)$, $\Omega(-1)$, $\omega(-1)$?

Exercício 1.2
Prove as equivalências (1.6), (1.7), (1.8) e (1.9).

Exercício 1.3
Prove as equações (1.10) até (1.15).

Exercício 1.4
Prove a proposição (1.3).

Exercício 1.5
Prove a proposição (1.4).

Exercício 1.6
Prove as características 1.10 até 1.15 (ou características equivalentes caso alguma não se aplica) para Ω.

Exercício 1.7
Prove ou mostre um contra-exemplo. Para qualquer constante $c \in \mathbb{R}$, $c > 0$

$$f \in O(g) \iff f + c \in O(g) \tag{1.23}$$

Exercício 1.8
Prove ou mostre um contra-exemplo.

(a) $\log(1+n) = O(\log n)$

(b) $\log O(n^2) = O(\log n)$

(c) $\log\log n = O(\log n)$

(d) $n! = O(2^{\binom{n}{2}})$

Exercício 1.9
Considere a função definido pela recorrência

$$f_n = 2f_{n-1}; \qquad f_0 = 1.$$

Professor Veloz afirme que $f_n = O(n)$, e que isso pode ser verificado simplesmente da forma

$$f_n = 2f_{n-1} = 2O(n-1) = 2O(n) = O(n)$$

Mas sabendo que a solução dessa recorrência é $f_n = 2^n$ duvidamos que $2^n = O(n)$. Qual o erro do professor Veloz?

Exercício 1.10
Mostre que a definição

$$\hat{o}(g(n)) = \{f : \mathbb{N} \to \mathbb{R}^+ \mid (\forall c > 0)\exists n_0(\forall n > n_0) : \ f(n) < cg(n)\}$$

(denotado com ô para diferenciar da definição o) e equivalente com a definição 1.2 para funções $g(n)$ que são diferente de 0 a partir de um n_0.

Exercício 1.11
Mostre que o números Fibonacci

$$f_n = \begin{cases} n & \text{se } 0 \leq n \leq 1 \\ f_{n-2} + f_{n-1} & \text{se } n \geq 2 \end{cases}$$

têm ordem assintótica $f_n \in \Theta(\Phi^n)$ com $\Phi = (1 + \sqrt{5})/2$.

Exercício 1.12
Prove a seguinte variação do princípio de absorção para $g : \mathbb{N} \to \mathbb{R}^+$:

$$g \in o(f) \Rightarrow f - g \in \Theta(f).$$

Exercício 1.13
Prove que

$$f \leq g \Rightarrow O(f) = O(g).$$

Exercício 1.14
Prove que $\Theta(f) = O(f)$, mas o contrário $O(f) = \Theta(f)$ não é correto.

Exercício 1.15
Para qualquer par das seguintes funções, analisa a complexidade mutual.

$$n^3, n^3 \log^{1/3} \log n / \log^{1/3} n, n^3 \log^{1/2} \log n / \log^{1/2} n, n^3 \log^{1/2},$$
$$n^3 \log^{5/7} \log n / \log^{5/7} n, n^3 \log^2 \log n / \log n, n^3 \log^{1/2} \log n / \log n,$$
$$n^3 \log n, n^3 \log^{5/4} \log n / \log^{5/4} n, n^3 \log^3 \log n / \log^2 n$$

Exercício 1.16
Prove: $2^{-m} = 1 + O(m^{-1})$.

Exercício 1.17

a) Suponha que f e g são funções polinomiais de $\mathbb{N}$ para $\mathbb{N}$ com $f(n) \in \Theta(n^r)$ e $g(n) \in \Theta(n^s)$. O que se pode afirmar sobre a função composta $g(f(n))$?

b) Classifique as funções $f(n) = 5 \cdot 2^n + 3$ e $g(n) = 3 \cdot n^2 + 5 \cdot n$ como $f \in O(g)$, $f \in \Theta(g)$ ou $f \in \Omega(g)$.

c) Verifique se $2^n \cdot n \in \Theta(2^n)$ e se $2^{n+1} \in \Theta(2^n)$.

Exercício 1.18
Mostra que $\log n \in O(n^\epsilon)$ para todo $\epsilon > 0$.

Exercício 1.19 (Levin (1973))
Duas funções $f(n)$ e $g(n)$ são *comparáveis* caso existe um k tal que

$$f(n) \leq (g(n) + 2)^k; \qquad g(n) \leq (f(n) + 2)^k.$$

Quais dos pares n e n^2, n^2 e $n^{\log n}$ e $n^{\log n}$ e e^n são comparáveis?

2. Análise de complexidade

2.1. Introdução

Para analisar a eficiência de algoritmos faz pouco sentido medir os recursos gastos em computadores específicos, porque devido a diferentes conjuntos de instruções, arquiteturas e desempenho dos processadores, as medidas são difíceis de comparar. Portanto, usamos um *modelo* de uma máquina que reflita as características de computadores comuns, mas é independente de uma implementação concreta. Um modelo comum é a *máquina de RAM* com as seguintes características:

- um processador com um ou mais registros, e com apontador de instruções,
- uma memória infinita de números inteiros e
- um conjunto de instruções elementares que podem ser executadas em tempo $O(1)$ (por exemplo funções básicas sobre números inteiros e de ponto flutuante, acesso à memória e transferência de dados); essas operações refletem operações típicas de máquinas concretas.

Observe que a escolha de um modelo abstrato não é totalmente trivial. Conhecemos vários modelos de computadores, cuja poder computacional não é equivalente em termos de complexidade (que não viola a tese de Church-Turing). Mas todos os modelos encontrados (fora da computação quântica) são polinomialmente equivalentes, e portanto, a noção de eficiência fica a mesma. A tese que todos modelos computacionais são polinomialmente equivalentes as vezes está chamado *tese de Church-Turing estendida*.

O plano

Uma hierarquia de abstrações:

$$\text{Tempo de execução [s]} \longrightarrow \text{Número de operações} \longrightarrow \text{Operações no caso pessimista} \to \text{Complexidade pessimista assintótica}$$

Custos de execuções

- Seja $E = O^*$ o conjunto de sequências de operações fundamentais O.
- Para um algoritmo A, com entradas D seja

$$\text{exec}[A] : D \to E$$

  a função que fornece a sequência de instruções executadas $\text{exec}[A](d)$ para cada entrada $d \in D$.

- Se atribuímos custos para cada operação básica, podemos calcular também o custo de uma execução

$$\text{custo} : E \to \mathbb{R}^+$$

- e o custo da execução do algoritmo a depende da entrada d

$$\text{desemp}[A] : D \to \mathbb{R}^+ = \text{custo} \circ \text{exec}[A]$$

Definição 2.1
O símbolo $\circ$ denota a composição de funções tal que

$$(f \circ g)(n) = f(g(n))$$

(leia: "f depois g").

Em geral, não interessam os custos específicos para cada entrada, mas o "comportamento" do algoritmo. Uma medida natural é como os custos crescem com o tamanho da entrada.

Condensação de custos

- Queremos condensar os custos para uma única medida.
- Essa medida depende somente do tamanho da entrada

$$\text{tam} : D \to \mathbb{N}$$

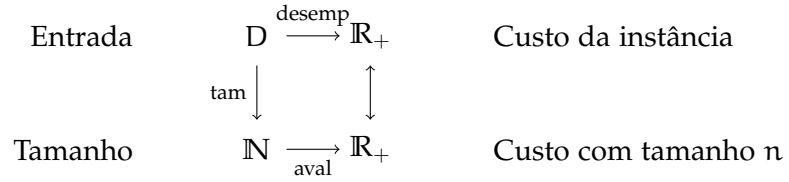
- O objetivo é definir uma função

$$\text{aval}[A](n) : \mathbb{N} \to \mathbb{R}^+$$

  que define o desempenho do algoritmo em relação ao tamanho.

- Como, em geral, tem várias entradas d tal que $\text{tam}(d) = n$ temos que definir como condensar a informação de $\text{desemp}[A](d)$ delas.

Condensação



Condensação

- Na prática, duas medidas condensadas são de interesse particular
- A complexidade pessimista

$$C_p^=[A](n) = \max\{\text{desemp}[A](d) \mid d \in D, \text{tam}(d) = n\}$$

- A complexidade média

$$C_m[A](n) = \sum_{\text{tam}(d)=n} P(d)\,\text{desemp}[A](d)$$

- Observe: A complexidade média é o valor esperado do desempenho de entradas com tamanho n.
- Ela é baseada na distribuição das entradas.

A complexidade média é menos usada na prática, por várias razões. Primeiramente, a complexidade pessimista garante um certo desempenho, independente da entrada. Em comparação, uma complexidade média $O(n^2)$, por exemplo, não exclui que algumas entradas com tamanho n precisam muito mais tempo. Por isso, se é importante saber quando uma execução de um algoritmo termina, preferimos a complexidade pessimista.
Para vários algoritmos com desempenho ruim no pior caso, estamos interessados como eles se comportam na média. Infelizmente, ela é difícil de determinar. Além disso, ela depende da distribuição das entradas, que frequentemente não é conhecida, difícil de determinar, ou é diferente em aplicações diferentes.

Definição alternativa

- A complexidade pessimista é definida como

$$C_p^{=}[A](n) = \max\{\text{desemp}[A](d) \mid d \in D, \text{tam}(d) = n\}$$

- Uma definição alternativa é

$$C_p^{\leq}[A](n) = \max\{\text{desemp}[A](d) \mid d \in D, \text{tam}(d) \leq n\}$$

- $C_p^{\leq}$ é monotônica e temos

$$C_p^{=}[A](n) \leq C_p^{\leq}[A](n)$$

- Caso $C_p^{=}$ seja monotônica as definições são equivalentes

$$C_p^{=}[A](n) = C_p^{\leq}[A](n)$$

$C_p^{=}[A](n) \leq C_p^{\leq}[A](n)$ é uma consequência da observação que

$$\{\text{desemp}[A](d) \mid d \in D, \text{tam}(d) = n\} \subseteq \{\text{desemp}[A](d) \mid d \in D, \text{tam}(d) \leq n\}$$

Analogamente, se $A \subseteq B$ tem-se que $\max A \leq \max B$.

Exemplo 2.1
Vamos aplicar essas noções num exemplo de um algoritmo simples. Queremos é decidir se uma sequência de números naturais contém o número 1.

Algoritmo 2.1 (Busca1)
Entrada Uma sequência $a_1, \ldots, a_n$ de números em $\mathbb{N}$.

Saída True, caso existe um i tal que $a_i = 1$, false, caso contrário.

```
for i:=1 to n do
  if (a_i = 1) then
    return true
  end if
end for
return false
```

Para analisar o algoritmo, podemos escolher, por exemplo, as operações básicas $O = \{\texttt{for}, \texttt{if}, \texttt{return}\}$ e atribuir um custo constante de 1 para cada um delas. (Observe que como "operação básica" são consideradas as operações de atribuição, incremento e teste da expressão booleana $i \leq n$.) Logo as execuções possíveis são $E = O^*$ e temos a função de custos

$$\text{custo} : E \to \mathbb{R}^+ : e \mapsto |e|.$$

Por exemplo $\text{custo}((\texttt{for}, \texttt{for}, \texttt{if}, \texttt{return})) = 4$. As entradas desse algoritmo são sequências de números naturais, logo, $D = \mathbb{N}^*$ e como tamanho da entrada escolhemos

$$\text{tam} : D \to \mathbb{N} : (a_1, \ldots, a_n) \mapsto n.$$

A função de execução atribui a sequência de operações executadas a qualquer entrada. Temos

$$\text{exec[Busca1]}(d) : D \to E :$$
$$(a_1, \ldots, a_n) \mapsto \begin{cases} (\texttt{for}, \texttt{if})^i\texttt{return}, & \text{caso existe } i = \min\{j \mid a_j = 1\}, \\ (\texttt{for}, \texttt{if})^n\texttt{return}, & \text{caso contrário,} \end{cases}$$

Com essas definições temos também a função de desempenho

$$\text{desemp[Busca1]}(n) = \text{custo} \circ \text{exec[Busca1]} :$$
$$(a_1, \ldots, a_n) \mapsto \begin{cases} 2i + 1, & \text{caso existe } i = \min\{j \mid a_j = 1\}, \\ 2n + 1, & \text{caso contrário,} \end{cases}$$

Agora podemos aplicar a definição da complexidade pessimista para obter

$$C_p^{\leq}[\text{Busca1}](n) = \max\{\text{desemp[Busca1]}(d) \mid \text{tam}(d) = n\} = 2n + 1 = O(n).$$

Observe que $C_p^{=}$ é monotônica, e portanto $C_p^{=} = C_p^{\leq}$.
Um caso que em geral é menos interessante podemos tratar nesse exemplo também: Qual é a complexidade otimista (complexidade no caso melhor)? Isso acontece quando 1 é o primeiro elemento da sequência, logo, $C_o[\text{Busca1}](n) = 2 = O(1)$.

$\Diamond$

2.2. Complexidade pessimista

2.2.1. Metodologia de análise de complexidade

Uma linguagem simples

- Queremos estudar como determinar a complexidade de algoritmos metodicamente.
- Para este fim, vamos usar uma linguagem simples que tem as operações básicas de
  (a) Atribuição: `v:=e`
  (b) Sequência: `c1;c2`
  (c) Condicional: `se b então c1 senão c2`
  (d) Iteração definida: `para i de j até m faça c`
  (e) Iteração indefinida: `enquanto b faça c`

A forma "`se b então c1`" vamos tratar como abreviação de "`se b então c1 senão skip`" com comando "`skip`" de custo 0.
Observe que a metodologia não implica que tem *um algoritmo* que, dado um algoritmo como entrada, computa a complexidade dele. Este problema não é computável (por quê?).

Convenção 2.1
A seguir vamos entender implicitamente todas operações sobre funções *pontualmente*, i.e. para alguma operação $\circ$, e funções f, g com $\mathrm{dom}(f) = \mathrm{dom}(g)$ temos

$$\forall d \in \mathrm{dom}(f) \quad (f \circ g) = f(d) \circ g(d).$$

Componentes

- A complexidade de um algoritmo pode ser analisada em termos de suas componentes (princípio de composicionalidade).
- Pode-se diferenciar dois tipos de componentes: *componentes conjuntivas* e *componentes disjuntivas*.
- Objetivo: Analisar as componentes independentemente (como sub-algoritmos) e depois compor as complexidades delas.

Composição de componentes

- Cuidado: Na composição de componentes o tamanho da entrada pode mudar.
- Exemplo: Suponha que um algoritmo A produz, a partir de uma lista de tamanho n, uma lista de tamanho n^2 em tempo $\Theta(n^2)$.

$$n \longrightarrow \boxed{A} \longrightarrow n^2 \longrightarrow \boxed{A} \longrightarrow n^4$$

- A sequência $A;A$, mesmo sendo composta pelos dois algoritmos com $\Theta(n^2)$ *individualmente*, tem complexidade $\Theta(n^4)$.
- Portanto, vamos diferençar entre
  - algoritmos que preservam (assintoticamente) o tamanho, e
  - algoritmos em que modificam o tamanho do problema.
- Neste curso tratamos somente o primeiro caso.

Componentes conjuntivas

A sequência

- Considere uma sequência $c_1; c_2$.
- Qual a sua complexidade $c_p[c_1; c_2]$ em termos dos componentes $c_p[c_1]$ e $c_p[c_2]$?
- Temos

  $$\text{desemp}[c_1; c_2] = \text{desemp}[c_1] + \text{desemp}[c_2] \geq \max(\text{desemp}[c_1], \text{desemp}[c_2])$$

  e portanto (veja A.8)

  $$\max(c_p[c_1], c_p[c_2]) \leq c_p[c_1; c_2] \leq c_p[c_1] + c_p[c_2]$$

  e como $f + g \in O(\max(f, g))$ tem-se que

  $$c_p[c_1; c_2] = \Theta(c_p[c_1] + c_p[c_2]) = \Theta(\max(c_p[c_1], c_p[c_2]))$$

Prova.

$$\begin{aligned}\max(\text{desemp}[c_1](d), \text{desemp}[c_2](d)) &\leq \text{desemp}[c_1; c_2](d)\\ &= \text{desemp}[c_1](d) + \text{desemp}[c_2](d)\end{aligned}$$

logo para todas entradas d com $\text{tam}(d) = n$

$$\begin{aligned}\max_d \max(\text{desemp}[c_1](d), \text{desemp}[c_2](d)) &\leq \max_d \text{desemp}[c_1; c_2](d)\\ &= \max_d (\text{desemp}[c_1](d) + \text{desemp}[c_2](d))\\ \iff \max(c_p[c_1], c_p[c_2]) &\leq c_p[c_1; c_2] \leq c_p[c_1] + c_p[c_2]\end{aligned}$$

■

Exemplo 2.2
Considere a sequência $S \equiv v := \text{ordena}(u); w := \text{soma}(u)$ com complexidades $c_p[v := \text{ordena}(u)](n) = n^2$ e $c_p[w := \text{soma}(u)](n) = n$. Então $c_p[S] = \Theta(n^2 + n) = \Theta(n^2)$. ◊

Exemplo 2.3
Considere uma partição das entradas do tamanho n tal que $\{d \in D \mid \text{tam}(d) = n\} = D_1(n) \dot\cup D_2(n)$ e dois algoritmos A_1 e A_2, A_1 precisa n passos para instâncias em D_1, e n^2 para instâncias em D_2. A_2, contrariamente, precisa n^2 para instâncias em D_1, e n passos para instâncias em D_2. Com isso obtemos

$$c_p[A_1] = n^2; \qquad c_p[A_2] = n^2; \qquad c_p[A_1; A_2] = n^2 + n$$

e portanto

$$\max(c_p[A_1], c_p[A_2]) = n^2 < c_p[A_1; A_2] = n^2 + n < c_p[A_1] + c_p[A_2] = 2n^2$$

◊

A atribuição: Exemplos

- Considere os seguintes exemplos.
- Inicializar ou atribuir variáveis tem complexidade $O(1)$

$$i := 0; \qquad j := i$$

- Calcular o máximo de n elementos tem complexidade $O(n)$

$$m := \max(v)$$

- Inversão de uma lista u e atribuição para w tem complexidade $2n \in O(n)$

$$w := \text{reversa}(u)$$

A atribuição

- Logo, a atribuição depende dos custos para a avaliação do lado direito e da atribuição, i.e.

$$\text{desemp}[v := e] = \text{desemp}[e] + \text{desemp}[\leftarrow_e]$$

- Ela se comporta como uma sequência desses dois componentes:

$$c_p[v := e] = \Theta(c_p[e] + c_p[\leftarrow_e])$$

- Frequentemente $c_p[\leftarrow_e]$ é absorvida pelo $c_p[e]$ e temos

$$c_p[v := e] = \Theta(c_p[e])$$

Exemplo 2.4
Continuando o exemplo 2.2 podemos examinar a atribuição $v := \texttt{ordene}(w)$. Com complexidade pessimista para a ordenação da lista $c_p[\texttt{ordene}(w)] = O(n^2)$ e complexidade $c_p[\leftarrow_e] = O(n)$ para a transferência, temos $c_p[v := \texttt{ordene}(w)] = O(n^2) + O(n) = O(n^2)$. ♢

Iteração definida

Seja C =`para i de j até m faça c`

- O número de iterações é fixo, mas j e m dependem da entrada d.
- Seja $N(n) = \max_d\{m(d) - j(d) + 1 \mid tam(d) \leq n\}$ e $N^*(n) = \max\{N(n), 0\}$.
- $N^*(n)$ é o máximo de iterações para entradas de tamanho até n.
- Tendo N^*, podemos tratar a iteração definida como uma sequência

$$\underbrace{c; c; \cdots; c}_{N^*(n)\text{ vezes}}$$

que resulta em

$$c_p[C] \leq N^*\, c_p[c]$$

Iteração indefinida

Seja C =`enquanto b faça c`

- Para determinar a complexidade temos que saber o número de iterações.
- Seja $H(d)$ o número da iteração (a partir de 1) em que a condição é falsa pela primeira vez
- e $h(n) = \max\{H(d) \mid tam(d) \leq n\}$ o número máximo de iterações com entradas até tamanho n.
- Em analogia com a iteração definida temos uma sequência

$$\underbrace{b; c; b; c; \cdots ; b; c;}_{h(n)-1 \text{ vezes}} b$$

  e portanto

$$c_p[C] \leq (h-1)c_p[c] + hc_p[b]$$

- Caso o teste b é absorvido pelo escopo c temos

$$c_p[C] \leq (h-1)c_p[c]$$

Observe que pode ser difícil determinar o número de iterações $H(d)$; em geral a questão não é decidível.

Componentes disjuntivas

Componentes disjuntivas

- Suponha um algoritmo c que consiste em duas componentes disjuntivas c_1 e c_2. Logo,

$$\text{desemp}[c] \leq \max(\text{desemp}[c_1], \text{desemp}[c_2])$$

  e temos

$$c_p[A] \leq \max(c_p[c_1], c_p[c_2])$$

Caso a expressão para o máximo de duas funções for difícil, podemos simplificar

$$c_p[A] \leq \max(c_p[c_1], c_p[c_2]) = O(\max(c_p[c_1], c_p[c_2])).$$

O condicional

Seja C =`se b então` c_1 `senão` c_2

- O condicional consiste em
  - uma avaliação da condição b
  - uma avaliação do comando c_1 ou c_2 (componentes disjuntivas).
- Aplicando as regras para componentes conjuntivas e disjuntivas obtemos

$$c_p[C] \leq c_p[b] + \max(c_p[c_1], c_p[c_2])$$

Para `se b então c1` obtemos com $c_p[skip] = 0$

$$c_p[C] \leq c_p[b] + c_p[c_1]$$

Exemplo 2.5
Considere um algoritmo a que, dependendo do primeiro elemento de uma lista u, ordena a lista ou determina seu somatório:

Exemplo

`se` $hd(u) = 0$ `então`
  $v := ordena(u)$
`senão`
  $s := soma(u)$

Assumindo que o teste é possível em tempo constante, ele é absorvido pelo trabalho em ambos casos, tal que

$$c_p[A] \leq \max(c_p[v := ordena(u)], c_p[s := soma(u)])$$

e com, por exemplo, $c_p[v := ordena(u)](n) = n^2$ e $c_p[s := soma(u)](n) = n$ temos

$$c_p[A](n) \leq n^2$$

$\Diamond$

2.2.2. Exemplos

Exemplo 2.6 (Bubblesort)
Nesse exemplo vamos estudar o algoritmo Bubblesort de ordenação.

Bubblesort

Algoritmo 2.2 (Bubblesort)
Entrada Uma sequência $a_1, \ldots, a_n$ de números inteiros.

Saída Uma sequência $a_{\pi(1)}, \ldots, a_{\pi(n)}$ de números inteiros onde π uma permutação de $[1, n]$ tal que para $i < j$ temos $a_{\pi(i)} \leq a_{\pi(j)}$.

```
for i:=1 to n
  { Inv: a_{n-i+2} ≤ ··· ≤ a_n são os i−1 maiores elementos }
  for j:=1 to n−i
    if a_j > a_{j+1} then
      swap a_j, a_{j+1}
    end if
  end for
end for
```

Bubblesort: Complexidade

- A medida comum para algoritmos de ordenação: o número de comparações (de chaves).
- Qual a complexidade pessimista?

$$\sum_{i \in [n]} \sum_{j \in [n]-i} 1 = n(n-1)/2.$$

- Qual a diferença se contamos as transposições também? (Ver exemplo 2.15.)

$\Diamond$

Exemplo 2.7 (Ordenação por inserção direta)
(inglês: straight insertion sort)

Ordenação por inserção direta

Algoritmo 2.3 (Ordenação por inserção direta)
Entrada Uma sequência $a_1, \ldots, a_n$ de números inteiros.

Saída Uma sequência $a_{\pi(1)}, \ldots, a_{\pi(n)}$ de números inteiros tal que π é uma permutação de $[1, n]$ e para $i < j$ temos $a_{\pi(i)} \leq a_{\pi(j)}$.

```
for i:=2 to n do
  { invariante: a1,...,ai−1 ordenado }
  { coloca item i na posição correta }
  c:=ai
  j:=i;
  while c < aj−1 and j > 1 do
    aj:=aj−1
    j:=j−1
  end while
  aj:=c
end for
```

(Nesse algoritmo é possível eliminar o teste $j > 1$ na linha 6 usando um elemento auxiliar $a_0 = -\infty$.)
Para a complexidade pessimista obtemos

$$c_p[SI](n) \leq \sum_{2 \leq i \leq n} \sum_{1 < j \leq i} O(1) = \sum_{2 \leq i \leq n} O(i) = O(n^2)$$

◊

Exemplo 2.8 (Máximo)
(Ver Toscani e Veloso (2005, cap. 3.3).)

Máximo

Algoritmo 2.4 (Máximo)
Entrada Uma sequência de números $a_1, \ldots, a_n$ com $n > 0$.

Saída O máximo $m = \max_i a_i$.

```
m := a1
for i := 2,...,n do
  if ai > m then
    m := ai
  end if
end for
return m
```

Para a análise supomos que toda operação básica (atribuição, comparação, aritmética) têm um custo constante. Podemos obter uma cota superior simples de $O(n)$ observando que o laço sempre executa um número fixo de operações (ao máximo dois no corpo). Para uma análise mais detalhada vamos denotar o custo em números de operações de cada linha como l_i e supomos que toda operação básico custa 1 e a linha 2 do laço custa dois ($l_2 = 2$, para fazer um teste e um incremento), então temos

$$l_1 + (n-1)(l_2 + l_3) + kl_4 + l_7 = 3n + k - 1$$

com um número de execuções da linha 4 ainda não conhecido k. No melhor caso temos $k = 0$ e custos de $3n - 1$. No pior caso $m = n - 1$ e custos de $4n - 2$. É fácil ver que assintoticamente todos os casos, inclusive o caso médio, têm custo $\Theta(n)$. ◊

Exemplo 2.9 (Busca sequencial)
O segundo algoritmo que queremos estudar é a busca sequencial.

Busca sequencial

Algoritmo 2.5 (Busca sequencial)
Entrada Uma sequência de números $a_1, \ldots, a_n$ com $n > 0$ e um chave c.

Saída A primeira posição p tal que $a_p = c$ ou $p = \infty$ caso não existe tal posição.

```
for i := 1,...,n do
  if a_i = c then
    return i
  end if
end for
return ∞
```

Busca sequencial

- Fora do laço nas linhas 1–5 temos uma contribuição constante.

- Caso a sequência não contém a chave c, temos que fazer n iterações.
- Logo temos complexidade pessimista $\Theta(n)$.

$\Diamond$

Counting-Sort

Algoritmo 2.6 (Counting-Sort)
Entrada Um inteiro k, uma sequência de números $a_1, \ldots, a_n$ e uma sequência de contadores $c_1, \ldots, c_n$.

Saída Uma sequência ordenada de números $b_1, \ldots, b_n$.

```
for i := 1,...,k do
  c_i := 0
end for
for i := 1,...,n do
  c_{a_i} := c_{a_i} + 1
end for
for i := 2,...,k do
  c_i := c_i + c_{i-1}
end for
for i := n,...,1 do
  b_{c_{a_i}} := a_i
  c_{a_i} := c_{a_i} - 1
end for
return b_1,...,b_n
```

Loteria Esportiva

Algoritmo 2.7 (Loteria Esportiva)
Entrada Um vetor de inteiros $r[1, \ldots, n]$ com o resultado e uma matriz A_{13xn} com as apostas dos n jogadores.

Saída Um vetor $p[1, \ldots, n]$ com os pontos feitos por cada apostador.

```
i := 1
while i ≤ n do
    p_i := 0
    for j de 1,...,13 do
        if A_{i,j} = r_j then p_i := p_i + 1
```

```
    end for
    i := i + 1
end while
for i de 1,...,n do
  if p_i = 13 then print(Apostador i é ganhador!)
end for
return p
```

Exemplo 2.10 (Busca Binária)

Busca Binária

Algoritmo 2.8 (Busca Binária)
Entrada Um inteiro x e uma sequência ordenada $S = a_1, a_2, \ldots a_n$ de números.

Saída Posição i em que x se encontra na sequência S ou -1 caso $x \notin S$.

```
i := 1
f := n
m := ⌊(f + i)/2⌋
while i ≤ f do
    if a_m = x then return m
    if a_m < x then f := m − 1
    else i := m + 1
    m := ⌊(f + i)/2⌋
end while
return −1
```

A busca binária é usada para encontrar um dado elemento numa sequência ordenada de números com gaps. Ex: 3,4,7,12,14,18,27,31,32...n. Se os números não estiverem ordenados um algoritmo linear resolveria o problema, e no caso de números ordenados e sem gaps (nenhum número faltante na sequência, um algoritmo constante pode resolver o problema. No caso da busca binária, o pior caso acontece quando o último elemento que for analisado é o procurado. Neste caso a sequência de dados é dividida pela metade até o término da busca, ou seja, no máximo $\log_2 n = x$ vezes, ou seja $2^x = n$.

Neste caso

$$C_p[A] = \sum_{1 \le i \le \log_2 n} c = O(\log_2 n)$$

◊

Exemplo 2.11 (Multiplicação de matrizes)
O algoritmo padrão da computar o produto $C = AB$ de matrizes (de tamanho $m \times n$, $n \times o$) usa a definição

$$c_{ik} = \sum_{j \in [n]} a_{ij} b_{jk} \qquad i \in [m]; k \in [o].$$

Algoritmo 2.9 (Multiplicação de matrizes)
Entrada Duas matrizes $A = (a_{ij}) \in \mathbb{R}^{m \times n}$, $B = (b_{jk}) \in \mathbb{R}^{n \times o}$.

Saída O produto $C = (c_{ik}) = AB \in \mathbb{R}^{m \times o}$.

```
for i := 1,...,m do
  for k := 1,...,o do
    c_ik := 0
    for j := 1,...,n do
      c_ik := c_ik + a_ij b_jk
    end for
  end for
end for
```

No total, precisamos $mno(M + A)$ operações, com M denotando multiplicações e A adições. É costume estudar a complexidade no caso $n = m = o$ e somente considerar as multiplicações, tal que temos uma entrada de tamanho $\Theta(n^2)$ e $\Theta(n^3)$ operações[1]. ◊

2.3. Complexidade média

Nesse capítulo, vamos estudar algumas técnicas de análise da complexidade média.

[1] Também é de costume contar as operações de ponto flutuante diretamente e não em relação ao tamanho da entrada. Senão a complexidade seria $2n/3^{3/2} = O(n^{3/2})$.

Motivação

- A complexidade pessimista é pessimista demais?
- Imaginável: poucas instâncias representam o pior caso de um algoritmo.
- Isso motiva a estudar a complexidade média.
- Para tamanho n, vamos considerar
  - O espaço amostral $D_n = \{d \in D \mid \text{tam}(d) = n\}$
  - Uma distribuição de probabilidade Pr sobre D_n
  - A variável aleatória desemp$[A]$
  - O custo médio

$$C_m[A](n) = E[\text{desemp}[A]] = \sum_{d \in D_n} P(d)\,\text{desemp}[A](d)$$

Tratabilidade?

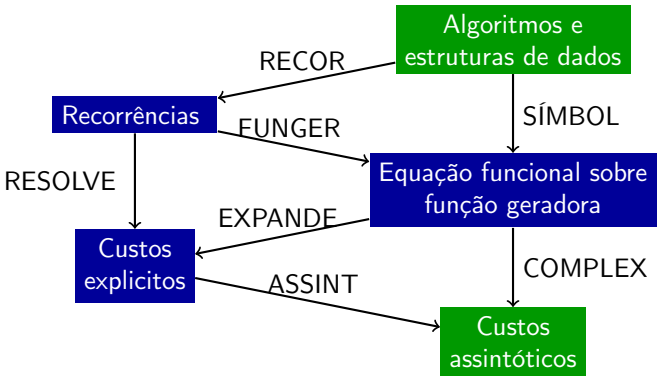
- Possibilidade: Problemas intratáveis viram tratáveis?
- Exemplos de tempo esperado:
  - Caminho Hamiltoniano: linear!
  - Parada não-determinístico em k passos: fica NP-completo.

(Resultados citados: (Gurevich e Shelah 1987; Du e Ko 1997) (Caminho Hamiltoniano), (Wang 1997) (Parada em k passos).)

Criptografia

- Alguns métodos da Criptografia dependem da existência de "funções sem volta" (inglês: one-way functions).
- Uma função sem volta $f : \{0,1\}^* \to \{0,1\}^*$ é tal que
  - dado x, computar $f(x)$ é fácil (eficiente)
  - dada $f(x)$ achar um x' tal que $f(x') = f(x)$ é difícil

Método



(Vitter e Flajolet 1990)

Exemplo 2.12 (Busca sequencial)
(Continuando exemplo 2.9.)

Busca sequencial

- Caso a chave esteja na i-ésima posição, temos que fazer i iterações.
- Caso a chave não ocorra no conjunto, temos que fazer n iterações.
- Supondo uma distribuição uniforme da posição da chave na sequência, temos
$$c_n[BS](n) = \frac{1}{n+1}(n + \sum_{i \in [n]} i)$$
iterações e uma complexidade média de $\Theta(n)$.

$\Diamond$

Exemplo 2.13
(Continuando o exemplo 2.1.)
Neste exemplo vamos analisar o algoritmo considerando que a ocorrência dos números siga uma outra distribuição que não a uniforme. Ainda, considere o caso em que não há números repetidos no conjunto. Seja n um tamanho fixo. Para BUSCA1 temos o espaço amostral $D_n = \{(a_1, \ldots, a_n) \mid a_1 \geq 1, \ldots, a_n \geq 1\}$. Supomos que os números sigam uma distribuição na qual cada elemento da sequência é gerado independentemente com a probabilidade $Pr(a_i = n) = 2^{-n}$ (que é possível porque $\sum_{1 \leq i} 2^{-i} = 1$).

Com isso temos

$$\Pr((a_1, \ldots, a_n)) = \prod_{i \in [n]} 2^{-a_i}$$

e, $\Pr(a_i = 1) = 1/2$ e $\Pr(a_i \neq 1) = 1/2$. Considere as variáveis aleatórias desemp$[A]$ e

$$p(d) = \begin{cases} \mu, & \text{se o primeiro 1 em } d \text{ está na posição } \mu, \\ n, & \text{caso contrário,} \end{cases}$$

Temos desemp$[A] = 2p + 1$ (veja os resultados do exemplo 2.1). Para i estar na primeira posição com elemento 1 as posições $1, \ldots, i-1$ devem ser diferente de 1, e a_i deve ser 1. Logo para $1 \leq i < n$

$$\Pr(p = i) = \Pr(a_1 \neq 1) \cdots \Pr(a_{i-1} \neq 1) \Pr(a_i = 1) = 2^{-i}.$$

O caso $p = n$ tem duas causas: ou a posição do primeiro 1 é n ou a sequência não contém 1. Ambas têm probabilidade 2^{-n} e logo $\Pr(p = n) = 2^{1-n}$.
A complexidade média calcula-se como

$$\begin{aligned} c_p[A](n) &= E[\text{desemp}[A]] = E[2p + 1] = 2E[p] + 1 \\ E[p] &= \sum_{i \geq 0} \Pr(p = i) i = 2^{-n} n + \sum_{i \in [n]} 2^{-n} n \\ &= 2^{-n} n + 2 - 2^{-n}(n + 2) = 2 - 2^{1-n} \qquad \text{(A.37)} \\ c_p[A](n) &= 5 - 2^{2-n} = O(1) \end{aligned}$$

A seguinte tabela mostra os custos médios para $n \in [9]$

n	1	2	3	4	5	6	7	8	9
C_m	3	4	4.5	4.75	4.875	4.938	4.969	4.984	4.992

◊

Exemplo 2.14 (Ordenação por inserção direta)
(Continuando exemplo 2.7.)

Ordenação por inserção direta

- Qual o número médio de comparações?
- Observação: Com as entradas distribuídas uniformemente, a posição da chave i na sequência já ordenada também é.

- Logo chave i precisa

$$\sum_{j \in [i]} j/i = (i+1)/2$$

comparações em média.

- Logo o número esperado de comparações é

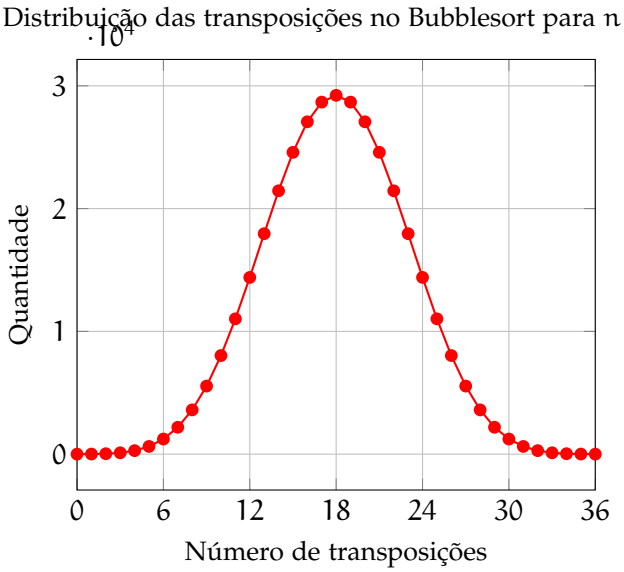
$$\sum_{2 \le i \le n} (i+1)/2 = 1/2 \sum_{3 \le i \le n+1} i = 1/2\,((n+1)(n+2)/2 - 3) = \Theta(n^2)$$

$\Diamond$

Exemplo 2.15 (Bubblesort)
(Continuando exemplo 2.6.)
O número de comparações do Bubblesort é independente da entrada. Logo, com essa operação básica temos uma complexidade pessimista e média de $\Theta(n^2)$.

Bubblesort: Transposições

- Qual o número de transposições em média?



Qual a complexidade contando o número de transposições? A figura acima mostra no exemplo das instâncias com tamanho 9, que o trabalho varia entre 0 transposições (sequência ordenada) e 36 transposições (pior caso) e na maioria dos casos, precisamos 18 transposições. A análise no caso em que todas entradas são permutações de $[1, n]$ distribuídas uniformemente resulta em $n/4(n-1)$ transposições em média, a metade do pior caso.

Inversões

- Uma *inversão* em uma permutação é um par que não está ordenado, i.e.

$$i < j \quad \text{tal que} \quad a_i > a_j.$$

- Exemplos

Permutação	#Inversões
123	0
132	1
213	1
231	2
312	2
321	3

- Frequentemente o número de inversões facilita a análise de algoritmos de ordenação.
- Em particular para algoritmos com transposições de elementos adjacentes: #Transposições = # Inversões

Número médio de transposições

- Considere o conjunto de todas permutações S_n sobre $[1, n]$.
- Denota-se por $\text{inv}(\pi)$ o número de inversões de uma permutação.
- Para cada permutação π existe uma permutação π^- correspondente com ordem inversa:

$$35124; \qquad 42153$$

- Cada inversão em π não é inversão em π^- e vice versa:

$$\text{inv}(\pi) + \text{inv}(\pi^-) = n(n-1)/2.$$

Número médio de transposições

- O número médio de inversões é

$$\begin{aligned}
1/n! \sum_{\pi \in S_n} \mathrm{inv}(\pi) &= 1/(2n!) \left(\sum_{\pi \in S_n} \mathrm{inv}(\pi) + \sum_{\pi \in S_n} \mathrm{inv}(\pi) \right) \\
&= 1/(2n!) \left(\sum_{\pi \in S_n} \mathrm{inv}(\pi) + \sum_{\pi \in S_n} \mathrm{inv}(\pi^{-}) \right) \\
&= 1/(2n!) \left(\sum_{\pi \in S_n} n(n-1)/2 \right) \\
&= n(n-1)/4
\end{aligned}$$

- Logo, a complexidade média (de transposições) é $\Theta(n^2)$.

$\Diamond$

Exemplo 2.16 (Máximo)
(Continuando exemplo 2.8.)
Queremos analisar o número médio de atribuições no cálculo do máximo, i.e. o número de atualizações do máximo.

Máximo

- Qual o número esperado de atualizações no algoritmo Máximo?
- Para uma permutação π considere a *tabela de inversões* $b_1, \ldots, b_n$.
- b_i é o número de elementos na esquerda de i que são maiores que i.
- Exemplo: Para 53142

b_1	b_2	b_3	b_4	b_5
2	3	1	1	0

- Os b_i obedecem $0 \leq b_i \leq n - i$.

Tabelas de inversões

- Observação: Cada tabela de inversões corresponde a uma permutação e vice versa.
- Exemplo: A permutação correspondente com

b_1	b_2	b_3	b_4	b_5
3	1	2	1	0

- Vantagem para a análise: Podemos escolher os b_i independentemente.
- Observação, na busca do máximo i é máximo local se todos números no seu esquerdo são menores, i.e. se todos números que são maiores são no seu direito, i.e. se $b_i = 0$.

Número esperado de atualizações

- Seja X_i a variável aleatória $X_i = [i \text{ é máximo local}]$.
- Temos $\Pr(X_i = 1) = \Pr(b_i = 0) = 1/(n-i+1)$.
- O número de máximos locais é $X = \sum_{i\in[n]} X_i$.
- Portanto, o número esperado de máximos locais é

$$\begin{aligned} \mathrm{E}[X] &= \mathrm{E}\left[\sum_{i\in[n]} X_i\right] = \sum_{i\in[n]} \mathrm{E}[X_i] \\ &= \sum_{i\in[n]} \Pr(X_i) = \sum_{i\in[n]} \frac{1}{n-i+1} = \sum_{i\in[n]} \frac{1}{i} = H_n \end{aligned}$$

- Contando atualizações: tem uma a menos que os máximos locais $H_n - 1$.

$\Diamond$

Exemplo 2.17 (Quicksort)
Nessa seção vamos analisar é Quicksort, um algoritmo de ordenação que foi inventado pelo C.A.R. Hoare em 1960 (Hoare 1962).

Quicksort

- Exemplo: o método Quicksort de ordenação por comparação.
- Quicksort usa divisão e conquista.
- Ideia:
  - Escolhe um elemento (chamado pivô).
  - Divide: Particione o vetor em elementos menores que o pivô e maiores que o pivô.
  - Conquiste: Ordene as duas partições recursivamente.

Particionar

Algoritmo 2.10 (Partition)
Entrada Índices l, r e um vetor a com elementos $a_l, \ldots, a_r$.

Saída Um índice $m \in [l, r]$ e a com elementos ordenados tal que $a_i \leq a_m$ para $i \in [l, m[$ e $a_m \leq a_i$ para $i \in]m, r]$.

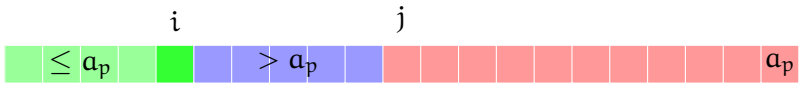
```
escolhe um pivô a_p
troca a_p e a_r
i := l−1 { último índice menor que pivô }
for j:=l to r−1 do
  if a_j ≤ a_r then
    i:=i+1
    troca a_i e a_j
  end if
end for
troca a_{i+1} e a_r
return i+1
```

Escolher o pivô

- Partition combina os primeiros dois passos do Quicksort..
- Operações relevantes: *Número de comparações* entre chaves!
- O desempenho de Partition depende da escolha do pivô.

- Dois exemplos
  - Escolhe o primeiro elemento.
  - Escolhe o maior dos primeiros dois elementos.
- Vamos usar a segunda opção.

Complexidade de particionar



- O tamanho da entrada é $n = r - l + 1$
- Dependendo da escolha do pivô: precisa nenhuma ou uma comparação.
- O laço nas linhas 4–9 tem $n - 1$ iterações.
- O trabalho no corpo do laço é $1 = \Theta(1)$ (uma comparação)
- Portanto temos a complexidade pessimista

$$c_p[\text{Partition}] = n - 1 = \Theta(n) \qquad \text{(Primeiro elemento)}$$
$$c_p[\text{Partition}] = n - 1 + 1 = n = \Theta(n) \qquad \text{(Maior de dois).}$$

Quicksort

Algoritmo 2.11 (Quicksort)
Entrada Índices l, r e um vetor a com elementos $a_l, \ldots, a_r$.

Saída a com os elementos em ordem não-decrescente, i.e. para $i < j$ temos $a_i \leq a_j$.

```
if l < r then
  m := Partition(l,r,a);
  Quicksort(l,m-1,a);
  Quicksort(m+1,r,a);
end if
```

$$\begin{aligned}\text{desemp}[QS](a_l,\ldots,a_r) &= \text{desemp}[P](a_l,\ldots,a_r)\\ &\quad + \text{desemp}[QS](a_l,\ldots,a_{m-1}) + \text{desemp}[QS](a_{m+1},\ldots,a_r)\end{aligned}$$

$$\text{desemp}[QS](a_l,\ldots,a_r) = 0 \qquad \text{se } l \geq r$$

Complexidade pessimista

- Qual a complexidade pessimista?
- Para entrada $d = (a_1,\ldots,a_n)$, sejam $d_l = (a_l,\ldots,a_{m-1})$ e $d_r = (a_{m+1},\ldots,a_r)$

$$\begin{aligned}c_p[QS](n) &= \max_{d\in D_n} \text{desemp}[P](d) + \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)\\ &= n + \max_{d\in D_n} \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)\\ &= n + \max_{i\in[n]} c_p[QS](i-1) + c_p[QS](n-i)\end{aligned}$$

- $c_p[QS](0) = c_p[QS](1) = 0$

Esse análise é válida para escolha do maior entre os dois primeiros elementos como pivô. Também vamos justificar o último passo na análise acima com mais detalhes. Seja $D_n = \bigcup_i D_n^i$ uma partição das entradas com tamanho n tal que para $d \in D_n^i$ temos $|d_l| = i-1$ (e consequentemente $|d_r| = n-i$). Então

$$\begin{aligned}&\max_{d\in D_n} \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)\\ =&\max_{i\in[n]}\max_{d\in D_n^i} \text{desemp}[QS](d_l) + \text{desemp}[QS](d_r) \qquad \text{separando } D_n\\ =&\max_{i\in[n]}\max_{d\in D_n^i} c_p[QS](i-1) + c_p[QS](n-i)\end{aligned}$$

e o último passo é justificado, porque a partição de uma permutação aleatória gera duas partições aleatórias independentes, e existe uma entrada d em que as duas sub-partições assumem o máximo. Para determinar o máximo da última expressão, podemos observar que ele deve ocorrer no índice $i = 1$ ou $i = \lfloor n/2 \rfloor$ (porque a função $f(i) = c_p[QS](i-1) + c_p[QS](n-i)$ é simétrico com eixo de simetria $i = n/2$).

Complexidade pessimista

- O máximo ocorre para $i = 1$ ou $i = n/2$
- Caso $i = 1$

$$\begin{aligned} c_p[QS](n) &= n + c_p[QS](0) + c_p[QS](n-1) = n + c_p[QS](n-1) \\ &= \cdots = \sum_{i \in [n]} (i-1) = \Theta(n^2) \end{aligned}$$

- Caso $i = n/2$

$$\begin{aligned} c_p[QS](n) &= n + 2c_p[QS](n/2) = n - 1 + 2((n-1)/2 + c_p(n/4)) \\ &= \cdots = \Theta(n \log_2 n) \end{aligned}$$

- Logo, no pior caso, Quicksort precisa $\Theta(n^2)$ comparações.
- No caso bem balanceado: $\Theta(n \log_2 n)$ comparações.

Complexidade média

- Seja X a variável aleatória que denota a posição (inglês: rank) do pivô a_m na sequência.
- Vamos supor que todos elementos a_i são diferentes (e, sem perda da generalidade, uma permutação de $[1, n]$).

$$\begin{aligned} c_m[QS](n) &= \sum_{d \in D_n} Pr(d)\, desemp[QS](d) \\ &= \sum_{d \in D_n} Pr(d)(desemp[P](d) + desemp[QS](d_l) + desemp[QS](d_r)) \\ &= n + \sum_{d \in D_n} Pr(d)(desemp[QS](d_l) + desemp[QS](d_r)) \\ &= n + \sum_{i \in [n]} Pr(X = i)(c_m[QS](i-1) + c_m[QS](n-i)) \end{aligned}$$

Novamente, o último passo é o mais difícil de justificar. A mesma partição que aplicamos acima leva a

$$\begin{aligned}
&\sum_{d\in D_n} \Pr(d)(\text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)) \\
&= \sum_{i\in[n]} \sum_{d\in D_n^i} \Pr(d)(\text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)) \\
&= \sum_{i\in[n]} \frac{1}{|D|} \sum_{d\in D_n^i} (\text{desemp}[QS](d_l) + \text{desemp}[QS](d_r)) \\
&= \sum_{i\in[n]} \frac{|D_n^i|}{|D|} (c_m[QS](i-1) + c_m[QS](n-i)) \\
&= \sum_{i\in[n]} \Pr(X=i)(c_m[QS](i-1) + c_m[QS](n-i))
\end{aligned}$$

é o penúltimo passo é correto, porque a média do desempenho sobre as permutações d_l e d_r é a mesma que sobre as permutações com $i-1$ e $n-i$ elementos: toda permutação ocorre com a mesma probabilidade e o mesmo número de vezes (Knuth (1998, p. 119) tem mais detalhes).
Se denotamos o desempenho com $T_n = c_m[QS](n)$, obtemos a recorrência

$$T_n = n + \sum_{i\in[n]} \Pr(X=i)\,(T_{i-1} + T_{n-i})$$

com base $T_n = 0$ para $n \leq 1$. A probabilidade de escolher o i-ésimo elemento como pivô depende da estratégia da escolha. Vamos estudar dois casos.

(a) Escolhe o primeiro elemento como pivô. Temos $\Pr(X = i) = 1/n$. Como $\Pr(X = i)$ não depende do i a equação acima vira

$$T_n = n - 1 + 2/n \sum_{0\leq i<n} T_i$$

(com uma comparação a menos no particionamento).

(b) Escolhe o maior dos dois primeiros elementos[2]. Temos $\Pr(X = i) = 2(i-1)/(n(n-1))$.

[2]Supomos para análise que todos elementos são diferentes. Um algoritmo prático tem que considerar o caso que um ou mais elementos são iguais (Toscani e Veloso 2005, p. 72).

$$
\begin{aligned}
T_n &= n + 2/(n(n-1)) \sum_{i \in [n]} (i-1)\,(T_{i-1} + T_{n-i}) \\
&= n + 2/(n(n-1)) \sum_{0 \le i < n} i\,(T_i + T_{n-i-1}) \\
&= n + 2/(n(n-1)) \sum_{0 \le i < n} iT_i + \sum_{0 \le i < n} iT_{n-i-1} \\
&= n + 2/(n(n-1)) \sum_{0 \le i < n} iT_i + \sum_{0 \le i < n} (n-i-1)T_i \\
&= n + 2/n \sum_{0 \le i < n} T_i
\end{aligned}
$$

Recorrência

- A solução final depende da escolha do pivô.
- Dois exemplos
  - Escolhe o primeiro elemento: $\Pr(X = i) = 1/n$.
  - Escolhe o maior dos primeiros dois elementos diferentes: $\Pr(X = i) = 2(i-1)/(n(n-1))$.
- Denota $T_n = c_m[QS](n)$
- Ambas soluções chegam (quase) na mesma equação recorrente

$$T_n = n + 2/n \sum_{0 \le i < n} T_i$$

Exemplo 2.18
Vamos determinar a probabilidade de escolher o pivô $\Pr(X = i)$ no caso $n = 3$ explicitamente, se o maior dos dois primeiros elementos é o pivô:

Permutação	Pivô
123	2
132	3
213	2
231	3
312	3
321	3

Logo temos as probabilidades

Pivô i	1	2	3
$Pr(X = i)$	0	1/3	2/3

$\Diamond$

Resolver a equação

- A solução da recorrência

$$T_n = n + 1 + 2/n \sum_{0 \le i < n} T_i$$

é $T_n = \Theta(n \ln n)$.

- Logo, em ambos casos temos a complexidade média de $\Theta(n \ln n)$.

$$T_n = n + 1 + 2/n \sum_{0 \le i < n} T_i \qquad \text{para } n > 0$$

multiplicando por n obtemos

$$nT_n = n^2 + n + 2 \sum_{0 \le i < n} T_i$$

a mesmo equação para $n - 1$ é

$$(n-1)T_{n-1} = (n-1)^2 + n - 1 + 2 \sum_{0 \le i < n-1} T_i \qquad \text{para } n > 1$$

subtraindo a segunda da primeira obtemos

$$nT_n - (n-1)T_{n-1} = 2n + 2T_{n-1} \qquad \text{para } n > 0, \text{ verificando } n = 1$$
$$nT_n = (n+1)T_{n-1} + 2n$$

multiplicando por $2/(n(n+1))$

$$\frac{2}{n+1}T_n = \frac{2}{n}T_{n-1} + \frac{4}{n+1}$$

substituindo $A_n = 2T_n/(n+1)$

$$A_n = A_{n-1} + \frac{2}{n+1} = \sum_{i \in [n]} \frac{4}{i+1}$$

e portanto

$$\begin{aligned} T_n &= 2(n+1) \sum_{i \in [n]} \frac{1}{i+1} \\ &= 2(n+1)(H_n - \frac{n}{n+1}) \\ &= \Theta(n \ln n) \end{aligned}$$

$\Diamond$

2.4. Outros típos de análise

2.4.1. Análise agregada

Para alguns algoritmos uma análise pessimista baseada na análise pessimista dos componentes resulta em uma complexidade "demais pessimista".

Exemplo 2.19 (Busca em Largura)

Busca em Largura

Algoritmo 2.12 (Busca em Largura)
Entrada Um nó origem s e um grafo não-direcionado $G = (V, E)$.

Saída Distância mínima (em número de arcos) d_v de s para cada vértice v.

$d_s := 0$
$d_u = \infty, \forall u \in V - \{s\}$
$Q := \emptyset$
Enqueue(Q, s)
`while` $Q \neq \emptyset$
  $u :=$ Dequeue(Q)
  `for cada` $v \in N(u)$

```
      if d_v = ∞ then
        d_v = d_u + 1
        Enqueue(Q, v)
      end if
    end for
  end while
```

Uma análise simples observa que o laço *while* nas linhas 7–13 executa no máximo $|V|$ iterações, uma para cada nó do grafo. O laço *for* interior nas linhas 7–12 executa d_u iterações, sendo d_u o grau do nó u. No caso pessimista $d_u = |V| - 1$, portanto esta análise resulta numa complexidade pessimista $O(|V|^2)$, supondo que "Enqueue" e "Dequeue" tem complexidade $O(1)$.
Uma *análise agregada* separa a análise do laço exterior, que tem complexidade pessimista $O(|V|)$, da análise do laço interior. O laço interior, tem, agregado sobre todas iterações do laço exterior, complexidade $\sum_{u \in V} d_u \leq 2|E|$. Portanto essa análise resulta numa complexidade pessimista $O(|V| + |E|)$. ◊

Exemplo 2.20

Algoritmo 2.13 (Dijkstra)
Entrada Grafo direcionado $G = (V, E)$ com pesos $c_e, e \in E$ nas arestas, e um vértice $s \in V$.

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

$d_s := 0; d_v := \infty, \forall v \in V \setminus \{s\}$
visited(v) := false, $\forall v \in V$
$Q := \emptyset$
insert$(Q, (s, 0))$
```
while Q ≠ ∅ do
  v := deletemin(Q)
  visited(v) := true
  for u ∈ N⁺(v) do
    if not visited(u) then
      if d_u = ∞ then
        d_u := d_v + d_vu
        insert(Q, (u, d_u))
      else if d_v + d_vu < d_u
```

```
            d_u := d_v + d_vu
            update(Q, (u, d_u))
          end if
        end if
      end for
    end while
```

Pelo mesmo argumento dado na análise da busca por profundidade, podemos ver

Proposição 2.1
O algoritmo de Dijkstra possui complexidade

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

$\Diamond$

Usando uma implementação da fila de prioridade por um heap binário que realiza todas operações em $O(\log n)$, a complexidade pessimista do algoritmo de Dijkstra é $O((m+n)\log n)$.

O caso médio do algoritmo de Dijkstra Dado um grafo $G = (V, E)$ e um vértice inicial arbitrário supõe que temos um conjunto $C(v)$ de pesos positivos com $|C(v)| = |N^-(v)|$ para cada $v \in V$. Atribuiremos permutações dos pesos em $C(v)$ aleatoriamente para os arcos entrantes em v.

Proposição 2.2 (Noshita (1985))
O algoritmo de Dijkstra chama update em média $n \log(m/n)$ vezes neste modelo.

Prova. Para um vértice v os arcos que podem levar a uma operação update em v são de forma (u, v) com $\text{dist}(s, u) \leq \text{dist}(s, v)$. Supõe que existem k arcos $(u_1, v), \ldots, (u_k, v)$ desse tipo, ordenado por $\text{dist}(s, u_i)$ não-decrescente. Independente da atribuição dos pesos aos arcos, a ordem de processamento é o mesmo. O arco (u_i, v) leva a uma operação update caso

$$\text{dist}(s, u_i) + d_{u_i v} < \min_{j:j<i} \text{dist}(s, u_j) + d_{u_j v}.$$

Com isso temos $d_{u_i v} < \min_{j:j<i} d_{u_j v}$, i.e., $d_{u_i v}$ é um mínimo local na sequência dos pesos dos k arcos. Pela análise no exemplo 2.16 sabemos que o número

esperado de máximos locais de uma permutação aleatória é $H_k - 1 \leq \ln k$ e considerando as permutações inversas, temos o mesmo número de mínimos locais. Como $k \leq \delta^-(v)$ temos um limite superior para o número de operações update em todos vértices de

$$\sum_{v \in V} \ln \delta^-(v) = n \sum_{v \in V} (1/n) \ln \delta^-(v) \leq n \ln \sum_{v \in V} (1/n)\delta^-(v) = n \ln m/n.$$

A desigualdade é justificada pela equação (A.23) observando que $\ln n$ é concava. ■

Com isso complexidade média do algoritmo de Dijkstra é

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

Usando uma fila de prioridade implementada por um heap binário que executa todas operações em $O(\log n)$ a complexidade média do algoritmo de Dijkstra é $O(m + n \log m/n \log n)$.

2.4.2. Análise amortizada

Exemplo 2.21

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

Uma outra forma da análise amortizada é através uma *função potencial* 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

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

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 chamadas de uma operação com custos $c_1, \ldots, c_n$ o custo amortizado da operação é $\sum_{1 \le i \le n} c_i / n$. Caso temos m operações diferentes, o custo amortizado da operação que ocorre nos índices $J \subseteq [1, m]$ é $\sum_{i \in J} c_i / |J|$.
As somas podem ser difíceis de avaliar diretamente. Um método para simplificar o cálculo do custo amortizado é o *método potencial*. Acha uma *função potencial* 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.

Exemplo 2.22
Queremos implementar uma tabela dinâmica para um número desconhecido de elementos. Uma estratégia é reservar espaço para n elementos, manter a última posição livre p, e caso $p > n$ alocara uma nova tabela de tamanho maior. Uma implementação dessa ideia é

```
insert(x):=
  if p > n then
    aloca nova tabela de tamanho t = max{2n, 1}
    copia os elementos x_i, 1 ≤ i < p para nova tabela
    n := t
  end if
  x_p := x
  p := p + 1
```

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

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

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

Caso temos que alocar uma nova tabela o custo é

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

$\Diamond$

2.5. Notas

O algoritmo 2.9 para multiplicação de matrizes não é o melhor possível: no capítulo 6.3 veremos um algoritmo proposto por Strassen (1969) que precisa somente $n^{\log_2 7} \approx n^{2.807}$ multiplicações. Ainda melhor é o algoritmo de Coppersmith e Winograd (1987) com $n^{2.376}$ multiplicações. Ambos algoritmos são pouco usados na prática porque seu desempenho real é melhor somente para n grande (no caso de Strassen $n \approx 700$; no caso de Coppersmith-Winograd o algoritmo não é praticável). O melhor algoritmo conhecido tem expoente 2.3727 (*Breaking the Coppersmith-Winograd barrier*).

2.6. Exercícios

(Soluções a partir da página 436.)

Exercício 2.1
Qual a complexidade pessimista dos seguintes algoritmos?

Algoritmo 2.14 (Alg1)
Entrada Um problema de tamanho n.

```
for i := 1...n do
  for j := 1...2^i
    operações constantes
```

```
    j := j + 1 //iterações com valores ímpares de j
  end for
end for
```

Algoritmo 2.15 (Alg2)
Entrada Um problema de tamanho n.

```
for i := 1...n do
  for j := 1...2^i
    operações com complexidade O(j^2)
    j := j + 1
  end for
end for
```

Algoritmo 2.16 (Alg3)
Entrada Um problema de tamanho n.

```
for i := 1...n do
  for j := i...n
    operações com complexidade O(2^i)
  end for
end for
```

Algoritmo 2.17 (Alg4)
Entrada Um problema de tamanho n.

```
for i := 1...n do
  j := 1
  while j ≤ i do
    operações com complexidade O(2^j)
    j := j+1
  end for
end for
```

Algoritmo 2.18 (Alg5)
Entrada Um problema de tamanho n.

```
for i := 1...n do
  j := i
  while j ≤ n do
    operações com complexidade O(2^j)
    j := j+1
  end for
end for
```

Exercício 2.2
Tentando resolver a recorrência

$$T_n = n - 1 + 2/n \sum_{0 \le i < n} T_i$$

que ocorre na análise do Quicksort (veja exemplo 2.17), o aluno J. Rapidez chegou no seguinte resultado: Supondo que $T_n = O(n)$ obtemos

$$\begin{aligned} T_n &= n - 1 + 2/n \sum_{0 \le i < n} O(i) \\ &= n - 1 + 2/n\, O(n^2) = n - 1 + O(n) = O(n) \end{aligned}$$

e logo, a complexidade média do Quicksort é $O(n)$. Qual o problema?

Exercício 2.3
Escreve um algoritmo que determina o segundo maior elemento de uma sequência $a_1, \ldots, a_n$. Qual a complexidade pessimista dele considerando uma comparação como operação básica?

Exercício 2.4
Escreve um algoritmo que, dado uma sequência $a_1, \ldots, a_n$ com $a_i \in \mathbb{N}$ determina um conjunto de índices $C \subseteq [1, n]$ tal que

$$\left| \sum_{i \in C} a_i - \sum_{i \notin C} a_i \right|$$

é mínimo. Qual a complexidade pessimista dele?

Exercício 2.5
Qual o número médio de atualizações no algoritmo

```
s := 0
for i = 1,...,n do
  if i > ⌊n/2⌋ then
    s:=s+i
  end if
end for
```

Exercício 2.6
Considere o algoritmo

Algoritmo 2.19 (Count6)
Entrada Uma sequência $a_1, \ldots, a_n$ com $a_i \in [1, 6]$.

Saída O número de elementos tal que $a_i = 6$.

```
k := 0
for i = 1,...,n do
  if a_i = 6 then
    k := k + 1
  end if
end for
```

Qual o número médio de atualizações $k := k + 1$, supondo que todo valor em cada posição da sequência tem a mesma probabilidade? Qual o número médio com a distribuição $\Pr(1) = 1/2$, $\Pr(2) = \Pr(3) = \Pr(4) = \Pr(5) = \Pr(6) = 1/10$?

Exercício 2.7
Suponha um conjunto de chaves numa árvore binária completa de k níveis e suponha uma busca binária tal que cada chave da árvore está buscada com a mesma probabilidade (em particular não vamos considerar o caso que uma chave buscada não pertence à árvore.). Tanto nós quanto folhas contém chaves. Qual o número médio de comparações numa busca?

Exercício 2.8
Usando a técnica para resolver a recorrência (veja p. 63)

$$T_n = n + 1 + 2/n \sum_{0 \le i < n} T_i$$

resolve as recorrências

$$T_n = n + 2/n \sum_{0 \le i < n} T_i$$
$$T_n = n - 1 + 2/n \sum_{0 \le i < n} T_i$$

explicitamente.

Exercício 2.9
Considere a seguinte implementação de uma fila usando duas pilhas. Uma pilha serve como pilha de entrada: um novo elemento sempre é inserido no topo dessa pilha. Outra pilha serve como pilha da saída: caso queremos remover um elemento da fila, removemos o topo da pilha de saída. Caso a pilha de saída é vazia, copiamos toda pilha de entrada para pilha de saída, elemento por elemento. (Observe que desta forma os elementos ficam na ordem reversa na pilha de saída).

1. Qual é a complexidade pessimista das operações "enqueue" (inserir um elemento a fila) e "dequeue" (remover um elemento da fila)?
2. Mostra que a complexidade amortizada de cada operação é $O(1)$.

Exercício 2.10
Mostra que $1 + n + n^2 + \cdots + n^{c-1} + n^c = \Theta(n^{c+1})$.

Exercício 2.11
Mostra que $\int x^p \log x = \Theta(x^{p+1} \log x)$.

Exercício 2.12
Compare as funções $\log_2 n^{\log_2 n}$, n^2 e $n^{\log_2 n}$ assintoticamente.

Parte II.

Projeto de algoritmos

3. Introdução

Resolver problemas

- *Modelar* o problema
  - Simplificar e abstrair
  - Comparar com problemas conhecidos
- *Inventar* um novo algoritmo
  - Ganhar experiência com exemplos
  - Aplicar ou variar técnicas conhecidas (mais comum)

Resolver problemas

- *Provar* a corretude do algoritmo
  - Testar só não vale
  - Pode ser informal
- *Analisar* a complexidade
- *Aplicar e validar*
  - Implementar, testar e verificar
  - Adaptar ao problema real
  - Avaliar o desempenho

4. Algoritmos gulosos

Radix omnium malorum est cupiditas.

(Seneca)

4.1. Introdução

(Veja Toscani e Veloso (2005, cap. 5.1.3).)

Algoritmos gulosos

- Algoritmos gulosos se aplicam a problemas de otimização.
- Ideia principal: Decide localmente.
- Um algoritmo guloso constrói uma solução de um problema
  - Começa com uma solução inicial.
  - Melhora essa solução com uma decisão *local* (gulosamente!).
  - Nunca revisa uma decisão.
- Por causa da localidade: Algoritmos gulosos frequentemente são apropriados para processamento online.

Trocar moedas

TROCA MÍNIMA

Instância Valores (de moedas ou notas) $v_1 > v_2 > \cdots > v_n = 1$, uma soma s.

Solução Números $c_1, \ldots, c_n$ tal que $s = \sum_{i \in [n]} c_i v_i$

Objetivo Minimizar o número de unidades $\sum_{i \in [n]} c_i$.

A abordagem gulosa

```
for i:=1,...,n do
  c_i := ⌊s/v_i⌋
  s := s − c_i v_i
end for
```

Exemplo

Exemplo 4.1
Com $v_1 = 500, v_2 = 100, v_3 = 25, v_4 = 10, v_5 = 1$ e $s = 3.14$, obtemos $c_1 = 0, c_2 = 3, c_3 = 0, c_4 = 1, c_5 = 4$.
Com $v_1 = 300, v_2 = 157, v_3 = 1$, obtemos $v_1 = 1, v_2 = 0, v_3 = 14$.
No segundo exemplo, existe uma solução melhor: $v_1 = 0, v_2 = 2, v_3 = 0$. No primeiro exemplo, parece que a abordagem gulosa acha a melhor solução. Qual a diferença? ◊

Uma condição simples é que todos valores maiores são múltiplos inteiros dos menores; essa condição não é necessária, porque o algoritmo guloso também acha soluções para outros sistemas de moedas, por exemplo no primeiro sistema do exemplo acima.

Lema 4.1
A solução do algoritmo guloso é a única que satisfaz

$$\sum_{i \in [m,n]} c_i v_i < v_{m-1}$$

para $m \in [2, n]$. (Ela é chamada a *solução canônica*.)

Proposição 4.1
Se $v_{i+1} | v_i$ para $1 \leq i < n$ a solução gulosa é mínima.

Prova. Sejam os divisores $v_i = f_i v_{i+1}$ com $f_i \geq 2$ para $1 \leq i < n$ e define $f_n = v_n = 1$. Logo cada valor tem a representação $v_i = f_i f_{i+1} f_{i+2} \cdots f_n$.
Seja $c_1, \ldots, c_n$ uma solução mínima. A contribuição de cada valor satisfaz $c_i v_i < v_{i-1}$ senão seria possível de substituir f_{i-1} unidades de v_i para uma de v_{i-1}, uma contradição com a minimalidade da solução (observe que isso somente é possível porque os f_i são números inteiros; senão o resto depois da substituição pode ser fracional e tem quer ser distribuído pelos valores

menores que pode causar um aumento de unidades em total). Logo $c_i \leq f_{i-1} - 1$ e temos

$$\begin{aligned}\sum_{i\in[m,n]} c_i v_i &\leq \sum_{i\in[m,n]} (f_{i-1} - 1)v_i \\ &= \sum_{i\in[m,n]} f_{i-1}v_i - \sum_{i\in[m,n]} v_i \\ &= \sum_{i\in[m,n]} v_{i-1} - \sum_{i\in[m,n]} v_i \\ &= v_{m-1} - v_n = v_{m-1} - 1 < v_{m-1}\end{aligned}$$

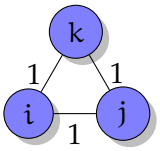
Agora aplique lema 4.1. ■

Otimalidade da abordagem gulosa

- A pergunta pode ser generalizada: Em quais circunstâncias um algoritmo guloso produz uma solução ótima?
- Se existe um solução gulosa: frequentemente ela tem uma implementação simples e é eficiente.
- Infelizmente, para um grande número de problemas não tem algoritmo guloso ótimo.
- Uma condição (que se aplica também para programação dinâmica) é a *subestrutura ótima*.
- A teoria de *matroides* e *greedoides* estuda as condições de otimalidade de algoritmos gulosos.

Definição 4.1 (Subestrutura ótima)
Um problema de otimização tem *subestrutura ótima* se uma solução ótima (mínima ou máxima) do problema consiste em soluções ótimas das subproblemas.

Exemplo 4.2
Considere caminhos (simples) em grafos. O caminho mais curto $v_1 v_2 \dots v_n$ entre dois vértices v_1 e v_n tem subestrutura ótima, porque um subcaminho também é mais curto (senão seria possível de obter um caminho ainda mais curto).
Do outro lado, o caminho mais longo entre dois vértices $v_1 \dots v_n$ não tem subestrutura ótima: o subcaminho $v_2 \dots v_n$, por exemplo, não precisa ser o caminho mais longo. Por exemplo no grafo



o caminho mais longo entre i e j é ikj, mas o subcaminho kj não é o subcaminho mais longo entre k e j. ◊

Para aplicar a definição 4.1 temos que conhecer (i) o conjunto de subproblemas de um problema e (ii) provar, que uma solução ótima contém uma (sub-)solução que é ótima para um subproblema. Se sabemos como estender uma solução de um subproblema para uma solução de todo problema, a subestrutura ótima fornece um algoritmo genérico da forma

---

Algoritmo 4.1 (Solução genérica de problemas com subestrutura ótima)
Entrada Uma instância I de um problema.

Saída Uma solução ótima S^* de I.

```
resolve(I):=
  S*:=nil                    { melhor solução }
  for todos subproblemas I' de I do
    S':=resolve(I')
    estende S' para uma solução S de I
    if S' é a melhor solução then
      S* := S
    end if
  end for
```

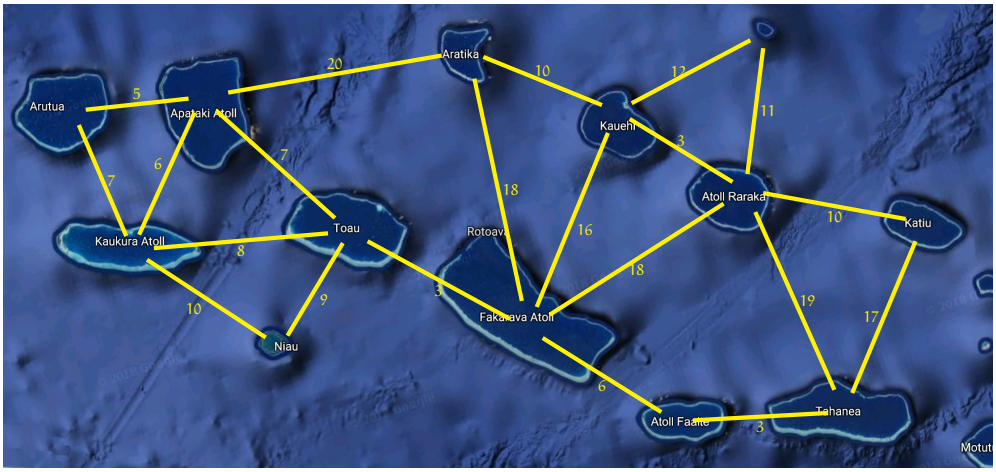
Informalmente, um algoritmo guloso é caracterizado por uma subestrutura ótima e a característica adicional, que podemos escolher o subproblema que leva a solução ótima através de uma regra simples. Portanto, o algoritmo guloso evite resolver todos subproblemas.
Uma subestrutura ótima é uma condição necessária para um algoritmo guloso ou de programação dinâmica ser ótimo, mas ela não é suficiente.

4.2. Algoritmos em grafos

4.2.1. Árvores geradoras mínimas

Motivação

Pontes para Polinésia Francesa!



Árvores geradoras mínimas (AGM)

Árvore geradora mínima (AGM)

Instância Grafo conexo não-direcionado $G = (V, A)$, pesos $c : A \to \mathbb{R}^+$.

Solução Um subgrafo $H = (V, A_H)$ conexo.

Objetivo Minimiza os custos $\sum_{a \in A_H} c(a)$.

- Um subgrafo conexo com custo mínimo deve ser uma árvore (por quê?).
- Grafo não conexo: Busca uma árvore em todo componente (floresta mínima).

Aplicações

- Redes elétricas
- Sistemas de estradas
- Pipelines
- Caixeiro viajante

- Linhas telefônicas alugadas

Observação 4.1
Nessa seção vamos supor que todos pesos nas arestas são diferentes. Isso simplifica o raciocínio sobre árvores geradoras mínimas, sem invalidar as nossas conclusões. ◊

Encontrar uma AGM

- O número de árvores geradoras pode ser exponencial.
- Como achar uma solução ótima?
- Observação importante

  Lema 4.2 (Propriedade do corte)
  Considere um *corte* $V = S \cup (V \setminus S)$ (com $S \neq \emptyset$, $V \setminus S \neq \emptyset$). O arco mínimo entre S e $V \setminus S$ faz parte de qualquer AGM.

Prova. Seja T uma AGM, e supõe que o arco mínimo $e = \{u, v\}$ entre S e $V \setminus S$ não faz de T. Em T existe um caminho de u para v que contém pelo menos um arco e' que atravessa o corte. Nossa afirmação: Podemos substituir e' com e, em contradição com a minimalidade de T (ver figura 4.1).
Prova da afirmação: Se substituirmos e' por e obtemos um grafo T'. Como e é mínimo, ele não custa mais que T. O novo grafo é conexo, porque para cada par de nós ou temos um caminho já em T que não usa e' ou podemos obter, a partir de um caminho em T que usa e' um novo caminho que usa um desvio sobre e. Isso sempre é possível, porque há um caminho entre u e v sem e, com dois sub-caminhos de u para u' e de v' para v, ambos sem usar e'. O novo grafo também é uma árvore, porque ele não contém um ciclo. O único ciclo possível é o caminho entre u e v em T com o arco e, porque T é uma árvore. ■

Encontrar uma AGM

- A característica do corte possibilita dois algoritmos simples:
  (a) Começa com algum nó e repetidamente adicione o nó ainda não alcançável com o arco de custo mínimo de algum nó já alcançável: *algoritmo de Prim*.
  (b) Começa sem arcos, e repetidamente adicione o arco com custo mínimo que não produz um ciclo: *algoritmo de Kruskal*.

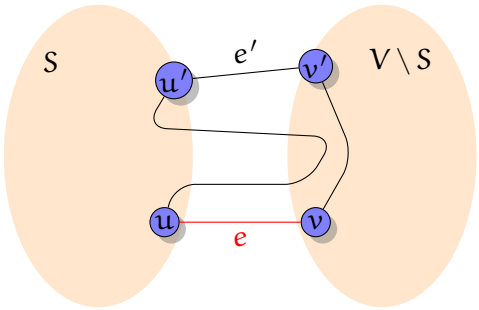


Figura 4.1.: O arco de menor custo e faz parte de toda árvore geradora mínima.

Existe uma outra propriedade que frequentemente é útil:

Lema 4.3 (Propriedade do ciclo)
Seja C um ciclo em G. A aresta mais pesada de C não faz parte de nenhuma árvore geradora mínima.

Prova. Seja T uma árvore geradora mínima que contém a resta mais pesada e de um ciclo C. A deleção de e separa T em dois componentes com vértices S e $V \setminus S$. No ciclo C existe uma outra aresta e' com um vértice em S e outro em $V \setminus S$ com peso menor que e. Por um argumento similar com aquela da propriedade de corte podemos ver que o conjunto $T' = T \setminus \{e\} \cup \{e'\}$ é conexo e acíclico, i.e., uma árvore geradora. Mas T' tem peso menor que T, uma contradição com a minimalidade de T. ■

AGM: Algoritmo de Prim

Algoritmo 4.2 (AGM-Prim)
Entrada Um grafo conexo não-orientado $G = (V, E_G)$ com pesos $c : V_G \to \mathbb{R}^+$

Saída Uma árvore $T = (V, E_T)$ com custo $\sum_{e \in E_T} c(e)$ mínimo.

$V' := \{v\}$ `para um` $v \in V$
$E_T := \emptyset$
`while` $V' \neq V$ `do`
  `escolhe` $e = \{u, v\}$ `com custo mínimo`
    `entre` V' `e` $V \setminus V'$ `(com` $u \in V'$`)`
  $V' := V' \cup \{v\}$

$E_T := E_T \cup \{e\}$
`end while`

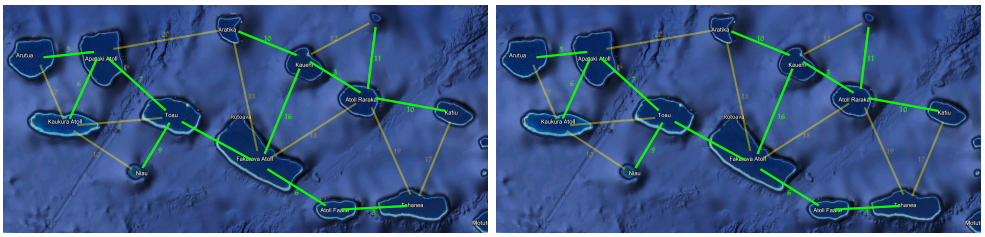
AGM: Algoritmo de Kruskal

Algoritmo 4.3 (AGM-Kruskal)
Entrada Um grafo conexo não-orientado $G = (V, E_G)$ com pesos $c : V_G \to \mathbb{R}^+$

Saída Uma árvore $T = (V, E_T)$ com custo $\sum_{e \in E_T} c(e)$ mínimo.

$E_T := \emptyset$
`while` (V, E_T) `não é conexo do`
  `escolha` e `com custo mínimo que não produz ciclo`
  $E_T := E_T \cup \{e\}$
`end while`

Exemplo 4.3
Resultado dos algoritmos de Prim e Kruskal para Polinésia Francesa:



O mesmo! ◊

Implementação do algoritmo de Prim

- Problema: Como manter a informação sobre a distância mínima de forma eficiente?
- Mantenha uma distância do conjunto V' para cada nó em $V \setminus V'$.
- Nós que não são acessíveis em um passo têm distância ∞.
- Caso um novo nó seja selecionado: atualiza as distâncias.

Implementação do algoritmo de Prim

- Estrutura de dados adequada:
  - Fila de prioridade Q de pares (e, v) (chave e elemento).
  - Operação $Q.\min()$ remove e retorna (e, c) com c mínimo.
  - Operação Q.atualiza(e, c') modifica a chave de e para v' caso v' é menor que a chave atual.
- Ambas operações podem ser implementadas com custo $O(\log n)$.

AGM: Algoritmo de Prim

Algoritmo 4.4 (AGM-Prim)
Entrada Um grafo conexo não-orientado $G = (V, E_G)$ com pesos $c : V_G \to \mathbb{R}^+$

Saída Uma árvore $T = (V, E_T)$ com custo $\sum_{e \in E_T} c(e)$ mínimo.

$E_T := \emptyset$
$Q := \{((v,u), c(v,u)) \mid u \in N(v)\}$ `{ nós alcançáveis }`
$Q := Q \cup \{((u,u), \infty) \mid u \in V \setminus N(v) \setminus \{v\}\}$ `{ nós restantes }`
`while` $Q \neq \emptyset$ `do`
  $((u,v), c) := Q.\min()$ `{` (u,v) `é arco mínimo }`
  `for` $w \in N(v)$ `do`
    Q.atualiza$((v,w), c(v,w))$
  `end for`
  $E_T := E_T \cup \{\{u,v\}\}$
`end while`

Algoritmo de Prim

- Complexidade do algoritmo com o refinamento acima:
- O laço 4–9 precisa $n - 1$ iterações.
- O laço 6–8 precisa no total menos que m iterações.
- $c_p[\text{AGM-PRIM}] = O(n \log n + m \log n) = O(m \log n)$

Uma implementação do algoritmo de Kruskal em tempo $O(m \log n)$ também é possível. Para esta implementação é necessário de manter conjuntos que representam nós conectados de maneira eficiente. Isso leva a uma estrutura de dados conhecida como *Union-Find* que tem as operações

- $C := \text{cria}(e)$: cria um conjunto com único elemento e.
- $\text{união}(C_1, C_2)$: junta os conjuntos C_1 e C_2.
- $C := \text{busca}(e)$: retorna o conjunto do elemento e.

Essas operações podem ser implementados em tempo $O(1)$, $O(1)$ e $O(\log n)$ (para n elementos) respectivamente.

4.2.2. Caminhos mais curtos

Caminhos mais curtos

- Problema comum: Encontra caminhos mais curtos em um grafo.
- Variações: Caminho mais curto
  - entre dois nós.
  - entre um nó e todos outros (inglês: single-source shortest paths, SSSP).
  - entre todas pares (inglês: all pairs shortest paths, APSP).

Caminhos mais curtos

CAMINHOS MAIS CURTOS

Instância Um grafo direcionado $G = (V, E)$ com função de custos $c : E \to \mathbb{R}^+$ e um nó inicial $s \in V$.

Solução Uma atribuição $d : V \to \mathbb{R}^+$ da distância do caminho mais curto $d(t)$ de s para t.

Aproximação: Uma idéia

1. Começa com uma estimativa viável: $d(s) = 0$ e $d(t) = \infty$ para todo nó em $V \setminus \{s\}$.
2. Depois escolhe uma aresta $e = (u, v) \in E$ tal que $d(u) + c(e) \leq d(v)$ e atualiza $d(v) = d(u) + c(e)$.
3. Repete até não ter mais arestas desse tipo.

Esse algoritmo é correto? Qual a complexidade dele?

4.3. Algoritmos de sequenciamento

Sequenciamento de intervalos

Considere o seguinte problema

SEQUENCIAMENTO DE INTERVALOS

Instância Um conjunto de intervalos $S = \{[c_i, f_i], i \in [n]\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.

Solução Um conjunto *compatível* $C \subseteq S$ de intervalos, i.e. cada par $i_1, i_2 \in C$ temos $i_1 \cap i_2 = \emptyset$.

Objetivo Maximiza a cardinalidade $|C|$.

(inglês: interval scheduling)

Como resolver?

- Qual seria uma boa estratégia gulosa para resolver o problema?
- Sempre selecionada o intervalo que
  - que começa mais cedo?
  - que termina mais cedo?
  - que começa mais tarde?
  - que termina mais tarde?
  - mais curto?
  - tem menos conflitos?

Implementação

Algoritmo 4.5 (Sequenciamento de intervalos)
Entrada Um conjunto de intervalos $S = \{[c_i, f_i] \mid i \in [n]\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.

Saída Um conjunto máximo de intervalos compatíveis C.

```
C := ∅
while S ≠ ∅ do
  Seja [c, f] ∈ S com f mínimo
  C := C ∪ [c, f]
  S := S \ {i ∈ S | i ∩ [c, f] ≠ ∅}
end while
return C
```

Seja $C = ([c_1, f_1], \ldots, [c_n, f_n])$ o resultado do algoritmo Sequenciamento de intervalos e $O = ([c'_1, f'_1], \ldots, [c'_m, f'_m])$ sequenciamento máximo, ambos em ordem crescente.

Proposição 4.2
Para todo $i \in [n]$ temos $f_i \leq f'_i$.

Prova. Como O é ótimo, temos $n \leq m$. Prova por indução. Base: Como o algoritmo guloso escolhe o intervalo cujo terminação é mínima, temos $f_1 \leq f'_1$. Passo: Seja $f_i \leq f'_i$ com $i < n$. O algoritmo guloso escolha entre os intervalos que começam após f_i aquele com terminação mínima. O próximo intervalo $[c'_{i+1}, m'_{i+1}]$ do sequenciamento ótimo está entre eles, porque ele começa depois f'_i e $f'_i \geq f_i$. Portanto, o próximo intervalo escolhido pelo algoritmo guloso termina antes de f'_{i+1}, i.e. $f_{i+1} \leq f'_{i+1}$. ■

Proposição 4.3
O sequenciamento do algoritmo guloso é ótimo.

Prova. Suponha que o algoritmo guloso retorna menos intervalos que o sequenciamento ótimo. Pela proposição 4.2, o último intervalo do C termina antes do último intervalo de O. Como O tem mais intervalos, existe mais um intervalo que poderia ser adicionado ao conjunto C pelo algoritmo guloso, a saber o $n+1$-ésimo intervalo de O. Isso é contradição com o fato, que o algoritmo somente termina se não sobram intervalos compatíveis. ■

Complexidade

- Uma implementação detalhada pode
  - Ordenar os intervalos pelo fim deles em tempo $O(n \log n)$
  - Começando com o primeiro intervalo e sempre escolher o intervalo atual e depois ignorar todos intervalos que começam antes que o intervalo atual termina.
  - Isso pode ser implementado em uma varredura de tempo $O(n)$.
  - Portanto o complexidade pessimista é $O(n \log n)$.

Conjunto independente máximo

Considere o problema

CONJUNTO INDEPENDENTE MÁXIMO, CIM

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

Solução Um conjunto *independente* $M \subseteq V$, i.e. todo $m_1, m_2 \in V$ temos $\{m_1, m_2\} \notin E$.

Objetivo Maximiza a cardinalidade $|M|$.

(inglês: maximum independent set, MIS)

Grafos de intervalo

- Uma instância S de sequenciamento de intervalos define um grafo não-direcionado $G = (V, E)$ com

$$V = S; \qquad E = \{\{i_1, i_2\} \mid i_1, i_2 \in S, i_1 \cap i_2 \neq \emptyset\}$$

- Grafos que podem ser obtidos pelo intervalos são *grafos de intervalo*.
- Um conjunto compatível de intervalos corresponde com um conjunto independente nesse grafo.
- Portanto, resolvemos CIM para grafos de intervalo!
- Sem restrições, CIM é NP-completo.

Variação do problema

Considere uma variação de Sequenciamento de intervalos:

Particionamento de intervalos

Instância Um conjunto de intervalos $S = \{[c_i, f_i], i \in [n]\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.

Solução Uma atribuição de rótulos para intervalos tal que cada conjunto de intervalos com a mesma rótula é compatível.

Objetivo Minimiza o número de rótulos diferentes.

Observação

- Uma superposição de k intervalos implica uma cota inferior de k rótulos.
- Seja d o maior número de intervalos super-posicionados (a *profundidade* do problema).
- É possível atingir o mínimo d?

Algoritmo

Algoritmo 4.6 (Particionamento de intervalos)

Instância Um conjunto de intervalos $S = \{[c_i, f_i], i \in [n]\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.

Solução Uma atribuição de rótulos para os intervalos tal que cada conjunto de intervalos com a mesma rótula é compatível.

Objetivo Minimiza o número de rótulos diferentes.

```
Ordene S em ordem de começo crescente.
for i := 1,...,n do
  Exclui rótulos de intervalos
    precedentes conflitantes
  Atribui ao intervalo i o número
    inteiro mínimo ≥ 1 que sobra
```

Corretude

- Com profundidade d o algoritmo precisa ao menos d rótulos.
- De fato ele precisa exatamente d rótulos. Por qûe?
- Qual a complexidade dele?

Observações: (i) Suponha que o algoritmo precise mais que d rótulos. Então existe um intervalo tal que todos números em $[1, d]$ estão em uso pelo intervalos conflitantes, uma contradição com o fato que a profundidade é d. (ii) Depois da ordenação em $O(n \log n)$ a varredura pode ser implementada em $O(n)$ passos. Portanto a complexidade é $O(n \log n)$.

Coloração de grafos

Considere o problema

COLORAÇÃO MÍNIMA

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

Solução Uma coloração de G, i.e. uma atribuição de cores $c : V \to C$ tal que $c(v_1) \neq c(v_2)$ para $\{v_1, v_2\} \in E$.

Objetivo Minimiza o número de cores $|C|$.

- PARTICIONAMENTO DE INTERVALOS resolve o problema COLORAÇÃO MÍNIMA para grafos de intervalo.
- COLORAÇÃO MÍNIMA para grafos sem restrições é NP-completo.

4.4. Tópicos

Compressão de dados

- Sequência genética (NM_005273.2, Homo sapiens guanine nucleotide binding protein)

  GATCCCTCCGCTCTGGGGAGGCAGCGCTGGCGGCGG...

  com 1666bp[1].

[1] "bp" é a abreviação do inglês "base pair" (par de bases)

- Como comprimir?
  - Com código fixo:

$$\begin{array}{ll} A = 00; & G = 01; \\ T = 10; & C = 11. \end{array}$$

    Resultado: 2b/bp e 3332b total.
  - Melhor abordagem: Considere as frequências, use códigos de diferente comprimento

A	G	T	C
.18	.30	.16	.36

Códigos: Exemplos

- Tentativa 1

$$\begin{array}{ll} T = 0; & A = 1; \\ G = 01; & C = 10 \end{array}$$

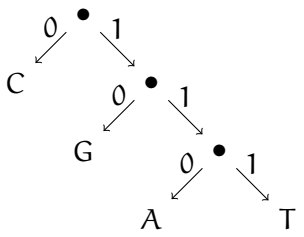
- Desvantagem: Ambíguo! $01 = TA$ ou $01 = G$?
- Tentativa 2

$$\begin{array}{ll} C = 0; & G = 10; \\ A = 110; & T = 111 \end{array}$$

- Custo: $0.36 \times 1 + 0.30 \times 2 + 0.18 \times 3 + 0.16 \times 3 = 1.98$b/bp, 3299b total.

Códigos livre de prefixos

- Os exemplos mostram
  - Em função das frequências, um código com comprimento variável pode custar menos.
  - Para evitar ambiguedades, nenhum prefixo de um código pode ser outro código: ele é *livre de prefixos* (inglês: prefix-free).
- Observação: Esses códigos correspondem com árvores binárias.



Cada código livre de prefixo pode ser representado usando uma árvore binária: Começando com a raiz, a sub-árvore da esquerda representa os códigos começando com 0 e a sub-árvore da direita os códigos começando com 1. Este padrão continua em cada sub-árvore considerando os demais bits. Caso todos bits de um código foram considerados, a árvore termina nessa posição com uma folha para esse código.
Essas considerações levam diretamente a um algoritmo. Na seguinte implementação, vamos representar árvores binárias como estrutura da dados abstrata que satisfaz

BinTree ::= Nil | Node(BinTree,Bintree)

uma folha sendo um nó sem filhos. Vamos usar a abreviação

Leaf ::= Node(Nil,Nil).

Algoritmo 4.7 (PrefixTree)
Entrada Um conjunto de códigos C livre de prefixos.

Saída Uma árvore binária, representando os códigos.

```
if |C| = 0 then
  return Nil                    { não tem árvore }
end if
if C = {ϵ} then
  return Leaf                { único código vazio }
end if
Escreve C = 0C1 ∪ 1C2
return Node(PrefixTree(C1),PrefixTree(C2))
```

Contrariamente, temos também

Proposição 4.4
O conjunto das folhas de cada árvore binária corresponde com um código livre de prefixo.

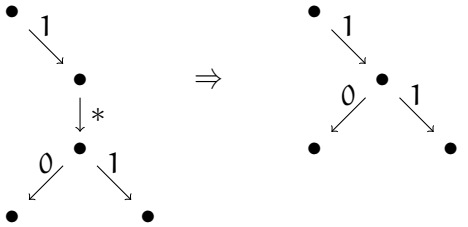
Prova. Dado uma árvore binária com as folhas representando códigos, nenhum código pode ser prefixo de outro: senão ocorreria como nó interno. ■

Qual o melhor código?

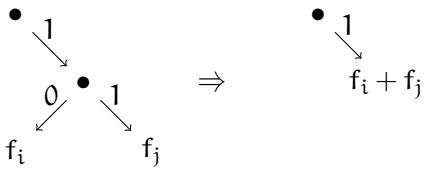
- A teoria de informação (Shannon) fornece um limite.
- A quantidade de informação contido num símbolo que ocorre com frequência f é
$$-f \log_2 f,$$
logo o número médio de bits transmitidos (para um número grande de símbolos) é
$$H = -\sum f_i \log_2 f_i.$$
- H é um limite inferior para qualquer código.
- Nem sempre é possível atingir esse limite. Com
$$A = 1/3, B = 2/3; \qquad H \approx 0.92b$$
mas o código ótimo precisa ao menos 1b por símbolo.
- Nosso exemplo: $H \approx 1.92$.

Como achar o melhor código?

- Observação 1: Uma solução ótima é uma árvore completa.



- Observação 2: Numa solução ótima, os dois símbolos com menor frequência ocorrem como irmãos no nível mais alto. Logo: Podemos substituir eles com um nó cujo frequência é a soma dos dois.



Algoritmo

Algoritmo 4.8 (Huffman)
Entrada Um alfabeto de símbolos S com uma frequência f_s para cada $s \in S$.

Saída Uma árvore binária que representa o melhor código livre de prefixos para S.

$Q := \{\text{Leaf}(f_s) \mid s \in S\}$ `{ fila de prioridade }`
`while` $|Q| > 0$ `do`
  $b_1 := Q.\text{min}()$ `com` $b_1 = \text{Node}(f_i, b_{i1}, b_{i2})$
  $b_2 := Q.\text{min}()$ `com` $b_2 = \text{Node}(f_j, b_{j1}, b_{j2})$
  $Q := Q.\text{add}(\text{Node}(f_i + f_j, b_1, b_2))$
`end while`

Exemplo 4.4

Saccharomyces cerevisiae

Considere a sequência genética do Saccharomyces cerevisiae (inglês: baker's yeast)

$$\text{MSITNGTSRSVSAMGHPAVERYTPGHIVCVGTHKVEVV}\ldots$$

com 2900352bp. O alfabeto nessa caso são os 20 amino-ácidos

$$S = \{A, C, D, E, F, G, H, I, K, L, M, N, P, Q, R, S, T, V, W, Y\}$$

que ocorrem com as frequências

A	C	D	E	F	G	H	I	K	L
0.055	0.013	0.058	0.065	0.045	0.050	0.022	0.066	0.073	0.096
M	N	P	Q	R	S	T	V	W	Y
0.021	0.061	0.043	0.039	0.045	0.090	0.059	0.056	0.010	0.034

Resultados
O algoritmo HUFFMAN resulta em

A	C	D	E	F	G	H	I	K	L
0010	100110	1110	0101	0000	1000	100111	1101	0011	100
M	N	P	Q	R	S	T	V	W	Y
000111	1011	11011	01011	10111	1111	0001	1010	000110	10110

que precisa 4.201 b/bp (compare com $\log_2 20 \approx 4.32$). ◊

4.5. Notas

O algoritmo guloso para sistemas de moedas Magazine, Nemhauser e Trotter (Magazine et al. 1975) dão critérios necessários e suficientes para uma solução gulosa do problema de troca ser ótima. Dado um sistema de moedas, Pearson (Pearson 2005) apresentou um algoritmo que descobre em tempo $O(n^3 \log^2 c_1)$, se o sistema é guloso. Um sistema de moedas tal que todo sufixo é guloso se chama *totalmente guloso*. Cowen et al. (Cowen et al. 2008) estudam sistemas de moedas totalmente gulosos.

4.6. Exercícios

(Soluções a partir da página 439.)

Exercício 4.1 (Sequênciamento guloso)
Dá um contra-exemplo para cada regra na página 89 para um algoritmo guloso que não é a regra correta.

Exercício 4.2 (Análise de series)
Suponha uma série de eventos, por exemplo, as transações feitos na bolsa de forma

compra Dell, vende HP, compra Google, ...

Uma certa ação pode acontecer mais que uma vez nessa sequência. O problema: Dado uma outra sequência, decida o mais rápido possível se ela é uma subsequência da primeira.
Achar um algoritmo eficiente (de complexidade $O(m+n)$ com sequência de tamanho n e m), prova a corretude a analise a complexidade dele.
(Fonte: (Kleinberg e Tardos 2005)).

Exercício 4.3 (Comunicação)
Imagine uma estrada comprida (pensa em uma linha) com casas ao longo dela. Suponha que todas casas querem acesso ao comunicação com celular. O

problema: Posiciona o número mínimo de bases de comunicação ao longo da estrada, com a restrição que cada casa tem quer ser ao máximo 4 quilômetros distante de uma base.
Inventa um algoritmo eficiente, prova a corretude e analise a complexidade dele.
(Fonte: (Kleinberg e Tardos 2005)).

5. Programação dinâmica

5.1. Introdução

Temos um par de coelhos recém-nascidos. Um par recém-nascido se torna fértil depois um mês. Depois ele gera um outro par a cada mês seguinte. Logo, os primeiros descendentes nascem em dois meses. Supondo que os coelhos nunca morrem, quantos pares temos depois de n meses?

n	0	1	2	3	4	5	6	$\cdots$
#	1	1	2	3	5	8	13	$\cdots$

Como os pares somente produzem filhos depois de dois meses, temos

$$F_n = \underbrace{F_{n-1}}_{\text{população antiga}} + \underbrace{F_{n-2}}_{\text{descendentes}}$$

com a recorrência completa

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{caso } n \geq 2 \\ 1 & \text{caso } n \in \{0, 1\} \end{cases}$$

Os números definidos por essa recorrência são conhecidos como os *números Fibonacci*. Uma implementação recursiva simples para calcular um número de Fibonacci é

```
fib(n) :=
  if n ≤ 1 then
    return 1
  else
    return fib(n−1)+fib(n−2)
  end if
end
```

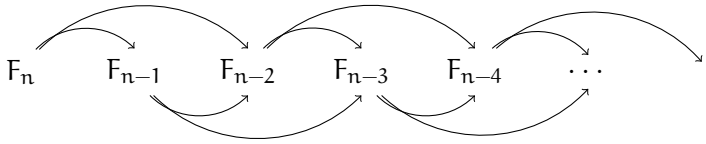
Qual a complexidade dessa implementação? Temos a recorrência de tempo

$$T(n) = \begin{cases} T(n-1) + T(n-2) + \Theta(1), & \text{caso } n \geq 2, \\ \Theta(1), & \text{caso contrário.} \end{cases}$$

É simples de ver (indução!) que $T(n) \geq F_n$, e sabemos que $F_n = \Omega(2^n)$ (outra indução), portanto a complexidade é exponencial. (A fórmula exata é

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \left\lfloor \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n}{\sqrt{5}} + \frac{1}{2} \right\rfloor$$

e foi publicado por Binet em 1843.)
Qual o problema dessa solução? De um lado temos um número exponencial de caminhos das chamadas recursivas

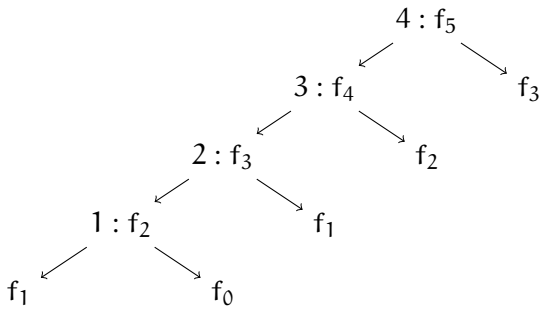


mas somente um número polinomial de valores diferentes! Ideia: usar uma *cache*!

$f_0 := 1$
$f_1 := 1$
$f_i := \perp$ `para` $i \geq 2$

```
fib(n) :=
  if f_n = ⊥ then
    f_n := fib(n−1)+fib(n−2)
  end if
  return f_n
end
```

Exemplo de uma execução:



	f_5	f_4	f_3	f_2	f_1	f_0
Inicial	$\bot$	$\bot$	$\bot$	$\bot$	1	1
1	$\bot$	$\bot$	$\bot$	2	1	1
2	$\bot$	$\bot$	3	2	1	1
3	$\bot$	5	3	2	1	1
4	8	5	3	2	1	1

O trabalho agora é $O(n)$, i.e. linear! Essa abordagem se chama *memoização*: usamos uma cache para evitar recalcular resultados intermediários utilizados frequentemente. Essa implementação é *top-down* e corresponde exatamente à recorrência acima. Uma implementação (ligeiramente) mais eficiente que preenche a "cache" de forma *bottom-up* é

```
fib(n) :=
  f_0 := 1
  f_1 := 1
  for i ∈ [2, n] do
    f_i := f_{i-1} + f_{i-2}
  end for
  return f_n
end
```

Finalmente, podemos otimizar essa computação ainda mais, evitando totalmente a cache

```
fib(n) :=
  f := 1
  g := 1
  for i ∈ [2, n] do
    { invariante: f = F_{i-2} ∧ g = F_{i-1} }
    g := f + g
    f := g − f
    { invariante: f = F_{i-1} ∧ g = F_i }
  end for
```

A ideia de armazenar valores intermediários usados frequentemente numa computação recursiva é uma das ideias principais da *programação dinâmica* (a outra senda o princípio de otimalidade, veja abaixo). A sequência de implementações no nosso exemplo dos números Fibonacci é típico para algoritmos de programação dinâmica, inclusive o último para reduzir a complexidade de espaço. Tipicamente usa-se implementações ascendentes (ingl. *bottom-up*).

Em resumo, para poder aplicar a programação dinâmica, e necessário decompor um problema um subproblemas menores, tal que o *princípio de oti-*

malidade (ou de *subestrutura ótima*) é satisfeito: uma solução ótima do problema pode ser obtida através de uma solução ótima de subproblemas. Isso define um grafo direcionado acíclico *subjacente*: ele possui o problema e todos subproblemas como vértices, e um arco de um problema para os seus subproblemas imediatos.
O exemplo dos número Fibonacci mostra, a programação dinâmica consegue reduzir a complexidade caso existe uma *superposição de subproblemas*. Em outras palavras: o número de vértices no grafo subjacente é menor que o número de caminhas da um problema para um vértice sem predecessores.

Passos do desenvolvimento de um algoritmo de PD

1. Verificar a subestrutura ótima.
2. Expressar a solução em forma de recorrência.
3. Para implementar: usar memoização ou armazenar os possíveis valores da recorrência numa tabela, preenchendo a tabela de forma ascendente.

5.2. Comparação de sequências

5.2.1. Subsequência comum mais longa

Uma subsequência de uma sequência é uma seleção de alguns elementos da sequência. A sequência "aba", por exemplo, é uma subsequência de "aberação". Para comparar duas sequências, frequentemente é interessante determinar a subsequência comum mais longe entre os dois.

SCML

Instância Duas sequências $x_1x_2\cdots x_m$ e $Y = y_1y_2\cdots y_n$.

Solução Uma subsequência comum $z_1z_2\cdots z_k$ das duas sequências, i.e., índices $i_1 < i_2 < \ldots < i_k$ e $j_1 < j_2 < \ldots < j_k$ tal que $x_{i_l} = y_{j_l}$ para $l \in [k]$.

Objetivo Maximizar o comprimento k da subsequência em comum.

Exemplo 5.1
A subsequência em comum mais longa de `abcbdab` e `bdcaba` é `bcab`. ◊

Exemplo 5.2
Comparar duas sequência genéticas

$$ACCGGTCGAGTG$$
$$GTCGTTCGGAATGCCGTTGCTCTGTAAA$$

$\Diamond$

Observação 5.1
Para resolver o SCML, podemos considerar prefixos das sequencias como subproblemas, e observar que uma solução ótima do problema pode ser obtida por soluções ótimas desses subproblemas. Usaremos a notação $x_{a:b}$ para a sequência $x_a x_{a+1} \cdots x_b$. Quais as possibilidades para a SCML entre $x_{1:n}$ e $y_{1:m}$?

- O último elemento é x_n, mas não y_m: logo a solução é um SCML entre $x_{1:n}$ e $y_{1:m-1}$.
- O último elemento é y_m, mas não x_n: logo a solução é um SCML entre $x_{1:n-1}$ e $y_{1:m}$.
- O último elemento é nenhum dos dois. Um dos dois subcasos acima se aplica.
- O última elemento é x_n e y_m. Para isso necessariamente $x_n = y_m$. A solução ótima é a SCML entre $x_{1:n-1}$ e $y_{1:m-1}$ seguido por x_n. De fato sempre se $x_n = y_m$ podemos supor que a SCML termina com estes elementos.

$\Diamond$

Logo,escrevendo $S(x_{1:n}, y_{1:m})$ para o tamanho da subsequência mais longa entre $x_{1:n}$ e $y_{1:m}$, temos uma solução recursiva

$$S(x_{1:n}, y_{1:m}) = \begin{cases} S(x_{1:n-1}, y_{1:m-1}) + 1 & \text{se } x_n = y_n \\ \max\{S(x_{1:n}, y_{1:m-1}), S(x_{1:n-1}, y_{1:m})\} & \text{se } x_n \neq y_n \\ 0 & \text{se } n = 0 \text{ ou } m = 0 \end{cases}$$

Qual a complexidade de implementação recursiva (naiva)? No pior caso executamos

$$T(n, m) = T(n-1, m) + T(n, m-1) + \Theta(1)$$

operações. Isso com certeza é mais que o número de caminhos de (n, m) até $(0, 0)$, que é maior que $\binom{m+n}{n}$, i.e. exponential no pior caso.
Usando memoização ou armazenando valores intermediários, podemos reduzir o tempo e espaço para $O(nm)$:

Algoritmo 5.1 (SCML)
Entrada Duas sequências $x_1x_2\cdots x_n$ e $y_1y_2\cdots y_m$.

Saída O tamanho da maior subsequência comum.

```
for i := 0 to n do S[i,0] := 0;
for j := 1 to m do S[0,j] := 0;
for i := 1 to n do
    for j := 1 to m do
        if x_i = y_j then
              S[i,j] := S[i-1,j-1] + 1
        else
              S[i,j] := max{S[i,j-1], S[i-1,j]}
        end if
    end for
return S[n,m]
```

Exemplo 5.3

	.	B	D	C	A	B	A
.	0	0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

$\Diamond$

Exemplo 5.4

			P	R	O	G	R	A	M	A
		0	1	2	3	4	5	6	7	8
	0	0	0	0	0	0	0	0	0	0
P	1	0	1	1	1	1	1	1	1	1
E	2	0	1	1	1	1	1	1	1	1
S	3	0	1	1	1	1	1	1	1	1
Q	4	0	1	1	1	1	1	1	1	1
U	5	0	1	1	1	1	1	1	1	1
I	6	0	1	1	1	1	1	1	1	1
S	7	0	1	1	1	1	1	1	1	1
A	8	0	1	1	1	1	2	2	2	2

$\Diamond$

Caso só o comprimento da maior subsequência em comum importa, podemos reduzir o espaço usado. Os valores de cada linha ou coluna dependem só dos valores da linha ou coluna anterior. Supondo, que o comprimento de uma linha e menor que o comprimento de uma coluna, podemos manter duas linhas e calcular os valores linha por linha. Caso as colunas são menores, procedemos das mesmo forma coluna por coluna. Com isso, podemos determinar o comprimento da maior subsequência em comum em tempo $O(nm)$ e espaço $O(\min\{n, m\})$.
Caso queiramos recuperar a própria subsequência, temos que manter essa informação adicionalmente:

SCML que permite mostrar a subsequência

Algoritmo 5.2 (SCML-2)
Entrada Dois strings X e Y e seus respectivos tamanhos m e n medidos em número de caracteres.

Saída O tamanho da maior subsequência comum entre X e Y e o vetor b para recuperar uma SCML.

```
m := comprimento[X]
n := comprimento[Y]
for i:=0 to m do c[i,0] := 0;
for j:=1 to n do c[0,j] := 0;
for i:=1 to m do
   for j:=1 to n do
```

```
        if x_i = y_j then
            c[i,j] := c[i-1,j-1] + 1
            b[i,j] := ↖
        else if c[i-1,j] ≥ c[i,j-1] then
                c[i,j] := c[i-1,j]
                b[i,j] := ↑
            else
                c[i,j] := c[i,j-1]
                b[i,j] := ←
return c e b
```

Exemplo

	.	B	D	C	A	B	A
.	0	0	0	0	0	0	0
A	0	↑0	↑0	↑0	↖1	←1	↖1
B	0	↖1	←1	←1	↑1	↖2	←2
C	0	↑1	↑1	↖2	←2	↑2	↑2
B	0	↖1	↑1	↑2	↑2	↖3	←3
D	0	↑1	↖2	↑2	↑2	↑3	↑3
A	0	↑1	↑2	↑2	↖3	↑3	↖4
B	0	↖ 1	↑2	↑2	↑3	↖4	↑4

Nesse caso, não tem método simples para reduzir o espaço de $O(nm)$ (veja os comentários sobre o algoritmo de Hirschberg abaixo). Mantendo duas linhas ou colunas de c, gasta menos recursos, mas para recuperar a subsequência comum, temos que manter $O(nm)$ elementos em b.
O algoritmo de Hirschberg (Hirschberg 1975), via Divisão e Conquista, resolve o problema da subsequência comum mais longa em tempo $O(mn)$, mas com complexidade de espaço linear $O(m+n)$. O algoritmo recursivamente divide a tabela em quatro quadrantes e ignora os quadrantes superior-direito e inferior-esquerdo, visto que a solução não passa por eles. Após, o algoritmo é chamado recursivamente nos quadrantes superior-esquerdo e inferior-direito. Em cada chamada recursiva é criada uma lista com as operações executadas, e tal lista é concatenada ao final das duas chamadas recursivas. A recuperação da sequências de operações pode ser feita percorrendo-se linearmente esta lista.

Print-SCML

Algoritmo 5.3 (Print-SCML)
Entrada Matriz $b \in \{\leftarrow, \nwarrow, \uparrow\}^{m \times n}$.

Saída A maior subsequência Z comum entre X e Y obtida a partir de b.

```
if i = 0 or j = 0 then return
if b[i,j] = ↖ then
  Print-SCML(b, X, i-1, j-1)
  print x_i
else if b[i,j] = ↑ then
  Print-SCML(b, X, i-1, j)
else
Print-SCML(b, X, i, j-1)
```

5.2.2. Similaridade entre strings

Considere o problema de determinar o número mínimo de operações que transformam um string s em um string t, se as operações permitidas são a inserção de um caractere, a deleção de um caractere ou a substituição de um caractere para um outro. O problema pode ser visto como um alinhamento de dois strings da forma

```
sonhar
vo--ar
```

em que cada coluna com um caractere diferente (inclusive a "falta" de um caractere -) tem custo 1 (uma coluna $(a, -)$ corresponde à uma deleção no primeiro ou uma inserção no segundo string, etc.). Esse problema tem subestrutura ótima: Uma solução ótima contém uma solução ótima do subproblema sem a última coluna, senão podemos obter uma solução de menor custo. Existem quatro casos possíveis para a última coluna:

$$\begin{bmatrix} a \\ - \end{bmatrix}; \begin{bmatrix} - \\ a \end{bmatrix}; \begin{bmatrix} a \\ a \end{bmatrix}; \begin{bmatrix} a \\ b \end{bmatrix}$$

com caracterees a, b diferentes. O caso (a, a) somente se aplica, se os últimos caracteres são iguais, o caso (a, b) somente, se eles são diferentes. Portanto, considerando todos casos possíveis fornece uma solução ótima. Com $d(i, j)$

e distância mínima entre $s_{1:i}$ e $t_{1:j}$ obtemos

$$d(i,j) = \begin{cases} \max(i,j), & \text{se } i = 0 \text{ ou } j = 0, \\ \min(d(i-1,j)+1, d(i,j-1)+1, d(i-1,j-1)+[s_i \neq t_j]), & \text{caso contrário,} \end{cases}$$

Essa distância está conhecida como *distância de Levenshtein* (Levenshtein 1966). Uma implementação direta é

Distância de Edição

Algoritmo 5.4 (Distância)
Entrada Dois strings $s = s_1 s_2 \cdots s_n$ e $t = t_1 t_2 \cdots t_m$.

Saída A distância mínima entre s e t.

```
distância(s_{1:n}, t_{1:m}):=
  if n = 0 return m
  if m = 0 return n
  if s_n = t_m then
    sol_0 = distância(s_{1:n-1}, t_{1:m-1})
  else
    sol_0 = distância(s_{1:n-1}, t_{1:m-1}) + 1
  end if
  sol_1 = distância(s_{1:n}, t_{1:m-1}) + 1
  sol_2 = distância(s_{1:n-1}, t_{1:m}) + 1
  return min{sol_0, sol_1, sol_2}
```

Essa implementação tem complexidade exponencial. Com programação dinâmica, armazenando os valores intermediários de d numa matriz m, obtemos

Distância de Edição

Algoritmo 5.5 (PD-distância)
Entrada Dois strings $s = s_1 s_2 \cdots s_n$ e $t = t_1 t_2 \cdots t_m$.

Saída A distância mínima entre s e t.

Comentário O algoritmo usa uma matriz $M = (m_{i,j}) \in \mathbb{N}^{(n+1)\times(m+1)}$ que armazena as distâncias mínimas $m_{i,j}$ entre os prefixos $s_{1:i}$ e $t_{1:j}$.

```
PD−distância(s_{1:n}, t_{1:m}) :=
  for i := 0,...,n do m_{i,0} := i; p_{i,0} := −1
  for i := 1,...,m do m_{0,i} := i; p_{0,i} := −1
  for i := 1,...,n do
    for j := 1,...,m do
      if (s_i = t_j) then
        sol_0 := m_{i−1,j−1}
      else
        sol_0 := m_{i−1,j−1} + 1
      end if
      sol_1 := m_{i,j−1} + 1
      sol_2 := m_{i−1,j} + 1
      m_{i,j} := min(sol_0, sol_1, sol_2);
      p_{i,j} := min{i | sol_i = m_{ij}}
    end for
  return m_{i,j}
```

Distância entre textos
Valores armazenados na matriz M para o cálculo da distância entre ALTO e LOIROS

	·	L	O	I	R	O	S
·	0	1	2	3	4	5	6
A	1	1	2	3	4	5	6
L	2	1	2	3	4	5	6
T	3	2	2	3	4	5	6
O	4	3	2	3	4	4	5

```
-ALTO-
LOIROS
```

O algoritmo possui complexidade de tempo e espaço $O(mn)$ para armazenar e calcular p e m. O espaço pode ser reduzido para $O(\min\{n, m\})$ usando uma adaptação do algoritmo de Hirschberg.

5.3. Problema da Mochila

MOCHILA (INGL. KNAPSACK)

Instância Um conjunto de n itens, cada item $i \in [n]$ com um valore v_i e um peso w_i, e um limite de peso da mochila W.

Solução Um subconjunto de itens $S \subseteq [n]$ que cabe na mochila, i.e. $\sum_{i \in S} w_i \leq W$.

Objetivo Maximizar o valor total $\sum_{i \in S} v_i$ dos itens selecionados.

Ideia: Ou item i faz parte da solução ótima com itens $i \dots n$ ou não.

- Caso sim: temos uma solução com valor v_i a mais do que a solução ótima para itens $i+1, \dots, n$ com capacidade restante $W - w_i$.
- Caso não: temos um valor correspondente à solução ótima para itens $i+1, \dots, n$ com capacidade W.

Seja $M(i, w)$ o valor da solução máxima para itens em $[i, n]$ e capacidade W. A ideia acima define uma recorrência

$$M(i,w) = \begin{cases} 0 & \text{se } i > n \text{ ou } w = 0 \\ M(i+1,w) & \text{se } w_i > w \text{ não cabe} \\ \max\{M(i+1,w), M(i+1,w-w_i)+v_i\} & \text{se } w_i \leq w \end{cases}$$

A solução desejada é $M(1, W)$. Para determinar a seleção de itens:

Mochila máxima (Knapsack)

- Seja $S^*(k, v)$ a solução de tamanho menor entre todas soluções que
  - usam somente itens $S \subseteq [1, k]$ e
  - tem valor exatamente v.
- Temos

$$\begin{aligned} S^*(k,0) &= \emptyset \\ S^*(1,v_1) &= \{1\} \\ S^*(1,v) &= \text{undef} \qquad \text{para } v \neq v_1 \end{aligned}$$

Mochila máxima (Knapsack)

- S^* obedece a recorrência

$$S^*(k,v) = \min_{\text{tamanho}} \begin{cases} S^*(k-1, v-v_k) \cup \{k\} & \text{se } v_k \leq v \text{ e } S^*(k-1, v-v_k) \text{ definido} \\ S^*(k-1, v) \end{cases}$$

- Menor tamanho entre os dois

$$\sum_{i \in S^*(k-1, v-v_k)} t_i + t_k \leq \sum_{i \in S^*(k-1, v)} t_i.$$

- Melhor valor: Escolhe $S^*(n, v)$ com o valor máximo de v definido.
- Tempo e espaço: $O(n \sum_i v_i)$.

5.4. Multiplicação de Cadeias de Matrizes

Qual é a melhor ordem para multiplicar n matrizes $M = M_1 \times \cdots \times M_n$? Como o produto de matrizes é associativo, temos várias possibilidades de chegar em M. Por exemplo, com quatro matrizes temos as cinco possibilidades

Possíveis multiplicações

Dadas (M_1, M_2, M_3, M_4) pode-se obter $M_1 \times M_2 \times M_3 \times M_4$ de 5 modos distintos, mas resultando no mesmo produto

$$M_1(M_2(M_3M_4))$$
$$M_1((M_2M_3)M_4)$$
$$(M_1M_2)(M_3M_4)$$
$$(M_1(M_2M_3))M_4$$
$$((M_1M_2)M_3)M_4$$

- Podemos multiplicar duas matrizes somente se $\mathtt{Ncol}(A) = \mathtt{Nlin}(B)$
- Sejam duas matrizes com dimensões $p \cdot q$ e $q \cdot r$ respectivamente. O número de multiplicações resultantes é $p \cdot q \cdot r$.

Dependendo do tamanho dos matrizes, um desses produtos tem o menor número de adições é multiplicações. O produto de duas matrizes $p \times q$ e $q \times r$ precisa prq multiplicações e $pr(q-1)$ adições. No exemplo acima, caso temos matrizes do tamanho 3×1, 1×4, 4×1 e 1×5 as ordens diferentes resultam em

Número de multiplicações para cada sequência

$$20 + 20 + 15 = 55$$
$$4 + 5 + 15 = 24$$
$$12 + 20 + 60 = 92$$
$$4 + 5 + 15 = 24$$
$$12 + 12 + 15 = 39$$

operações, respectivamente. Logo, antes de multiplicar as matrizes vale a pena determinar a ordem ótima (caso o tempo para determinar ela não é proibitivo). Dada uma ordem, podemos computar o número de adições e multiplicações em tempo linear. Mas quantas ordens tem? O produto final consiste em duas matrizes que são os resultados dos produtos de i e $n-i$ matrizes; o ponto de separação i pode ser depois qualquer matriz $1 \leq i < n$. Por isso o número de possibilidades C_n satisfaz a recorrência

$$C_n = \sum_{1 \leq i < n} C_i C_{n-i}$$

para $n \geq 1$ e as condições $C_1 = 1$ e $C_2 = 1$. A solução dessa recorrência é $C_n = \binom{2n}{n}/(2(2n-1)) = O(4^n/n^{3/2})$ e temos $C_n \geq 2^{n-2}$, logo têm um número exponencial de ordens de multiplicação possíveis[1].

Solução por Recorrência

O número de possibilidades T_n satisfaz a recorrência

$$T(n) = \sum_{1 \leq i < n-1} T(i)T(n-i)$$

para $n \geq 1$ e as condições $T(1) = 1$.
A solução dessa recorrência é $T(n) = \binom{2n}{n}(2(2n-1)) = O(4^n/n^{3/2})$ e temos $C_n \geq 2^{n-2}$.
Então não vale a pena avaliar o melhor ordem de multiplicação, enfrentando um número exponencial de possibilidades? Não, existe uma solução com programação dinâmica, baseada na mesma observação que levou à nossa recorrência.

$$m_{ik} = \begin{cases} \min_{i \leq j < k} m_{ij} + m_{(j+1)k} + b_{i-1} b_j b_k & \text{caso } i < k \\ 0 & \text{caso } i = kx \end{cases}$$

[1] Podemos obter uma solução usando funções geratrizes. $(C_{n-1})_{n \geq 1}$ são os números Catalan, que têm diversas aplicações na combinatórica.

Multiplicação de Cadeias de Matrizes

- Dada uma cadeia $(A_1, A_2, \ldots, A_n)$ de n matrizes, coloque o produto $A_1 A_2 \ldots A_n$ entre parênteses de forma a minimizar o número de multiplicações.

Retorna o número mínimo de multiplicações necessárias para multiplicar a cadeia de matrizes passada como parâmetro.

Algoritmo 5.6 (MM)
Entrada Matrizes $A_1, A_2, \ldots, A_n$ com dimensões b_i, $0 \leq i \leq n$. A matriz A_i tem dimensão $b_{i-1} \times b_i$.

Saída Número mínimo de multiplicações para calcular o produto $\prod_{i \leq k \leq j} A$ em $m_{i,j}$, e ponto de separação do produto $p_{i,j}$ no nível mais alto.

```
for i ∈ [n] do m_{i,j} := 0
for u:=1 to n−1 do {diagonais superiores}
  for i:=1 to n−u do {posição na diagonal}
    j:=i+u          {u = j−i}
    m_{i,j} := ∞
    for k:=i to j−1 do
      c:= m_{i,k} + m_{k+1,j} + b_{i−1} b_k b_j
      if c < m_{i,j} then m_{i,j}:=c; p_{i,j} := k
    end for
  end for
end for
return m_{1,n}
```

Para analisar a complexidade do algoritmo vamos supor que as operações aritméticas sobre os reais são operações fundamentais. O tamanho da entrada é o número de matrizes n. Como o número de iterações em cada laço é limitado por n, obtemos uma complexidade pessimista de $O(n^3)$.

5.5. Tópicos

Essa seção apresente exemplos adicionais de algoritmos usando programação dinâmica. Um outro exemplo encontra-se no capítulo 10.2.3.

5.5.1. Algoritmo de Floyd-Warshall

O algoritmo de Floyd-Warshall calcula o caminho mínimo entre todos os pares de vértices de um grafo. Supõe que $G = (V, A)$ é uma grafo direcionado com distâncias l_a para $a \in A$.

Algoritmo de Floyd-Warshall

- Calcula o caminho mínimo entre cada par de vértices de um grafo.
- Considera que o grafo não tenha ciclos negativos, embora possa conter arcos de custo negativo.

Subestrutura ótima de caminhos mais curtos

Lema 5.1
Dado um grafo direcionado $G = (V, A)$ com pesos l_a nos arcos $a \in A$, caso $v_1 v_2 \cdots v_k$ é o caminho mais curto entre v_1 e v_k então para qualquer $i \leq j$ o caminho $v_i v_{i+1} \cdots v_j$ é o mais curto entre v_i e v_j.

Prova. Supõe que existe um caminho mais curto entre v_i e v_j que o caminho acima. Podemos substituir este trecho no caminho de v_1 para v_k para obter um caminho menor, uma contradição com a minimalidade do caminho $v_1 v_2 \cdots v_k$. ■

Logo para calcular o caminho mais curto d_{ik} entre vértices i e k uma primeira tentativa de definir uma recorrência poderia ser

$$d_{ij} = \begin{cases} 0, & \text{caso } i = j, \\ \min\{\min_{k \in [n]} d_{ik} + d_{kj}, l_{ij}\}, & \text{caso contrário,} \end{cases}$$

(supondo que $l_{ij} = \infty$ para $ij \notin A$).
Infelizmente, os valores d_{ij} possuem uma mutual. Para quebrá-la, introduziremos um limite para os vértices intermediários de uma caminho mais curto. Seja d_{ij}^k a menor distância entre vértices i e j com caminhos usando somente vértices intermédiários com índice no máximo k no interior. Com isso temos

$$d_{ij}^k = \begin{cases} l_{ij}, & \text{caso } k = 0, \\ \min\{d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1}\}, & \text{caso contrário.} \end{cases}$$

O algoritmo para calcular os caminhos mais curtos entre todos pares de vértices é conhecido como algoritmo de Floyd-Warshall.

Algoritmo de Floyd-Warshall

Algoritmo 5.7 (Algoritmo de Floyd-Warshall)
Entrada Um grafo $G = (V, E)$ com $n = |V|$ vértices, representado por uma matriz quadrada $D = (d_{ij}) \in \mathbb{R}^{n \times n}$ com distância d_{ij} entre $ij \in E$ ou $d_{ij} = \infty$, caso $ij \notin E$.

Saída Uma matriz quadrada com cada célula contendo a distância mínima entre i e j.

$D^0 := D$
for $k := 1$ **to** n
  for $i := 1$ **to** n
    for $j := 1$ **to** n
      $d_{ij}^k := \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$
return D^n

Observe que não é necessário armazenar as matrizes D^k explicitamente. O algoritmo de Floyd-Warshall continua sendo correto usando a mesma matriz D para todas operações e portanto possui complexidade de espaço $\Theta(n^2)$.

Excurso 5.1
Podemos substituir as operações sobre $(\mathbb{R}, \min, +)$ no algoritmo de Floyd-Warshall por outras operações, para resolver problemas similares. Por exemplo:

- Sobre o semi-anel $(\mathbb{R}, \max, \min)$, o algoritmo resolve o problema do caminho gargalho entre todos pares (ingl. all pairs bottleneck paths problem).

Sobre o semi-anel $(\mathbb{R}, \min, +)$ a matriz D^k representa o menor caminho com no máximo k hops entre todos pares de vértices, e portanto D^n é a matriz calculada pelo algoritmo de Floyd-Warshall. (Observe que a matriz D^k no algoritmo do Floyd-Warshall não é D elevado a k.)
Portanto, podemos aplicar n vezes uma multiplicação de matrizes para obter D^n em $O(n \times n^3)$. Como $D^i = D^n$ para $i \geq n$, podemos calcular $D^n = D^{\lceil \log_2 n \rceil}$ mais rápido quadrando D $\lceil \log_2 n \rceil$ vezes (ver os algoritmos de potenciação), uma abordagem que possui complexidade $O(n^3 \log n)$.
É uma observação importante que o algoritmo de Strassen (e algoritmos mais avançados como o algoritmo de Coppersmith-Winograd) só funcionam sobre anéis, porque eles precisam um inverso para a adição. Um algoritmo subcúbico para a multiplicação de matrizes em semi-anéis implicaria em um

algoritmo sub-cúbico para o problema do caminho mínimo entre todos pares de vértices e problemas similares. Para mais informações ver por exemplo Chan (2007) e Williams e Williams (2010).

Outra observação é que o algoritmo de Floyd-Warshall somente calcula as distâncias entre todos pares de vértices. Para determinar os caminhos mais curtos, veja por exemplo (Alon et al. 1992). ◊

5.5.2. Caixeiro viajante

O problema do caixeiro viajante é um exemplo em que a programação dinâmica ajuda reduzir uma complexidade exponencial: o algoritmo continua ser exponencial (que não é surpresa, já que o problema é NP-completo), mas significadamente mais eficiente.

Problema do Caixeiro Viajante

Instância Um grafo $G = (V, A)$ com distância d_a para $a \in A$. Vamos sem perda de generalidade supor que $V = [1, n]$.

Solução Uma rota que visita todos vértices exatamente uma vez, i.e. uma permutação π de $[n]$.

Objetivo Minimizar a distância total da rota

$$d(\pi) = d_{\{\pi(n),\pi(1)\}} + \sum_{i \in [n]} d_{\{\pi(i),\pi(i+1)\}}.$$

O algoritmo é baseado na seguinte ideia (Bellman 1962). Seja $v_1, v_2, \ldots$ uma solução ótima. Sem perda de generalidade, podemos supor que $v_1 = 1$. Essa solução tem como sub-solução ótima o caminho $v_2, v_3, \ldots$ que passa por todos vértices exceto v_1 e volta. Da mesmo forma a última sub-solução tem o sub-caminho $v_3, v_4, \ldots$ que passa por todos vértices exceto v_1, v_2 e volta, como sub-solução. Essas soluções têm sub-estrutura ótima, porque qualquer outro caminho menor pode ser substituído para o caminho atual.

Logo, podemos definir $T(i, V)$ como menor rota começando no vértice i e passando por todos vértices em V exatamente uma vez e volta para vértice 1. A solução desejada então é $T(1, [2, n])$. Para determinar o valor de $T(i, V)$ temos que minimizar sobre todas continuações possíveis. Isso leva à recor-

rência

$$T(i, V) = \begin{cases} \min_{v \in V} d_{iv} + T(v, V \setminus \{v\}) & \text{caso } V \neq \emptyset \\ d_{i1} & \text{caso } V = \emptyset \end{cases}$$

Se ordenamos todos os sub-conjuntos dos vértices $[n]$ em ordem de $\subseteq$, obtemos uma matrix de dependências

	V_1	V_2	$\cdots$	V_{2^n}
1				
2				
$\vdots$				
n				

em que qualquer elemento depende somente de elementos em colunas mais para esquerda. Uma implementação pode representar um subconjunto de $[n]$ por um número entre 0 e $2^n - 1$. Nesse caso, a ordem natural já respeita a ordem $\subseteq$ entre os conjuntos, e podemos substituir um teste $v \in V_j$ por $2^v \& j = 2^v$ e a operação $V_j \setminus \{v\}$ por $j - 2^v$.

```
for i ∈ [n] do T_{i,0} := d_{i1} { base }
for j ∈ [2^n − 1] do
for i ∈ [n] do
T_{i,j} := min_{k|2^k &j=2^k} d_{ik} + T_{i,j−2^k} { tempo O(n) ! }
end for
end for
```

A complexidade de tempo desse algoritmo é $n^2 2^n$ porque a minimização na linha 4 precisa $O(n)$ operações. A complexidade de espaço é $O(n2^n)$. Isso atualmente é o melhor algoritmo exato conhecido para o problema do caixeiro viajante (veja também xkcd.com/399).

5.5.3. Cobertura por conjuntos

Considere o problema NP-completo

Cobertura por conjuntos

Instância Uma coleção de subconjuntos $\mathcal{C} = \{C_1, \ldots, C_n\}$ de um universo U.

Solução Uma seleção $S \subseteq [n]$ de subconjuntos que cobre U, i.e., $\cup_{i \in S} C_i = U$.

Objetivo Determinar a menor cobertura, i.e., minimizar $|S|$.

O problema possui uma solução com força bruta em $O(2^n m)$ sendo $m = |U|$.

Teorema 5.1
O problema de cobertura por conjuntos pode ser resolvido em tempo $O(2^m nm)$.

Prova. Construiremos um algoritmo usando programação dinâmica. Seja $S(C, i)$ o tamanho da menor cobertura para C com conjuntos $C_1, \ldots, C_i$. Temos

$$S(C,i) = \begin{cases} 0, & \text{caso } i = 0 \text{ e } C = \emptyset, \\ \infty, & \text{caso } i = 0 \text{ e } C \neq \emptyset, \\ \min\{S(C \setminus C_i, i-1), S(C, i-1)\}, & \text{caso contrário.} \end{cases}$$

A tabela correspondente possui $2^m n$ entradas e cada entrada pode ser calculada em tempo $O(m)$. ■

5.5.4. Caminho mais longo

Encontrar o caminho mais longo é um problema NP-completo. Em particular, não podemos aplicar programação dinâmica da mesma forma que no problema do caminho mais curto, porque o caminho mais longo não possui a mesma subestrutura ótima: dado um caminho mais longo $u \cdots vw$, $u \cdots v$ não é um caminho mais longo entre u e v, porque podemos passar pelo vértice w para achar um caminho ainda mais longo (ver Fig 5.1). Porém, excluindo o vértice w, $u \cdots v$ é o caminho mais longo entre u e v. Isso nos permite definir $C(s, t, V)$ como caminho mais longo entre s e t sem passar pelos vértices em V, e temos

$$C(s,t,V) = \begin{cases} \max_{u \in N^-(t) \setminus V} C(s, u, V \cup \{t\}) + d_{ut} & \text{caso } s \neq t \text{ e } N^-(t) \setminus V \neq \emptyset \\ -\infty & \text{caso } s \neq t \text{ e } N^-(t) \setminus V = \emptyset \\ 0 & \text{caso } s = t \end{cases}$$

(Observe que a recorrência possui valor $-\infty$ caso não existe caminho entre s e t.)
A tabela de programação dinâmica possui $O(n^2\, 2^n)$ entradas, e cada entrada pode ser computado em tempo $O(n)$, que resulta numa complexidade total de $O(n^3\, 2^n)$.
Um corolário é que caso o grafo é acíclico, o caminho mais longo pode ser calculado em tempo polinomial.

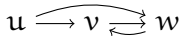


Figura 5.1.: Esquerda: O subcaminho uv de uvw não é o caminho mais longo entre u e v.

5.5.5. Notas

Mais um exemplo de programação dinâmica pode ser encontrado na página 199 no capítulo sobre estruturas de dados.

5.6. Exercícios

Soluções a partir da página 441.

Exercício 5.1
Da três exemplos de problemas que não possuem uma subestrutura ótima, i.e. a solução ótima de um problema não contém soluções ótimas de subproblemas.

Exercício 5.2
O problema do caminho mais longo em grafos direcionados acíclicos possui uma subestrutura ótima? Justifique. Caso sim, propõe um algoritmo de programação dinâmica que resolve o problema em tempo polinomial.

Exercício 5.3
O problema do empacotamento unidimensional consiste em encontrar o menor número de contêineres de um dado tamanho $T \in \mathbb{Z}_+$ que permitem empacotar n itens com tamanhos $t_1, \ldots, t_n \in \mathbb{Z}_+$. O problema de decidir se existe um empacotamento com no máximo $K \in \mathbb{Z}_+$ contêineres é NP-completo. Mostra que o problema pode ser resolvido em tempo pseudo-polinomial caso o número K de contêineres é fixo.

Exercício 5.4
Temos um vetor $a_1, \ldots, a_n, b_1, \ldots, b_n$ e queremos reordenar os elementos para produzir $a_1, b_1, \ldots a_n, b_n$. O problema é que n pode ser grande, e temos somente $O(\log n)$ memoria adicional disponível, logo temos que reordenar *in situ* (ingl. in-place).
Projete um algoritmo de programação dinâmica que consegue isso em tempo $O(n \log n)$.
Consegues resolver o problema em tempo linear?

6. Divisão e conquista

6.1. Introdução

O método de divisão e conquista *divide* um problema em subproblemas independentes, resolve ("*conquista*") os subproblemas recursivamente até obter um caso base que pode ser resolvido diretamente, e depois *combina* as soluções dos subproblemas, a fim de formar a solução do problema original. O algoritmo 6.1 resume estes passos.

Algoritmo 6.1 (DC)
Entrada Uma instância I de tamanho n.

```
if n < n_0 then
  return Solução direta do caso base
else
  Divide I em subproblemas I_1,...,I_k, k > 0.
  Resolve s_i = DC(I_i), i ∈ [k] recursivamente.
  Resolve I combinando soluções s_1,...,s_k.
end if
```

A estrutura da divisão e conquista sugere: (i) uma prova indutiva da corretude do algoritmo: prove que o algoritmo é correto nos casos base e que a combinação de soluções corretas dos subproblemas produz uma solução correta do problema; e (ii) a definição da complexidade do algoritmo por uma recorrência: a análise da complexidade consiste na solução desta recorrência. Com $d(n)$ o tempo para dividir o problema em subproblemas e $s(n)$ o tempo para combinar as sub-soluções para uma solução completa, e com n_i o tamanho do subproblema $i \in [k]$ obtemos a *recorrência natural*

$$T(n) = \begin{cases} \Theta(1), & \text{para } n < n_0, \\ \sum_{i \in [k]} T(n_i) + f(n), & \text{caso contrário,} \end{cases}$$

com $f(n) = d(n) + c(n)$. No caso balanceado a recorrência pode ser simplifi-

cada para

$$T(n) = \begin{cases} \Theta(1), & \text{para } n < n_0, \\ kT(\lceil n/m \rceil) + f(n), & \text{caso contrário.} \end{cases}$$

Para simplificar a recorrência em geral podemos (i) supor que os argumentos são inteiros, i.e., omitir pisos e tetos, e (ii) omitir a condição limite da recorrência. (Isso pode ser justificado formalmente usando o método de Akra-Bazzi explicado na seção .)

Exemplo 6.1
Um algoritmo para ordenar n números por comparação é

Algoritmo 6.2 (Mergesort)
Entrada Índices l, r e um vetor a com elementos $a_l, \ldots, a_r$.

Saída O vetor a ordenado em ordem não-decrescente.

```
MergeSort(a, l, r) :=
   if l < r then
      m := ⌊(l + r)/2⌋
      MergeSort(a, m, q)
      MergeSort(a, m + 1, r)
      Merge(a, l, m, r)
      end if
end
```

A equação de recorrência do Mergesort é

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1, \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + \Theta(n), & \text{se } n > 1, \end{cases}$$

que pode ser simplificado para

$$T(n) = 2T(n/2) + \Theta(n) \tag{6.1}$$

◊

6.2. Resolver recorrências

Existem vários métodos para resolver recorrências, entre eles a substituição, a expansão da árvore de recorrências, e os métodos Mestre e Akra-Bazzi.

6.2.1. Método da substituição

O *método da substituição* demonstra por indução que uma recorrência satisfaz um dado limite. Ele se aplica principalmente em casos onde o limite é conhecido ou simples de advinhar.

Exemplo 6.2 (Mergesort usando o método da substituição)
Supõe-se que a recorrência

$$T(n) = 2T\big(n/2\big) + \Theta(n)$$

tem limite superior $cn \log n$ para alguma constante $c > 0$.
A demonstração de $T(n) \leq cn \log n$ é por indução. Para o caso baso é suficiente observar que o trabalho para n suficientemente pequeno é constante. No passo da indução podemos supor o limite para casos menores, e demonstrar que ele vale para n por substitui-lo na recorrência:

$$\begin{aligned} T(n) &\leq 2T(n/2) + c'n \\ &\leq 2cn/2 \log(n/2 + c'n) \\ &\leq cn \log n/2 + c'n = cn \log n - cn + c'n \\ &\leq cn \log n \end{aligned}$$

para $c \geq c'$. ◊

A expressão $-cn + c'n$ na desigualdade

$$cn \log n \underbrace{-cn + c'n}_{\text{resíduo}} \leq cn \log n$$

é o *resíduo*. O objetivo da prova é mostrar que o resíduo é não-positivo.
O método de substituição sempre pode ser aplicado, tanto para demonstrar limites superiores, quanto para limites inferiores, mas ela precisa um bom palpite. Para isso podemos, por exemplo,

i) usar o resultado de recorrências semelhantes. Por exemplo, considerando a recorrência $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$, tem-se que $T(n) \in O(n \log n)$.

ii) provar limites superiores (ou inferiores) e reduzir o intervalo de incerteza. Por exemplo, para equação do Mergesort podemos provar que $T(n) = \Omega(n)$ e $T(n) = O(n^2)$. Podemos gradualmente diminuir o limite superior e elevar o inferior, até obter $T(n) = \Theta(n \log n)$.

iii) usar o resultado do método de árvore de recursão como limite hipotético para o método da substituição.

Exemplo 6.3 (Máximo de uma sequência por divisão é conquista)

Algoritmo 6.3 (Máximo)
Entrada Uma sequência de números $a_l, \ldots, a_{r-1}$.

Saída $\max_{l \le i < r} a_i$

$m_1 := \text{Máximo}(a, l, \lfloor (l+r)/2 \rfloor)$
$m_2 := \text{Máximo}(a, \lceil (l+r)/2 \rceil, r)$
`return` $\max(m_1, m_2)$

Isso leva a recorrência

$$T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$$

$\Diamond$

6.2.2. Substituição direta

Caso tem uma única chamada recursiva muitas vezes é possível substituir a recorrência diretamente e resolver o somatório resultante.

Proposição 6.1
A solução de

$$T(n) = \begin{cases} f(1), & \text{caso } n = 1, \\ T(n-1) + f(n), & \text{caso contrário.} \end{cases}$$

é $T(n) = \sum_{i \in [n]} f(i)$.

Prova. Temos

$$\begin{aligned} T(n) &= f(n) + T(n-1) = f(n) + f(n-1) + T(n-1) + f(n-2) \\ &= \cdots = \sum_{i \in [n]} f(i) \end{aligned}$$

onde $\cdots$ representa uma prova simples por indução. ■
Similarmente, caso o único subproblema é um fator constante menor temos

Proposição 6.2
A solução de

$$T(n) = \begin{cases} O(1), & \text{caso } n \leq n_0, \\ T(bn) + f(n), & \text{caso contrário,} \end{cases}$$

para $b \in (0,1)$ é $T(n) = \sum_{0 \leq i \leq \log_{1/b} n/n_0} f(b^i n) + O(1)$.

Prova. Por indução. Caso $n \leq n_0$ o somatório é vazio, porque $\log_{1/b} n/n_0 < 0$. Para $n > n_0$ obtemos

$$\begin{aligned} T(n) &= f(n) + T(bn) = f(n) + \sum_{0 \leq i \leq \log_{1/b} bn/n_0} f(b^{i+1} n) + O(1) \\ &= f(n) + \sum_{1 \leq i \leq \log_{1/b} n/n_0} f(b^i n) + O(1) = \sum_{0 \leq i \leq \log_{1/b} n/n_0} f(b^i n) + O(1). \end{aligned}$$

■

Generalizando para múltiplas chamadas recursivas homogêneas temos

Proposição 6.3
A solução de

$$T(n) = \begin{cases} O(1), & \text{caso } n \leq n_0, \\ aT(bn) + f(n), & \text{caso contrário,} \end{cases}$$

para $a \in \mathbb{N}$ e $b \in (0,1)$ é $\sum_{0 \leq i \leq \log_{1/b} n/n_0} a^i f(b^i n) + O(n^{\log_{1/b} a})$.

Prova. Primeiramente vamos remover a constante a. Dividindo a recorrência por $a^{\log_{1/b} n/n_0}$ temos

$$T(n)/a^{\log_{1/b} n/n_0} = T(bn)/a^{\log_{1/b} bn/n_0} + f(n)/a^{\log_{1/b} n/n_0}$$

e substituindo $T'(n) = T(n)/a^{\log_{1/b} n/n_0}$ e $f'(n) = f(n)/a^{\log_{1/b} n/n_0}$ obtemos a recorrência

$$T'(n) = T'(bn) + f'(n)$$

cuja solução, pela proposição 6.2, é

$$\begin{aligned} T'(n) &= \sum_{0 \leq i \leq \log_{1/b} n/n_0} f'(b^i n) + O(1) = \sum_{0 \leq i \leq \log_{1/b} n} f(b^i n)/a^{\log_{1/b} b^i n/n_0} + O(1) \\ &= 1/a^{\log_{1/b} n/n_0} \sum_{0 \leq i \leq \log_{1/b} n/n_0} a^i f(b^i n) + O(1) \end{aligned}$$

e logo

$$T(n) = a^{\log_{1/b} n/n_0} T'(n) = \sum_{0 \le i \le \log_{1/b} n} a^i f(b^i n) + O(a^{\log_{1/b} n/n_0}).$$

Com $O(a^{\log_{1/b} n/n_0}) = O(a^{\log_{1/b} n}) = O(n^{\log_{1/b} a})$ a proposição segue. ■

Exemplo 6.4
Na aula sobre a complexidade média do algoritmo Quicksort (veja página 60), encontramos uma recorrência da forma

$$T(n) = n + T(n-1)$$

cuja solução é $\sum_{i \in [n]} i = n(n-1)/2$ pela Proposição 6.1. ◊

Exemplo 6.5
Ainda na aula sobre a complexidade média do algoritmo Quicksort (veja página 60), encontramos outra recorrência da forma

$$T(n) = n + 2T(n/2).$$

cuja solução é $T(n) = \sum_{0 \le i \le \log_2 n} n + O(n) = O(n \log n)$ pela Proposição 6.3. ◊

Exemplo 6.6 (Multiplicação de números binários)
Dado dois números p, q de n bits, podemos separá-los em duas partes forma

$$p = \boxed{p_l}\boxed{p_r} = p_l 2^{n/2} + p_r$$
$$q = \boxed{q_l}\boxed{q_r} = q_l 2^{n/2} + q_r.$$

Logo o produto possui a representação

$$\begin{aligned} pq &= (p_l 2^{n/2} + p_r)(q_l 2^{n/2} + q_r) \\ &= 2^n p_l q_l + 2^{n/2} (\, p_l q_r + p_r q_l \,) + p_r q_r. \end{aligned}$$

Observação:

$$p_l q_r + p_r q_l = (p_l + p_r)(q_l + q_r) - p_l q_l - p_r q_r.$$

Portanto a multiplicação de dois números binários com n bits é possível com três multiplicações de números binários com aproximadamente $n/2$ bits.

Algoritmo 6.4 (mult-bin (Karatsuba e Ofman 1962))
Entrada Dois números binários p, q com n bits.

Saída O produto pq (com $\leq 2n$ bits).

```
if n = 1 then return pq
else
  x1 := MULT-BIN(pl, ql)
  x2 := MULT-BIN(pr, qr)
  x3 := MULT-BIN(pl + pr, ql + qr)
  return x1 2^n + (x3 − x2 − x1) 2^{n/2} + x2
end if
```

Por exemplo, com $p = (1101.1100)_2 = (220)_{10}$ e $q = (1001.0010)_2 = (146)_{10}$ temos $n = 8$, $x = p_1 + p_2 = 1.1001$ tal que $x_1 = 1$ e $x_2 = 1001$ e $y = q_1 + q_2 = 1011$ tal que $y_1 = 0$ e $y_2 = 1011$. Logo $z = x_2 y_2 = 0110.0011$, $t = y_2 2^4 + z = 21.0001.0011$, $u = p_1 q_1 = 0111.0101$, $v = p_2 q_2 = 0001.1000$ e finalmente $r = 1111.1010.111.1000 = 32120$.
O algoritmo multiplica dois números binários com no máximo $\lceil n/2 \rceil$ bits, e os números $p_l + p_r$ e $q_l + q_r$ com no máximo $\lceil n/2 \rceil + 1$ bits, e portanto possui complexidade

$$T(n) = \begin{cases} \Theta(1), & \text{se } n \leq 1, \\ 2T(\lceil n/2 \rceil) + T(\lceil n/2 \rceil + 1) + \Theta(n), & \text{caso contrário,} \end{cases}$$

que podemos simplificar para

$$T(n) = 3T(n/2) + cn.$$

Logo, pela Proposição 6.3, com $a = 3$, $b = 1/2$,

$$T(n) = \sum_{0 \leq i \leq \log_2 n} (3/2)^i cn + O(n^{\log_2 3}) = O(n^{\log_2 3}) = O(n^{1.59})$$

◊

6.2.3. Método da árvore de recursão

O método da árvore de recursão
Uma árvore de recursão apresenta uma forma bem intuitiva para a análise de complexidade de algoritmos recursivos.

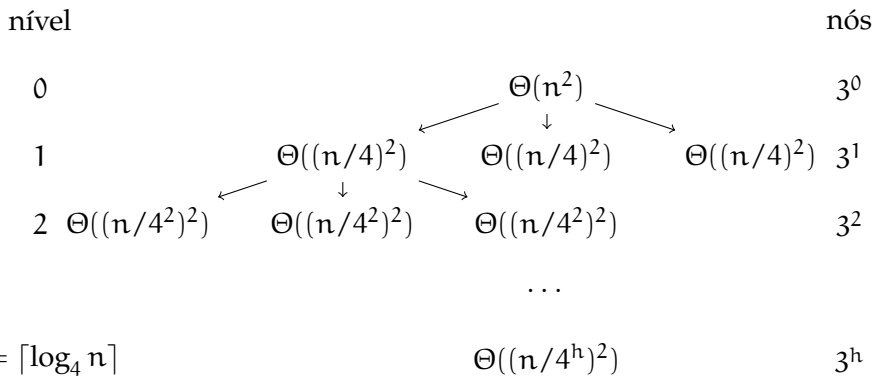
- Numa árvore de recursão cada nó representa o custo de um único subproblema da respectiva chamada recursiva
- Somam-se os custos de todos os nós de um mesmo nível, para obter o custo daquele nível
- Somam-se os custos de todos os níveis para obter o custo da árvore

Exemplo

Dada a recorrência $T(n) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$

- Em que nível da árvore o tamanho do problema é 1? No nível $i = \log_4 n = 1/2 \log_2 n$.
- Quantos níveis tem a árvore? A árvore tem $\log_4 n + 1$ níveis $(0, \ldots, \log_4 n)$.
- Quantos nós têm cada nível? 3^i.
- Qual o tamanho do problema em cada nível? $n/4^i$.
- Qual o custo de cada nível i da árvore? $3^i c(n/4^i)^2$.
- Quantos nós tem o último nível? $\Theta(n^{\log_4 3})$.
- Qual o custo da árvore? $\sum_{i=0}^{\log_4(n)} n^2 \cdot (3/16)^i = O(n^2)$.

Exemplo



Prova por substituição usando o resultado da árvore de recorrência

- O limite de $O(n^2)$ deve ser um limite restrito, pois a primeira chamada recursiva é $\Theta(n^2)$.
- Prove por indução que $T(n) = \Theta(n^2)$

$$\begin{aligned} T(n) &\leq 3T(\lfloor n/4 \rfloor) + cn^2 \\ &\leq 3d \lfloor n/4 \rfloor^2 + cn^2 \\ &\leq 3d(n/4)^2 + cn^2 \\ &= \frac{3}{16} dn^2 + cn^2 \leq dn^2. \end{aligned}$$

  para $3d/16 + c \leq d$, ou seja, para valores $d \geq 16/13\, c$

Exemplo 6.7
Considere a recorrência $T(n) = 3T(n/2) + cn$ do algoritmo de multiplicação de números binários (exemplo 6.6). A árvore tem $\log_2 n$ níveis, o nível i com 3^i nós, tamanho do problema $n/2^i$, trabalho $cn/2^i$ por nó e portanto $(3/2)^i n$ trabalho total por nível. O número de folhas é $3^{\log_2 n}$ e logo temos

$$\begin{aligned} T(n) &= \sum_{0 \leq i < \log_2 n} (3/2)^i n + \Theta(3^{\log_2 n}) \\ &= n \left(\frac{(3/2)^{\log_2 n} - 1}{3/2 - 1} \right) + \Theta(3^{\log_2 n}) \\ &= 2(n^{\log_2 3} - 1) + \Theta(n^{\log_2 3}) \\ &= \Theta(n^{\log_2 3}) \end{aligned}$$

Observe que a recorrência $T(n) = 3T(n/2) + c$ tem a mesma solução. ◊

Resumindo o método

1. Desenha a árvore de recursão
2. Determina
   - o número de níveis
   - o número de nós e o custo por nível
   - o número de folhas
3. Soma os custos dos níveis e o custo das folhas
4. (Eventualmente) Verifica por substituição

Árvore de recorrência: ramos desiguais
Calcule a complexidade de um algoritmo com a seguinte equação de recorrência

$$T(n) = T(n/3) + T(2n/3) + O(n).$$

6.2.4. Método Mestre

Método Mestre
Para aplicar o método mestre deve ter a recorrência na seguinte forma:

$$T(n) = aT(n/b) + f(n)$$

onde $a \geq 1$, $b > 1$ e $f(n)$ é uma função assintoticamente positiva. Se a recorrência estiver no formato acima, então $T(n)$ é limitada assintoticamente como:

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0$, então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \log n)$
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algum $\epsilon > 0$, e se $af(n/b) \leq cf(n)$ para $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$

Considerações

- Nos casos 1 e 3 $f(n)$ deve ser polinomialmente menor, resp. maior que $n^{\log_b a}$, ou seja, $f(n)$ difere assintoticamente por um fator n^{ϵ} para um $\epsilon > 0$.
- Os três casos não abrangem todas as possibilidades

Algoritmo Potenciação

Algoritmo 6.5 (Potenciação)
Entrada Uma base $a \in \mathbb{R}$ e um exponente $n \in \mathbb{N}$.

Saída A potência a^n.

```
P(a, n) :=
  if n = 0
```

```
    return 1
  else
    return aP(a, n − 1)
  end if
end
```

Complexidade da potenciação

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 0, \\ T(n-1) + 1, & \text{se } n > 0. \end{cases}$$

A complexidade dessa recorrência é linear, ou seja, $T(n) \in O(n)$.

Algoritmo Potenciação para $n = 2^i$

Algoritmo 6.6 (Potenciação-Npotência2)
Entrada Uma base $a \in \mathbb{R}$ e um exponente $n \in \mathbb{N}$ tal que $n = 2^i$.

Saída A potência a^n.

```
P2(a, n) :=
  if n = 1 then
    return a
  else
     x := P2(a, n/2)
     return x^2
  end if
end
```

Complexidade da potenciação-Npotência2

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 0, \\ T(\lfloor n/2 \rfloor) + c, & \text{se } n > 0. \end{cases}$$

A complexidade dessa recorrência é logarítmica, ou seja, $T(n) \in O(\log n)$

Busca Binária

Algoritmo 6.7 (Busca-Binária)
Entrada Um inteiro x, índices i e f e uma sequência $S = a_1, a_2, \ldots a_n$ de números ordenados.

Saída Posição i em que x se encontra na sequência S ou ∞ caso $x \notin S$.

```
if i = f then
  if a_i = x return i
  else return ∞
end if
m := ⌊(f + i)/2⌋
if x < a_m then
  return Busca-Binária(i, m − 1)
else
  return Busca-Binária(m + 1, f)
end if
```

Complexidade da Busca-Binária

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1, \\ T(\lfloor n/2 \rfloor) + c, & \text{se } n > 1. \end{cases}$$

A complexidade dessa recorrência é logarítmica, ou seja, $T(n) \in O(\log n)$

Quicksort

Algoritmo 6.8 (Quicksort)
Entrada Índices l, r e um vetor a com elementos $a_l, \ldots, a_r$.

Saída a com os elementos em ordem não-decrescente, i.e. para $i < j$ temos $a_i \leq a_j$.

```
if l < r then
  m := Partition(l,r,a);
  Quicksort(l,m − 1,a);
  Quicksort(m + 1,r,a);
end if
```

Complexidade do Quicksort no pior caso

$$T(n) = \begin{cases} \Theta(1), & \text{se } n = 1, \\ T(n-1) + \Theta(n), & \text{se } n > 1. \end{cases}$$

A complexidade dessa recorrência é quadrática, ou seja, $T(n) \in O(n^2)$

Complexidade do Quicksort no melhor caso

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é $T(n) \in O(n \log n)$

Complexidade do Quicksort com Particionamento Balanceado

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\frac{9n}{10}) + T(\frac{n}{10}) + \Theta(n) & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é $T(n) \in O(n \log n)$
Agora, vamos estudar dois exemplos, em que o método mestre não se aplica.

Exemplo 6.8 (Contra-exemplo 1)
Considere a recorrência $T(n) = 2T(n/2) + n \log n$. Nesse caso, a função $f(n) = n \log n$ não satisfaz nenhum dos critérios do teorema Mestre (ela fica "entre" casos 2 e 3), portanto temos que analisar com árvore de recorrência que resulta em

$$T(n) = \sum_{0 \le i < \log_2 n} (n \log n - in) + \Theta(n) = \Theta(n \log^2 n)$$

$\Diamond$

Exemplo 6.9 (Contra-exemplo 2)
Considere a recorrência $T(n) = 2T(n/2) + n/\log n$. De novo, a função $f(n) = n/\log n$ não satisfaz nenhum dos critérios do teorema Mestre (ela fica "entre" casos 1 e 2). Uma análise da árvore de recorrência resulta em

$$\begin{aligned} T(n) &= \sum_{0 \le i < \log_2 n} (n/(\log n - i)) + \Theta(n) \\ &= \sum_{1 \le j \le \log n} n/j + \Theta(n) = nH_n + \Theta(n) = \Theta(n \log \log n) \end{aligned}$$

$\Diamond$

Prova do Teorema Mestre

- Consideremos o caso simplificado em que $n = 1, b, b^2, ...$, ou seja, n é uma potência exata de dois e assim não precisamos nos preocupar com tetos e pisos.

Prova do Teorema Mestre

Lema 4.2: Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Defina $T(n)$ sobre potências exatas de b pela recorrência:

$$\begin{cases} T(n) = \Theta(1) & \text{se } n = 1 \\ T(n) = aT(\frac{n}{b}) + f(n) & \text{se } n = b^i \end{cases}$$

onde i é um inteiro positivo. Então

$$T(n) = \Theta(n^{\log_b a}) + \sum_{j=0}^{\log_b(n-1)} a^j \cdot f(n/b^j)$$

Prova do Teorema Mestre

Lema 4.4: Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Uma função $g(n)$ definida sobre potências exatas de b por

$$g(n) = \sum_{j=0}^{\log_b(n-1)} a^j \cdot f(n/b^j)$$

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0 \in \mathbb{R}^+$, então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$
3. Se $a \cdot f(n/b) \leq c \cdot f(n)$ para $c < 1$ e para todo $n \geq b$, então $g(n) = \Theta(f(n))$

Prova do Teorema Mestre
Lema 4.4: Sejam $a \geq 1$ e $b > 1$ constantes, e seja $f(n)$ uma função não negativa definida sobre potências exatas de b. Defina $T(n)$ sobre potências exatas de b pela recorrência

$$\begin{cases} T(n) = \Theta(1) & \text{se } n = 1 \\ T(n) = aT(\frac{n}{b}) + f(n) & \text{se } n = b^i \end{cases}$$

onde i é um inteiro positivo. Então $T(n)$ pode ser limitado assintoticamente para potências exatas de b como a seguir:

Prova do Teorema Mestre

1. Se $f(n) = O(n^{\log_b a - \epsilon})$ para algum $\epsilon > 0 \in \mathbb{R}^+$, então $T(n) = \Theta(n^{\log_b a})$
2. Se $f(n) = \Theta(n^{\log_b a})$, então $T(n) = \Theta(n^{\log_b a} \cdot \lg n)$
3. Se $f(n) = \Omega(n^{\log_b a + \epsilon})$ para algum $\epsilon > 0 \in \mathbb{R}^+$, e se $a \cdot f(n/b) \leq c \cdot f(n)$ para $c < 1$ e para todo n suficientemente grande, então $T(n) = \Theta(f(n))$

6.2.5. O método de Akra-Bazzi

O teorema Mestre se aplica somente no caso que a árvore de recursão está balanceado. O método de *Akra-Bazzi* (Akra e Bazzi 1998) é uma generalização do teorema Mestre, que serve também em casos não balanceados[1]. O que segue é uma versão generalizada de Leighton (1996).

Teorema 6.1 (Método Akra-Bazzi e Leighton)
Dado a recorrência

$$T(x) = \begin{cases} \Theta(1), & \text{se } x \leq x_0, \\ \sum_{i \in [k]} a_i T(b_i x + h_i(x)) + g(x), & \text{caso contrário,} \end{cases}$$

com constantes $a_i > 0$, $0 < b_i < 1$ e funções g, h, que satisfazem

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

para um $\epsilon > 0$ e a constante x_0 e suficientemente grande[2] temos que

$$T(x) \in \Theta\left(x^p \left(1 + \int_1^x \frac{g(u)}{u^{p+1}} du\right)\right)$$

com p tal que $\sum_{i \in [k]} a_i b_i^p = 1$.

[1] Uma abordagem similar foi proposta por (Roura 2001).
[2] As condições exatas são definidas em Leighton (1996).

Observação 6.1
As funções $h_i(x)$ servem particularmente para aplicar o teorema para com pisos e tetos. Com

$$h_i(x) = \lceil b_i x \rceil - b_i x$$

(que satisfaz a condição de h, porque $h_i(x) \in O(1)$) obtemos a recorrência

$$T(x) = \begin{cases} \Theta(1), & \text{se } x \leq x_0, \\ \sum_{i \in [k]} a_i T(\lceil b_i x \rceil) + g(x), & \text{caso contrário,} \end{cases}$$

demonstrando que obtemos a mesmo solução aplicando tetos. ◊

Exemplo 6.10
Considere a recorrência

$$T(n) = T(n/5) + T(7n/10 + 6) + O(n)$$

(que ocorre no algoritmo da seleção do k-ésimo elemento). Primeiro temos que achar um p tal que $(1/5)^p + (7/10)^p = 1$ que é o caso para $p \approx 0.84$. Com isso, teorema (6.1) afirma que

$$\begin{aligned} T(n) \in \Theta(n^p + (1 + \int_1^n c_1 u / u^{p+1} du)) &= \Theta(n^p (1 + c_1 \int_1^n u^{-p} du)) \\ &= \Theta(n^p (1 + \frac{c_1}{1-p} n^{1-p})) \\ &= \Theta(n^p + \frac{c_1}{1-p} n) = \Theta(n) \end{aligned}$$

◊

Exemplo 6.11
Considere $T(n) = 2T(n/2) + n \log n$ do exemplo 6.8 (que não pode ser resolvido pelo teorema Mestre). $2(1/2)^p = 1$ define $p = 1$ e temos

$$\begin{aligned} T(n) \in \Theta(n + (1 + \int_1^n \log u / u \, du)) &= \Theta(n(1 + [\log^2(u)/2]_1^n) \\ &= \Theta(n(1 + \log^2(n)/2)) \\ &= \Theta(n \log^2(n)) \end{aligned}$$

◊

Exemplo 6.12
Considere $T(n) = 2T(n/2) + n/\log n$ do exemplo 6.9 (que não pode ser resolvido pelo teorema Mestre). Novamente $p = 1$ e temos

$$\begin{aligned} T(n) \in \Theta(n + (1 + \int_1^n 1/(u \log u)\, du)) &= \Theta(n(1 + [\log\log u]_1^n) \\ &= \Theta(n(1 + \log\log n)) \\ &= \Theta(n \log(\log n)) \end{aligned}$$

◊

Observação 6.2
Caso $p > c$ temos

$$\int_1^x \frac{g(u)}{u^{p+1}} = \frac{1 - x^{c-p}}{p - c} \leq 1/(p - c)$$

e logo $T(x) = \Theta(x^p)$. ◊

6.3. Algoritmos de divisão e conquista

6.3.1. O algoritmo de Strassen

No capítulo 2.2.2 analisamos um algoritmo para multiplicar matrizes quadradas de tamanho (número de linhas e colunas) n com complexidade de $O(n^3)$ multiplicações. Um algoritmo mais eficiente foi proposto por Strassen (1969). A ideia do algoritmo é: subdivide os matrizes num produto $A \times B = C$ em quatro sub-matrizes com a metade do tamanho (e, portanto, um quarto de elementos):

$$\left(\begin{array}{c|c} A_{11} & A_{12} \\ \hline A_{21} & A_{22} \end{array}\right) \times \left(\begin{array}{c|c} B_{11} & B_{12} \\ \hline B_{21} & B_{22} \end{array}\right) = \left(\begin{array}{c|c} C_{11} & C_{12} \\ \hline C_{21} & C_{22} \end{array}\right).$$

Com essa subdivisão, o produto AB obtem-se pelas equações

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

e precisa-se de oito multiplicações de matrizes de tamanho $n/2$ ao invés de uma multiplicação de matrizes de tamanho n. A recorrência correspondente, considerando somente multiplicações é

$$T(n) = 8T(n/2) + O(1)$$

e possui solução $T(n) = O(n^3)$, que demonstra que essa abordagem não é melhor que algoritmo simples. Strassen inventou as equações

$$\begin{aligned}
M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\
M_2 &= (A_{21} + A_{22})B_{11} \\
M_3 &= A_{11}(B_{12} - B_{22}) \\
M_4 &= A_{22}(B_{21} - B_{11}) \\
M_5 &= (A_{11} + A_{12})B_{22} \\
M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\
M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \\
C_{11} &= M_1 + M_4 - M_5 + M_7 \\
C_{12} &= M_3 + M_5 \\
C_{21} &= M_2 + M_4 \\
C_{22} &= M_1 - M_2 + M_3 + M_6
\end{aligned}$$

(cuja verificação é simples). Essas equações contém somente *sete* multiplicações de matrizes de tamanho $n/2$, que leva à recorrência

$$T(n) = 7T(n/2) + O(1)$$

para o número de multiplicações, cuja solução é $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

6.3.2. Menor distância

Dado pontos $p_1, \ldots, p_n$ no plano uma abordagem direta de encontrar o par de menor distância euclidiana possui complexidade $\Theta(n^2)$. Usando divisão e conquista um algoritmo de complexidade $O(n \log n)$ é possível. A ideía é ordenar os pontos por uma coordenada, dividir em dois subconjuntos iguais, buscar o par de menor distância em cada subconjunto e juntar os resultados. Um candidato para o par de menor distância é o par de menor distância δ dos dois subconjuntos.

A combinação dos resultados é o parte mais difícil do algoritmo, porque temos que tratar o caso que o par de menor distância é entre os dois subconjuntos. Para conseguir a complexidade $O(n \log n)$ temos que encontrar um tal par, caso existe, em tempo linear no número de pontos.

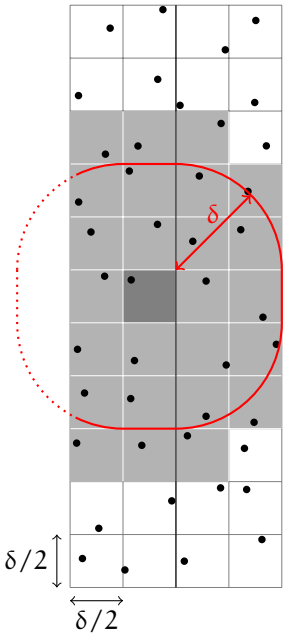


Figura 6.1.: Uma faixa de largura 2δ na divisão dos subconjuntos, dividido em celulas de tamanho $\delta/2$.

Figura 6.1 mostra como isso é possível. Sabemos que caso o par de pontos de menor distância é entre os dois subconjuntos, os dois pontos se encontram numa faixa da distância no máximo δ da linha de divisão. Subdividindo esta faixa por células quadradas de tamanho $\delta/2$, em cada célula se encontra no máximo um ponto (por quê?). Portanto numa lista dos pontos nesta faixa, ordenada pelo coordenado ortogonal ao coordenada de divisão, um par de pontos de distância menor que δ é no máximo 13 posições distante. Portanto podemos testar para cada ponto em tempo constante, se existe outro de distância menor que δ.

Algoritmo 6.9 (Menor distância)
Entrada Conjunto de pontos $p_1, \ldots, p_n$ no plano.

Pré-processamento Ordenar os pontos por x para obter $x_1, \ldots, x_n$ e por y para obter $y_1, \ldots, y_n$.

Saída O par p, q de pontos de menor distância euclidiana.

```
md(x_1,...,x_n,y_1,...,y_n) :=
  if n ≤ 3 then
    resolve o problema diretamente
  end if
  m := ⌊n/2⌋
  sejam Y_1 os pontos com x ≤ x_m ordenados por y
  sejam Y_2 os pontos com x > x_m ordenados por y
  (p, q) := md(x_1,...,x_m, Y_1)
  (p', q') := md(x_{m+1},x_n, Y_2)
  if ||p' − q'|| < ||p − q|| then (p, q) := (p', q')
  δ := ||p − q||

  Sejam y'_1,...,y'_k os pontos ordenados por y
    com distância em x menos que δ para x_m.
  for i ∈ [k] do
    for j ∈ {i,...,i + 13} ∩ [k] do
      if ||y_i − y_j|| < ||p − q|| then (p, q) := (y_i, y_j)
    end for
  end for
  return (p, q)
```

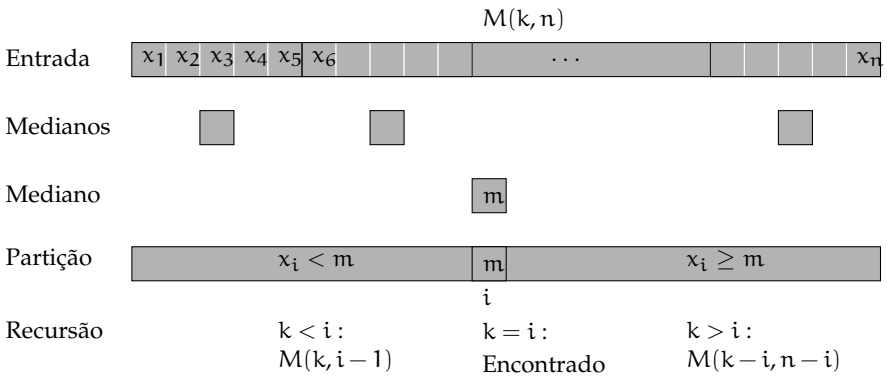


Figura 6.2.: Funcionamento do algoritmo recursivo para seleção.

Observação 6.3
O pré-processamento garante que podemos obter os pontos dos subproblemas ordenados por x e por y em tempo linear. Ordenar os pontos na rotina leva a recorrência $T_n = T_{\lfloor n/2 \rfloor} + T_{\lceil n/2 \rceil} + O(n \log n)$ com solução $O(n \log^2 n)$. ◊

6.3.3. Seleção

Dado um conjunto de números, o problema da seleção consiste em encontrar o k-ésimo maior elemento. Com ordenação o problema possui solução em tempo $O(n \log n)$. Mas existe um outro algoritmo mais eficiente. Podemos determinar o mediano de grupos de cinco elementos, e depois o recursivamente o mediano m desses medianos. Com isso, o algoritmo particiona o conjunto de números em um conjunto L de números menores que m e um conjunto R de números maiores que m. O mediano m é na posição $i := |L| + 1$ desta sequência. Logo, caso $i = k$ m é o k-ésimo elemento. Caso $i > k$ temos que procurar o k-ésimo elemento em L, caso $i < k$ temos que procurar o $k - i$-ésimo elemento em R (ver figura 6.2).

O algoritmo é eficiente, porque a seleção do elemento particionador m garante que o subproblema resolvido na segunda recursão é no máximo um fator $7/10$ do problema original. Mais preciso, o número de medianos é maior que $n/5$, logo o número de medianos antes de m é maior que $n/10 - 1$, o número de elementos antes de m é maior que $3n/10 - 3$ e com isso o número de elementos depois de m é menor que $7n/10 + 3$. Por um argumento similar, o número de elementos antes de m é também menor que $7n/10 + 3$.

Portanto temos um custo no caso pessimista de

$$T(n) = \begin{cases} \Theta(1), & \text{se } n \leq 5, \\ T(\lceil n/5 \rceil) + \Theta(7n/10 + 3) + \Theta(n), & \text{caso contrário,} \end{cases}$$

e com $5^{-p} + (7/10)^p = 1$ temos $p = \log_2 7 \approx 0.84$ e

$$\begin{aligned} T(n) &= \Theta\left(n^p \left(1 + \int_1^n u^{-p} du\right)\right) \\ &= \Theta(n^p(1 + (n^{1-p}/(1-p) - 1/(1-p)))) \\ &= \Theta(c_1 n^p + c_2 n) = \Theta(n). \end{aligned}$$

Algoritmo 6.10 (Seleção)
Entrada Números $x_1, \ldots, x_n$, posição k.

Saída O k-ésimo maior número.

$S(k, \{x_1, \ldots, x_n\}) :=$
  `if` $n \leq 5$
    `calcula e retorne o k-ésimo elemento`
  `end if`
  $m_i := \text{median}(x_{5i+1}, \ldots, x_{\min(5i+5,n)})$ `para` $0 \leq i < \lceil n/5 \rceil$.
  $m := S(\lceil \lceil n/5 \rceil / 2 \rceil, m_1, \ldots, m_{\lceil n/5 \rceil - 1})$
  $L := \{x_i \mid x_i < m, i \in [n]\}$
  $R := \{x_i \mid x_i \geq m, i \in [n]\}$
  $i := |L| + 1$
  `if` $i = k$ `then`
    `return` m
  `else if` $i > k$ `then`
    `return` $S(k, L)$
  `else`
    `return` $S(k - i, R)$
  `end if`

6.3.4. Convoluções

Dado duas sequências $a_0, \ldots, a_{n-1}$ e $b_0, \ldots, b_{n-1}$ de n números a *convolução* é uma sequência $c_0, \ldots, c_{2n-2}$ de $2n-1$ números definido por

$$c_k = \sum_{(i,j):i+j=k} a_i b_j. \tag{6.2}$$

Os próximos exemplos mostram a utilidade de convoluções.

Exemplo 6.13 (Multiplicação de polinômios)
Dois polinômios de grau $n-1$ em forma de coeficientes

$$A(x) = \sum_{0 \le i < n} a_i x^i; \qquad B(x) = \sum_{0 \le i < n} b_i x^i$$

possuem o produto de grau $2n-2$

$$C(x) = \Big(\sum_{0 \le i < n} a_i x^i \Big) \Big(\sum_{0 \le i < n} b_i x^i \Big) = \sum_{0 \le i < 2n-1} \sum_{0 \le j \le i} a_j b_{i-j},$$

cujos coeficientes são convoluções das coeficientes dos polinômios. ◊

Exemplo 6.14
Dado duas variáveis aleatórias inteiras independentes X e Y com valor em $[0, n)$, seja $x_i = \Pr(X = i)$ e $y_i = \Pr(Y = i)$. A distribuição da variável $X + Y$ é

$$\Pr(X + Y = k) = \sum_{(i,j):i+j=k} \Pr(X = i) \Pr(Y = j),$$

uma convolução das distribuições da variáveis individuais. ◊

Exemplo 6.15 (Bluestein (1970), Algoritmo "chirp-z")
A transformada de Fourier discreta de uma sequência $x_0, \ldots, x_{n-1}$ e a sua inversa é definido por

$$X_k = \sum_{0 \le j < n} x_j \omega^{kj}; \qquad x_k = 1/n \sum_{0 \le j < n} X_j \omega^{-kj}$$

com $\omega = e^{-2\pi i/n}$. Usando $kj = (k^2 + j^2 - (k-j)^2)/2$ podemos escrever

$$\begin{aligned} X_k &= \sum_{0 \le j < n} x_j \omega^{(k^2 + j^2 - (k-j)^2)/2} \\ &= \omega^{k^2/2} \sum_{0 \le j < n} (x_j \omega^{j^2/2}) \omega^{(-(k-j)^2)/2} \end{aligned}$$

como convolução

$$\omega^{k^2/2} \sum_{0 \le j < n} a_j b_{k-j}$$

das sequências $a_j = x_j \omega^{j^2/2}$ e $b_j = \omega^{-j^2/2}$. Uma transformação similar se aplica para a inversa. ◊

A convolução de duas sequências pode ser calculado em $O(n^2)$, calculando cada um dos $2n-1$ coeficientes da equação (6.2) em tempo $O(n)$. Para calcular a convolução mais eficientemente, vamos focar no caso da multiplicação de polinômios. Observando que um polinômio de grau d é definido por $d+1$ pontos o seguinte processo pode ser aplicado no exemplo 6.13:

Avaliação Avalia $A(x)$ e $B(x)$ em $2n-1$ pontos $x_0, \ldots, x_{2n-2}$. Trataremos $A(x)$ e $B(x)$ como polinômios de grau $2n-2$ com coeficientes $a_i = b_i = 0$ para $i \ge n$ para simplificar a discussão que segue. O problema deste passo então é avaliar um polinômio de grau d em $d+1$ pontos.

Multiplicação Calcula $C(x_i) = A(x_i)B(x_i)$. $C(x)$ possui grau $2n-2$.

Interpolação Determina os coeficientes de $C(x)$.

Com isso a multiplicação possui complexidade $O(n)$. Para obter um algoritmo mais eficiente que a abordagem direta temos que avaliar e interpolar em tempo $o(n^2)$.

Observação 6.4 (Avaliação e interpolação)
A avaliação de um polinômio $A(x)$ de grau $n-1$ em n pontos $x_0, \ldots, x_{n-1}$ pode ser escrito da forma

$$\begin{pmatrix} A(x_0) \\ A(x_1) \\ \vdots \\ A(x_{n-1}) \end{pmatrix} = \begin{pmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \cdots & & & & \\ 1 & x_{n-1} & x_{n-1}^2 & \cdots & x_{n-1}^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix}$$

Uma matriz desta forma se chama uma *matriz Vandermonde*. Uma matriz Vandermonde possui determinante $\prod_{0 \le i < j < n} x_i - x_j$. Logo, caso todos pares x_i e x_j são diferentes, ela possui inversa e a multiplicação de um vetor de valores nos pontos $A(x_0), \ldots, A(x_{n-1})$ nos fornece os coeficientes a_i, i.e., resolve o problema da interpolação. ◊

Avaliação em $O(n \log n)$ A avaliação de $A(x) = \sum_{0 \le i < n} a_i x^i$ pode ser separado em contribuições pares e impares

$$A(x) = \sum_{0 \le 2i < n} a_{2i} x^{2i} + \sum_{0 \le 2i+1 < n} a_{2i+1} x^{2i+1} = A_p(x^2) + x A_i(x^2)$$

com

$$A_p(x) = \sum_{0 \le 2i < n} a_{2i} x^i; \qquad A_i(x) = \sum_{0 \le 2i+1 < n} a_{2i+1} x^i.$$

Aplicando essa recursão para o ponto x tem custo $\Theta(n)$, logo um custo total de $\Theta(n^2)$ para a avaliação de n pontos. Observação: a avaliação do par $\pm x$ possui o mesmo custo! Logo podemos avaliar o polinômio $A(x)$ de grau $n-1$ nos n pontos $\pm x_0, \ldots, \pm x_{n/2-1}$ com duas avaliações de polinômios de grau $(n-1)/2$ nos $n/2$ pontos $x_0^2, \ldots, x_{n-1}^2$ e um trabalho adicional de $O(n)$. Sendo $T(n)$ o tempo para avaliar um polinômio de grau $n-1$ em n pontos temos

$$T(n) \le 2T(n/2 + 1/2) + O(n) = O(n \log n).$$

Esta recorrência somente está correta, caso temos sempre pares de números $\pm x_i$ em todos níveis. Como no primeiro nível temos números x_i^2, isso não é obviamente possível. Para garantir isso, podemos observar que os n n-ésimas raízes complexas $e^{k2\pi i/n}$, $0 \le k < n$ da unidade satisfazem essa condição: temos $-e^{k2\pi i/n} = e^{\pi i} e^{k2\pi i/n} = e^{(n/2+k)2\pi i/n}$ para $0 \le k < n/2$ e os quadrados desses números são os $n/2$ $n/2$-ésimas raízes complexas $e^{k2\pi i/(n/2)}$, $0 \le k < n/2$ da unidade. Em total a avaliação procede em $\log_2 n$ níveis

$$\begin{aligned}
1, e^{2\pi i/n}, e^{2\cdot 2\pi i/n}, e^{3\cdot 2\pi i/n}, \ldots, e^{(n-1)\cdot 2\pi i/n} \\
1, e^{2\pi i/(n/2)}, e^{2\cdot 2\pi i/(n/2)}, e^{3\cdot 2\pi i/(n/2)}, \ldots, e^{(n/2-1)\cdot 2\pi i/(n/2)} \\
\ldots \\
1, i, -1, -i \\
1, -1 \\
1
\end{aligned}$$

Algoritmo 6.11 (FFT)
Entrada Polinômio $A(x) = \sum_{0 \le i < n} a_i x^i$ de grau $n-1$ ($n = 2^k$) e a n-ésima raiz da unidade $\omega = e^{2\pi i/n}$.

Saida Os valores $A(\omega^j)$, $0 \le j < n$.

```
fft(a_0,...,a_{n-1},ω) :=
  if ω = 1 return A(1)
  { A(x) = A_p(x^2) + xA_i(x^2) }
  fft(a_0,a_2,...,a_{n-2},ω^2) { avalia A_p }
  fft(a_1,a_3,...,a_{n-1},ω^2) { avalia A_i }
  for j ∈ [0,n) do
    A(ω^j) = A_p(ω^{2j}) + w^j A_i(ω^{2j})
  return A(ω^0),...,A(ω^{n-1})
```

Observação: O algoritmo funciona igualmente para ω^{-1}.

Interpolação em $O(n \log n)$ A matriz de avaliação usando as raizes complexas é

$$V = V(\omega) := \begin{pmatrix} 1 & \omega^0 & \omega^0 & \cdots & \omega^0 \\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{n-1} \\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(n-1)} \\ & & \cdots & & \\ 1 & \omega^{n-1} & \omega^{2n-2} & \cdots & \omega^{(n-1)(n-1)} \end{pmatrix} = (\omega^{ij})_{i,j}$$

Pela observação 6.4 essa matriz possui inversa e a interpolação é simplesmente a multiplicação com essa inversa.
Neste caso específico é simples obter a inversa observando que a inversa de uma matriz unitária é a sua conjugada transposta e que $V/\sqrt{n}$ é unitária.

Fato 6.1
Lembramos que o número complexo $a + ib$ possui conjugada $(a+ib)^* = a - ib$, em particular $(e^{i\varphi})^* = e^{-i\varphi}$. O produto de dois vetores complexos é $\langle x, y \rangle = xy^*$. A conjugada transposta de uma matriz é $V^\dagger = V^{t*}$. Matrizes unitárias satisfazem $VV^\dagger = I$.

Lema 6.1
A matriz $V/\sqrt{n}$ é unitária.

Prova. Um critério para uma matriz $\mathbb{C}^{n \times n}$ ser unitária é que as colunas

formam uma base ortonormal de $\mathbb{C}^n$. O produto das colunas j e k é

$$1/n \sum_{0 \le l < n} \omega^{lj} (\omega^{lk})^* = 1/n \sum_{0 \le l < n} (\omega^{j-k})^l = \frac{1 - \omega^{(j-k)n}}{1 - \omega^{j-k}} = [j = k]$$

O segundo passo é justificado porque se trata de uma série geométrica, e o último passo porque $\omega^n = 1$ tal que o produto é 0, exceto no caso $j = k$ no qual a soma é n. ■

Logo $VV^\dagger = nI$ que implica $V^{-1} = V^\dagger/n = V(\omega^{-1})/n$. Essa última observação nos fornece uma maneira fácil de interpolar n valores $A(\omega^0), \ldots, A(\omega^{n-1})$: é suficiente chamar

`fft(`$A(\omega^0), \ldots, A(\omega^{n-1})$`,`$\omega^{-1}$`)`

e dividir os coeficientes obtidos por n.

6.4. Notas

O algoritmo 6.4 para multiplicar números binários se aplica igualmente para números em bases arbitrárias. Ele é conhecido como algoritmo de Karatsuba (Karatsuba e Ofman 1962). Um algoritmo mais eficiente é do Schönhage e Strassen (Schönhage e Strassen 1971) que multiplica em $O(n \log n \log\log n)$. Fürer (2007) apresenta um algoritmo que multiplica em $n \log n\, 2^{O(\log^* n)}$, um pouco acima do limite inferior $\Omega(n \log n)$.

6.5. Exercícios

(Soluções a partir da página 441.)

Exercício 6.1
Resolva as seguintes recorrências

(a) $T(n) = 9T(n/3) + n$

(b) $T(n) = T(2n/3) + 1$

(c) $T(n) = 3T(n/4) + n \log n$

(d) $T(n) = 2T(n/2) + n \log n$

(e) $T(n) = 4T(n/2) + n^2 \lg n$

(f) $T(n) = T(n-1) + \log n$

(g) $T(n) = 2T(n/2) + n/\log n$

(h) $T(n) = 3T(n/2) + n\log n$

Exercício 6.2
Aplique o teorema mestre nas seguintes recorrências:

(a) $T(n) = 9T(n/3) + n$

(b) $T(n) = T(2n/3) + 1$

(c) $T(n) = 3T(n/4) + n\log n$

(d) $T(n) = 2T(n/2) + n\log n$

Exercício 6.3
Prove a complexidade do algoritmo 6.4 por indução. Porque a prova com a hipótese $T(n) \leq cn^{\log_2 3}$ falha? Qual é uma hipótese que permite demonstrar a complexidade?

Exercício 6.4
Prove a complexidade do algoritmo de Strassen (página 139) usando o método da árvore de recursão e por indução.

Exercício 6.5
Prove a complexidade do algoritmo da seleção (página 143) do k-ésimo elemento usando o método da árvore de recursão.

Exercício 6.6
A recorrência na análise do algoritmo de Strassen leva em conta somente multiplicações. Determina e resolve a recorrência das multiplicações e adições.

Exercício 6.7
Porque o algoritmo 6.10 para selecionar o k-ésimo elemento não trabalha com grupos de três elementos? Analisa a complexidade do algoritmo neste caso.

Exercício 6.8
Aplica o método de substituição ao recorrência do MergeSort (6.1) com palpite $O(n^2)$ e $O(n)$. Explica porque a prova por indução funciona no primeiro mas não no segundo caso.

Exercício 6.9
- Considere $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$. Prove que $T(n) = O(n)$.

- Considere $T(n) = 2T(\lfloor n/2 \rfloor) + n$. É possível provar que $T(n) = O(n)$?
- Considere $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$. Prove que $T(n) = O(\log n \log\log n)$

Exercício 6.10
Formula uma versão iterativa do Mergesort.

Exercício 6.11
Aplica o método de substituição direta para resolver recorrências da forma $T(n) = cT(n/d)$.

7. Árvores de busca, backtracking e branch-and-bound

7.1. Backtracking

Motivação

- Conhecemos diversas técnicas para resolver problemas.
- O que fazer se eles não permitem uma solução eficiente?
- Para resolver um problema: pode ser necessário buscar em todo espaço de solução.
- Mesmo nesse caso, a busca pode ser mais ou menos eficiente.
- Podemos aproveitar
  - restrições conhecidas: *Backtracking* (retrocedimento).
  - limites conhecidos: *Branch-and-bound* (ramifique-e-limite).

Supomos que uma solução $(s_1, s_2, \ldots, s_n)$ do problema pode ser construído elemento por elemento, tal que cada elemento s_i pertence a um conjunto S_i de candidatos. Para solução parcial $(s_1, s_2, \ldots, s_i)$ escreveremos $\overline{s}_i$.

Backtracking: Árvore de busca

- Seja uma solução dado por um vetor $(s_1, s_2, \ldots, s_n)$ com $s_i \in S_i$.
- Queremos somente soluções que satisfazem uma propriedade $P_n(s_1, \ldots, s_n)$.
- Ideia: Refinar a busca por força bruta, aproveitando restrições cedo
- Define propriedades $P_i(\overline{s}_i)$ tal que

$$\neg P_i(\overline{s}_i) \Rightarrow \neg P_{i+1}(\overline{s}_{i+1})$$

- Os P_i são condições necessários das soluções parciais que soluções maiores (ou completas) tem que satisfazer.

Com essa ideia podemos definir um algoritmo genérico de backtracking como segue.

Algoritmo 7.1 (Backtracking)
Entrada Um problema de busca ou otimização, tal que soluções parciais $(\bar{s}_i)$ satisfazem a condição $P_i(\bar{s}_i)$.

Solução Uma ou todas soluções em $\{s \mid P_n(s)\}$.

```
BT(s̄_i) :=
  if i = n then
    processa solução (s_1, s_2, ..., s_n)
  else
    for s_{i+1} ∈ S_{i+1} | P_{i+1}(s̄_{i+1}) do
      BT(s̄_{i+1})
    end for
  end if
end
```

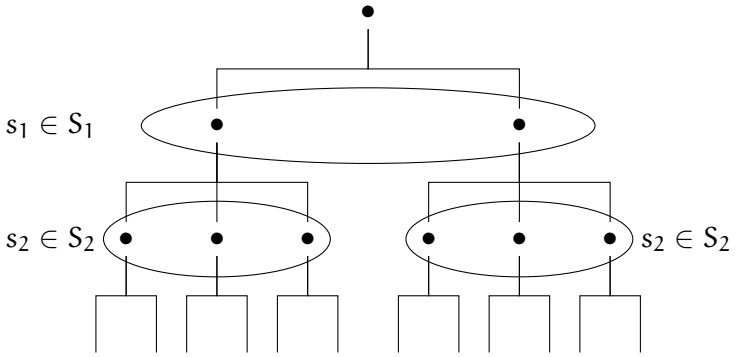
Backtracking: Árvore de busca

- A árvore de busca $T = (V, A, r)$ é definido por

$$
\begin{aligned}
V &= \{(s_1, \ldots, s_i) \mid P_i(s_1, \ldots, s_i)\} \\
A &= \{(v_1, v_2) \in V \mid v_1 = (s_1, \ldots, s_i), v_2 = (s_1, \ldots, s_{i+1})\} \\
r &= ()
\end{aligned}
$$

- Backtracking busca nessa árvore em profundidade.

- Observe que é suficiente manter o caminho da raiz até o nodo atual na memória.



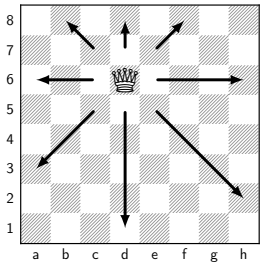
7.1.1. Exemplo: O problema das n-rainhas

O problema das n-rainhas

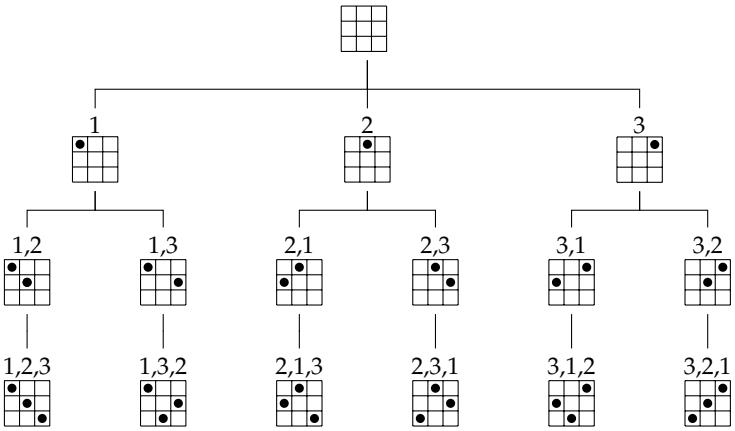
PROBLEMA DAS N-RAINHAS

Instância Um tablado de xadrez de dimensão $n \times n$, e n rainhas.

Solução Todas as formas de posicionar as n rainhas no tablado sem que duas rainhas estejam na mesma coluna, linha ou diagonal.



O problema das n-rainhas (simplificado: sem restrição da diagonal)
O que representam as folhas da árvore de busca para este problema?



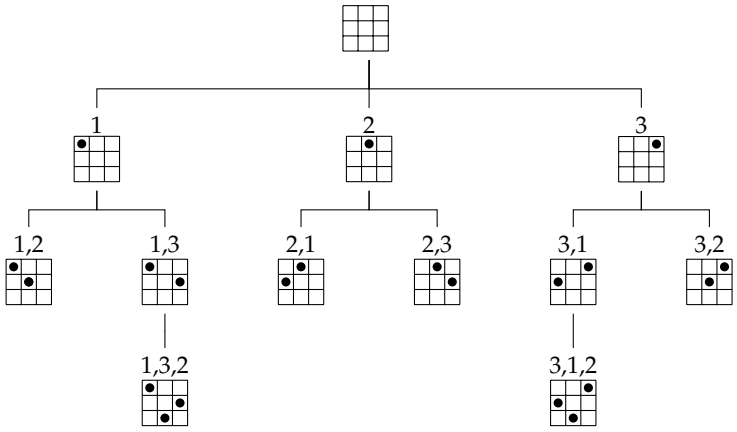
O problema das n-rainhas

- A melhor solução conhecida para este problema é via Backtracking.
- Existem $\binom{n^2}{n}$ formas de posicionar n rainhas no tablado.
- Restringe uma rainha por linha: n^n.
- Restringe ainda uma rainha por coluna problema: $n!$.
- Pela aproximação de Stirling

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n (1 + O(1/n)) \tag{7.1}$$

O problema das n-rainhas

Se considerarmos também a restrição de diagonal podemos reduzir ainda mais o espaço de busca (neste caso, nenhuma solução é factível)



Backtracking

- Testa soluções sistematicamente até que a solução esperada seja encontrada
- Durante a busca, se a inserção de um novo elemento "não funciona", o algoritmo retorna para a alternativa anterior (*backtracks*) e tenta um novo ramo
- Quando não há mais elementos para testar, a busca termina
- É apresentado como um algoritmo recursivo
- O algoritmo mantém somente uma solução por vez

Backtracking

- Durante o processo de busca, alguns ramos podem ser evitados de ser explorados
  1. O ramo pode ser infactível de acordo com restrições do problema
  2. Se garantidamente o ramo não vai gerar uma solução ótima

7.1.2. Exemplo: Caixeiro viajante

O problema do caixeiro viajante
Encontrar uma rota de menor distância tal que, partindo de uma cidade inicial, visita todas as outras cidades uma única vez, retornando à cidade de partida ao final.

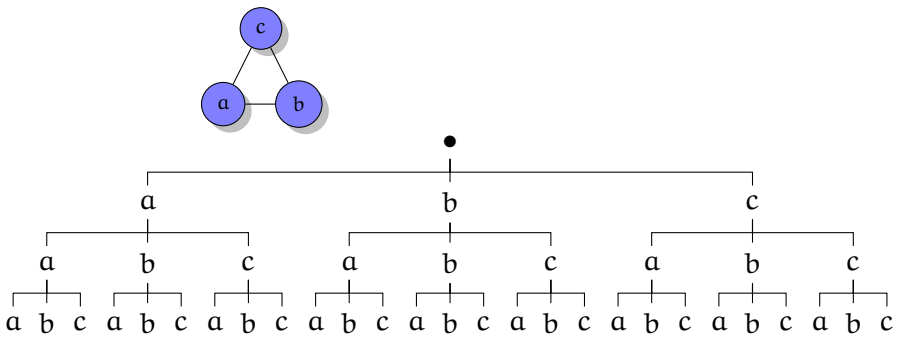
PROBLEMA DO CAIXEIRO VIAJANTE

Instância Um grafo G=(V,E) com pesos p (distâncias) atribuídos aos links. $V = [1, n]$ sem perda de generalidade.

Solução Uma rota que visita todos vértices exatamente uma vez, i.e. uma permutação de $[1, n]$.

Objetivo Minimizar o custo da rota $\sum_{1 \leq i < n} p_{\{i,i+1\}} + p_{\{n,1\}}$.

O problema do caixeiro viajante



7.1.3. Exemplo: Cobertura por vértices

Considere

COBERTURA POR VÉRTICES (INGL. VERTEX COVER)

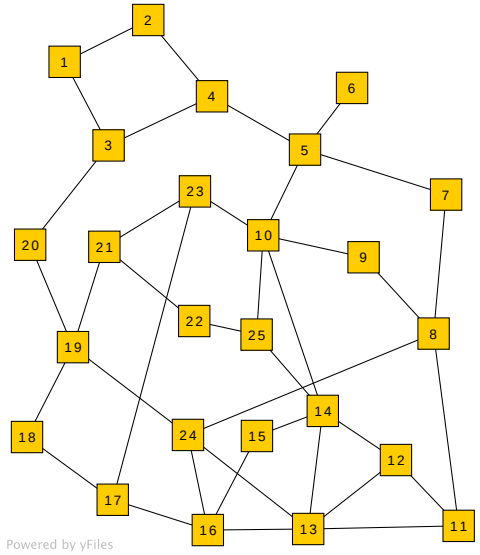
Instância Grafo não-direcionado $G = (V, E)$.

Solução Uma cobertura $C \subseteq V$, i.e. $\forall e \in E : e \cap C \neq \emptyset$.

Objetivo Minimiza $|C|$.

A versão de decisão de COBERTURA POR VÉRTICES é NP-completo.

O que fazer?



Simplificando o problema

- Vértice de grau 1: Usa o vizinho.
- Vértice de grau 2 num triângulo: Usa os dois vizinhos.

Algoritmo 7.2 (Redução de cobertura por vértices)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Um conjunto $C \subseteq V$ e um grafo G', tal que cada cobertura de vértices contém C, e a união de C com a cobertura mínima de G' é uma cobertura mínima de G.

```
Reduz(G) :=
  while (alguma regra em baixo se aplica) do
    Regra 1:
      if ∃u ∈ V : deg(u) = 1 then
        seja {u, v} ∈ E
        C := C ∪ {v}
        G := G − {u, v}
      end if
    Regra 2:
      if ∃u ∈ V : deg(u) = 2 then
        seja {u, v}, {u, w} ∈ E
        if {v, w} ∈ E then
          C := C ∪ {v, w}
          G := G − {u, v, w}
        end if
  end while
  return (C, G)
```

Uma solução exata com busca exaustiva:

Árvore de busca

Algoritmo 7.3 (Árvore de busca)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura por vértices $S \subseteq V$ mínima.

```
minVertexCover(G):=
  if E = ∅ return ∅
  escolhe {u, v} ∈ E
  C1 := minVertexCover(G − u) ∪ {u}
  C2 := minVertexCover(G − v) ∪ {v}
  return a menor cobertura C1 ou C2
```

Solução ótima?

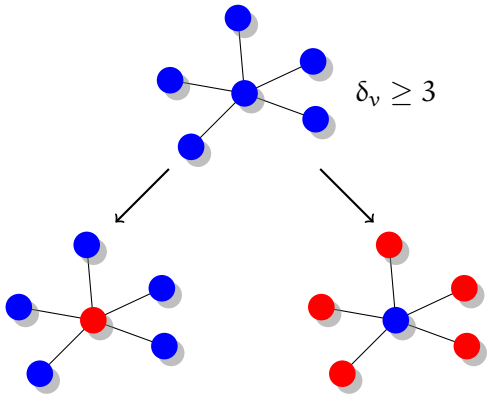
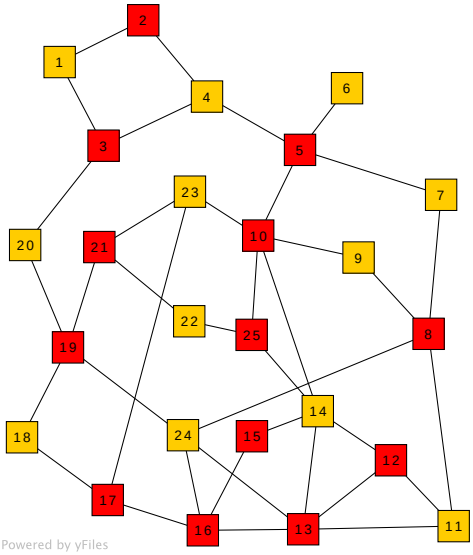


Figura 7.1.: Cobertura mínima: Dois subcasos para vértice v.



A complexidade da solução acima satisfaz $T_n = 2T_{n-1} + \Theta(n) = \Theta(2^n)$.

Observação 7.1

- 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 $N(v)$ (ver figura 7.1).

$\Diamond$

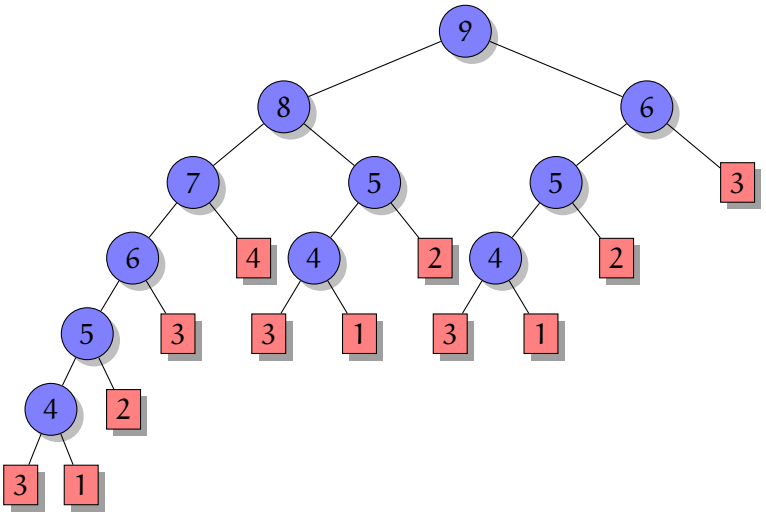


Figura 7.2.: Cobertura mínima: Árvore de busca para $n = 9$

```
mvc'(G) :=
  if Δ(G) ≤ 2 then
    determina a cobertura mínima C em tempo O(n)
    return C
  end if
  seleciona um vértice v com grau δ_v ≥ 3
  C_1 := {v} ∪ mvc'(G \ {v})
  C_2 := N(v) ∪ mvc'(G \ N(v) \ {v})
  return a menor cobertura entre C_1 e C_2
```

Qual a complexidade do algoritmo? Observe que existem mais folhas que nós internos: para simplificar vamos só contar folhas. O número de folhas obedece

$$T(n) \leq T(n-1) + T(n-4)$$

Para resolver essa recorrência podemos aplicar teorema 7.1 com vetor de bifurcação $(1, 4)$. O polinômio característico é $z^4 - z^3 + 1$ e possui a maior raiz $\alpha \approx 1.39$ com multiplicidade 1. Portanto o número de folhas é $\Theta(\alpha^n)$ e a complexidade $O(\alpha^n)$.

3-SAT

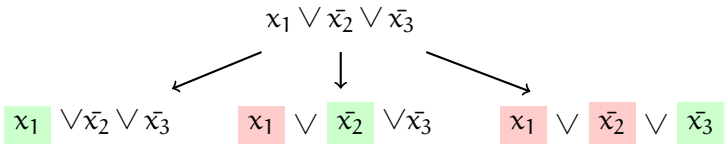
- Fórmula em *forma normal conjuntiva* com 3 literais por claúsula

$$\varphi(x_1, \ldots, x_n) = \bigwedge_i l_{i1} \vee l_{i2} \vee l_{i3}$$

- Problema: Existe *atribuição* $a : x_i \mapsto \{F, V\}$ que satisfaz φ?
- Força bruta: $\Theta(2^n)$ testes.

3-SAT

- Árvore de busca (Monien e Speckenmeyer 1985)



T(n) = T(n-1) + T(n-2) + T(n-3)

- Vetor de bifurcação $(1, 2, 3)$.
- Polinômio característico: $z^3 - z^2 - z^1 - 1$.
- Maior raiz: $\alpha \approx 1.84$.
- Número de folhas: $\Theta(\alpha^n)$.
- Complexidade: $O(\alpha^n)$.

Exemplo 7.1 (Conjunto independente máximo)

CONJUNTO INDEPENDENTE MÁXIMO

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

Solução Um conjunto *independente* $M \subseteq V$, i.e. para todo $m_1, m_2 \in V$ temos $\{m_1, m_2\} \notin E$.

Objetivo Maximiza a cardinalidade $|M|$.

Uma abordagem similar com a cobertura por vértices resolve o problema: caso o grau máximo do grafo é dois, o problema possui uma solução em tempo linear. Caso contrário existe ao menos um vértice de grau três. Caso o vértice faz parte do conjunto independente máximo podemos remover $\{v\} \cup N(v)$ do grafo, para encontrar a conjunto independente máximo no grafo restante. Caso contrário podemos remover v. Com isso obtemos a recorrência

$$T(n) \leq T(n-1) + T(n-4)$$

para o número de folhas da árvore de busca. A recorrência possui solução $O(1.39^n)$, melhor que a abordagem direta de testar os 2^n subconjuntos. $\Diamond$

7.1.4. Tópicos

O desempenho de um algoritmo de backtracking depende fortemente do número de nodos na árvore de busca correspondente e do custo para percorrer os nodos. O seguinte método de Knuth (1975) pode ser usado para estimar estas quantidades via amostragem.

Algoritmo 7.4 (Estimativa do custo do backtracking)
Entrada Um problema de busca ou otimização, tal que soluções parciais $(\overline{s}_i)$ satisfazem a condição $P_i(\overline{s}_i)$.

Saída Uma estimativa dos custos c e do número de nodos n na árvore de backtracking.

```
E(s̄_i, n, c) :=
  if i = n then
    return (n, c)
  else
    seja S := {s_{i+1} | P_{i+1}(s̄_{i+1})}
    seleciona s̄_{i+1} ∈ S uniformemente
    return E(s̄_{i+1}, n|S|, c(s̄_i)n|S|)
  end if
end
```

7.2. Branch-and-bound

Branch-and-bound

Ramifica-e-limite (ingl. branch-and-bound, Land e Doig (1960))

- Técnica geral para problemas combinatoriais.

  > Branch and Bound is by far the most widely used tool for solving large scale NP-hard combinatorial optimization problems. (Clausen 1999)

- Ideia básica:
  - Particiona um problema em subproblemas disjuntos e procura soluções recursivamente.
  - Evite percorrer toda árvore de busca, calculando limites e cortando sub-árvores.
- Particularmente efetivo para programas inteiras: a relaxação linear fornece os limites.

Limitar

- Para cada sub-árvore mantemos um limite inferior e um limite superior.
  - Limite inferior: Valor da melhor solução encontrada na sub-árvore.
  - Limite superior: Estimativa (p.ex. valor da relaxação linear na PI)
- Observação: A eficiência do método depende crucialmente da qualidade do limite superior.

Cortar sub-árvores

Podemos cortar ...

(1) por inviabilidade: Sub-problema é inviável.

(2) por limite: Limite superior da sub-árvore $\bar{z}_i$ menor que limite inferior global $\underline{z}$ (o valor da melhor solução encontrada).

(3) por otimalidade: Limite superior $\bar{z}_i$ igual limite inferior $\underline{z_i}$ da sub-árvore.

Observação: Como os cortes dependem do limite $\underline{z}$, uma boa solução inicial pode reduzir a busca consideravelmente.

Ramificar

- Não tem como cortar mais? Escolhe um nó e particiona.
- Qual a melhor ordem de busca?
- Busca por profundidade
  - V: Limite superior encontrado mais rápido.
  - V: Pouca memória ($O(\delta d)$, para δ subproblemas e profundidade d).
  - V: Re-otimização eficiente do pai (método Simplex dual)
  - D: Custo alto, se solução ótima encontrada tarde.
- Melhor solução primeiro ("best-bound rule")
  - V: Procura ramos com maior potencial.
  - V: Depois encontrar solução ótima, não produz ramificações supérfluas.
- Busca por largura? Demanda de memória é impraticável.

Em resumo: um algoritmo de branch-and-bound consiste de quatro componentes principais:

- Uma heurística que encontra uma boa solução inicial;
- um limite inferior (no caso de minimização) ou superior (para maximização) do valor de um subproblema;
- uma estratégia de ramificação, que decompõe um problema em subproblemas;
- uma estratégia de seleção, que escolhe o próximo subproblema entre os subproblemas ativos.

Algoritmos B&B

Algoritmo 7.5 (B&B)
Instância Programa inteiro $P = \max\{c^t x \mid Ax \leq b, x \in Z^n_+\}$.

Saida Solução inteira ótima.

```
{ usando função z̄ para estimar limite superior }
```

```
z := −∞          { limite inferior }
A := {(P, g(P))}          { nós ativos }
while A ≠ ∅ do
  Escolhe: (P, g(P)) ∈ A; A := A \ (P, g(P))
  Ramifique: Gera subproblemas P1,...,Pn.
  for all Pi, 1 ≤ i ≤ n do
    { adiciona, se permite melhor solução }
    if z̄(Pi) > z then
      A := A ∪ {(Pi, z̄(Pi))}
    end if
    { atualize melhor solução }
    if (solução z̄(Pi) é viável) then
      z := z̄(Pi)
    end if
  end for
end while
```

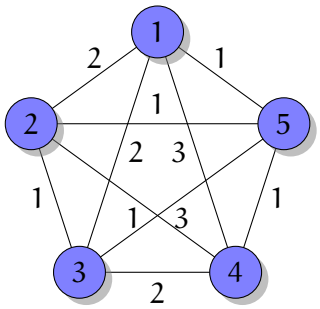
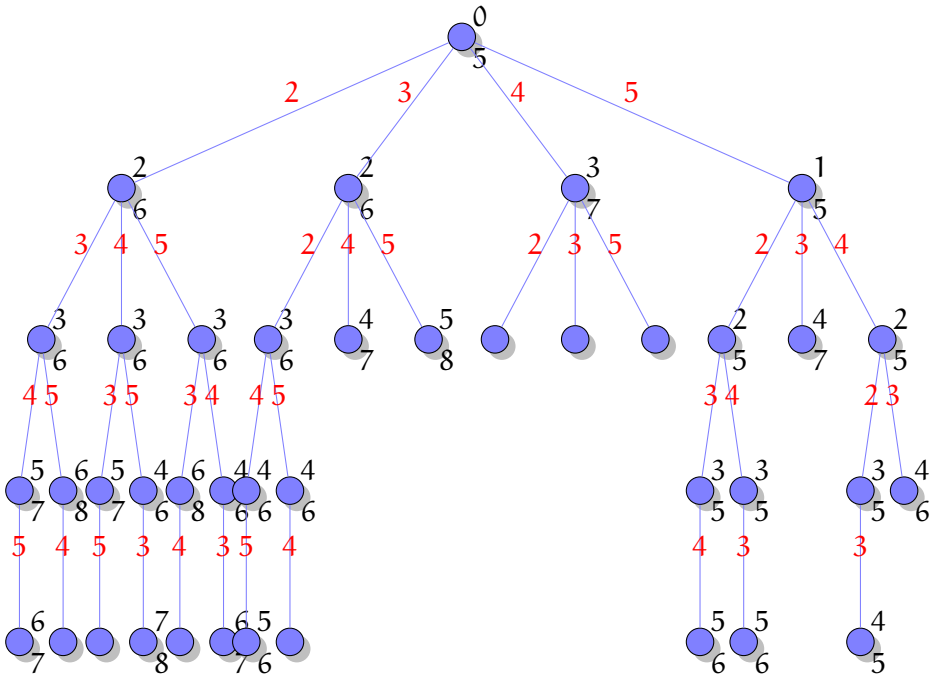


Figura 7.3.: Exemplo de uma instância do PCV.

Exemplo 7.2 (Aplicação Branch-and-Bound no PCV)
Considera uma aplicação do PCV no grafo da Figura 7.3.
Aplicando somente backtracking obtemos a seguinte árvore de busca:



A árvore de backtracking completa possui 65 vértices (por nível: 1,4,12,24,24).
Usando como limite inferior o custo atual mais o número de arcos que faltam

vezes a distância mínima e aplicando branch-and-bound obtemos os custos parciais e limites indicados na direita de cada vértice. Com isso podemos aplicar uma série de cortes: busca da esquerda para direito obtemos

- uma nova solução 7 em 2345;
- um corte por limite em 235;
- um corte por otimalidade em 243;
- um corte por otimalidade em 2453;
- um corte por limite em 253;
- um corte por otimalidade em 2543;
- uma nova solução 6 em 3245;
- um corte por otimalidade em 32;
- um corte por otimalidade em 3;
- um corte por limite em 4;
- um corte por otimalidade em 5234;
- um corte por otimalidade 5243;
- um corte por limite em 53;
- um corte por otimalidade 543.

$\Diamond$

Exemplo 7.3 (Escalonamento de tarefas)
Considera o problema de escalonamento $1 \mid r_j \mid L_{max}$: temos n tarefas a serem executadas numa única máquina. Cada tarefa possui um tempo de execução p_j e é disponível a partir do tempo r_j (release date) e idealmente tem que terminar antes do prazo d_j (due date). Caso a tarefa j termina no tempo C_j o seu atraso é $L_j = \max\{0, C_j - d_j\}$. Uma tarefa tem que ser executada sem interrupção. Queremos encontrar uma sequenciamento das tarefas tal que o atraso máximo é minimizado. (Observe que uma solução é uma permutação das tarefas.)
Um exemplo de uma instância com quatro tarefas é

Tarefa	1	2	3	4
p_j	4	2	6	5
r_j	0	1	3	5
d_j	8	12	11	11

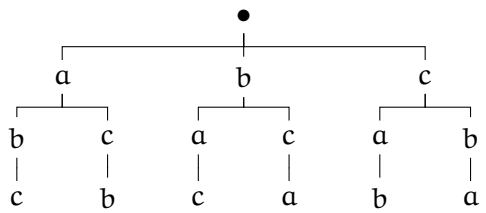
Uma abordagem via branch-and-bound é explorar todas permutações possíveis. Um limite inferior bom para a função objetivo pode ser obtido como segue: o problema sem *release dates* $1 \mid\mid L_{max}$ possui uma solução simples polinomial, conhecida como EDD (earliest due date): ordene as tarefas por *due date*. No nosso caso é possível que durante a execução de uma tarefa passamos o release de uma outra tarefa com *due date* menor. Para considerar isso, o nosso limite inferior será o sequenciamento obtido pela regra EDD, permitindo interrupções. $\Diamond$

O problema do caixeiro viajante

- Para cada chamada recursiva existem diversos vértices que podem ser selecionados
- Vértices ainda não selecionados são os candidatos possíveis
- A busca exaustiva é gerada caso nenhuma restrição for imposta
- Todas as permutações de cidades geram as soluções factíveis ($P_n = n(n-1)(n-2)...1 = n!$)
- Este problema têm solução $n^2 2^n$ usando programação dinâmica.
- Mas: para resolver em PD é necessário $n2^n$ de memória!

O problema do caixeiro viajante

- Alguma ideia de como diminuir ainda mais o espaço de busca?



Problema de Enumeração de conjuntos

ENUMERAÇÃO DE CONJUNTOS

Instância Um conjunto de n itens $S=a_1, a_2, a_3, ...a_n$.

Solução Enumeração de todos os subconjuntos de S.

- A enumeração de todos os conjuntos gera uma solução de custo exponencial 2^n.

Problema da Mochila

PROBLEMA DA MOCHILA

Instância Um conjunto de n itens $a_1, a_2, ...a_n$ e valores de importância v_i e peso w_i referentes a cada elemento i do conjunto; um valor K referente ao limite de peso da mochila.

Solução Quais elementos selecionar de forma a maximizar o valor total de "importância" dos objetos da mochila e satisfazendo o limite de peso da mochila?

- O problema da Mochila fracionário é polinomial
- O problema da Mochila 0/1 é NP-Completo
  - A enumeração de todos os conjuntos gera uma solução de custo exponencial 2^n
  - Solução via PD possui complexidade de tempo $O(Kn)$ (pseudo-polinomial) e de espaço $O(K)$

Problema de coloração em grafos

PROBLEMA DE COLORAÇÃO EM GRAFOS

Instância Um grafo G=(V,A) e um conjunto infinito de cores.

Solução Uma coloração do grafo, i.e. uma atribuição $c : V \to C$ de cores tal que vértices vizinhos não têm a mesma cor: $c(u) \neq c(v)$ para $(u, v) \in E$.

Objetivo Minimizar o número de cores $|C|$.

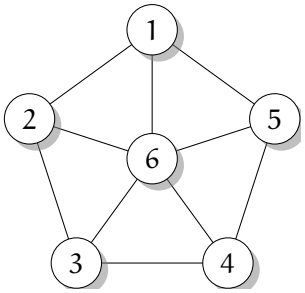
- Coloração de grafos de intervalo é um problema polinomial.
- Para um grafo qualquer este problema é NP-completo.
- Dois números são interessantes nesse contexto:
  - O *número de clique* $\omega(G)$: O tamanho máximo de uma clique que se encontra como sub-grafo de G.
  - O *número cromático* $\chi(G)$: O número mínimo de cores necessárias para colorir G.
- Obviamente: $\chi(V) \geq \omega(G)$
- Um grafo G é *perfeito*, se $\chi(H) = \omega(H)$ para todos sub-grafos H.
- Verificar se o grafo permite uma 2-coloração é polinomial (grafo bipartido).
- Um grafo k**-partido** é um grafo cujos vértices podem ser particionados em k conjuntos disjuntos, nos quais não há arestas entre vértices de um mesmo conjunto. Um grafo 2-partido é o mesmo que grafo bipartido.

Coloração de Grafos

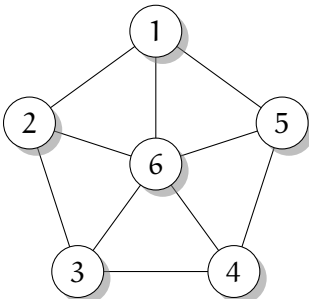
- A coloração de mapas é uma abstração do problema de colorir vértices.
- Projete um algoritmo de backtracking para colorir um grafo planar (mapa).
- Um grafo planar é aquele que pode ser representado em um plano sem qualquer intersecção entre arestas.
- Algoritmo $O(n^n)$, supondo o caso em que cada área necessite uma cor diferente
- Teorema de Kuratowski: um grafo é planar se e somente se não possuir *minor* K_5 ou $K_{3,3}$.
- Teorema das Quatro Cores: Todo grafo planar pode ser colorido com até quatro cores (1976, Kenneth Appel e Wolfgang Haken, University of Illinois)

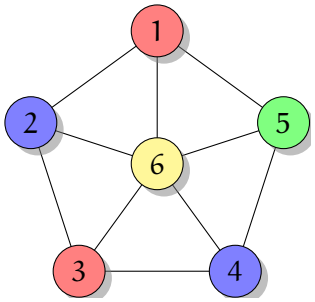
- Qual o tamanho máximo de um clique de um grafo planar?
- Algoritmo $O(4^n)$, supondo todas combinações de área com quatro cores.
- Existem 3^{n-1} soluções possíveis (algoritmo $O(3^n)$) supondo que duas áreas vizinhas nunca tenham a mesma cor.

De quantas cores precisamos?



De quantas cores precisamos?





(Precisamos de 4 cores!)

Backtracking

Algoritmo 7.6 (bt-coloring)
Entrada Grafo não-direcionado G=(V,A), com vértices $V = [1, n]$.

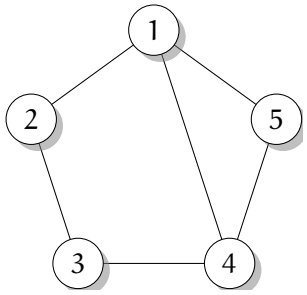
Saída Uma coloração $c : V \rightarrow [1, 4]$ do grafo.

Objetivo Minimiza o número de cores utilizadas.

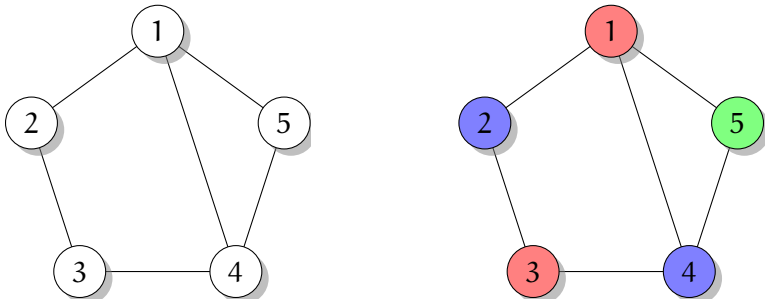
Para um vértice $v \in V$, vamos definir o conjunto de cores adjacentes $C(v) := \{c(u) \mid \{u, v\} \in A\}$.

```
return bt − coloring(1, 1)

boolean bt − coloring(v, c_v) :=
  if v > n
    return true

  if c ∉ C(v) then { v colorível com c_v? }
    c(v) := c_v
    for c_u ∈ [1, 4] do
      if bt − coloring(v + 1, c_u)
        return true
    end for
  else
    return false
  end if
end
```

De quantas cores precisamos?



De quantas cores precisamos?

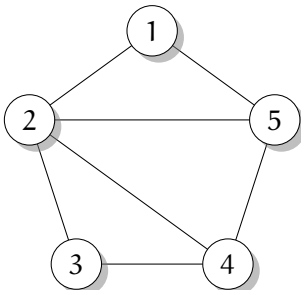


Precisamos de 3 cores!

Coloração de Grafos

- Existe um algoritmo $O(n^2)$ para colorir um grafo com 4 cores (1997, Neil Robertson, Daniel P. Sanders, Paul Seymour).
- Mas talvez sejam necessárias menos que 4 cores para colorir um grafo!
- Decidir se para colorir um grafo planar são necessárias 3 ou 4 cores é um problema NP-completo.

De quantas cores precisamos?



Roteamento de Veículos

ROTEAMENTO DE VEÍCULOS

Instância Um grafo G=(V,A), um depósito v_0, frota de veículos com capacidade Q (finita ou infinita), demanda $q_i > 0$ de cada nó ($q_0 = 0$, distância $d_i > 0$ associada a cada aresta

Solução Rotas dos veículos.

Objetivo Minimizar a distância total.

- Cada rota começa e termina no depósito.
- Cada cidade $V \setminus v_0$ é visitada uma única vez e por somente um veículo (que atende sua demanda total).
- A demanda total de qualquer rota não deve superar a capacidade do caminhão.

Roteamento de Veículos

- Mais de um depósito
- veículos com capacidades diferentes
- Entrega com janela de tempo (período em que a entrega pode ser realizada
- Periodicidade de entrega (entrega com restrições de data)
- Entrega em partes (uma entrega pode ser realizada por partes)
- Entrega com recebimento: o veículo entrega e recebe carga na mesma viagem
- Entrega com recebimento posterior: o veículo recebe carga após todas entregas
- ...

Backtracking versus Branch & Bound

- Backtracking: enumera todas as possíveis soluções com exploração de busca em profundidade.
  - Exemplo do Problema da Mochila: enumerar os 2^n subconjuntos e retornar aquele com melhor resultado.
- Branch&Bound: usa a estratégia de backtracking
  - usa limitantes inferior (relaxações) e superior (heurísticas e propriedades) para efetuar cortes
  - explora a árvore da forma que convier
  - aplicado apenas a problemas de otimização.

Métodos Exatos

- Problemas de Otimização Combinatória: visam minimizar ou maximizar um objetivo num conjunto finito de soluções.
- Enumeração: Backtracking e Branch&Bound.
- Uso de cortes do espaço de busca: Planos de Corte e Branch&Cut.
- Geração de Colunas e Branch&Price.

Métodos não exatos

- Algoritmos de aproximação: algoritmos com garantia de aproximação $S = \alpha S^*$.
- Heurísticas: algoritmos aproximados sem garantia de qualidade de solução.
  - Ex: algoritmos gulosos não ótimos, busca locais, etc.
- Metaheurísticas: heurísticas guiadas por heurísticas (*meta*=além + *heuriskein* = encontrar).
  - Ex: Algoritmos genéticos, Busca Tabu, GRASP (*greedy randomized adaptive search procedure*), Simulated annealing, etc.

7.3. Analisar árvores de busca

As recorrências obtidas na análise de árvores de busca geralmente não podem ser resolvidos pelo teorema de Akra-Bazzi, porque o tamanho da instância diminui somente por uma constante. Aplicaremos uma técnica geral para resolver recorrências no nosso caso particular.

Fato 7.1 (Graham et al. (1988, cap. 7.3)**)**
Uma expressão para os termos g_n de sequência g_n que satisfaz uma recorrência pode ser obtido da seguinte maneira:

1. Expressa a recorrência em forma de uma equação única que é válida para todo inteiro n, supondo que $g_{-1} = g_{-2} = \cdots = 0$.
2. Multiplica a equação por z^n e soma sobre todos n. Com isso obtemos uma equação com a função geradora $G(z) = \sum_n g_n^n$ no lado da esquerda. Manipula o lado da direita para obter uma expressão que depende de $G(z)$.

3. Resolve a equação para obter uma fórmula fechada para $G(z)$.
4. Expande $G(z)$ em uma série de potências. O coeficiente de z^n é uma expressão fechada para g_n.

Caso $G(z) = P(z)/Q(z)$ é uma função racional temos ainda

Teorema 7.1 (General expansion theorem for rational generating functions (Graham et al. 19
Caso $G(z) = P(z)/Q(z)$ com $Q(z) = q_0(1-\rho_1 z)^{m_1} \cdots (1-\rho_l z)^{m_l}$ com $\rho_1, \ldots, \rho_l$ números diferentes, e $P(z)$ é um polinômio de grau menos que $m_1 + \cdots + m_l$ então

$$[z^n]G(z) = f_1(n)\rho_1^n + \cdots + f_l(n)\rho_l^n \qquad \text{para todo } n \geq 0 \qquad (7.2)$$

com $f_k(n)$ um polinômio de grau $m_k - 1$.

Corolário 7.1 (Graham et al. (1988))
Dado a recorrência

$$t_n = \begin{cases} \Theta(1), & n \leq \max_{i\in[k]} d_i, \\ \sum_{i\in[k]} \alpha_i t_{n-d_i}, & \text{caso contrário,} \end{cases}$$

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

$$t_n = \Theta(n^{l-1}\alpha^n) = \Theta^*(\alpha^n).$$

($(d_1, \ldots, d_k)$ é o *vetor de bifurcação*.)

Prova. A recorrência satisfaz

$$t_n = \sum_{i\in[k]} \alpha_i t_{n-d_i} + c[0 \leq n < d]$$

e logo obtemos

$$G(z) = \sum_n t_n z^n = \sum_{i\in[k]} \alpha_i z^{d_i} G(z) + c \sum_{i\in[d-1]} z^i$$

e assim

$$G(z) = \frac{c\sum_{i\in[d-1]} z^i}{1 - \sum_{i\in[k]} \alpha_i z^{d_i}} = \frac{-c\sum_{i\in[d-1]} z^i}{-\sum_{i\in[0,k]} \alpha_i z^{d_i}}$$

com $\alpha_0 = -1$ e $d_0 = 0$. Logo os critérios do teorema 7.1 são satisfeitos. Mas os coeficientes ρ_l são as raízes do polinômio

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

e portanto, pelo teorema 7.1 obtemos assintoticamente para a maior raiz ρ com multiplicidade m

$$t_n = \Theta(f(n)\rho^n)$$

com $f(n)$ um polinômio de grau $m-1$. ■

7.4. Notas

Mais sobre como estimar o tamanho de árvores de busca pode ser encontrado em Kullmann (2008). Kilby et al. ("Estimating search tree size"') propuseram uma generalização para casos em que a árvore não está disponível sem executar o algoritmo e que pode ser aplicado para branch-and-bound.

7.5. Exercícios

Soluções a partir da página 444.

Exercício 7.1
O problema do empacotamento unidimensional consiste em encontrar o menor número de contêineres de um dado tamanho $T \in \mathbb{Z}_+$ que permitem empacotar n itens com tamanhos $t_1, \ldots, t_n \in \mathbb{Z}_+$. O problema de decidir se existe um empacotamento com no máximo $K \in \mathbb{Z}_+$ contêineres é NP-completo. Mostra que o problema pode ser resolvido em tempo polinomial por busca exaustiva caso o tamanho T dos contêineres é fixo.

Parte III.

Estruturas de dados

8. Estruturas abstratas de dados

Um tipo abstrato de dados representa uma classe de objetos abstratos definidos completamente por suas operações (Liskov e Zilles 1974). Um tipo abstrato de dados separa a interface de uma estrutura de dados da implementação. Para usar o tipo de dados o conhecimento do interface é suficiente. Para uma separação ideal em que podemos provar a corretude de um algoritmo usando a estrutura abstrata de dados sem conhecer a implementação temos que definir a sua semântica formalmente, por exemplo através de um definição axiomática.
Um exemplo simples de uma estrutura abstrata de dados são números inteiros: para usa-los é suficiente conhecer as operações possíveis (adição, multiplicação, etc.) sem saber que a implementação usa uma representação binária.

8.1. Exemplos de estruturas abstratos de dados

Conjunto Um conjunto não contém elementos repetidos e permite inserir ("insert"), remover ("remove"), testar a pertinência ("member") de elementos, e testar se o conjunto é vazio ("empty").

Tipo abstrato de dados 8.1 (Conjunto)
Seja C um conjunto C, e sejam v e w elementos.
Equações:

$\text{new}()$	novo conjunto.	(8.1)
$\text{member}(v, \text{new}()) = \text{false}$		(8.2)
$\text{member}(v, \text{add}(v, C)) = \text{true}$		(8.3)
$\text{member}(v, \text{add}(w, C)) = \text{member}(v, C)$	caso $v \neq w$	(8.4)
$\text{remove}(v, \text{new}()) = \text{new}()$		(8.5)
$\text{remove}(v, \text{add}(v, C)) = \text{remove}(v, C)$		(8.6)
$\text{remove}(v, \text{add}(w, C)) = \text{add}(w, \text{remove}(v, C))$	caso v$\neq w$	(8.7)
$\text{empty}(\text{new}()) = \text{true}$		(8.8)
$\text{empty}(\text{add}(v, C)) = \text{false}$		(8.9)

Exemplo 8.1
Provaremos que $C_1 = C_2$ para C_1:=add(v,add(v,C)) e C_2:=add(v,C), C e v arbitrário. A prova da igualdade é por extensão, i.e. provaremos que todas operações obtém o mesmo resultado para C_1 e C_2 por indução estrutural sobre C. Observe que as seguintes identidades são corretas para qualquer conjunto C:

$\text{member}(v, C_1) = \text{true} = \text{member}(v, C_2)$	por 3
$\text{member}(w, C_1) = \text{member}(w, C) = \text{member}(w, C_2)$	por 4, para $w \neq v$
$\text{remove}(v, C_1) = \text{remove}(v, C) = \text{remove}(v, C_2)$	por 6
$\text{empty}(C_1) = \text{false} = \text{empty}(C_2)$	por 9.

Por indução podemos ver que $\text{remove}(w, C_1) = \text{remove}(w, C_2)$ para $w \neq v$: para $C = \text{new}()$ temos

$$\text{remove}(w, C_1) = \text{add}(v, \text{add}(v, \text{new}()))$$
$$\text{remove}(w, C_2) = \text{add}(v, \text{new}())$$

$$\text{remove}(w, C_1) = \text{add}(v, \text{add}(v, \text{remove}(w, C)))$$
$$\text{remove}(w, C_2) = \text{add}(v, \text{remove}(w, C))$$

$\Diamond$

Dicionário Um dicionário (ou mapa, vetor associativo) armazena valores que são identificados por chaves.

Tipo abstrato de dados 8.2 (Dicionário)
Seja D um dicionário, k e j chaves, e v um valor v.

Equações:

$$\text{new}() \qquad \text{novo dicionário} \tag{8.10}$$
$$\text{find}(k, \text{insert}(k, v, D)) = v \tag{8.11}$$
$$\text{find}(k, \text{insert}(j, v, D)) = \text{find}(k, D) \quad \text{caso } k \neq j \tag{8.12}$$
$$\text{delete}(k, \text{new}()) = \text{new}() \tag{8.13}$$
$$\text{delete}(k, \text{insert}(k, v, D)) = \text{delete}(k, D) \tag{8.14}$$
$$\text{delete}(k, \text{insert}(j, v, D)) = \text{insert}(j, v, \text{delete}(k, D)) \quad \text{caso } k \neq j \tag{8.15}$$

Pilha Uma pilha armazena valores de forma que temos acesso somente pelo topo da pilha. Uma operação "push" empilha um novo elemento no topo. O topo pode ser acessado pela operação "top" e removido pela operação "pop". A operaçõa "empty" testa sem existe algum elemento no topo.

Tipo abstrato de dados 8.3 (Pilha)
Seja P uma pilha e v um elemento v.
Equações:

$$\text{new}() \qquad \text{nova pilha} \tag{8.16}$$
$$\text{pop}(\text{push}(v, P)) = P \tag{8.17}$$
$$\text{top}(\text{push}(v, P)) = v \tag{8.18}$$
$$\text{empty}(\text{new}()) = \text{true} \tag{8.19}$$
$$\text{empty}(\text{push}(v, P)) = \text{false} \tag{8.20}$$
$$\tag{8.21}$$

Fila Uma fila armazena uma sequência de valores de forma que podemos inserir novos elementos em um lado da sequência usando uma operação "add", mas remover elementos somente no outro lado (via uma operação "remove"). Uma fila implementa uma estratégia de FIFO (inglês first in, first out). Uma operação "front" permite o acesso ao elemento mais antigo da fila. Para uma fila Q e elementos v e w temos

Tipo abstrato de dados 8.4 (Fila)
Equações:

$$\text{new}() \qquad \text{nova fila} \tag{8.22}$$

$$\text{front}(\text{add}(v, \text{new}())) = v \tag{8.23}$$

$$\text{remove}(\text{add}(v, \text{new}())) = \text{new}() \tag{8.24}$$

$$\text{front}(\text{add}(v, \text{add}(w, Q))) = \text{front}(\text{add}(w, Q)) \tag{8.25}$$

$$\text{remove}(\text{add}(v, \text{add}(w, Q))) = \text{add}(v, \text{remove}(\text{add}(w, Q))) \tag{8.26}$$

Fila de prioridade Uma fila de prioridade mantem um conjunto de elementos com prioridades associados de forma que podemos acessar ("min") e remover ("dm" do deletemin) facilmente o elemento de menor prioridade.

Tipo abstrato de dados 8.5 (Fila de prioridade)
Seja Q uma fila de prioridade Q, e sejam v e w elementos.
Equações:

$$\text{new}() \qquad \text{cria nova fila} \tag{8.27}$$

$$\text{min}(\text{insert}(v, \text{new}()) = v \tag{8.28}$$

$$\text{dm}(\text{insert}(v, \text{new}()) = \text{new}() \tag{8.29}$$

$$\text{min}(\text{insert}(v, \text{insert}(w, Q))) = \tag{8.30}$$

```
if prio(v) < prio(min(insert(w,Q))) then
  v
else
  min(insert(w,Q))
```

$$\text{dm}(\text{insert}(v, \text{insert}(w, Q))) = \tag{8.31}$$

```
if prio(v) < prio(min(insert(w,Q))) then
  insert(w,Q)
else
  insert(v,dm(insert(w,Q)))
```

9. Estruturas da dados elementares

Para armazenar uma sequência de elementos uma estrutura de dados tem que apoiar as operações elementares de *inserir*, *procurar* e *remover* um elemento. Uma outra operação freqüente é a *concatenação* de duas sequências.

As estruturas de dados elementares se orientam no modelo da memória do computador, que pode ser visto como uma sequência de algum tipo de dado elementar. Um elemento é acessado pelo *endereço* dele na memória, i.e. o seu número na sequência. Um *vetor* representa esse modelo diretamente, com, por conveniência, um índice de base diferente. Caso os dados não são armazenadas contiguamente no memória, precisa-se um mecanismo para encontrar o endereço do elemento. A abordagem mais simples, uma *lista*, armazeno junto com o elemento o endereço do seus sucessor na sequência.

Vetor Um *vetor* (ou *tabela fixa*, *lista sequencial*) armazena uma sequência continua da dados indexados por números inteiros. Por isso, o acesso e a atualização do elemento dado o índice tem tempo $O(1)$. Inserir ou remover um elemento no final da sequência, caso ainda tem espaço, também custa $O(1)$. Porém, inserir ou remover um elemento em um ponto diferente tem custo linear $O(n)$: temos que deslocar todos elementos após do ponto de inserção ou remoção. Pela mesma razão a concatenação de duas sequências de tamanho n e m custa $O(\min(n, m))$ caso a menor ainda tem espaço para os elementos da maior, e $O(m + n)$ caso temos que alocar uma nova sequência e copiar os elementos das duas.

Buscar um elemento em um vetor custa $O(n)$, porque temos que percorrer todos elementos. Uma caso particular importante é a busca em um vetor ordenado: como podemos acessar os elementos livremente, neste caso é possível encontrar um elemento em $O(\log n)$ passos usando uma *busca binária*:

Algoritmo 9.1 (Busca binária)
Entrada Um vetor $v_1, \ldots, v_n$ ordenado, i.e., $v_1 < v_2 < \cdots < v_n$ e um valor v.

Saída Uma posição i tal que $v_i = v$, caso existe, ∞ caso contrário.

```
find(l,u,v) :=
  while l < u do
```

```
    { invariante: ∃i v_i = v ⇒ l ≤ i ≤ u }
    m := ⌊(l+u)/2⌋
    if v_m = v then
      return m
    end if
    if v_m < v then
      l := m+1
    else
      r := m−1
    end if
  end while
  if l = u ∧ v_l = v then
    return l
  else
    return ∞
  end if
end
```

Como um vetor é uma sequência contigua, tipicamente temos que informar o número máximo de elementos no vetor na hora da sua alocação. O exemplo 2.22 mostra como realocar um vetor tal que o custo amortizado de uma inserção no final da tabela continua a ser $O(1)$.

Lista Uma *lista* simples (ou *lista encadeada simples*) armazena junto com cada elemento um *ponteiro* (o endereço) para o sucessor do elemento. Ela é representada por um ponteiro para o primeiro elemento da lista. Um *ponteiro nulo* marca o final da lista.
Em comparação com o vetor a inserção ou remoção de um elemento após de um elemento dado por um ponteiro é possível em tempo $O(1)$. A inserção de um novo elemento na posição L, por exemplo, pode ser implementando por

```
insert(L,v) :=
  e:=make-element
  e.v = v
  e.s = L.s
  L.s = e
```

Similarmente, o acesso e a atualização do elemento igualmente é possível em tempo $O(1)$ usando um ponteiro para o elemento. Porém o acesso do i-ésimo elemento, dado somente um ponteiro para o primeiro elemento da

lista requer uma busca linear em tempo $O(n)$. Por consequência uma busca possui complexidade $O(n)$, mesmo quando a lista é ordenada: uma lista não permite uma busca binária eficiente.
Uma outra desvantagem de uma lista simples é que podemos percorrer-la somente do início até o final. Uma *lista dupla* (ou *lista duplamente encadeada*) evita esse problema armazenando um ponteiro adicional para o sucessor de cada elemento.
Para remover um elemento de uma lista dupla temos que implementar

```
delete(L) :=
  if L.p ≠ undefined then
    L.p.s = L.s
  end if
  if L.s ≠ undefined then
    L.s.p = L.p
  end if
  { libera L caso necessário }
```

Para simplificar isso podemos representar uma lista vazio por um elemento nulo ϵ com $\epsilon.s = \epsilon.p = \epsilon$ e manter uma lista circular em que o sucessor do ϵ é o primeiro elemento da lista, o predecessor de ϵ é o último elemento da lista, e o sucessor do último elemento é ϵ. Com isso todo elemento sempre possui predecessor e sucessor definido, a implementação do *delete* é simplesmente

```
delete(L) :=
  { pré-condição: L ≠ ε }
  L.p.s = L.s
  L.s.p = L.p
  { libera L caso necessário }
```

Como temos acesso eficiente ao último elemento da lista essa representação permite concatenar duas listas em tempo constante.

Lista de blocos Listas e vetores podem ser combinadas em *listas de vetores* ou *listas de blocos*. Nessa estrutura de dados, cada elemento da lista representa um vetor de k elementos, sendo k a granularidade da lista de blocos. Uma lista de blocos combina elementos de listas e vetores: para $k = 1$ obtemos uma lista, para $k \to \infty$ um vetor. Escolhendo uma granularidade adequada para a aplicação, uma lista de blocos pode ser mais eficiente.

Pilhas Uma *pilha* (ingl. *stack*) permite acesso aos elementos armazenados somente em ordem inversa da sua inserção. Igualmente podemos ver uma

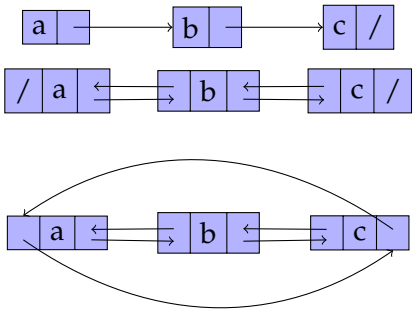


Figura 9.1.: Lista simples, lista dupla, lista circular.

pilha como uma sequência quais elementos podemos acessar somente por um lado. O último elemento inserido pode ser removido da estrutura por uma operação *pop*. Podemos inserir ("empilhar") um novo elemento usando uma operação *push*. Implementado por uma lista as duas operações possuem complexidade $O(1)$.

Filas Uma *fila* (ingl. *queue*) permite inserção e remoção de elementos de forma que os elementos são removidos na ordem em que eles foram inseridos. A inserção é chamada *enqueue*, a remoção *dequeue*. Em comparação com a pilha, a inserção sempre acontece num lado, e a remoção no outro lado. Igual a uma pilha em uma implementação por uma lista circular as duas operações possuem complexidade $O(1)$. Uma versão comum da fila possui capacidade limitada. Essa versão pode ser implementado usando um vetor e armazenado o índice do primeiro e último elemento da fila (modulo o tamanho da capacidade da fila).

Fila dupla (dequeue) Uma fila dupla ou deque (do inglês double-ended queue) é uma fila em que podemos inserir ou remover elementos dos dois lados. Igualmente podemos ver uma fila dupla como uma pilha que pode ser acessada por dois lados. Novamente uma lista circular permite uma implementação em que todas operações (inserção e deleção em um dos dois lados) possui complexidade $O(1)$.

Matrizes Uma matriz M de tamanho $n \times m$ pode ser armazenado em um vetor com nm elementos. Temos a opção de armazenar os elementos da matriz por linha (inglês row major order) ou por coluna (inglês column major

order). O elemento m_{ij} se encontra na posição $im + j$ ou $i + jn$, respectivamente. Essa técnica pode ser generalizado para matrizes de dimensões maiores. Linguagens de programação tipicamente possuem apoio para estas operações.
Matrizes grandes encontradas na prática frequentemente são *esparsas*, i.e. somente uma pequena fração dos elementos é diferente de zero. Nesta caso o consumo de memória pode ser limitado armazenando somente os elementos diferentes de zero. Para este fim temos que encontrar uma maneira de localizar estes elementos.
Uma representação simples é em forma de uma lista de triplas (i, j, v) em que i, j representam a linha e coluna do elemento na matriz e v o valor. Além dos valores essa representação armazena 2nnz índices, com nnz o número de elementos diferente de zero.
Uma abordagem que precisa menos memória é conhecido como linhas esparsas comprimidas (inglês: compressed sparse rows): um vetor v de tamanho nnz armazena todos elementos diferentes de zero por linha e um outro vetor c do mesmo tamanho as colunas desses elementos. Um terceiro vetor r de tamanho n armazena o índice do primeiro elemento da cada linha. Essa representação armazena somente $nnz + n$ índices além dos valores e é padrão na computação científica (Saad 2003).

9.1. Notas

Uma observação interessante é após um `delete` do elemento L da uma lista as operações

L.p.s = L
L.s.p = L

re-inserem o elemento L na posição anterior da lista. Knuth (*Dancing Links*) mostra como essa fato pode ser aproveitado para implementar algoritmos de *backtracking*.
Buluç et al. (2009) propõem representar matrizes esparsas em *blocos esparsos comprimidos*.

10. Dicionários

Um dicionário é uma estrutura abstrata de dados que permite inserir, remover e procurar por chaves (“insert”, “find”, “delete”) que podem ser associados com valores. A seção 8.1 contém uma descrição do tipo abstrato de dados de um dicionário. Frequentemente estruturas de dados que implementam um dicionário permitem ainda listar (“traverse”) todos elementos contidos. Outras operações úteis incluem encontrar o menor ou maior elemento (“min”, “max”) e obter o próximo elemento menor ou maior (“pred”, “succ”). Observe que as operações recebem a chave
Com um vetor ou uma lista não-ordenado podemos implementar um dicionário com custo $O(1)$ para inserir, e custo $O(n)$ para procurar e remover um elemento ou listar todos elementos. Com um vetor ordenado o custo de procurar um elemento pode ser reduzido para $O(\log n)$ usando uma busca binária, aumentando o custo de inserir um elemento para $O(n)$. Com uma lista ordenada obtemos custo $O(n)$ para todos operações.

10.1. Listas skip

Uma lista *skip* mantém um conjunto ordenado de elementos. Ela permite manter um conjunto dinâmico de elementos com uma busca eficiente em tempo esperado $O(\log n)$ (com alta probabilidade) (Pugh 1990) e permite uma implementação simples.
Uma lista skip armazena todos elementos numa lista ordenada. Uma desvantagem de uma lista simples é que uma busca tem custo $O(n)$. Para acessar os elementos da lista mais rapidamente, podemos criar uma segunda lista com um subconjunto de elementos, mantendo a correspondência com a lista completa. Usaremos a segunda lista para encontrar a posição aproximada de um elemento, e depois localizar o elemento na lista completa. Para n elementos, e uma segunda lista com m elementos distribuída uniformemente, o custo de uma busca no pior caso agora é $m + n/m$, cujo mínimo é $2\sqrt{n}$ para $m = \sqrt{n}$.
Para três níveis obtemos $3n^{1/3}$ e para k níveis $kn^{1/k}$ pela condição que a razão entre os níveis tem que ser constante. Em particular para $k = \log n$ obtemos custo $2 \log n$ e cada nível superior contém cada segundo elemento do nível anterior. Isso leva a uma *lista skip ideal*. (Ver figura X). Uma busca

nessa lista começão no nível superior, e desce um nível caso o sucessor é maior que o elemento procurado.

```
find(v,L) :=
  e := L.k { top level }
  do
    while e.s.v < v do e := e.s
    if e.d = undefined
      return e
    e := e.d
  forever
```

'E caro manter a estrutura de uma lista skip ideal inserindo e removendo elementos. A solução desse problema é randomizar a decisão quais elementos ocorrem nos diferentes níveis, mantendo aproximadamente a garantia de uma lista skip ideal. Para inserir um elemento, vamos procurar o ponto de inserção no nível inferior e repetidamente, com probabilidade 1/2 promover o elemento para o próximo nível.

```
insert(v,L) :=
  e := find(v, L)
  insere v na lista atual
  repete com probabilidade 1/2
    sobe para o próximo nível,
      (cria novo nível caso necessário)
    insere v na lista atual
  end

delete(v,L) :=
  e := search(v, L)
  if e.v = v then
    remove a coluna do elemento v
  end if
```

Teorema 10.1
Com alta probabilidade cada busca numa lista skip com n elementos custa $O(\log n)$. Mais precisamente, para cada $\alpha \geq 1$ existem constantes tal que a busca possui complexidade $O(\log n)$ com probabilidade $1 - O(n^{-\alpha})$.

10.2. Árvores de busca

Algumas características básicas de uma árvore:

Definição 10.1
Uma árvore possui um único nodo sem pai, a *raiz*. Cada nodo possui uma série de *filhos*. Um nodo sem filhos é uma *folha*. A *altura* de uma árvore é o caminho mais longo entre a raiz e uma folha.

10.2.1. Árvores binárias

Uma árvore binária consiste em nodos com no máximo dois filhos. Uma representação alternativa substitui cada filho não presente por um *nodo externo*. Desta forma cada nodo *interno* possui dois filhos, e somente nodos externos são folhas.

Definição 10.2
A *distância total interna* é a soma dos comprimentos dos caminhos da raiz para cada nodo interno. A *distância total externa* é a soma dos comprimentos dos caminhos da raiz para cada nodo externo.

Lema 10.1
Numa árvore binária com n chaves, a distância total interna I e a distância total externa E satisfazem $E = I + 2n$.

Prova. Por indução sobre o número de nodos internos. Para uma árvore vazia temos $E = I = 0$, para uma árvore com único nodo interno $E = 2$ e $I = 0$. Dado uma árvore com $n \geq 2$ nodos internos, temos duas sub-árvores com n_1 e n_2 nodos, tal que $n_1 + n_2 + 1 = n$. Pela hipótese de indução $E_1 = I_1 + n_1$ e $E_2 = I_2 + n_2$. A contribuição da primeira sub-árvore para o comprimento interno é $I_1 + n_1$ porque a distância de cada nodo interno cresce por um. Similarmente ela contribui para o comprimento externo $E_1 + n_1 + 1$ porque temos um nodo externo mais que nodos internos. Em total obtemos $E = E_1 + n_1 + 1 + E_2 + n_2 + 1$ e $I = I_1 + n_1 + I_2 + n_2$ que satisfaz

$$\begin{aligned} E &= E_1 + n_1 + 1 + E_2 + n_2 + 1 \\ &= I_1 + 3n_1 + I_2 + 3n_2 + 2 = I + 2n_1 + 2n_2 + 2 = I + 2n. \end{aligned}$$

■

Para procurar um elemento em $\log_2 n$ uma árvore binária mantem a propriedade que a chave de cada vértice interno é maior ou igual à cada chave da sub-árvore esquerda e menor ou igual a cada chave da sub-árvore direita. Com isso uma busca começa na raiz, compara a chave procurada com a chave do nodo visitado. Caso o elemento não encontra-se no nodo atual, a busca continua com a filho da esquerda, caso o chave procurada é menor que a chave do nodo atual, e com o filho da direita, caso contrário.

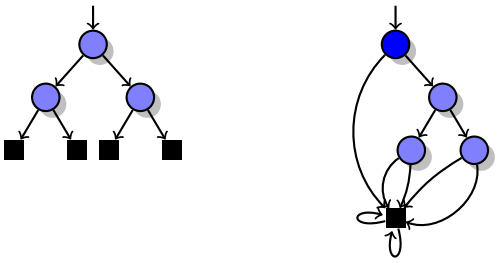


Figura 10.1.: Representação de uma árvore binária.

Definição 10.3
Uma árvore binária é uma *árvore de busca* caso a chave de cada nodo interno é maior ou igual que todas chaves da sub-árvore da esquerda e menor ou igual que todas chaves da sub-árvore da direita.

Para inserir um elemento, podemos buscar a posição na árvore em que a nova chave será inserida, e criar um novo nodo como filho do pai dessa posição.
Remover um elemento é simples, caso ao menos um dos filhos é vazio: o pai recebe como novo filho o filho não-vazio do nodo a ser removido. Caso contrário podemos remover o nodo e substituí-lo com a maior chave menor (o *predecessor simétrico*) ou a menor chave maior (o *sucessor simétrico*) na árvore. Isso é possível porque o predecessor simétrico não possui filho da direita e o sucessor simétrico não possui filho da esquerda (por quê?). Para simplificar a implementação podemos sempre substituir o nodo com o sucessor simétrico.
Podemos traversar uma árvore binária em ordem crescente de chaves usando uma busca por profundidade, visitando primeiro o filho da esquerda, depois o próprio nodo, e depois o filho da direita.
Observe que as operações frequentemente precisam modificar o pai de um elemento e testar se um elemento não possui filho. Para evitar testes em casos especiais, p.ex. ao remover a raiz que não possui pai, é conveniente implementar uma árvore binária de forma que sempre possui dois elementos: (i) uma raiz, com filho da esquerda sempre vazio, e o filho da direita é o nodo mais alto da árvore, (ii) um "nodo vazio" ϵ que representa um filho não presente. Nodos que não tem filho terão este "nodo vazio" como filho. Por convenção ϵ é filho da esquerda e da direita de si mesmo. Um exemplo encontra-se na figura 10.1.
Com isso podemos implementar

```
find(B,c) :=
```

```
    ϵ.c = c
    B := B.r
    while B.c ≠ c do
      if B.c < c then B := B.r else B := B.l
    if B = ϵ
      return ‘‘chave não encontrada’’
    else
      return B

insert(B,c) :=
    P := B
    B := B.r
    while B ≠ ϵ do
      P := B
      if B.c < c then B := B.r else B := B.l
    B := Node(c)
    if P.c < c
      P.r := B
    else
      P.l := B
    return B
end

delete(B,c) :=
    ϵ.c = c
    P := B
    { busca chave c }
    B := B.r
    while B.c ≠ c do
      P := B
      if B.c < c then B := B.r else B := B.l
    R := B { substitui R com um novo nodo }
    { Fig. 10.2 (a) }
    if R.r = ϵ
      B := R.l
    { Fig. 10.2 (b) }
    else if R.r.l = ϵ
      B := R.r
      B.l := R.l
    { Fig. 10.2 (c) }
```

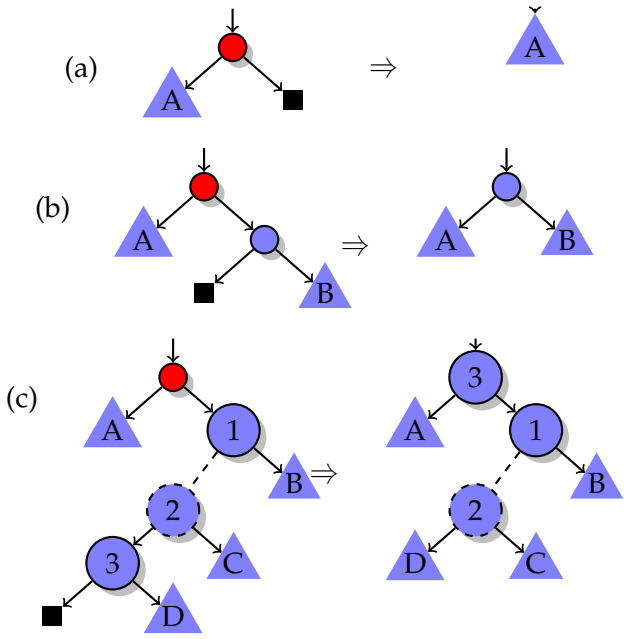


Figura 10.2.: Remover um elemento de uma árvore binária. (a) Filho da direita não existe. (b) Filho da direita não possui filho da esquerda. (c) Filho da direita possui filho da esquerda. Legenda: ●: nodo normal; ●: nodo a ser removido; ■: nodo não presente; s: sub-árvore, eventualmente vazia.

```
  else
    PB := R.r { pai do sucessor simétrico }
    while PB.l.l ≠ ϵ
      PB := PB.l
    B := PB.l { B agora é o sucessor simétrico, PB seu pai }
    PB.l := B.r
    B.l := R.l; B.r := R.r
  end if
  delete R
  if P.c < c then P.r := B else P.l := B
end
```

A complexidade das três operações é limitada pela altura da árvore. No caso de uma árvore balanceada isso implica uma complexidade pessimista de $O(\log n)$. Um problema potencial é que algumas sequências de chaves

geram uma árvore desbalanceada. Por exemplo, a inserção de $1, 2, \ldots, n$ gera uma lista linear de altura n. Felizmente, o seguinte lema mostra no caso médio uma árvore binária se comporta bem.

Lema 10.2 (Hibbard (1962))
O número médio de comparações numa busca com sucesso C_n e o número médio de comparações numa busca sem sucesso C'_n satisfazem

$$C_n = (1 + 1/n)C'_n - 1. \tag{10.1}$$

Prova. Temos $C_n = 1 + I/n$ porque uma busca com sucesso precisa uma comparação mais que o comprimento do caminho da raiz para o nodo encontrado. Similarmente $C'_n = E/(n+1)$, porque uma busca sem sucesso precisa um número de comparações igual o comprimento de um caminho para um nodo externo. Logo,

$$C_n = 1 + I/n = 1 + (E - 2n)/n = E/n - 1 = (1 + 1/n)C'_n - 1.$$

■

Proposição 10.1
Buscar ou inserir um elemento numa árvore binária custa aproximadamente $2 \ln n \approx 2.88 \log_2 n$ comparações em média sobre sequências de n chaves randômicos.

Prova. O número de comparações para uma busca com sucesso, é mais um que o número de comparações para uma busca sem sucesso, antes de inserir a chave. Portanto

$$C_n = 1 + (C'_0 + C'_1 + \cdots + C'_{n-1})/n.$$

Aplicando lema 10.2 obtemos

$$(n+1)C'_n = 2n + C'_0 + C'_1 + \cdots + C'_{n-1}$$

e subtraindo

$$nC'_{n-1} = 2(n-1) + C'_0 + C'_1 + \cdots + C'_{n-2}$$

resulta em

$$(n+1)C'_n - nC'_{n-1} = 2 + C'_{n-1}$$

ou seja $C'_n = 2/(n+1) + C'_{n-1} = 2/(n+1) + 2/n + C'_{n-2} = \cdots = 2H_{n+1} - 2$ com $C'_0 = 0$. Aplicando lema 10.2 novamente resulta em

$$C_n = 2(1 + 1/n)H_{n+1} - 3.$$

■

É possível demonstrar que a deleção de elementos não leva a um desbalanceamento grave na prática. Em particular isso é verdadeiro para nossa implementação asimétrica da deleção, que substitui o nodo deletado pelo sucessor simétrico (para detalhes ver Knuth (1998, p. 432) e Sedgewick (1992, p. 250)).

10.2.2. Treaps

Um *treap* (uma combinação de *tree* e *heap*, i.e. uma *filárvore*) é uma árvore binária que armazena junto com as chaves uma prioridade escolhido aleatoriamente na inserção. Além de ser uma árvore de busca, ele é um *min-heap* das prioridades: a prioridade de um nodo é menor ou igual que a prioridade dos seus filhos (ver também capítulo 11.1.1). Essa abordagem garante que a estrutura da árvore após de inserir e remover um número arbitrário de chaves é igual a estrutura da árvore obtido por uma inserção das chaves na ordem das prioridades, sem rebalanceamento. Em outras palavras, obtemos uma árvore binária aleatória independente das chaves (Aragon e Seidel 1989).

10.2.3. Árvores binárias de busca ótimas

Para um conjunto conhecido de chaves podemos construir uma árvore de busca ótima, que minimiza o número esperado de operações dado uma distribuição da probabilidade de busca.

Motivação

- Suponha que temos um conjunto de chaves com probabilidades de busca conhecidas.
- Caso a busca é repetida muitas vezes, vela a pena construir uma estrutura de dados que minimiza o tempo médio para encontrar uma chave.
- Uma estrutura de busca eficiente é uma árvore binária.

Portanto, vamos investigar como construir a árvore binária ótima. Para um conjunto de chaves com distribuição de busca conhecida, queremos minimizar o número médio de comparações (nossa medida de custos).

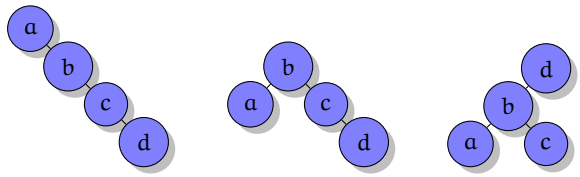
Exemplo 10.1
Considere a sequência ordenada $a < b < c < d$ e as probabilidades

Exemplo

Elemento e	a	b	c	d
$\Pr(e)$	0.2	0.1	0.6	0.1

qual seria uma árvore ótima? Alguns exemplos

Árvore correspondente



que possuem um número médio de comparações $0.2 \times 1 + 0.1 \times 2 + 0.6 \times 3 + 0.1 \times 4 = 2.6$, $0.2 \times 2 + 0.1 \times 1 + 0.6 \times 2 + 0.1 \times 3 = 2.0$, $0.2 \times 3 + 0.1 \times 2 + 0.6 \times 3 + 0.1 \times 1 = 2.7$, respectivamente. ◊

Árvore de Busca Binária Ótima
Em geral, temos que considerar as probabilidades de procurar uma chave junto com as probabilidades que uma chave procurada não pertence à árvore. Logo, supomos que temos

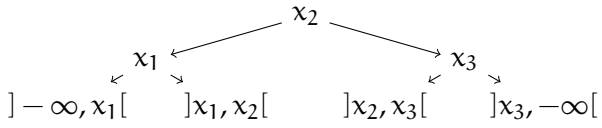
(a) uma sequência ordenada $a_1 < a_2 < \cdots < a_n$ de n chaves e

(b) probabilidades

$$\Pr(c < a_1), \Pr(c = a_1), \Pr(a_1 < c < a_2), \ldots$$
$$\ldots, \Pr(a_{n-1} < c < a_n), \Pr(c = a_n), \Pr(a_n < c)$$

que a chave procurada c é uma das chaves da sequência ou cai num intervalo entre elas.

A partir dessas informações queremos minimizar a complexidade média da busca. Em uma dada árvore, podemos observar que o número de comparações para achar uma chave existente é igual a profundidade dela na árvore (começando com profundidade 1 na raiz). Caso a chave não pertence à árvore, podemos imaginar chaves artificiais que representam os intervalos entre as chaves, e o número de comparações necessárias é um menos que a profundidade de uma chave artificial. Um exemplo de uma árvore com chaves artificiais (representadas pelos intervalos correspondentes) é



Para facilitar a notação, vamos introduzir chaves adicionais $a_0 = -\infty$ e $a_{n+1} = \infty$ e escrever $J_i = (a_i, a_{i+1})$ tanto para intervalo entre a_i e a_{i+1} quanto para o nodo que representa este intervalo. Seja $d(v)$ a profundidade de um nodo. Com isso, obtemos a complexidade média de busca

$$c_M = \sum_{1 \le i \le n} \Pr(c = a_i) d(a_i) + \sum_{0 \le i \le n} \Pr(c \in J_I)(d(J_i) - 1);$$

ela depende da árvore concreta.

Como achar a árvore ótima? A observação crucial é: *Uma das chaves deve ser a raiz e as duas sub-árvores da esquerda e da direita devem ser árvores ótimas das sub-sequências correspondentes.*

Para expressar essa observação numa equação, seja $c_M(e, d)$ a complexidade média de uma busca numa sub-árvore ótima para os elementos $a_e, \ldots, a_d$. Para a complexidade da árvore inteira, definido acima, temos $c_M = c_M(1, n)$. Da mesma forma, obtemos

$$c_M(e, d) = \sum_{e \le i \le d} \Pr(c = a_i) d(a_i) + \sum_{e-1 \le i \le d} \Pr(c \in J_i)(d(J_i) - 1)$$

Árvore de Busca Binária Ótima

Supondo que a_r é a raiz desse sub-árvore, essa complexidade pode ser escrito

como

$$\begin{aligned}
c_M(e,d) &= \Pr(c = a_r) \\
&\quad + \sum_{e \le i < r} \Pr(c = a_i) d(a_i) + \sum_{e-1 \le i < r} \Pr(c \in J_i)(d(J_i) - 1) \\
&\quad + \sum_{r < i \le d} \Pr(c = a_i) d(a_i) + \sum_{r \le i \le d} \Pr(c \in J_i)(d(J_i) - 1) \\
&= \left(\sum_{e-1 \le i \le d} \Pr(c \in J_i) + \sum_{e \le i \le d} \Pr(c = a_i) \right) \\
&\quad + c_M(e, r-1) + c_M(r+1, d) \\
&= \Pr(a_{e-1} < c < a_{d+1}) + c_M(e, r-1) + c_M(r+1, d)
\end{aligned}$$

Árvore de Busca Binária Ótima

(O penúltimo passo é justificado porque, passando para uma sub-árvore a profundidade e um a menos.) Essa equação define uma recorrência para a complexidade média ótima: Escolhe sempre a raiz que minimiza essa soma. Como base temos complexidade $c_M(e,d) = 0$ se $d > e$:

$$c_M(e,d) = \begin{cases} \min_{e \le r \le d} \Pr(a_{e-1} < c < a_{d+1}) + c_M(e, r-1) + c_M(r+1, d) & \text{caso } e \le d \\ 0 & \text{caso } e > d \end{cases} \tag{10.2}$$

Árvore de Busca Binária Ótima

Ao invés de calcular o valor c_M recursivamente, vamos usar a programação (tabelação) dinâmica com três tabelas:

- c_{ij}: complexidade média da árvore ótima para as chaves a_i até a_j, para $1 \le i \le n$ e $i-1 \le j \le n$.
- r_{ij}: raiz da árvore ótima para as chaves a_i até a_j, para $1 \le i \le j \le n$.
- p_{ij}: $\Pr(a_{i-1} < c < a_{j+1})$

Árvore de Busca Binária Ótima

Algoritmo 10.1 (ABB-ÓTIMA)
Entrada Probabilidades $p_i = Pr(c = a_i)$ e $q_i = Pr(c \in J_i)$.

Saída Vetores c e r como descrita acima.

```
for i:=1 to n+1 do
  p_{i(i-1)} := q_{i-1}
  c_{i(i-1)} := 0
end for

for d:=0 to n-1 do { para todas diagonais }
  for i:=1 to n-d do { da chave i }
    j:=d+i          { até chave j }
    p_{ij} := p_{i(j-1)} + p_j + q_j
    c_{ij} := ∞
    for r:=i to j do
      c := p_{ij} + c_{i(r-1)} + c_{(r+1)j}
      if c < c_{ij} then
        c_{ij} := c
        r_{ij} := r
    end for
  end for
end for
```

i/j	1	2	3	···	n
1					
2					
.					
.					
n					

Finalmente, queremos analisar a complexidade desse algoritmo. São três laços, cada com não mais que n iterações e com trabalho constante no corpo. Logo a complexidade pessimista é $O(n^3)$.

10.2.4. Árvores rubro-negros

Árvores rubro-negros são árvores binárias cujos nodos possuem cor preto ou vermelho e satisfazem:

Definição 10.4 (Propriedades de árvores rubro-negros)

a) A raiz é os nodos externos são pretos.

b) Cada nodo vermelho possui um pai preto.

c) Todos caminhos simples de um nodo para uma nodo externo contém o mesmo número de nodos pretos sem contar o próprio nodo, chamada a sua *altura preta*.

Observação 10.1
Alguns autores usam a cor do nodo para indicar a cor da aresta para o pai. O significado de nodos ou cores vermelhos é o mesmo nas duas interpretações: eles representam *nodos internos* respectivamente *arestas internas* de nodos maiores: contraindo nodos vermelhos com os seus pais ou, equivalentemente, contraindo arestas vermelhos, obtemos uma árvore 2,3,4, porque um nodo preto possui no máximo dois filhos vermelhos. ◊

Observação 10.2
Uma árvore binária balanceada com todos nodos pretos é uma árvore rubro-negro. ◊

Lema 10.3
A altura de uma árvore rubro-negro com n elementos é no máximo $2\log(n+1)$.

Prova. Pela observação 10.1 uma árvore rubro-negro corresponde com uma árvore 2,3,4. Essa árvore é balanceada, porque temos o mesmo número de nodos pretos nos caminhos da raiz para as folhas. O número de folhas $n+1$ tem que ser entre 2^h e 4^h, sendo h a altura dessas árvore, portanto $h \leq \log(n+1)$. Logo, a árvore original possui altura no máximo $2\log(n+1)$, porque nenhum caminho da raiz para folha contém uma sequência de dois nodos vermelhos. ■

Corolário 10.1
Encontrar o elemento mínimo ou máximo, o predecessor ou sucessor de um elemento ou procurar um elemento em uma árvore rubro-negra possui complexidade $O(\log n)$.

Para garantir a complexidade das operações temos que explicar manter as propriedades de uma árvore rubro-negra sobre inserções e deleções de elementos. Para este fim usaremos duas operações, uma *rotação* (ver figura 10.3) e uma *inversão de cores* (ver figura 10.4). Segue uma implementação dessas operações. Observe que as rotações retornam a nova raiz da subárvore em que a rotação foi aplicada.

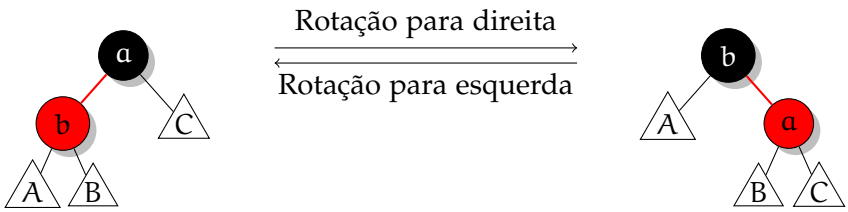


Figura 10.3.: Rotações para direita e esquerda.

```
inverte_cores(A):=
  inverte_cor(A)
  inverte_cor(A.l)
  inverte_cor(A.r)
end

rotaciona_esquerda(A) :=
  R := A.r
  A.r := R.l
  R.l := A
  R.l := A
  R.cor := A.cor { cor constante para cima }
  A.cor := vermelho
  return R
end

rotaciona_direita(A) :=
  R := A.l
  A.l := R.r
  R.r := A
  R.cor := A.cor { cor constante para cima }
  A.cor := vermelho
  return R
end
```

Observação 10.3
A rotação e a inversão de cores mantem a propriedade 10.3 de árvores de busca. Eles também mantem propriedades 10.4 de uma árvore rubro-negro. (Uma rotação é definida para qualquer árvore binária, mas ela mantem as propriedades da árvore rubro-negro no caso das cores indicadas na figura 10.3. ◊

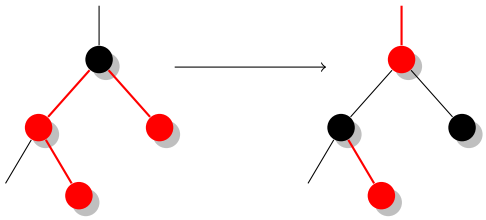


Figura 10.4.: Recoloração.

Para inserir uma nova chave, procuramos a folha que corresponde com a sua posição esperada e inserimos um novo nodo vermelho com a chave. O novo nodo é vermelho para manter a propriedade 10.4c). Caso o novo nodo é a raiz ou possui pai vermelho, as propriedades 10.4a) respectivamente 10.4b) são violados, e temos que modificar a árvore para estabelecer-las novamente.

1. Caso o novo nodo é raiz ela recebe cor preto.

2. Caso o irmão do pai também é vermelho aplicaremos a recoloração. O processo se repete com o pai do pai que agora é vermelho.

3. Caso o irmão do pai vermelho é preto aplicaremos uma ou duas rotações. A direção das rotações depende da posição do novo nodo e o seu pai. Caso os dois são filhos esquerdos ou direitos, rotacionamos o pai do pai para direita ou esquerda, respectivamente. O pai do nodo é colorido preto com dois filhos vermelhos. Caso o novo nodo é filho direito de um pai que é filho esquerdo, rotacionamos primeiro o pai para esquerda e depois o novo pai para direita. Similarmente, caso o nodo é filho esquerdo de um pai que é filho direito, rotacionamos primeiro o pai para direita e depois o novo pai para esquerda. A figura 10.5 mostra os quatro casos possíveis. Depois de aplicar uma ou duas rotações a árvore satisfaz as condições 10.4 e o processo termina.

Uma implementação das rotações é simples porém tedioso pelo fato que existem vários casos simétricos. Por isso vamos adotar uma convenção proposta por Sedgewick (*Algorithms for the masses*): caso um nodo possui somente um filho vermelho, será o filha da esquerda. Com isso obtemos árvores rubro-negros inclinados para esquerda (inglês: left-leaning red-black-trees). Com essa convenção sabemos que o pai vermelho de um nodo novo vermelho é o filho da esquerda do seu pai. Portanto as rotações são somente as duas do lado da esquerda na figura 10.5.

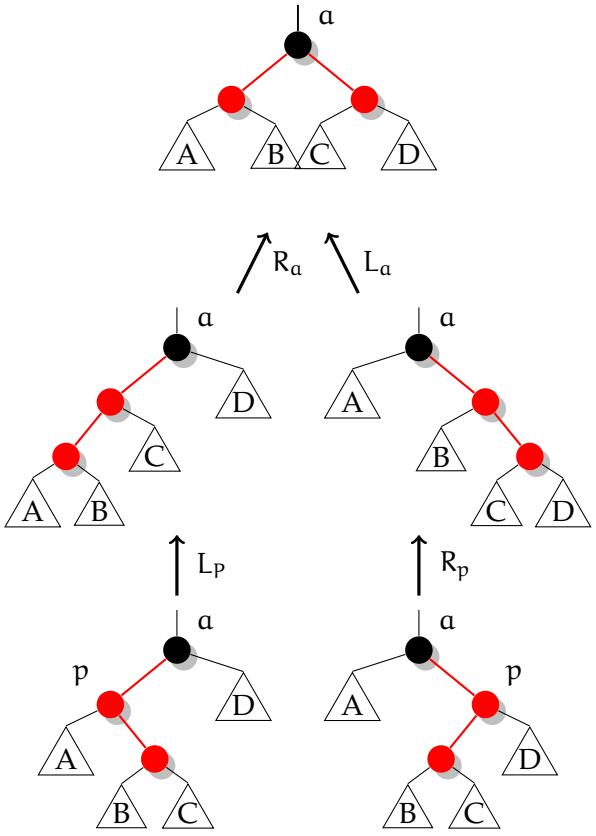


Figura 10.5.: Uma ou duas rotações estabelecem novamente as propriedades de uma árvore rubro-negro.

Ainda para simplificar a implementação vamos garantir que nenhum nodo possui dois filhos vermelhos. Neste caso aplicaremos uma inversão de cores. Equivalentemente podemos dizer que vamos manter uma árvore $(2,3)$ e separar eventuais 4-nodos em dois 2-nodos.

```
insert(A,c) :=
  A.r := insert_rec(A.r, c)
  A.r.cor := preto
end

{ garante que as propriedades da árvore rubro−negro }
{ estão satisfeitas em A }
estabelece_rn(A) :=
  { mantem a árvore inclinada para esquerda }
  if preto(A.l) e vermelho(A.r) then
    A = rotaciona_esquerda(A)

  { mantem propriedade 10.4b) }
  if vermelho(A.l) e vermelho(A.l.l) then
    A = rotaciona_direita(A)

  { separa 4−nodos em dois 2-nodos }
  if vermelho(A.l) e vermelho(A.r) then
    inverte_cores(A)

  return A
end

insert_rec(A,c) :=
  if A = ϵ then
    return Node(c)
  end if

  { insere recursivamente }
  if A.c = c then
    return
  else if A.c > c
    A.l := insert_rec(A.l,c)
  else
    A.r := insert_rec(A.r,c)
```

```
  end if

  return estabelece_rn(A)
end
```

Para remover um elemento usaremos a mesma estratégia de árvores binárias: caso o nodo a ser removido possui somente um filho interno é simples de remover, caso contrário substituirmos-lo pelo sucessor simétrico e removemos este sucessor. Se o nodo removido estava vermelho a árvore ainda satisfaz as propriedades 10.4, caso contrário temos que rebalancear-lo.

Começaremos com um procedimento que remove o menor elemento da árvore. Isso é particularmente simples, porque esse elemento não possui filho da esquerda. Para garantir que o nodo removido é vermelho, a implementação mantem a invariante que o nodo atual ou o seu filho da esquerda é vermelho. Chegando no menor elemento, o nodo tem que ser vermelho e pode ser simplesmente removido. Na caminho da raiz para o menor elemento temos que manter essa invariante.

```
moveVermelhoEsquerda(A) :=
  { pré-condição: vermelho(A) e preto(A.l) e preto(A.r) }
  inverte_cores(A)

  { propriedade 10.4b) violado na direita? re-estabelece }
  if vermelho(A.r.l) then
    A.r = rotaciona_direita(A.r)
    A = rotaciona_esquerda(A)
    inverte_cores(A)
  end if
  return A
end

deletemin(A) :=
  A.r := deletemin_rec(A.r)
  A.r.cor := preto
end

deletemin_rec(A) :=
  { Invariante: vermelho(A) ou vermelho(A.l)

  { encontramos o mínimo, A é vermelho: remove }
  if A.l = ϵ
```

```
    return ϵ

  { Invariante violado em A.l? re−estabelece }
  if preto(A.l) e preto(A.l.l) then
    A := moveVermelhoEsquerda(A)

  { remove recursivamente }
  A.l := deletemin_rec(A.l)

  return estabelece_rn(A)
```

Observação 10.4

- Por ter uma árvore $(2,3)$ a linha 6 testa corretamente a única violação da propriedade 10.4b) após a inversão de cores. Isso corresponde com o caso que o nodo da direita é um 3-nodo.

- Caso moveVermelhoEsquerda somente inverte as cores uma vez deixamos um nodo vermelho para a direita e eventualmente um 4-nodo neste ponto. A chamada `estabelece_rn` inclinará a árvore para esquerda após a remoção do elemento.

$\Diamond$

Para remover um elemento arbitrário temos que manter a invariante também nas chamadas recursivas para a direita. Caso o filho da esquerda é vermelho, podemos rotacionar o nodo para direita para garantir isso. Caso contrário um procedimento similar de uma chamada para esquerda garante a invariante. Como em todos casos serão removidos somente nodos com no máximo um filho; caso contrário removeremos o sucessor simétrico.

```
moveVermelhoDireita(A) :=
  { pré−condição: vermelho(A) e preto(A.l) }
  {                 e preto(A.r) e preto(A.r.l) }
  inverte_cores(A)

  { propriedade 10.4b) violado na esquerda? re−estabelece }
  if vermelho(A.l.l) then
    A = rotaciona_direita(A)
    inverte_cores(A)
  end if
  return A
end
```

```
delete(A,c) :=
  A.r := delete_rec(A.r,c)
  A.r.cor := preto
end

delete_rec(A,c) :=
  { chave na esquerda? }
  if c < A.c then
    { Invariante violado em A.l? re-estabelece }
    if preto(A.l) e preto(A.l.l) then
      A := moveVermelhoEsquerda(A)
    { remove recursivamente }
    A.l := delete(A.l, c)
  else
    { vermelho na esquerda? usa para garantir invariante }
    if vermelho(A.l) then
      rotaciona_direita(A)
    { chave encontrado numa folha? remove }
    if c = A.c e A.r = ϵ then
      return ϵ
    { re-estabelece invariante para a direita }
    if preto(A.r) e preto(A.r.l) then
      moveVermelhoDireita(A)
    { chave encontrado num nodo interno? }
    { troca com sucessor simétrico }
    if c = A.c then
      swap(A,min(A.r)) { troca chave e valor }
      A.r := deletemin_rec(A.r)
    else { chave na direita }
      A.r := delete_rec(A.r,c)
    end if
    return estabele_rn(A)
  end if
```

Observação 10.5

- Na linha 32 sabemos que $A.l = \epsilon$: pela rotação anterior caso $A.l \neq \epsilon$ tem que ser preto, uma violação da propriedade10.4c).

$\Diamond$

10.2.5. Árvores AVL

Árvores AVL são árvores de busca binária satisfazendo o critério de balanceamento da

Definição 10.5
Uma árvore binária é *balanceada* caso para cada nodo interno a diferença das alturas das duas sub-árvores é no máximo um.

A sigla AVL é representa os inicias dos inventores Adelson-Velskii e Landis desse tipo de árvore (Adelson-Velskii e Landis 1962). Uma consequência do balanceamento é que a altura é logarítmico no número de chaves.

Lema 10.4
Uma árvore AVL com n chaves possui altura $O(\log n)$.

Prova. Seja $N(h)$ o menor número de nodos internos de uma árvore AVL com altura h. Temos $N(0) = 0$ e $N(1) = 1$ e $N(h) = 1 + \min\{2N(h-1), N(h-1) + N(h-2)\} = 1 + N(h-1) + N(h-2)$. Logo $N(h) \geq F(h)$ com F os números de Fibonacci, i.e. $N(h) \geq \Phi^h/\sqrt{5}$ com $\Phi = (1+\sqrt{2})/2$. Logo para uma árvore AVL com n chaves e altura h sabemos que $n \geq N(h) \geq \Phi^h/\sqrt{5}$, i.e. $h \leq \log \sqrt{5}n / \log \Phi = O(\log n)$. ■

Corolário 10.2
Encontrar o elemento mínimo ou máximo, o predecessor ou sucessor de um elemento ou procurar um elemento em uma árvore AVL possui complexidade $O(\log n)$.

Para completar a definição de uma árvore AVL temos que especificar como inserir e remover elementos mantendo o critério 10.5. A inserção procura o ponto de inserção de novo elemento e adiciona um novo nodo. Depois ela percorre o caminho do novo nodo até a raiz para corrigir eventuais violações do balanceamento. A remoção procura o elemento e – como sempre – caso ele possui dos filhos substitui-lo pelo sucessor simétrico que será removido. Para remover um elemento com menos que dois filhos o pai recebe o único filho, caso existe, ou um nodo interno caso contrário, como novo filho no lugar do elemento. Depois temos que percorrer o caminho do novo filho até a raiz para garantir o balanceamento.
Para poder balancear uma árvore eficientemente vamos armazenar no campo b se o filho da esquerda possui maior altura ($b = -1$), os dois filhos possuem as mesma altura ($b = 0$), ou o filho da direita possui maior altura ($b = 1$).
Detalhes seguem.

10.2.6. Árvores Splay

Uma árvore Splay é uma árvore binário de busca. Árvores Splay reduzem todas operações para uma única operação "splay" (inglês para "chanfrar"). A operação splay(A,c) procura a chave c na árvore e depois aplica rotações até o nodo p, que é igual c, caso c faz parte da árvore, ou o pai de c, caso contrário, é a raiz da árvore:

- Caso p possui avô aplica duas rotações (no pai do p e depois no novo pai do p) tal que p sobe dois níveis.
- Caso contrário aplica uma rotação no pai do p tal que p sobe um nível.

O resultado da operação splay(A,c) é uma árvore com raiz c, caso c está contido na árvore, ou o seu predecessor ou sucessor simétrico, caso contrário. Para inserir uma chave c executamos splay(A,c). Caso c se encontra na raiz já faz parte da árvore; caso contrário podemos inserir uma nova raiz com chave c com um filho sendo a antigo raiz, e outro uma sub-árvore da antiga raiz. A ordem das chaves decide a seleção da sub-árvore e a posição dos filhos.
Para remover uma chave c podemos testar usando splay(A,c) se c faz parte da árvore. Caso positivo um splay(A.l,∞) transforma a sub-árvore de esquerda tal que a raiz não possui filho de direita. Logo podemos podemos substituir este filho pelo sub-árvore da direita da raiz atual, e remover a raiz. A nova raiz da árvore é a raiz da sub-árvore de esquerda.
Detalhes seguem.

10.2.7. Árvores (a, b)

Numa árvore (a, b) cada nodo exceto a raiz possui entre a e b filhos (incluindo nodos externos), e consequentemente entre $a - 1$ e $b - 1$ chaves. Deste forma, uma árvore binária é uma árvore $(2, 2)$. Uma árvore (a, b) é particularmente interessante para $a \geq 2$ e $b \geq 2a - 1$. A última condição garante uma flexibilidade suficiente na implementação das operações como veremos abaixo. Logo a menor árvore (a, b) interessante que satisfaz estas condições é uma árvore $(2, 3)$ proposto por Aho et al. (1974). Estudaremos árvores $(2, 4)$ ou árvores $2, 3, 4$. TBD: Por quê?
A busca numa árvore (a, b) é uma simples generalização da busca numa árvore binária: temos que comparar com todas chaves presentes num nodo para encontrá-lo ou decidir qual o filho em que a busca continuará. Para a e b grande, essa busca pode ser implementada como busca binária.
A inserção similarmente busca o ponto de inserção de uma nova chave. Caso o nodo encontrado possui menos que b filhos podemos inserir a nova chave

diretamente. Caso contrário uma estratégia é separar primeiramente o nodo com b filhos. Para discutir isso, vamos supor no momento que trata-se de uma árvore $(2,4)$, i.e., o nodo que queremos inserir a chave possui 4 filhos. Podemos separar este nodo em dois nodos com 2 filhos e inserir a terceira chave no pai. Isso pode levar uma "cascata" de separações do nodos até a raiz. Uma maneira de evitar isso, e separar nodos com 4 filhos já no caminho da raiz para o nodo de inserção. Essa versão é conhecido por árvore $2,3,4$ "top-down". Depois de separação tem espaço para inserir a nova chave num dos dois nodos com 2 filhos criados. No caso geral de uma árvore (a,b), podemos separar um nodo com b filhos em dois com $\lfloor b/2 \rfloor$ e $\lceil b/2 \rceil$ filhos.
DETALHES SEGUEM.

10.2.8. Árvores B

Uma árvore B é uma árvore $(\lceil m/2 \rceil, m)$. Uma aplicação típica de uma árvore B é armazenar um número grande de chaves em memória externa. Por isso o objetivo de uma árvore B é manter o maior número de chaves em cada nodo e por consequência reduzir a altura os mais possível, porque cada nível implica em uma operação de entrada/saída que domina o tempo das operações. Um valor típico de m é igual ao número de chaves que cabem num bloco de disco, por exemplo $m = 512$.
DETALHES SEGUEM.

10.2.9. Tries

Um *trie* é uma árvore de busca especializado para palavras sobre uma dado alfabeto. Cada aresta de uma trie é rotulado com um caráter do alfabeto, a o caminho da raiz para um nodo interno representa um prefixo de uma palavra definido pelas caráteres do caminho. Caso uma palavra no dicionário é definido pelo prefixo chegamos em uma folha da árvore que contém a palavra completa.
Uma caso particular interessante são tries sobre o alfabeto $\{0,1\}$ que representam códigos binários. Um subconjunto de códigos interessante são códigos livre de prefixos. Eles permitem representar uma sequência de palavras sem introduzir um separador, porque nenhuma palavra é prefixo de outra. (O capítulo sobre algoritmos gulusos mostra como construir um trie ótima para um código livre de prefixos.)

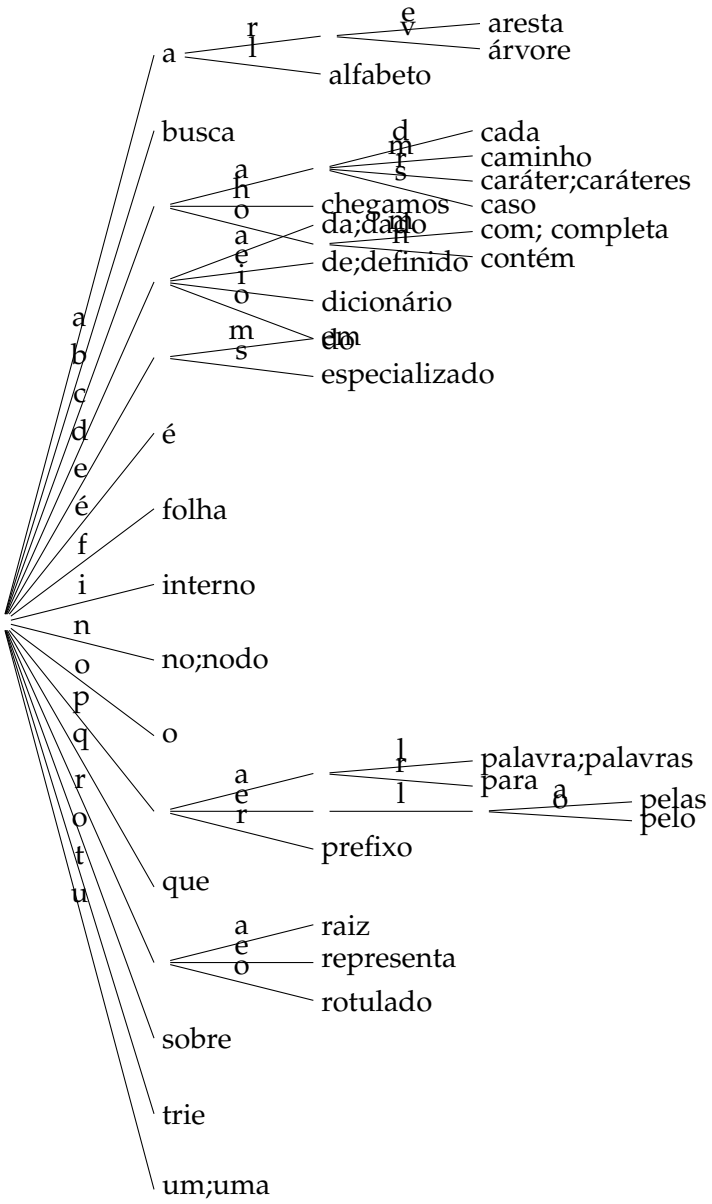


Figura 10.6.: Exemplo de um trie que contém as palavras do primeiro parágrafo dessa seção.

11. Filas de prioridade e heaps

11.1. Filas de prioridade e heaps

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

Exemplo 11.1
Árvore geradora mínima através do algoritmo de Prim.

Algoritmo 11.1 (Árvore geradora mínima)
Entrada Um grafo conexo não-orientado ponderado $G = (V, E, c)$

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

$V' := \{v_0\}$ `para um` $v_0 \in V$
$T := \emptyset$
`while` $V' \neq V$ `do`
  `escolhe` $e = \{u, v\}$ `com custo mínimo`
    `entre` V' `e` $V \setminus V'$ `(com` $u \in V', v \in V \setminus V'$`)`
  $V' := V' \cup \{v\}$
  $T := T \cup \{e\}$
`end while`

Algoritmo 11.2 (Prim refinado)
Implementação mais concreta:

$T := \emptyset$
`for` $u \in V \setminus \{v\}$ `do`
  `if` $u \in N(v)$ `then`
    $\text{value}(u) := c_{uv}$
    $\text{pred}(u) := v$

```
    else
      value(u) := ∞
    end if
    insert(Q, (value(u), u)) { pares (chave,elemento) }
  end for
  while Q ≠ ∅ do
    v := deletemin(Q)
    T := T ∪ {pred(v)v}
    for u ∈ N(v) do
    if u ∈ Q e c_vu < value(u) then
        value(u) := c_uv
        pred(u) := v
        update(Q, u, c_vu)
      end if
    end for
  end while
```

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

◊

Observação 11.1
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. $m = \Omega(n^2)$. ◊

Observação 11.2
Implementação com lista ordenada: insert $= O(n)$, deletemin $= O(1)$, update $= O(n)$, e temos custo $O(n^2 + n + mn) = O(mn)$[2]. ◊

Observação 11.3
Implementação com uma lista de $\sqrt{n}$ blocos de $\sqrt{n}$ elementos, insert, deletemin e update podem ser implementados em tempo $O(\sqrt{n})$, logo o algoritmo de Prim e de Dijkstra tem complexidade $O(m\sqrt{n})$. ◊

Exemplo 11.2
Caminhos mais curtos com o algoritmo de Dijkstra

[1] Com chaves compactas $[1, n]$.
[2] Na hipótese razoável que $m \geq n$

Algoritmo 11.3 (Dijkstra)
Entrada Grafo $G = (V, E)$ com pesos $c_e \geq 0$ nas arestas $e \in E$, e um vértice $s \in V$.

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

$d_s := 0; d_v := \infty, \forall v \in V \setminus \{s\}$
visited$(v) :=$ false$, \forall v \in V$
$Q := \emptyset$
insert$(Q, (s, 0))$
while $Q \neq \emptyset$ **do**
  $v :=$ deletemin(Q)
  visited$(v) :=$ true
  for $u \in N(v)$ **do**
    if not visited(u) **then**
      if $d_u = \infty$ **then**
        $d_u := d_v + d_{vu}$
        insert$(Q, (u, d_u))$
      else if $d_v + d_{vu} < d_u$
        $d_u := d_v + d_{vu}$
        update$(Q, (u, d_u))$
      end if
    end if
  end for
end while

A fila de prioridade contém pares de vértices e distâncias.

Proposição 11.1
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 concreta depende da implementação desses operações. ■

Proposição 11.2
O algoritmo de Dijkstra é correto.

Prova. Seja $\mathrm{dist}(s,x)$ a menor distância entre s e x. Provaremos por indução que para cada vértice v selecionado na linha 6 do algoritmo $d_v = \mathrm{dist}(s,x)$. 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

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

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

Observação 11.4
Podemos ordenar n elementos usando um heap com n operações "insert" e n operações "deletemin". Pelo limite de $\Omega(n \log n)$ para ordenação via comparação, podemos concluir que o custo de "insert" mais "deletemin" é $\Omega(\log n)$. Portanto, pelo menos uma das operações é $\Omega(\log n)$. ◊

O caso médio do algoritmo de Dijkstra Dado um grafo $G = (V, E)$ e um vértice inicial arbitrário supõe que temos um conjunto $C(v)$ de pesos positivos com $|C(v)| = |N^-(v)|$ para cada $v \in V$. Atribuiremos permutações dos pesos em $C(v)$ aleatoriamente para os arcos entrantes em v.

Proposição 11.3 (Noshita (1985))
O algoritmo de Dijkstra chama update em média $n \log(m/n)$ vezes neste modelo.

Prova. Para um vértice v os arcos que podem levar a uma operação update em v são de forma (u,v) com $\mathrm{dist}(s,u) \leq \mathrm{dist}(s,v)$. Supõe que existem k arcos $(u_1,v),\ldots,(u_k,v)$ desse tipo, ordenado por $\mathrm{dist}(s,u_i)$ não-decrescente. Independente da atribuição dos pesos aos arcos, a ordem de processamento é o mesmo. O arco (u_i,v) leva a uma operação update caso

$$\mathrm{dist}(s,u_i) + d_{u_iv} < \min_{j:j<i} \mathrm{dist}(s,u_j) + d_{u_jv}.$$

Com isso temos $d_{u_iv} < \min_{j:j<i} d_{u_jv}$, i.e., d_{u_iv} é um mínimo local na sequência dos pesos dos k arcos. Pela análise no exemplo 2.16 sabemos que o número

esperado de máximos locais de uma permutação aleatória é $H_k - 1 \leq \ln k$ e considerando as permutações inversas, temos o mesmo número de mínimos locais. Como $k \leq \delta^-(v)$ temos um limite superior para o número de operações update em todos vértices de

$$\sum_{v \in V} \ln \delta^-(v) = n \sum_{v \in V} (1/n) \ln \delta^-(v) \leq n \ln \sum_{v \in V} (1/n)\delta^-(v) = n \ln m/n.$$

A desigualdade é justificada pela equação (A.23) observando que $\ln n$ é concava. ■

Com isso complexidade média do algoritmo de Dijkstra é

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

Usando uma fila de prioridade implementada por um heap binário que executa todas operações em $O(\log n)$ a complexidade média do algoritmo de Dijkstra é $O(m + n \log m/n \log n)$.

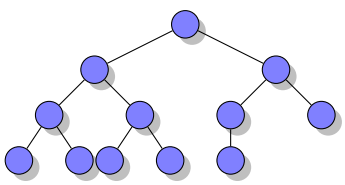
11.1.1. Heaps binários

Teorema 11.1
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.

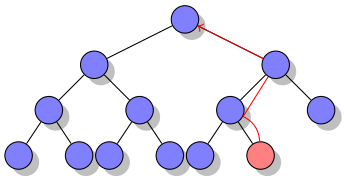
- *min-heap*: as chaves dos filhos são maior ou igual que a chave do pai;
- *max-heap*: as chaves dos filhos são menor ou igual que a chave do pai.

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.

```
insert(H,c) :=
  insere c na última posição p
  heapify-up(H,p)

heapify-up(H,p) :=
  if root(p) return
  if key(parent(p))>key(p) then
    swap(key(parent(p)),key(p))
    heapify-up(H,parent(p))
  end if
```

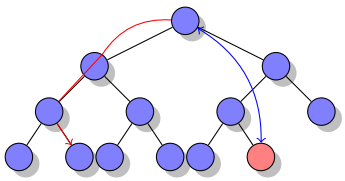


Lema 11.1
Seja T um min-heap. Decremente a chave do nó p. Após heapify-up(T, P) temos novamente um min-heap. A operação custa $O(\log n)$.

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.

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

```
  troca última posição com p
  heapify-down(H,p)

heapify-down(H,p):=
  if p não possui filhos return
  if p possui um filho then
    if key(left(p))<key(p)) then swap(key(left(p)),key(p))
  end if
  { p possui dois filhos }
  if key(p)>key(left(p)) or key(p)>key(right(p)) then
    if (key(left(p))<key(right(p)) then
      swap(key(left(p)),key(p))
      heapify-down(H,left(p))
    else
      swap(key(right(p)),key(p))
      heapify-down(H,right(p))
    end if
  end if
```



Lema 11.2
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.

```
update(H,p,v) :=
  if v < key(p) then
    key(p):=v
    heapify-up(H,p)
  else
    key(p):=v
    heapify-down(H,p)
  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

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

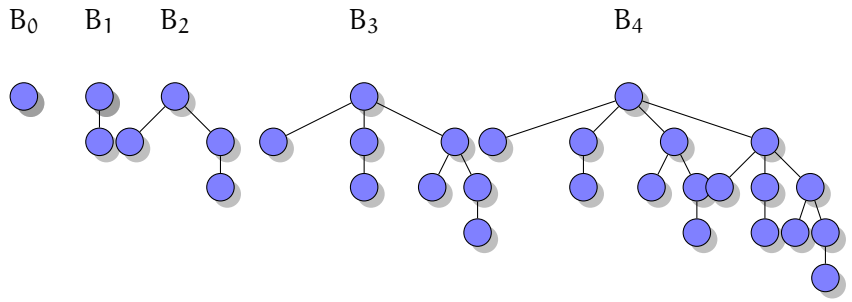
Outras observações:

- 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 $\texttt{pos}[c]$ é o índice da chave c no heap.

- A fila de prioridade não possui teste $u \in Q$ (linha 15 do algoritmo 11.2) eficiente. O teste pode ser implementado usando um vetor visited, tal que $\text{visited}[u]$ sse $u \notin Q$.

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

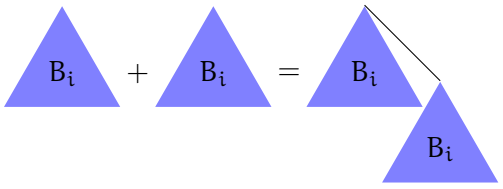


Lema 11.3
Uma árvore binomial tem as seguintes características:

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 11.5
Podemos combinar dois B_i obtendo um B_{i+1} e mantendo a ordenação do heap: Escolhe a árvore com menor chave na raiz, e torna a outra filho da primeira. Chamaremos essa operação "link". Ela tem custo $O(1)$ (veja observações sobre a implementação).



◊

Observação 11.6
Um B_i possui 2^i vértices. Um heap com n chaves consiste em $O(\log n)$ árvores. Isso permite juntar dois heaps binomiais em tempo $O(\log n)$. A operação é semelhante à soma de dois números binários com "carry". Começa juntar os B_0. Caso tem zero, continua, case tem um, inclui no heap resultante. Caso tem dois o heap resultante não recebe um B_0. Define como "carry" o link dos dois B_0's. Continua com os B_1. Sem tem zero ou um ou dois, procede como no caso dos B_0. Caso tem três, incluindo o "carry", inclui um no resultado, e define como "carry" o link dos dois restantes. Continue desse forma com os restantes árvores. Para heaps h_1, h_2 chamaremos essa operação 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,c): Como no caso do heap binário completo com custo $O(\log n)$.
- delete(h,c): decreasekey(h,c,$-\infty$); deletemin(h)

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

Custo amortizado do heap binomial Nosso potencial no caso do heap binomial é o número de árvores no heap. O custo de getmin e updatekey não altera o potencial e por isso permanece o mesmo. makeheap cria uma árvore que custa mais uma operação, mas permanece $O(1)$. deletemin pode criar $O(\log n)$ árvores novas, porque o heap contém no máximo um $B_{\lceil \log n \rceil}$ que tem $O(\log n)$ filhos, e permanece também com custo $O(\log n)$. Finalmente, insert reduz o potencial para cada link no meld e portanto agora custa somente $O(1)$ amortizado, com o mesmo argumento que no exemplo 2.21.
Desvantagem: a complexidade (amortizada) assintótica de calcular uma árvore geradora mínima permanece $O(n \log n + m \log n)$.

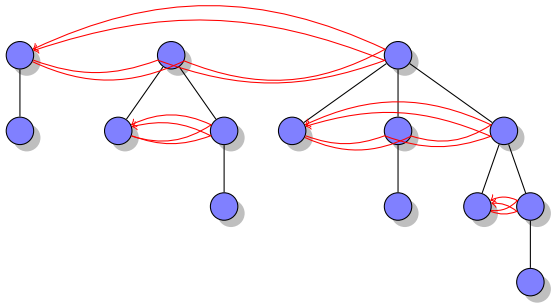
Meld preguiçosa Ao invés de reorganizar os dois heaps em um meld, podemos simplesmente concatená-los em tempo $O(1)$. Isso pode ser implementado sem custo adicional nas outras operações. A única operação que não tem complexidade $O(1)$ é deletemin. Agora temos uma coleção de árvores binomiais não necessariamente de posto diferente. O deletemin reorganiza

o heap, tal que obtemos um heap binomial com árvores de posto único novamente. Para isso, mantemos um vetor com as árvores de cada posto, inicialmente vazio. Sequencialmente, cada árvore no heap, será integrado nesse vetor, executando operações link só for necessário. O tempo amortizado de deletemin permanece $O(\log n)$.
Usaremos um potencial 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 posto diferente.

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



11.1.3. 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 no posto da árvore. A análise da complexidade das operações como deletemin depende desse fato para garantir que temos $O(\log n)$ árvores no heap. Consequência: Temos que garantir, que uma árvore não fica "podado" 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. 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.

```
insert(h, c) :=
  meld(makeheap(c))

getmin(h) :=
  return minroot(h)

delete(h,p) :=
  decreasekey(h,p,−∞)
  deletemin(h)

meld(h1,h2) :=
  h := lista com raízes de h1 e h2 (em O(1))
  minroot(h) :=
    if key(minroot(h1))<key(minroot(h2)) h1 else h2

decreasekey(h,p,c) :=
  key(p):= c
  if c < key(minRoot(h))
    minRoot(h) := p
  if not root(p)
    if key(parent(p))>key(p)
```

```
      corta p e adiciona na lista de raízes de h
      cut(p) := false
      cascading-cut(h,parent(p))

cascading-cut(h,p) :=
  { p perdeu um filho }
  if root(p)
    return
  if (not cut(p)) then
    cut(p) := true
  else
    corta p e adiciona na lista de raízes de h
    cut(p) := false
    cascading-cut(h,parent(p))
  end if

deletemin(h) :=
  remover minroot(h)
  juntar as listas do resto de h e dos filhos de minroot(h)
  { reorganizar heap }
  determina o posto máximo M = M(n) de h
  r_i := undefined para 0 ≤ i ≤ M
  for toda raíz r do
    remove r da lista de raízes
    d := degree(r)
    while (r_d not undefined) do
      r := link(r,r_d)
      r_d := undefined
      d := d+1
    end while
    r_d := r
  end for
  definir a lista de raízes pelas entradas definidas r_i
  determinar o novo minroot

link(h_1,h_2) :=
  if (key(h_1)<key(h_2))
    h := makechild(h_1,h_2)
  else
    h := makechild(h_2,h_1)
```

```
cut(h1) := false
cut(h2) := false
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 em que o corte em cascata remove mais que uma subárvore. Supondo que cortamos n árvores, o número de raízes é $t + n$ após dos cortes. Para todo corte em cascata, a árvore cortada é desmarcada, logo temos no máximo $m - (n - 1)$ marcas depois. Portanto custo amortizado é

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

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

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

Seja m o número de marcas antes do deletemin e m' o número depois. Como deletemin marca nenhum vértice, temos $m' \leq m$. O número de árvores t' depois de deletemin satisfaz $t' \leq M$ porque deletemin garante que existe no máximo uma árvore de cada posto. Portanto, o potencial depois de deletemin e $\varphi' = c_1 t + c_2 m' \leq c_1 M + c_2 m$, e o custo amortizado é

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

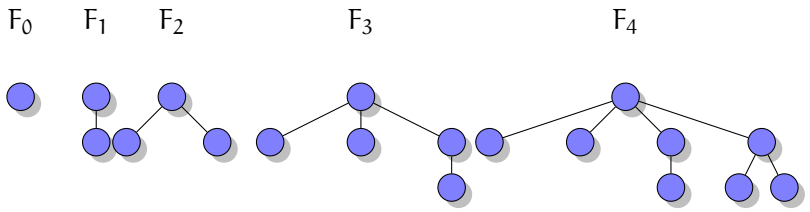
e com $c_1 \geq c_0$ temos custo amortizado $O(M)$.

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 11.4
Seja p um vértice arbitrário de um heap Fibonacci. Considerando os filhos na ordem temporal em que eles foram introduzidos, filho i possui pelo menos $i-2$ filhos.

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

Quais as menores árvores, que satisfazem esse critério?



Lema 11.5
Cada subárvore com uma raiz p com k filhos possui pelo menos F_{k+2} vértices.

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

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

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 11.1
O posto máximo de um heap Fibonacci com n elementos é $O(\log n)$.

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

11.1.4. Rank-pairing heaps

Haeupler et al. (2009) propõem um rank-pairing heap (um heap "emparelhando postos") com as mesmas garantias de complexidade que um heap Fibonacci e uma implementação simplificada e mais eficiente na prática (ver observação 11.9).

Torneios Um *torneio* é uma representação alternativa de heaps. Começando com todos elementos, vamos repetidamente comparar pares de elementos, e promover o vencedor para o próximo nível (Fig. 11.1(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 11.1(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 11.1(c) mostra uma representação compacta em forma de *semi-árvore*. Numa semi-árvore cada elemento possui um filho *ordenado* (na figura o filha da esquerda) e um filho *não-ordenado* (na figura o filho da direita). O filho ordenado é o perdedor da comparação direta com o elemento, enquanto o filho não-ordenado é o perdedor da comparação com o irmão vazio. A raiz possui somente um filho ordenado.
Cada elemento de um torneio possui um *posto*. Por definição, o posto de uma folha é 0. Uma comparação *justa* entre dois elementos do mesmo posto r resulta num elemento com posto $r+1$ no próximo nível. Numa comparação *injusta* entre dois elementos com postos diferentes, o posto do vencedor é definido pelo maior dois postos dos participantes (uma alternativa é que o posto fica o mesmo). O posto de um elemento representa um limite inferior do número de elementos que perderam contra-lo:

Lema 11.6
Um torneio com campeão de posto k possui pelo menos 2^k elementos.

Prova. Por indução. Caso um vencedor possui posto k temos duas possibilidades: (i) foi o resultado de uma comparação justa, com dois participantes

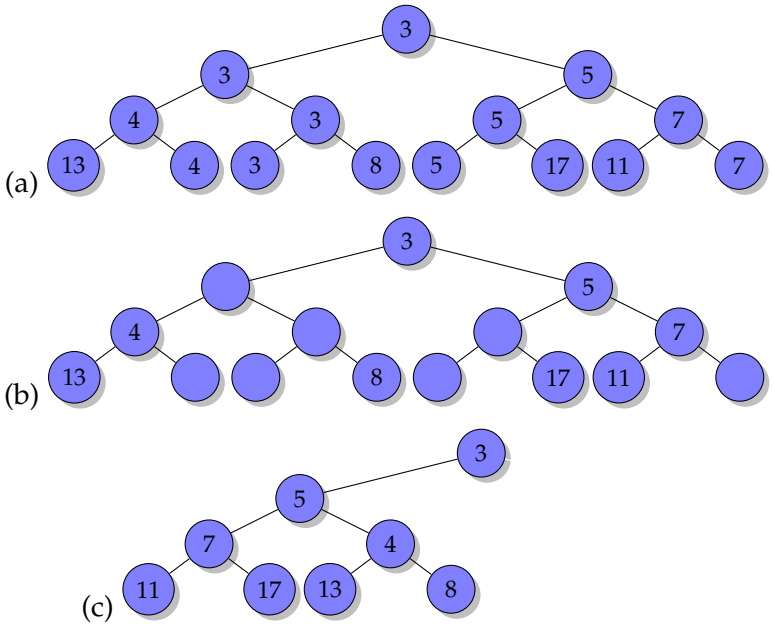


Figura 11.1.: Representações de heaps.

com posto $k-1$ e pela hipótese da indução com pelo menos 2^{k-1} elementos, tal que o vencedor ganhou contra pelo menos 2^k elementos. (ii) foi resultado de uma comparação injusta. Neste caso um dos participantes possuíu posto k e o vencedor novamente ganhou contra pelo menos 2^k elementos. ■

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

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

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

```
{ compara duas árvores }
link(t1,t2) :=
  if t1.c < t2.c then
    return makechild(t1,t2)
  else
```

```
    return makechild(t2,t1)
  end if

makechild(s,t) :=
  t.u := s.o
  s.o := t
  setrank(t)
  s.r := s.r + 1
  return s

setrank(t) :=
  if t.o.r = t.u.r
    t.r = t.o.r + 1
  else
    t.r = max(t.o.r,t.u.r)
  end if

{ cria um heap com um único elemento com chave c }
make-heap(c) := return Node(c,0,undefined,undefined)

{ insere chave c no heap }
insert(h,c) := link(h,make-heap(c))

{ união de dois heaps }
meld(h1,h2) := link(h1,h2)

{ elemento mínimo do heap }
getmin(h) := return h

{ deleção do elemento mínimo do heap }
deletemin(h) :=
  aloca array r0...rh.o.r+1
  t = h.o
  while t not undefined do
    t′ := t.u
    t.u := undefined
    register(t,r)
    t := t′
  end while
  h′ := undefined
```

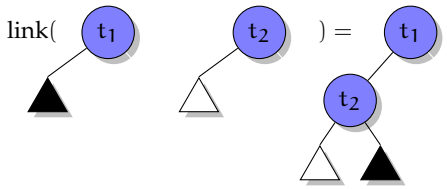


Figura 11.2.: A operação "link" para semi-árvores no caso $t_1.c < t_2.c$.

```
  for i = 0,...,h.o.r+1 do
    if r_i not undefined
      h' := link(h',r_i)
    end if
  end for
  return h'
end

register(t,r) :=
  if r_{t.o.r+1} is undefined then
    r_{t.o.r+1} := t
  else
    t:=link(t,r_{t.o.r+1})
    r_{t.o.r+1} := undefined
    register(t,r)
  end if
end
```

(A figura 11.2 visualiza a operação "link".)

Observação 11.7
Todas comparações de "register" são justas. As comparações injustas ocorrem na construção da árvore final nas linhas 35–39. ♢

Lema 11.7
Num torneio balanceado o custo amortizado de "make-heap", "insert", "meld" e "getmin" é O(1), o custo amortizado de "deletemin" é $O(\log n)$.

Prova. Usaremos o número de comparações injustas no torneio como potencial. "make-heap" e "getmin" não alteram o potencial, "insert" e "meld" aumentam o potencial por no máximo um. Portanto a complexidade amortizada dessas operações é O(1). Para analisar "deletemin" da raiz r do torneio vamos supor que houve k comparações injustas com r. Além dessas comparações injustas, r participou em no máximo $\log n$ comparações

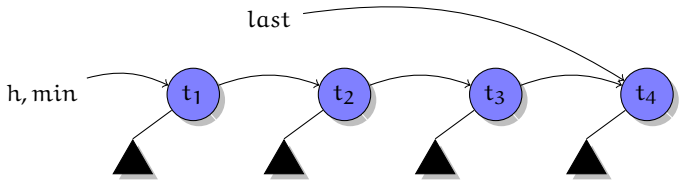


Figura 11.3.: Representação de um heap binomial.

justas pelo lema 11.6. Em soma vamos liberar no máximo $k + \log n$ árvores, que reduz o potencial por k, e com no máximo $k + \log n$ comparações podemos produzir um novo torneio. Dessas $k + \log n$ comparações no máximo $\log n$ são comparações injustas. Portanto o custo amortizado é $k + \log n - k + \log n = 2 \log n = O(\log n)$. ■

Heaps binomiais com varredura única O custo de representar o heap numa árvore única é permitir comparações injustas. Uma alternativa é permitir somente comparações justas, que implica em manter uma coleção de $O(\log n)$ árvores. A estrutura de dados resultante é similar com os heaps binomiais: manteremos uma lista (simples) de raízes das árvores, junto com um ponteiro para a árvore com a raíz de menor valor. O heap é representado pela raíz de menor valor, ver Fig. 11.3.

```
insert(h,c) :=
  insere make-heap(c) na lista de raizes
  atualize a árvore mínima

meld(h1,h2) :=
  concatena as listas de h1 e h2
  atualize a árvore mínima
```

Somente "deletemin" opera diferente agora:

```
deletemin(h) :=
  aloca um array de listas r0...r⌈log n⌉
  remove a árvore mínima da lista de raizes
  distribui as restantes árvores sobre r

  t := h.o
  while t not undefined do
    t′ := t.u
    t.u := undefined
```

```
    insere t na lista r_{t.o.r+1}
    t := t'
  end while

  { executa o maior número possível }
  { de comparações justas num único passo }

  h := undefined { lista final de raizes }
  for i = 0,...,⌈log n⌉ do
    while |r_i| ≥ 2
      t := link(r_i.head, r_i.head.next)
      insere t na lista h
      remove r_i.head, r_i.head.next da lista r_i
    end if
    if |r_i| = 1 insere r_i.head na lista h
  end for
  return h
```

Observação 11.8
Continuando com comparações justas até sobrar somente uma árvore de cada posto, obteremos um heap binomial. ♢

Lema 11.8
Num heap binomial com varredura única o custo amortizado de "make-heap", "insert", "meld", "getmin" é $O(1)$, o custo amortizado de "deletemin" é $O(\log n)$.

Prova. Usaremos o dobro do número de árvores como potencial. "getmin" não altera o potencial. "make-heap", "insert" e "meld" aumentam o potencial por no máximo dois (uma árvore), e portanto possuem custo amortizado $O(1)$. "deletemin" libera no máximo $\log n$ árvores, porque todas comparações foram justas. Com um número total de h árvores, o custo de deletemin é $O(h)$. Sem perda de generalidade vamos supor que o custo é h. A varredura final executa pelo menos $(h - \log n)/2 - 1$ comparações justas, reduzindo o potencial por pelo menos $h - \log n - 2$. Portanto o custo amortizado de "deletemin" é $h - (h - \log n - 2) = \log n + 2 = O(\log n)$. ■

rp-heaps O objetivo do rp-heap é adicionar ao heap binomial de varredura única uma operação "decreasekey" com custo amortizado $O(1)$. A 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.

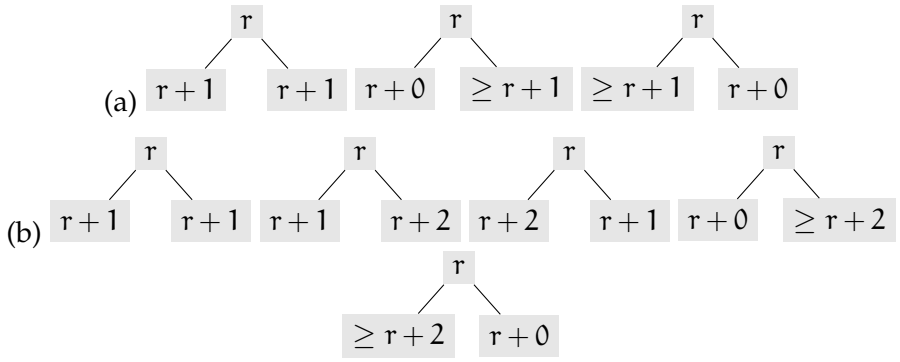


Figura 11.4.: Diferenças no posto de rp-heaps do tipo 1 (a) e tipo 2 (b).

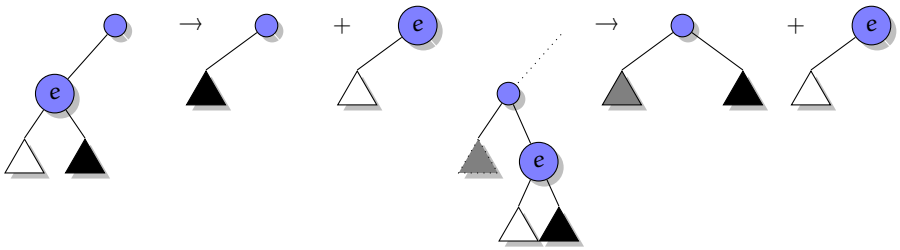


Figura 11.5.: A operação "decreasekey".

(ii) o heap Fibonacci usava cortes em cascata para manter um número suficiente de elementos na árvore; no rp-heap ajustaremos os postos do heap que perde uma sub-árvore. Para poder cortar sub-árvores temos que permitir uma folga nos postos. Num heap binomial a diferença do posto de um elemento com o posto do seu pai (caso existe) sempre é um. Num rp-heap do tipo 1, exigimos somente que os dois filhos de um elemento possuem diferença do posto 1 e 1, ou 0 e ao menos 1. Num rp-heap do tipo 2, exigimos que os dois filhos de um elemento possuem diferença do posto 1 e 1, 1 e 2 ou 0 e pelo menos 2. (Figura 11.4.)
Com isso podemos implementar o "decreasekey" (para rp-heaps do tipo 2) como segue:

```
decreasekey(h,e,Δ) :=
  e.c := e.c − Δ
  if root(e)
    return
  if parent(e).o = e then
```

```
    parent(e).o := e.u
  else
    parent(e).u := e.u
  end if
  parent(e).u := parent(e)
  e.u := undefined
  u := parent(e)
  parent(e) := undefined
  insere e na lista de raízes de h
  decreaserank(u)

rank(e) :=
  if e is undefined
    return −1
  else
    return e.r

decreaserank(u) :=
  if root(u)
    return
  if rank(u.o) > rank(u.u)+1 then
    k := rank(u.o)
  else if rank(u.u) > rank(u.o)+1 then
    k := rank(u.u)
  else
    k = max(rank(u.o),rank(u.u))+1
  end if
  if u.r = k then
    return
  else
    u.r := k
    decreaserank(parent(u))

delete(h,e) :=
  decreasekey(h,e,−∞)
  deletemin(h)
```

Observação 11.9
Para implementar o rp-heap precisamos além dos ponteiros para o filho ordenado e não-ordenado um ponteiro para o pai do elemento. A (suposta)

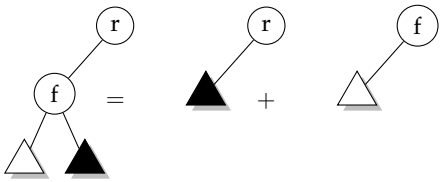


Figura 11.6.: Separar uma semi-árvore de posto k em duas.

eficiência do rp-heap vem do fato que o decreasekey altera os postos do heap, e pouco da estrutura dele e do fato que ele usa somente três ponteiros por elemento, e não quatro como o heap Fibonacci. ♢

Lema 11.9
Uma semi-árvore do tipo 2 com posto k contém pelo menos ϕ^k elementos, sendo $\phi = (1+\sqrt{5})/2$ a razão áurea.

Prova. Por indução. Para folhas o lema é válido. Caso a raiz com posto k não é folha podemos obter duas semi-árvores: a primeira é o filho da raiz sem o seu filho não-ordenado, e a segunda é a raiz com o filho não ordenado do seu filho ordenado (ver Fig. 11.6). Pelas regras dos postos de árvores de tipo dois, essas duas árvores possuem postos $k-1$ e $k-1$, ou $k-1$ e $k-2$ ou k e no máximo $k-2$. Portanto, o menor número de elementos n_k contido numa semi-árvore de posto k satisfaz a recorrência

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

que é a recorrência dos números Fibonacci. ■

Lema 11.10
As operações "decreasekey" e "delete" possuem custo amortizado $O(1)$ e $O(\log n)$

Prova. Ver (Haeupler et al. 2009). ■

11.1.5. Heaps ocos

(Versão: 10619.)

Introdução

Objetivo: operações com a mesma complexidade amortizada que heaps de Fibonacci. Para um heap h, chave k e elemento e temos as operações:

- make-heap(): $O(1)$
- find-min(h)/getmin(h): $O(1)$
- meld(h_1,h_2): $O(1)$
- insert(e,k,h): $O(1)$
- decrease-key(e,k,h): $O(1)$
- delete(e,h): $O(\log n)$
- delete-min(h): $O(\log n)$

Ideia principal: a operação delete esvazia nós, produzindo nós ocos (ingl. hollow nodes), a operação decrease-key é um delete, seguido por um insert. Teremos duas medidas:

n Número de elementos no heap

N Número de nós no heap = # de elementos + # de nós ocos = # operações insert + # operações decrease-key

Variantes de heaps ocos:

- Heaps ansiosos (ingl. "eager heaps") com múltiplas raízes.
- Heaps ansiosos com uma única raíz.
- Heaps preguiçosos.

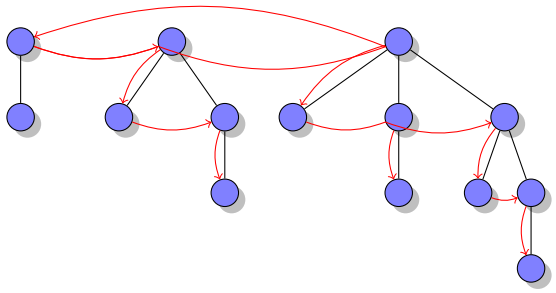
```
def Node =
  item // elemento
  key  // chave
  fc   // ponteiro para primeiro filho
  ns   // ponteiro para próximo irmão
  rank // posto do nó

def Item =
  no   // nó correspondente
  // mais dados satelites
```

Operação básica: link Um *link* gera um vencedor e um perdedor, que se torna filho do vencedor, e aumenta o posto do vencedor.

```
(ranked)link(t1,t2) :=
  if t1.key ≤ t2.key
    return makechild(t1,t2)
  else
    return makechild(t2,t1)

makechild(w,l) :=
  l.ns   := w.fc
  w.fc   := l
  w.rank := w.rank+1
  return w
```

Representação básica

- Lista simples circular de árvores com ordenação do heap, representada por um ponteiro à árvore cuja raíz contém a menor chave (chamada a *raíz mínima*).
- Cada *nó cheia* armazena um item. Podem existir *nós ocos* sem item.
- Nós ocos nunca mais ficam cheias, eles podem somente ser destruídos.
- Filhos ficam armazenados em listas simples, em ordem não-crescente de postos.



```
make-heap() := return null

make-heap(e,k) := return Node(e,k,null,self,0)
```

```
getmin(h) := h

findmin(h) := return h is not null? h.item : null

meld(h1,h2) :=
  if h1 is null return h2
  if h2 is null return h1
  swap(h1.ns,h2.ns) // cria uma lista circular simples
  if h1.key ≤ h2.key return h1 else return h2

insert(e,k,h) := meld(make-heap(e,k),h)

decrease-key(e,k,h) :=
  u = e.node
  v = make-heap(e,k)
  v.rank = max{0,u.rank-2}
  // desloca os filhos de postos 0,...,rank-2 para v
  if u.rank ≥ 2
    v.fc := u.fc.ns.ns
    u.fc.ns.ns := null
  return meld(v,h)

delete(e,h) :=
  e.node.item := null
  if e.node = h
    delete-min(h)

delete-min(h) :=
  if h is null: return
  h.node.item := null

  aloca um array R0,R1,...,RM
  // repetidamente remove raízes ocos e une os heaps
  r:=h
  repeat
    rn := r.ns
    link-heap(r,R)
    r:=rn
  until r==h
```

```
    // reconstrói o heap
    h:=null
    for i=0,...,M
      if R_i is not null
        R_i.ns := R_i
        h := meld(h,R_i)
    return h

link-heap(h,R) :=
  if h is hollow
    r:=h.fc
    while r is not null
      rn := r.ns
      link-heap(r,R)
      r := rn
    destroy node h
  else
    i := h.rank
    while R_i is not null
      h := link(h,R_i)
      R_i := null
      i := i + 1
    end
    R_i := h
```

Invariantes

1. Ordenação do heap.
2. Invariante do posto: cada nó de posto r possui r filhos com postos $0,\ldots,r-1$, exceto no caso $r \geq 2$ e o nó foi esvaziada por uma operação decrease-key. Neste caso o nó possui dois filhos de postos $r-1$ e $r-2$.

Corretude

Teorema 11.2
Heaps com nós ocos implementam corretamente todas operação e mantém as invariantes.

Prova. Por indução sobre o número de operações. ■

Lembrança: os números de Fibonacci são definidos por $F_0 = 0, F_1 = 1, F_{i+2} = F_i + F_{i+1}$, para $i \geq 0$ e temos $F_{i+2} \geq \Phi^i$, com a razão áurea $\Phi = (1 + \sqrt{5})/2$.

Teorema 11.3
Um nó de posto r possui pelo menos $F_{r+3} - 1$ descendentes (cheios ou ocos), incluindo o próprio nó, na árvore.

Prova. Por indução sobre r. Para $r = 0$, temos $F_3 - 1 = 1$, e para $r = 1$ temos $F_4 - 1 = 2$ e a afirmação está correta, porque para $r < 2$ um nó não perde filhos caso for esvaziado. Para $r \geq 2$ pela invariante do posto temos pelo menos dois filhos com postos $r - 1$ e r_2. Pela hipótese da indução eles tem pelo menos $F_{r+1} - 1$ e $F_{r+2} - 1$ descendentes e logo r possui pelo menos $F_{r+1} - 1 + F_{r+2} - 1 + 1 = F_{r+3} - 1$ descendentes. ■

Corolário 11.2
Depois uma operação delete-min o número de árvores é no máximo $\lceil \log_\Phi N \rceil = O(\log N)$ porque temos no máximo uma árvore por posto. Logo podemos escolher $M = \lceil \log_\Phi N \rceil$ na operação delete-min.

Teorema 11.4
O tempo amortizado por operação num heap oco é $O(1)$, exceto para as operações delete e delete-min, que tem complexidade $O(\log N)$ para um heap com N nós.

Prova. Todas operações exceto a deleção do elemento mínimo possuem tempo $O(1)$ no caso pessimista. O custo de uma deleção é $O(H + T)$ com H o número de nós ocos destruídos, e T o número de árvores antes das operações link. Depois das operações link temos no máximo $\log_\Phi N$ árvores, logo faremos pelo menos $T - \log_\Phi N$ operações link e no máximo $\log_\Phi N$ operações meld. Logo o custo total é $O(1)$ por destruição de um nó oco, e por link, mas $O(\log N)$.
Para contabilizar a destruição do um nó, aumentamos o custo de cada criação (insert, decrease-key) por 1.
Para contabilizar as operações link: define um potencial igual ao número de nós cheias, que não são filho de outro nó cheia (i.e. raízes e filhos de nós ocos). Para todas operações diferente de delete-min e delete, o aumento do potencial é constante (no máximo 1 para insert, 3 para decrease-key, 0 para as demais). Para o delete que remove o elemento mínimo e delete-min, o custo amortizado de cada link é 0, porque um link combina duas raízes cheias, reduzindo o potencial por 1. Além disso, ao remover um elemento, o potencial aumenta por no máximo $\log_\Phi N$, um por cada filho do novo nó oco. Logo o custo amortizado de delete e delete-min é $O(\log N)$.

■

Tabela 11.1.: Complexidade das operações de uma fila de prioridade. Complexidades em negrito são amortizados. (1): meld preguiçoso.

	insert	getmin	deletemin	update	decreasekey	delete
Vetor	$O(1)$	$O(1)$	$O(n)$	$O(1)$	(update)	$O(1)$
Lista ordenada	$O(n)$	$O(1)$	$O(1)$	$O(n)$	(update)	$O(1)$
Heap binário	$O(\log n)$	$O(1)$	$O(\log n)$	$O(\log n)$	(update)	$O(\log n)$
Heap binomial	$\mathbf{O(1)}$	$O(1)$	$O(\log n)$	$O(\log n)$	(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	$O(1)$	$O(1)$	$\mathbf{O(\log n)}$	-	$\mathbf{O(1)}$	$\mathbf{O(\log n)}$

Re-otimizando o heap A análise acima é em função de N. Caso $\log N = O(\log n)$ temos um heap assintoticamente ótimo. Caso executamos muitas operações decrease-key, temos que reconstruir o heap periodicamente, para garantir $N = O(n)$. O método mais simples é: escolhe uma constante $c > 1$ e para $N > cn$ reconstrói o heap completamente, destruindo os nós ocos, criando heaps de um único nó de todos nós cheios, e aplicando operações meld para unir todos heaps. O custo é $O(N)$ para percorrer todo nó uma vez e pode ser atribuído na análise amortizada para as operações insert e delete-min.

Resumo: Filas de prioridade A tabela 11.1 resume a complexidade das operações para diferentes implementações de uma fila de prioridade.

11.1.6. Árvores de van Emde Boas

Pela observação 11.4 é impossível implementar uma fila de prioridade baseado em comparação de chaves com todas operações em $o(\log n)$. Porém existem algoritmos que ordenam n números em $o(n \log n)$, aproveitando o fato que as chaves são números com k bits, como por exemplo o radix sort que ordena em tempo $O(kn)$, ou aproveitando que as chaves possuem um domínio limitado, como por exemplo o counting sort que ordena n números em $[k]$ em tempo $O(n + k)$.
Uma *árvore de van Emde Boas* (árvore vEB) T realiza as operações

- member(T, e): elemento e pertence a T?
- insert(T, e): insere e em T
- delete(T, e): remove e de T

- min(T) e max(T): elemento mínimo e máximo de T, ou "undefined" caso não existe
- succ(T, e) e pred(T, e): successor e predecessor de e em T; e não precisa pertencer a T

no universo de chaves $[0, u-1]$ em tempo $O(\log\log u)$ e espaço $O(u)$.
Outras operações compostas podem ser implementados, por exemplo

```
deletemin(T) :=
    e := min(T); delete(e); return e
deletemax(T) :=
    e := max(T); delete(e); return e
```

Árvores binárias em ordem vEB Na discussão da implementação de árvores binárias na página 222 discutimos uma representação em ordem da busca por profundidade (BFS order). A ideia da ordem vEB é "cortar" a altura (número de níveis) h de uma árvore binária (que possui $n = 2^h - 1$ nodos e 2^{h-1} folhas) pela metade. Com isso obtemos

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

Os nodos dessa árvore são armazenados em ordem $T_0, T_1, \ldots, T_b$ e toda árvore T_i é ordenado recursivamente da mesma maneira, até chegar numa árvore de altura $h = 1$, como a Figura 11.7 mostra.
Armazenar uma árvore binária em ordem de vEB não altera a complexidade das operações. Uma busca, por exemplo, continua com complexidade $O(h)$. Porém, armazenado em ordem da busca por profundidade, uma busca pode gerar $\Theta(h)$ falhas no *cache*, no pior caso. Na ordem de vEB, a busca sempre atravessa $\Omega(\log_2 B)$ níveis, com B o tamanho de uma linha de cache, antes de gerar uma nova falha no *cache*. Logo uma busca gera somente $O(\log_2 n / \log_2 B) = O(\log_B n)$ falhas no *cache*. O layout se chama *cache oblivious* porque funciona sem conhecer o tamanho de uma linha de cache B.

Árvores vEB A estrutura básica de uma árvore de vEB é

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

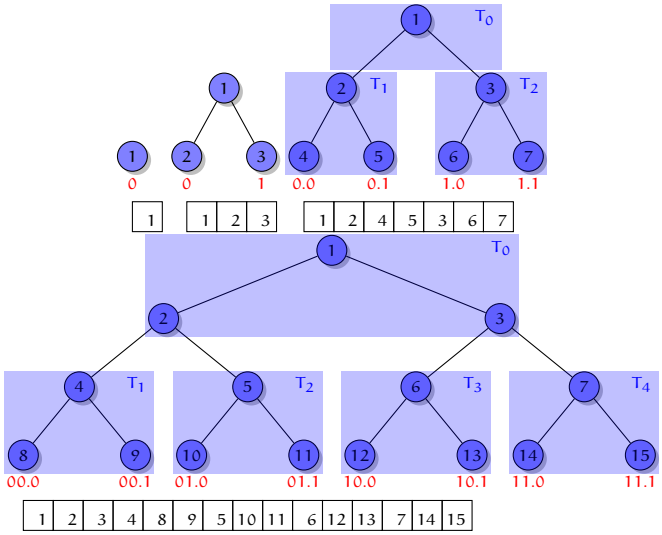
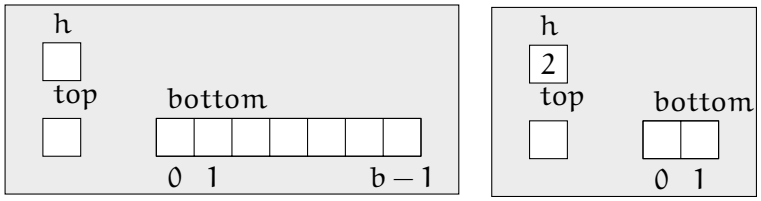


Figura 11.7.: Organização de árvores binárias em ordem de van Emde Boas para $h \in [4]$. As folhas são rotuladas por "cluster.subíndice". Abaixo da árvore a ordem do armazenamento do vértices é dado. Os T_i correspondem com as subárvores do primeiro nível de recursão.



(a) (b)

Figura 11.8.: Representação da primeira versão de uma árvore vEB. (a) Forma geral. (b) Caso base.

2. Cada folha armazena um bit, que é 1 caso o elemento correspondente pertence ao conjunto representado.

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

Todas as operações da estrutura acima podem ser implementadas em tempo $O(h) = O(\log u)$. Para melhorar isso, vamos aplicar a mesma ideia da ordem de van Emde Boas: a árvore é separada em uma árvore superior, e uma série de árvores inferiores, cada uma com altura $\approx h/2$. As folhas da árvore superior contém o resumo das raízes das árvores inferiores: por isso a árvore superior possui altura $\lfloor h/2 \rfloor + 1$, uma a mais comparado com a ordem de vEB.
Fig. 11.8 mostra essa representação. A altura da árvore está armazenada no campo h. Além disso temos um ponteiro "top" para a árvore superior, e um vetor de ponteiros "bottom" de tamanho $b = 2^{\lfloor h/2 \rfloor}$ para as raízes das árvores inferiores. No caso base com $h = 2$, abusaremos os campos "top" e "bottom" para armazenar os bits da raiz e dos dois filhos: um ponteiro arbitrário diferente de undefined representa um bit 1, o ponteiro undefined o bit 0. Para isso servem as funções auxiliares

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

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

```
subtree(e) := e ≫ ⌈h/2⌉ − 1
subindex(e) := e&(1 ≪ ⌈h/2⌉ − 1) − 1
element(s,i) := (s ≪ ⌈h/2⌉ − 1) | i
```

para extrair de um elemento o número da sub-árvore correspondente, ou o seu índice nesta sub-árvore, e para determinar o índice na árvore atual do i-ésimo elemento da sub-árvore s.
Com isso podemos implementar as operações como segue.

```
member(T,e) :=
  if T.h = 2
    return bit(T.bottom[e])
  return member(T.bottom[subtree(e)],subindex(e))

min(T,e) :=
  if T.h = 2
    if bit(T.bottom[0])
      return 0
    if bit(T.bottom[1])
      return 1
    return undefined

  c := min(T.top)
  if c = undefined
    return c
  return element(c,min(T.bottom[c]))

succ(T,e) :=
  if T.h = 2
    if e = 0 and bit(T.bottom[1])=1
      return 1
    return 0

  s := succ(T.bottom[subtree(e)],subindex(e))
  if s ≠ undefined
    return element(subtree(e),s)

  c := succ(T.top,subtree(e))
  if c = undefined
    return c
  return element(c,min(T.bottom[c]))
```

```
insert(T,e) :=
  if T.h = 2
    set(T.bottom[e])
    set(T.top)
  else
    insert(T.bottom[subtree(e)],subindex(e))
    insert(T.top,subtree(e))

delete(T,e) :=
  if T.h = 2
    clear(T.bottom[e])
    if (bit(T.bottom[1-e])=0
      clear(T.top)
  else
    delete(T.bottom[subtree(e)],subindex(e))
    s:=min(T.bottom[subtree(e)])
    if s = undefined
      delete(T.top,subtree(e))
```

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

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

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

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

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

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

1. Armazenar somente o mínimo, a operação "min" custa somente $O(1)$ é "insert", "succ" e "delete" consequentemente somente $O(h)$.

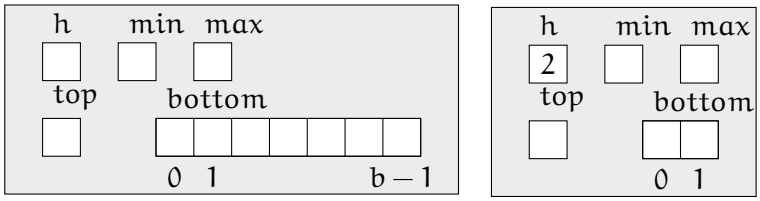


Figura 11.9.: Representação uma árvore vEB. (a) Forma geral. (b) Caso base.

2. Armazenado também o máximo, sabemos na operação "succ" se o sucessor está na árvore atual sem buscar, logo a operação "succ" pode ser implementada em $O(\log\log u)$.

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

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

```
clear(T) :=
  T.min:=1; T.max:=0; // convenção

empty(T) :=
  return T.min>T.max

singleton(T) :=
  return T.min=T.max

full(T) :=
  return T.min<T.max

member(T,e) :=
  if empty(T)
    return false
```

```
  if T.min = e or T.max = e
    return true

  { não é ‘‘min’’ nem ‘‘max’’? a base não contém o elemento }
  if T.h = 2
    return false

  return member(T.bottom[subtree(e)],subindex(e))

min(T) :=
  if empty(T)
    return undefined
  return T.min

max(T) :=
  if empty(T)
    return undefined
  return T.max

succ(T,e) :=
  if T.h=2
    if e = 0 and T.max = 1
      return 1
    return undefined

  if not empty(T) and e < T.min
    return T.min

  { sucessor na árvore atual }
  m:=max(T.bottom[subtree(e)])
  if m ≠ undefined and subindex(e)<m
    return element(subtree(e),
                   succ(T.bottom[subtree(e)],subindex(e)))

  { mínimo na árvore sucessora }
  c := succ(T.top,subtree(e))
  if c = undefined
    return c
  return element(c,min(T.bottom[c]))
```

```
pred(T,e) :=
  if T.h=2
    if e = 1 and T.min=0
      return 0
    return undefined

  if not empty(T) and T.max < e
    return T.max

  { predecessor na árvore atual }
  m:=min(T.bottom[subtree(e)])
  if m ≠ undefined and m <subindex(e)
    return element(subtree(e),
                   pred(T.bottom[subtree(e)],subindex(e)))

  { máximo na árvore predecessora }
  c:= pred(T.top,subtree(e))
  if c = undefined
    if not empty(T) and T.min<e
      return T.min
    else
      return undefined

  return element(c,max(T.bottom[c]))

insert(T,e) :=
  if empty(T)
    T.min := T.max := e
    return

  { novo mínimo: setar min, insere min anterior }
  if e < T.min
    swap(T.min,e)

  { insere recursivamente }
  if T.h > 2
    if empty(T.bottom[subtree(e)])
      insert(T.top,subtree(e))
```

```
    insert(T.bottom[subtree(e)],subindex(e))

  { novo máximo: atualiza }
  if T.max < e
    T.max := e

delete(T,e) :=
  if empty(T)
    return

  if singleton(T)
    if T.min = e
      clear(T)
    return

  { novo mínimo? }
  if e = T.min
    T.min := element(min(T.top),min(T.bottom[min(T.top)]))
    e := T.min

  { remove e da árvore }
  delete(T.bottom[subtree(e)],subindex(e))

  if empty(T.bottom[subtree(e)])
    delete(T.top,subtree(e))
    if e = T.max
      c:=max(T.top)
      if c = undefined
        T.max := T.min
      else
        T.max := element(c,max(T.bottom[c]))
  else
    T.max := element(subtree(e),max(T.bottom[subtree(e)]))
```

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

o último elemento e custo somente $O(1)$.

11.1.7. Tópicos

Fast marching method

A equação Eikonal (grego eikon, imagem)

$$\begin{aligned} \|\nabla T(x)\| F(x) &= 1, & x \in \Omega, \\ T|_{\partial\Omega} &= 0, \end{aligned}$$

define o tempo de chegada de uma superfície que inicia no tempo 0 na fronteira $\partial\Omega$ de um subconjunto aberto $\Omega \subseteq \mathbb{R}^3$ e se propaga com velocidade $F(x) > 0$ na direção normal[3]. O fast marching method resolve a equação Eikonal por discretizar o espaço regularmente, aproximar as derivadas do gradiente $\|\nabla T\|$ por diferenças finitas e propagar os valores com um método igual ao algoritmo de Dijkstra.
Com

$$\nabla T = (\partial T/\partial x, \partial T/\partial y, \partial T/\partial z)$$

temos

$$\|\nabla T\|^2 = (\partial T/\partial x)^2 + (\partial T/\partial z)^2 + (\partial T/\partial z)^2 = 1/F^2.$$

Definindo as diferenças finitas

$$D^{+x}T = T(x_1 + 1, x_2, x_3) - T(x); \qquad D^{-x}T = T(x) - T(x_1 - 1, x_2, x_3)$$

podemos aproximar

$$\partial T/\partial x \approx T_x = \max\{D^{-x}T, -D^{+x}T, 0\}$$

e com aproximações similares para as direções y e z obtemos uma equação quadrática em $T(x)$

$$\|\nabla T\|^2 \approx T_x^2 + T_y^2 + T_z^2 = 1/F^2 \tag{11.1}$$

Na solução dessa equação valores ainda desconhecidos de T são ignorados. O fast marching method define $T = 0$ para os pontos iniciais em $\partial\Omega$ e coloca-os numa fila de prioridade. Repetidamente o ponto de menor tempo é extraído da fila, os vizinhos ainda não visitados são atualizados de acordo com (11.1) e entram na fila, caso ainda não fazem parte. (Na terminologia do fast marching method, os pontos com distância já conhecida são "vivos" (*alive*), os pontos na fila formam a "faixa estreita" (*narrow band*), os restantes pontos são "distantes" (*far away*).)

[3] O método também funciona para $F(x) < 0$, mas não para $F(x)$ com sinais diferentes.

Busca informada

O algoritmo de Dijkstra encontra o caminho mais curto de um vértice origem $s \in V$ para todos os outros vértices num grafo ponderado $G = (V, E, d)$. Caso estamos interessados somente no caminho mais curto para um único vértice destino $t \in T$, podemos parar o algoritmo depois de processar t. Isso é uma aplicação muito comum, por exemplo na busca da rota mais curta em sistemas de navegação. Uma *busca informada* processa vértices que estimadamente são mais próximos do destino com preferência. O objetivo é processar menos vértices antes de encontrar o destino. Um dos algoritmos mais conhecidos de busca informada é o algoritmo A^*. Para cada vértice $v \in V$ com distância $g(v)$ do origem s, ele usa uma função heurística $h(v)$ que estima a distância para o destino t e processa os vértices em ordem crescente do custo total estimado

$$f(v) = g(v) + h(v). \tag{11.2}$$

O desempenho do algoritmo A^* depende da qualidade de heurística h. Ele pode, diferente do algoritmo de Dijkstra, processar vértices múltiplas vezes, depois de descobrir um caminho mais curto para um vértice já processado. Isso é a principal diferença com o algoritmo de Dijkstra. Uma outra é que substituímos o campo "visited" usando no algoritmo Dijkstra 11.3 por um conjunto V de vértices já visitados, porque o A^* é frequentemente aplicado em grafos com um número grande de vértices, que são explorados passo a passo sem armazenar todos vértices do grafo na memoria.

```
g(s) := 0
f(s) := g(s) + h(s)
V := ∅ { vértices já visitados }
Q := ∅
insert(Q, (s, f(s)))
while Q ≠ ∅ do
  v := deletemin(Q)
  V := V ∪ {v}
  if v = t        { destino encontrado }
    return
  for u ∈ N⁺(v) do
    if u ∈ Q then { ainda aberto: atualiza }
      g(u) := min(g(v) + d_vu, g(u))
      f(u) := g(u) + h(u)
      update(Q, (u, f(u)))
```

```
    else if u ∈ V then
      if g(v) + d_vu < g(u) then
        { caminho menor p/ vértice já processado }
        V := V \ {u}
        g(u) := g(v) + d_vu
        f(u) := g(u) + h(u)
        insert(Q, (u, f(u)))
      end if
    else                          { novo vértice }
        g(u) := g(v) + d_vu
        f(u) := g(u) + h(u)
        insert(Q, (u, f(u)))
    end if
  end for
end while
```

Observação 11.10
O algoritmos de Dijkstra e A^* funcionam de forma idêntica quando substituímos o vértice destino $t \in V$ por um conjunto de vértices destino $T \subseteq V$. ◊

Existe uma formulação alternativa, equivalente do algoritmo A^*. Ao invés de sempre processar o vértice aberto de menor valor f podemos processar sempre o vértice aberto de menor distância $\hat{g}$ num grafo com pesos modificados $\hat{d}_{uv} = d_{uv} - h(u) + h(v)$. Com pesos modificados obtemos para a distância total de um caminho uv arbitrário P

$$\begin{aligned}\hat{g}(u,v) = \sum_{(u',v')\in P} \hat{d}_{u'v'} &= \sum_{(u',v')\in P} d_{u'v'} - h(u') + h(v') \\ &= h(v) - h(u) + \sum_{(u',v')\in P} d_{u'v'} = h(v) - h(u) + g(u,v).\end{aligned}$$

Com $\hat{g}(u) = \hat{g}(s,u)$ obtemos

$$\begin{aligned}f(u) \leq f(v) &\iff g(u) + h(u) \leq g(v) + h(v) \\ &\iff \hat{g}(u) + h(s) \leq \hat{g}(v) + h(s) \\ &\iff \hat{g}(u) \leq \hat{g}(v).\end{aligned}$$

Logo a ordem de processamento por menor $\hat{g}$ ou por menor valor f é equivalente.

Para garantir a otimalidade de uma solução a heurística h tem que ser *admissível*. Caso h é *consistente* o algoritmo A^* não somente retorna a solução ótima, mas processa cada vértice somente uma vez.

Definição 11.1 (Admissibilidade e consistência)
Seja $\delta(v)$ a distância mínima do vértice v ao destino t. Uma heurística h é *admissível* caso h é um limitante inferior à distância mínima, i.e.

$$h(v) \leq \delta(v). \tag{11.3}$$

Uma heurística é consistente caso o seu valor diminui de acordo com o pesos do grafo: para um arco $(u, v) \in A$

$$h(v) \geq h(u) - d_{uv}. \tag{11.4}$$

Na representação alternativa, o critério de consistência (11.4) é equivalente com $\hat{d}_{uv} = d_{uv} - h(u) + h(v) \geq 0$. Com isso temos diretamente o

Teorema 11.5
Caso h é consistente o algoritmo A^* nunca processa um vértice mais que uma vez.

Prova. Neste caso $\hat{d}_{uv} \geq 0$. Logo todas distâncias são positivas é o algoritmo A^* é equivalente com o algoritmo de Dijkstra. Por um argumento similar ao da proposição (11.2) o A^* nunca processa um vértice duas vezes. ■

Lema 11.11
Caso h é consistente, h é admissível.

Prova. Seja $P = v_0 v_1 \dots v_k$ um caminho de $v_0 = u$ a $v_k = t$. Então

$$d(P) = \sum_{i \in [k]} d_{v_{i-1}, v_i} \geq \sum_{i \in [k]} h(v_{i-1}) - h(v_i) = h(u) - h(t) \geq h(u).$$

Em particular, para um caminho P^* ótimo de u a t temos $h(u) \leq d(P^*) = \delta(P^*)$. ■

Teorema 11.6
Caso existe uma solução mínima e h é admissível o algoritmo A^* encontra a solução mínima.

Prova. Seja $P^* = v_0 v_1 \dots v_k$ um caminho ótimo de $v_0 = s$ a $v_k = t$. Caso A^* não terminou, t ainda não foi explorado. Logo existe um vértice aberto de menor índice v_i em P^*. Agora supõe que o próximo vértice explorado é t, mas o valor de t não é ótimo, i.e. $f(t) > d(P^*)$. Mas então $f(v_i) \leq d(P^*) < f(t)$, porque h é admissível, em contradição com a exploração de t. ■

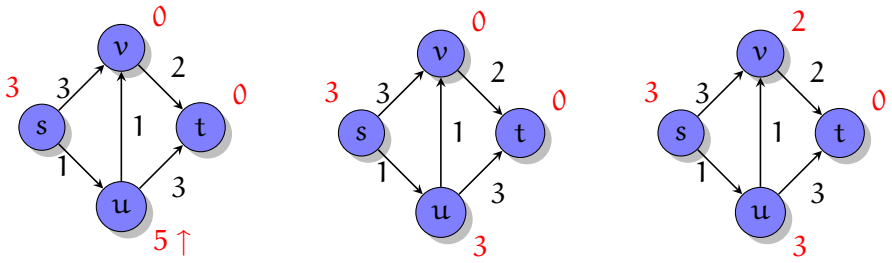


Figura 11.10.: Esquerda: Heurística não-admissível. A^* produz o valor errado 5. Centro: Heurística admissível, mas inconsistente. A^* visita v duas vezes. Direita: Heurística admissível e consistente. A^* visita cada vértice somente uma vez.

Exemplo 11.3
Figure 11.10 mostra uma grafo com três funções heurísticos h diferentes. A heurística no grafo da esquerda não é admissível em u (marcado por $\uparrow$). O A^* expande s, v e depois t e termina com a distância errada de 5 para chegar em t. A heurística no grafo do meio é admissível, mas não consistente: $h(u) \leq h(v) + 1$ não é satisfeito. O A^* expande s, v, u, v, t, i.e. o vértice v é processado duas vezes. Finalmente a heurística no grafo da direita é consistente (e por isso admissível). O A^* expanda cada vértice uma vez, na ordem s, u, t (ou s, u, v, t).

◊

Exemplo 11.4
A Figura 11.11 compara uma busca com o algoritmo de Dijkstra com uma busca com o A^* num grafo geométrico com 5000 vértices e uma aresta entre vértices de distância no máximo 0.02. Vértices não explorados são pretos, vértices explorados claros. A claridade corresponde com a ordem de exploração.

◊

11.1.8. Notas

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 e Zwick 2009).
Karger propôs uma variante de heaps de Fibonacci que substituem a marca "cut" usado nos cortes em cascata por uma decisão randômica: com pro-

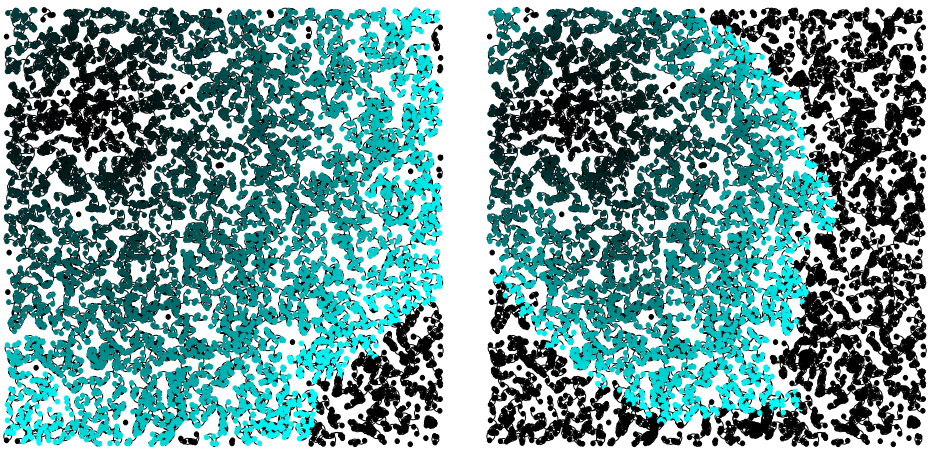


Figura 11.11.: Comparação de uma busca com o algoritmo de Dijkstra (esquerda) e o A^* (direita).

babilidade 0.5 continua cortando, senão para. Caso além disso o heap é construído novamente com probabilidade $1/n$ depois de cada operação, "deletemin" possui complexidade $\Theta(\log^2 n / \log \log n)$ (Li e Peebles 2015).
Armazenar e atravessar árvores em ordem de van Emde Boas usando índices, similar ao ordem por busca em largura é possível (Brodal et al. 2001). O consumo de memoria das árvores de van Emde Boas pode ser reduzido para $O(n)$ (Dementiev et al. 2004; Cormen et al. 2009).
Mais sobre o fast marching method se encontra em Sethian (1999). Uma aplicação interessante é a solução do caixeiro viajante contínuo (Andrews e Sethian 2007).

11.1.9. Exercícios

Exercício 11.1
Prove lema 11.3. Dica: Use indução sobre n.

Exercício 11.2
Prove que um heap binomial com n vértices possui $O(\log n)$ árvores. Dica: Por contradição.

Exercício 11.3 (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 11.4 (Laboratório 2)

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

Exercício 11.5

A proposição 11.2 continua ser correto para grafos com pesos negativos? Justifique.

12. 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* $h : U \to [m]$ 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 c_1, c_2 com $h(c_1) = h(c_2)$, que geram *colisões*. Logo uma tabela hash precisa definir um método de *resolução de colisões*. Uma solução é *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.

12.1. Hashing com listas encadeadas

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

$$\Pr(h(c) = i) = 1/m. \tag{12.1}$$

Seja $n_i := |l_i|$ o tamanho da lista i e $c_{ji} := \Pr(h(j) = i)$ 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 o *fator de ocupação* da tabela hash.

```
insert(c,H) :=
  insert(c, l_h(c))

lookup(c,H) :=
  lookup(c, l_h(c))

delete(c,H) :=
  delete(c, l_h(c))
```

Teorema 12.1
Uma busca sem sucesso precisa tempo esperado $\Theta(1 + \alpha)$.

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 12.2
Uma busca com sucesso precisa tempo esperado $\Theta(1 + \alpha)$.

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 sequência na qual as chaves foram inseridas. Supondo que inserimos as chaves no inicio da lista, $E[p]$ é um mais que o número de chaves inseridos depois de c na mesma lista.
Seja X_{ij} um variável aleatória que indica se chaves c_i e c_j tem o mesmo valor hash. $E[X_{ij}] = \Pr(h(c_i) = h(c_j)) = \sum_{1 \le k \le m} \Pr(h(c_i) = k) \Pr(h(c_j) = k) = 1/m$. Seja p_i a posição da chave c_i na sua lista. Temos

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

e para uma chave aleatória c

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

Portanto, o tempo esperado de uma busca com sucesso é

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

■

Seleção de uma função hash Para implementar uma tabela hash, temos que escolher uma função hash, que satisfaz (12.1). Para facilitar isso, supomos que o universo de chaves é um conjunto $U = [u]$ de números inteiros. (Para tratar outros tipos de chaves, 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 ímpares é ímpar, e se $m = 2^k$ o valor hash consiste nos primeiros k bits. Uma escolha que funciona na prática é um número primo "suficientemente" distante de uma potência de 2.
O *método de multiplicação* define

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

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

Hashing universal Outra idéia: Para qualquer função hash h fixa, sempre existe um conjunto de chaves, tal que essa função hash gera muitas colisões. (Em particular, um "adversário" que conhece a função hash pode escolher chaves $c \in h^{-1}(i)$ para qualquer posição $i \in [m]$, tal que $h(c) = i$ é constante. Para evitar isso podemos escolher uma função hash aleatória 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 12.3
Se escolhemos uma função hash $h \in \mathcal{H}$ uniformemente, para uma chave arbitrária c 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 de 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

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

Hashing perfeito Hashing é *perfeito* sem colisões. Isso podemos garantir somente caso conheçemos a chaves a serem inseridos na tabela. Para uma função aleatória de uma família universal de funções hash para uma tabela hash de tamanho m, o número esperado de colisões é $E[\sum_{i \neq j} X_{ij}] = \sum_{i \neq j} E[X_{ij}] \leq n^2/m$. Portanto, caso esolhemos uma tabela de tamanho $m > n^2$ o número esperado de colisões é menos que um. Em particular, para $m > cn^2$ com $c > 1$ a probabilidade de uma colisão é $\Pr(\sum_{i \neq j} X_{ij} \geq 1) \leq E[\sum_{i \neq j} X_{ij}] \leq n^2/m < 1/c$ onde a primeira desigualdade segue da desigualdade de Markov.

12.2. Hashing com endereçamento aberto

Uma abordagem para resolução de colisões, chamada *endereçamento aberto*, é escolher uma 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 $h(c,1),\ldots,h(c,m)$ é uma permutação de $[m]$, representa essa estratégia.

```
insert(c,H) :=
  for i in [m]
    if H[h(c,i)] = free
      H[h(c,i)]=c
      return

lookup(c,H) :=
  for i in [m]
    if H[h(c,i)] = free
      return false
    if H[h(c,i)] = c
      return true
  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 12.4
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 encontrar 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 12.1
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 12.5
Caso $\alpha < 1$ a função lookup precisa esperadamente $1/\alpha \ln 1/(1-\alpha)$ testes caso a chave esteja na tabela, e cada chave tem a mesma probabilidade de ser procurada.

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

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

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

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

■

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

12.3. 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 $O(1)$:

```
lookup(c,H) :=
  if H[h1(c)] = c or H[h2(c)] = c
    return true
  return false

delete(c,H) :=
  if H[h1(c)] = c
    H[h1(c)] := free
  if H[h2(c)] = c
    H[h2(c)] := free
```

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

```
insert(c,H) :=
  if H[h1(c)] = c or H[h2(c)] = c
    return
  p := h1(c)
```

```
do n vezes
  if H[p] = free
    H[p] := c
    return
  swap(c,H[p])
  { escolhe a outra posição da chave atual }
  if p = h1(c)
    p := h2(c)
  else
    p := h1(c)
rehash(H)
insert(c,H)
```

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

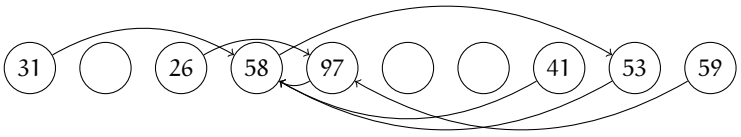
Exemplo 12.1
Para chaves de dois dígitos c_1c_2 seja $h_1(c) = 3c_1 + c_2 \mod m$ e $h_2(c) = 4c_1 + c_2$. Para $m = 10$ obtemos para uma sequencia aleatória de chaves

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

e a seguinte sequencia de tabelas hash

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

O grafo cuco correspondente é



$\Diamond$

Lema 12.2
Para posições i e j e um $c > 1$ tal que $m \geq 2cn$, a probabilidade de existir um caminho mínimo de i para j de comprimento $d \geq 1$ é no máximo c^{-d}/m.

Prova. Observe que a probabilidade de um item c ter posições i e j como alternativas é no máximo $\Pr(h_1(c) = i, h_2(c) = j) + \Pr(h_1(c) = j, h_2(c) = i) = 2/m^2$. Portanto a probabilidade de pelo menos uma das n chaves ter posições alternativas i e j é no máximo $2n/m^2 = c^{-1}/m$.
A prova do lema é por indução sobre d. Para $d = 1$ a afirmação está correto pela observação acima. Para $d > 1$ existe um caminho mínimo de comprimento $d-1$ de i para um k. A probabilidade disso é no máximo $c^{-(d-1)}/m$ e a probabilidade de existir um elemento com posições alternativas k e j no máximo c^{-1}/m. Portanto, para um k fixo, a probabilidade existir um caminho de comprimento d é no máximo c^{-d}/m^2 e considerando todas posições k possíveis no máximo c^{-d}/m. ■
Com isso a probabilidade de existir um caminho entre duas chaves i e j, é igual a probabilidade de existir um caminho começando em $h_1(i)$ ou $h_2(i)$ e terminando em $h_1(j)$ ou $h_2(j)$, que é no máximo $4\sum_{i\geq 1} c^{-i}/m \leq 4/m(c-1) = O(1/m)$. Logo o número esperado de itens visitados numa inserção é $4n/m(c-1) = O(1)$, caso não é necessário reconstruir a tabela hash.

12.4. 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 $h_1, \ldots, h_k$.

```
insert(c,B) :=
  for i in 1...k
    b_{h_i(c)} := 1
  end for

lookup(c,B) :=
  for i in 1...k
    if b_{h_i(c)} = 0
      return false
  return true
```

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

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

que é igual ao valor esperado da fração de bits não setados[1]. 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 e Mitzenmacher 2003). Broder e 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, primeiramente A filtra os elementos, manda um filtro de Bloom S_A para B e depois B executa o join baseado em S_A. Para eliminação de eventuais elementos classificados erradamente, B manda os resultados para A e A filtra os elementos errados.

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

Tabela 12.1.: Complexidade das operações em tabelas hash. Complexidades em negrito são amortizados.

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

13. Grafos

(As definições elementares de grafos encontram-se na seção A.5 do apêndice.)

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

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

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

Uma escolha comum é um vetor de vértices que armazena listas de vizinhos. Esse estrutura permite uma inserção e deleção simples de arcos. Para facilitar a deleção de um vértice em grafos não-direcionados, podemos armazenar junto com o vizinho u do vértice v a posição do vizinho v do vértice u. A representação dos vizinhos por vetores é mais eficiente, e por isso preferível caso a estrutura do grafo é estático (Black Jr. e Martel 1998; Park et al. 2004).

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

[1] Ainda mais espaço consuma uma *matrix de incidência* entre vértices e arestas em $\mathbb{B}^{n\times m}$.

Tabela 13.1.: Operações típicas em grafos.

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

Supõe que $V = [n]$. Uma outra representação compacta e eficiente conhecido como *forward star* para grafos estáticos usa um *vetor de arcos* $a_1, \ldots, a_m$. Mantemos a lista de arestas ordenado pelo começo do arco. Uma permutação σ nos dá as arestas em ordem do término. (O uso de uma permutação serve para reduzir o consumo de memória.) Para percorrer eficientemente os vizinhos de um vértice armazenamos o índice s_v do primeiro arco sainte na lista de arestas ordenado pelo começo e o índice e_v do primeiro arco entrante na lista de arestas ordenado pelo término com $s_{n+1} = e_{n+1} = m + 1$ por definição. Com isso temos $N^+(v) = \{a_{s_v}, \ldots, a_{s_{v+1}-1}\}$ com $\delta_v^+ = s_{v+1} - s_v$, e $N^-(v) = \{a_{\sigma(e_v)}, \ldots, a_{\sigma(e_{v+1}-1)}\}$ com $\delta_v^- = e_{v+1} - e_v$. A representação precisa espaço $O(n+m)$.
Tabela 13.1 mostra a complexidade de operações típicas nas diferentes representações.

Parte IV.

Algoritmos

14. Algoritmos em grafos

14.1. Fluxos em redes

Definição 14.1
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

$$f(v) := \sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e = 0 \tag{14.1}$$

(ver Fig. 14.1).

Lema 14.1
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,

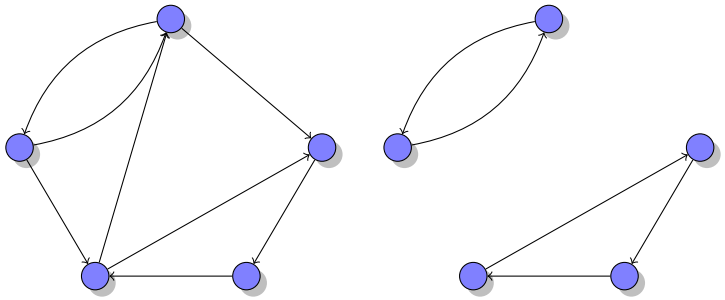


Figura 14.1.: Grafo (esquerda) com circulação (direita)

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 14.2
Um fluxo s–t satisfaz $f(s) + f(t) = 0$.

Prova. Pelo lema 14.1 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

$$\begin{aligned} \textbf{maximiza} \quad & f(s) && && (14.2)\\ \textbf{sujeito a} \quad & f(v) = 0 && \forall v \in V \setminus \{s, t\} \\ & 0 \leq f_e \leq c_e && \forall e \in E. \end{aligned}$$

Observação 14.1
O programa (14.2) 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. ◊

14.1.1. O 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.

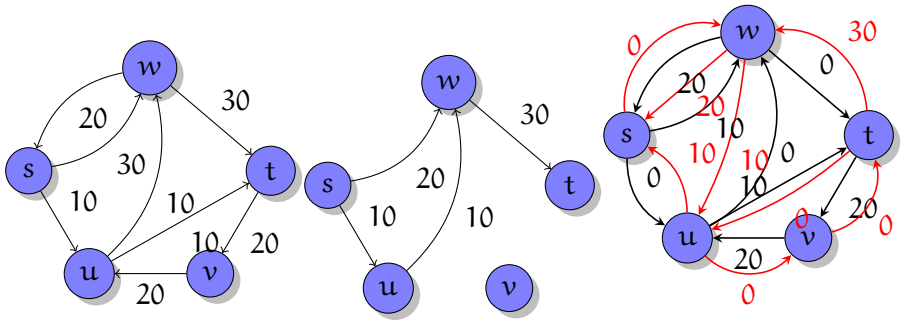


Figura 14.2.: Esquerda: Grafo com capacidades. Centro: Fluxo com valor 30. Direita: O grafo residual correspondente.

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"

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

Observação 14.2
Repetidamente procurar um caminho com gargalo positivo e aumentar nem sempre produz um fluxo máximo. Na Fig. 14.2 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. 14.3).
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 $f_{(u,v)} > 0$.

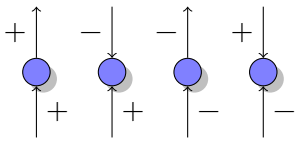


Figura 14.3.: Manter a conservação do fluxo.

Observe que na Fig. 14.2 o grafo residual possui um caminho $P = (s, w, u, t)$ que aumenta o fluxo por 10. O algoritmo de Ford-Fulkerson (Ford e Fulkerson 1956) consiste em, repetidamente, aumentar o fluxo num caminho s–t no grafo residual.

Algoritmo 14.1 (Ford-Fulkerson)
Entrada Grafo $G = (V, E, c)$ com capacidades c_e no arcos.

Saída Um fluxo f.

```
for all e ∈ E: f_e := 0
while existe um caminho s--t em G_f do
  Seja P um caminho s--t simples
  Aumenta o fluxo f um valor g(f,P)
end while
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 14.3
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 14.4
Em cada iteração, o fluxo aumenta por pelo 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 14.5
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 14.2
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 (14.1) para conjuntos: $f(X) := f(X, \bar{X}) - f(\bar{X}, X)$ é o fluxo neto do saindo do conjunto X.
Analogamente, escrevemos para as capacidades $c(X, Y) := \sum_{e \in F(X,Y)} c_e$. Uma partição $(X, \bar{X})$ é um *corte* s–t, se $s \in X$ e $t \in \bar{X}$.
Um arco e se chama *saturado* para um fluxo f, caso $f_e = c_e$.

Lema 14.6
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 14.7
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 14.1 (Fluxo máximo – corte mínimo)
O valor do fluxo máximo entre dois vértices s e t é igual ao valor do corte mínimo.

Lema 14.8
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 14.7, o valor de um fluxo arbitrário é menor ou igual que $c(X, \bar{X})$, portanto f é um fluxo máximo. ■
Prova. (Do teorema 14.1) 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. 14.4). 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. 14.4). Usando uma busca por profundidade para achar caminhos s–t ele termina, mas é ineficiente (Dean et al. 2006).

14.1.2. O algoritmo de Edmonds-Karp

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

Teorema 14.2
O algoritmo de Edmonds-Karp precisa $O(nm)$ iterações, e portanto termina em tempo $O(nm^2)$.

Lema 14.9
Seja $\delta_f(v)$ a distância entre s e v em G_f. Durante a execução do algoritmo de Edmonds-Karp $\delta_f(v)$ cresce monotonicamente para todos vértices em V.

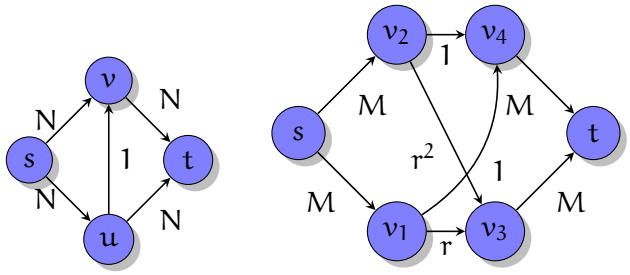


Figura 14.4.: Esquerda: Pior caso para o algoritmo de Ford-Fulkerson com pesos inteiros aumentando o fluxo por 2N vezes por 1 nos caminhos (s,u,v,t) e (s,v,u,t). Direita: Menor grafo com pesos irracionais em que o algoritmo de Ford-Fulkerson falha (Zwick 1995). $M \geq 3$, e $r = (1+\sqrt{1-4\lambda})/2 \approx 0.682$ 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.

Prova. Para $v = s$ o lema é evidente. Supõe que uma iteração modificando o fluxo f para f' diminuirá o valor de um vértice $v \in V \setminus \{s\}$, i.e., $\delta_f(v) > \delta_{f'}(v)$. Supõe ainda que v é o vértice de menor distância $\delta_{f'}(v)$ em $G_{f'}$ com essa característica. Seja $P = (s,\ldots,u,v)$ um caminho mais curto de s para v em $G_{f'}$. O valor de u não diminuiu nessa iteração (pela escolha de v), i.e., $\delta_f(u) \leq \delta_{f'}(u)$ (*).
O arco (u,v) não existe in G_f, senão a distância do v in G_f é no máximo a distância do v in $G_{f'}$: Supondo $(u,v) \in E(G_f)$ temos

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

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

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

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

Prova. (do teorema 14.2)
Chama um arco num caminho que aumenta o fluxo com capacidade igual ao gargalo *crítico*. Em cada iteração existe ao menos um arco crítico que desaparece do grafo residual. Provaremos que cada arco pode ser crítico no máximo $n/2-1$ vezes, que implica em no máximo $m(n/2-1) = O(mn)$ iterações.
No grafo G_f em que um arco $uv \in E$ é crítico pela primeira vez temos $\delta_f(u) = \delta_f(v) - 1$. O arco só aparece novamente no grafo residual caso alguma iteração diminui o fluxo em uv, i.e., aumenta o fluxo vu. Nessa iteração, com fluxo f', $\delta_{f'}(v) = \delta_{f'}(u) - 1$. Em soma temos

$$\begin{aligned}\delta_{f'}(u) &= \delta_{f'}(v) + 1 \\ &\geq \delta_f(v) + 1 && \text{pelo lema 14.9} \\ &= \delta_f(u) + 2,\end{aligned}$$

i.e., a distância do u entre dois instantes em que uv é crítico aumenta por pelo menos dois. Enquanto u é alcançável por s, a sua distância é no máximo $n-2$, porque o caminho não contém s nem t, e por isso a aresta uv pode ser crítico por no máximo $(n-2)/2 = n/2-1$ vezes. ■
Zadeh (1972) apresenta instâncias em que o algoritmo de Edmonds-Karp precisa $\Theta(n^3)$ iterações, logo o resultado do teorema 14.2 é o melhor possível para grafos densos.

14.1.3. O algoritmo "caminho mais gordo" ("fattest path")

Idéia (Edmonds e Karp 1972): usar o caminho de maior gargalo para aumentar o fluxo. (Exercício 14.1 pede provar que isso é possível com uma modificação do algoritmo de Dijkstra em tempo $O(n \log n + m)$.)

Lema 14.10
Um fluxo f pode ser decomposto em fluxos $f_1, \ldots, f_k$ com $k \leq m$ tal que o fluxo f_i é positivo somente num caminho p_i entre s e t.

Prova. Dado um fluxo f, encontra um caminho p de s para t usando somente arcos com fluxo positivo. Define um fluxo no caminho cujo valor é o valor do menor fluxo de algum arco em p. Subtraindo esse fluxo do fluxo f obtemos um novo fluxo reduzido. Repete até o valor do fluxo f é zero.
Em cada iteração pelo menos um arco com fluxo positivo tem fluxo zero depois da subtração do caminho p. Logo o algoritmo termina em no máximo m iterações. ■

Teorema 14.3
O caminho com o maior gargalo aumenta o fluxo por pelo menos OPT/m.

Prova. Considera o fluxo máximo. Pelo lema 14.10 existe uma decomposição do fluxo em no máximo m fluxos em caminhos s-t. Logo um dos caminhos possui valor pelo menos OPT/m. ■

Teorema 14.4
A complexidade do algoritmo de Ford-Fulkerson usando o caminho de maior gargalo é $O((n \log n + m) m \log C)$ para um limitante superior C do fluxo máximo.

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

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

A desigualdade é válida pelo teorema 14.3, considerando que o grafo residual possui no máximo $2m$ arcos. Logo

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

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

Para todo $i > 0$ e x temos

$$\left(1 + \frac{x}{i}\right)^i \leq e^x.$$

Logo, com $x = -i/2m$

$$\left(1 - \frac{i/2m}{i}\right)^i \leq e^{-i/2m}.$$

Corolário 14.1
Caso U é um limite superior da capacidade de um arco, o algoritmo termina em no máximo $O(m \log mU)$ passos.

14.1.4. O algoritmo push-relabel

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

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

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

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

Observe que o segundo parte da condição precisa ser satisfeita somente para arcos no grafo residual.

Observação 14.3
Pela condição (*), para um caminho $v_0, v_1, \ldots, v_k$ em G_f temos $p_{v_0} \leq p_{v_1} + 1 \leq p_{v_2} + 2 \leq \cdots \leq p_{v_k} + k$. ♢

Lema 14.11
A condição (*) pode ser satisfeita sse G_f não possui caminho s–t.

Prova. "⇒": Supõe existe um caminho s–t simples $v_0 = s, v_1, \ldots, v_k = t$. Pela observação (14.3)

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

uma contradição. "⇐": Sejam X os vértices alcançáveis em G_f a partir de s (incluindo s). Define $p_v = n$ para $v \in X$ e $p_v = 0$ para $v \in \overline{X}$. ■

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

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

```
push(u,v) :=
  { pré-condição: u é ativo }
  { pré-condição: p_v = p_u − 1 }
  { pré-condição: (u,v) ∈ A(G_f) }
  aumenta o fluxo em (u,v) por min{c_(u,v), e(u)}
  { atualiza G_f de acordo }
end

relabel(v) :=
  { pré-condição: v é ativo }
  { pré-condição: não existe (u,v) ∈ A(G_f) com p_v = p_u − 1 }
  p_v := p_v + 1
end
```

Observe que as duas operações mantém a condição (*).

Algoritmo 14.2 (Push-relabel)
Entrada Grafo $G = (V, A, c)$ com capacidades c_a no arcos.

Saída Um fluxo f.

```
p_s := n;  p_v := 0,        ∀v ∈ V \ {s}
f_a := c_a,  ∀a ∈ δ⁺(s) senão f_a := 0
while existe vértice ativo do
  escolhe o vértice ativo u de maior p_u
  repete até u é inativo
    if existe arco (u,v) ∈ G_f com p_v = p_u − 1 then
      push(u,v)
    else
      relabel(u)
    end if
  end
end while
return f
```

Lema 14.12
O algoritmo push-relabel é parcialmente correto (i.e. correto caso termina).

Prova. Ao terminar não existe vértice ativo. Logo f é um fluxo. Pelo lema 14.11 não existe caminho s–t em G_f. Logo pelo teorema 14.1 o fluxo é ótimo. ■

A terminação é garantido por

Teorema 14.5
O algoritmo push-relabel executa $O(n^3)$ operações push e $O(n^2)$ operações relabel.

Prova. Um vértice ativo v tem excesso de fluxo, logo existe um caminho v–s em G_f. Por (14.3) $p_v \leq p_s + (n-1) < 2n$, e o número de operações relabel é no $O(n^2)$. Supõe que uma operação push satura um arco $a = (u, v)$ (i.e. manda fluxo c_a). Para mandar fluxo novamente, temos que mandar primeiramente fluxo de v para u; isso só pode ser feito depois de pelo menos duas operações relabel em v. Logo o número de operações push saturantes é $O(mn)$. Para operações push não-saturantes, podemos observar que entre duas operações relabel temos no máximo n desses operações, porque cada uma torna o vértice de maior p_v inativo (talvez ativando vértices de menor potentical), logo tem no máximo $O(n^3)$ deles. ■

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

Uma busca linear na lista de vértices encontra o vértice de maior potencial. Entre dois operações relabel a busca pode continuar no último ponto e precisa tempo $O(n)$ em total, logo a busca custa no máximo $O(n^3)$ sobre toda execução do algoritmo. Para a operação push podemos simplesmente consultar a lista de candidatos. Para um push saturante, o candidato será removido. Isso custa $O(1)$. Finalmente no caso de um relabel temos que encontrar em $O(n)$ a nova posição do vértice na lista, e reconstruir a lista de candidatos, que também precisa tempo $O(n)$. Logo todas operações relabel custam não mais que $O(n^3)$.

14.1.5. Variações do problema

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

$$\begin{aligned} \textbf{maximiza} \quad & f(S) \\ \textbf{sujeito a} \quad & f(v) = 0 && \forall v \in V \setminus (S \cup T) \\ & f_e \leq c_e && \forall e \in E. \end{aligned} \tag{14.3}$$

Tabela 14.1.: Complexidade de diversos algoritmos de fluxo máximo (Schrijver 2003).

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

O problema (14.3) pode ser reduzido para um problema de fluxo máximo simples em $G' = (V', E', c')$ (veja Fig. 14.5(a)) com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ E' &= E \cup \{(s^*, s) \mid s \in S\} \cup \{(t, t^*) \mid t \in T\} \\ c'_e &= \begin{cases} c_e & e \in E \\ c(S, \bar{S}) & e = (s^*, s) \\ c(\bar{T}, T) & e = (t, t^*) \end{cases} \end{aligned} \tag{14.4}$$

Lema 14.13
Se f' é solução máxima de (14.4), $f = f'|_E$ é uma solução máxima de (14.3). Conversamente, se f é uma solução máxima de (14.3),

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

é uma solução máxima de (14.4).

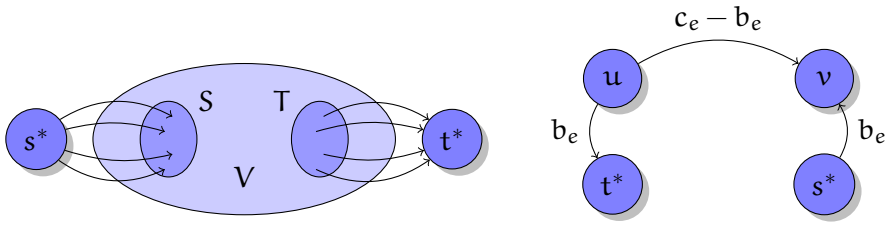


Figura 14.5.: Reduções entre variações do problema do fluxo máximo. Esquerda: Fontes e destinos múltiplos. Direita: Limite inferior e superior para a capacidade de arcos.

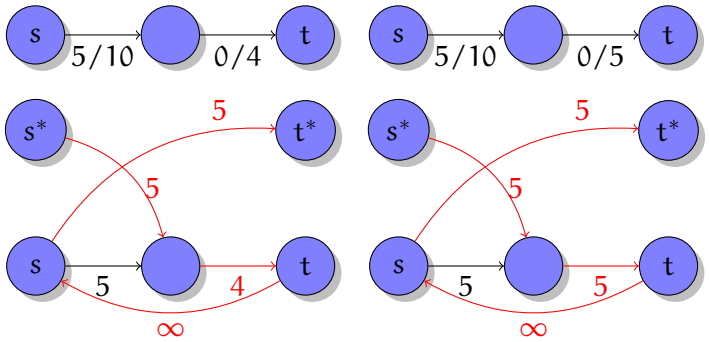


Figura 14.6.: Dois exemplos da transformação do lema 14.14. Esquerda: Grafo sem solução viável e grafo transformado com fluxo máximo 4. Direita: Grafo com solução viável e grafo transformado com fluxo máximo 5.

Prova. Supõe f é solução máxima de (14.3). Seja f' uma solução de (14.4) com valor $f'(s^*)$ maior. Então $f'|_E$ é um fluxo válido para (14.3) 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

$$\begin{aligned} \textbf{maximiza} \quad & f(s) \\ \textbf{sujeito a} \quad & f(v) = 0 && \forall v \in V \setminus \{s, t\} \\ & b_e \leq f_e \leq c_e && e \in E. \end{aligned} \tag{14.5}$$

O problema (14.5) pode ser reduzido para um problema de fluxo máximo simples em $G' = (V', E', c')$ (veja Fig. 14.5(b)) com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ E' &= E \cup \{(u, t^*) \mid (u, v) \in E\} \cup \{(s^*, v) \mid (u, v) \in E\} \cup \{(t^*, s^*)\} \\ c'_e &= \begin{cases} c_e - b_e & e \in E \\ \sum_{v \in N^+(u)} b_{(u,v)} & e = (u, t^*) \\ \sum_{u \in N^-(v)} b_{(u,v)} & e = (s^*, v) \\ \infty & e = (t, s) \end{cases} \end{aligned} \tag{14.6}$$

Lema 14.14
Problema (14.5) possui uma solução viável sse (14.6) possui uma solução máxima com todos arcos auxiliares $E' \setminus E$ saturados. Neste caso, se f é um fluxo máximo em (14.5),

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

é um fluxo máximo de (14.6) com arcos auxiliares saturados. Conversamente, se f' é um fluxo máximo para (14.6) com arcos auxiliares saturados, $f_e = f'_e + b_e$ é um fluxo máximo em (14.5).

Prova. (Exercício.) ■

Para obter um fluxo máximo de (14.5) podemos maximizar o fluxo a partir da solução viável obtida, com qualquer variante do algoritmo de Ford-Fulkerson. Uma alternativa para obter um fluxo máximo com limites inferiores nos arcos é primeiro mandar o limite inferior de cada arco, que torna o problema num problema de encontrar o fluxo s-t máximo num grafo com demandas.

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

$$\begin{aligned} \textbf{existe} \quad & f \\ \textbf{s.a} \quad & f(v) = -d_v && \forall v \in V \\ & f_e \leq c_e && e \in E. \end{aligned} \tag{14.7}$$

Evidentemente $\sum_{v \in V} d_v = 0$ é uma condição necessária (lema (14.1)). O problema (14.7) pode ser reduzido para um problema de fluxo máximo em $G' = (V', E')$ com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ E' &= E \cup \{(s^*, v) \mid v \in V, d_v < 0\} \cup \{(v, t^*) \mid v \in V, d_v > 0\} \\ c_e &= \begin{cases} c_e & e \in E \\ -d_v & e = (s^*, v) \\ d_v & e = (v, t^*) \end{cases} \end{aligned} \tag{14.8}$$

Lema 14.15
Problema (14.7) possui uma solução sse problema (14.8) 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

$$\begin{aligned} \textbf{existe} \quad & f \\ \textbf{s.a} \quad & f(v) = d_v && \forall v \in V \\ & b_e \leq f_e \leq c_e && e \in E. \end{aligned} \tag{14.9}$$

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

$$V' = V$$
$$E' = E \tag{14.10}$$
$$c_e = c_e - b_e$$
$$d'_v = d_v - \sum_{e \in \delta^-(v)} b_e + \sum_{e \in \delta^+(v)} b_e \tag{14.11}$$

Lema 14.16
O problema (14.9) possui solução sse problema (14.10) possui solução.

Prova. (Exercício.) ■

14.1.6. Aplicações

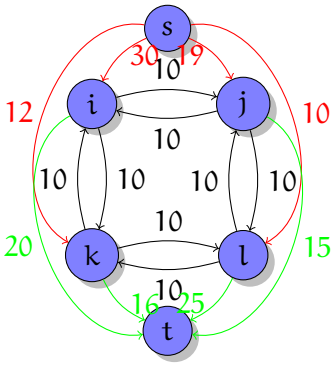
Projeto de pesquisa de opinião O objetivo é projetar uma pesquisa de opinião, com as restrições

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

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

Segmentação de imagens O objetivo é segmentar 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. 14.8 (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

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



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

Figura 14.7.: Exemplo da construção para uma imagem 2×2. Direita: Tabela com valores pele/não-pele. Esquerda: Grafo com penalidade fixa $p_{ij} = 10$.

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. 14.7).

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

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

Figura 14.8.: Segmentação de imagens com diferentes penalidades p. Acima: (a) Imagem original (b) Segmentação somente com probabilidades ($p = 0$) (c) $p = 1000$ (d) $p = 10000$. Abaixo: (a) Walter Gramatté, Selbstbildnis mit rotem Mond, 1926 (b) Segmentação com $p = 10000$. A probabilidade de um pixel representar pele foi determinado conforme Jones e Rehg (1998).

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.

O problema $P \mid \text{pmtn}, r_i \mid L_{max}$ Primeiramente resolveremos um problema mais simples: será que existe um sequenciamento tal que toda tarefa i executa dentro do seu intervalo $[r_i, d_i]$? Equivalentemente, será que existe uma solução com $L_{max} = 0$?
Seja $\{t_1, t_2, \ldots, t_k\} = \{r_1, r_2, \ldots r_n\} \cup \{d_1, d_2, \ldots, d_n\}$, com $t_1 \leq t_2 \leq \cdots \leq t_k$. (Observe que $k \leq 2n$, e $k < 2n$ no caso de tempos repetidos.) Podemos ver os t_i como eventos em que uma tarefa fica disponível ou tem que terminar o seu processamento. Os t_i definem $k-1$ intervalos $I_i = [t_i, t_{i+1}]$ para $i \in [k-1]$ com duração $S_i = t_{i+1} - t_i$ correspondente. Cada tarefa j pode ser executada no intervalo T_i caso $I_i \subseteq [r_i, d_i]$. Logo podemos modelar o problema via um grafo direcionado bipartido com vértices $T \dot{\cup} I$, sendo $T = [n]$ o conjunto de tarefas e $I = \{I_i \mid i \in [k-1]\}$ o conjunto de intervalos, e com arcos (j, i) caso tarefa j pode ser executada no intervalo i. Para completar o grafo adicionaremos um arco (s, j) de um vértice origem s para cada tarefa j, e um arco (i, t) de cada intervalo para um vértice destino t. Um fluxo nesse grafo representa tempo, e teremos capacidades p_j entre s e tarefa j, S_i entre tarefa j e intervalo i, e mS_i entre T_i e t, sendo mS_i o tempo total disponível durante o intervalo i. A figura 14.9 mostra a construção completa.
Logo $P \mid \text{pmtn}, r_i \mid L_{max}$ pode ser resolvido em tempo $O(mn \log \bar{L})$.

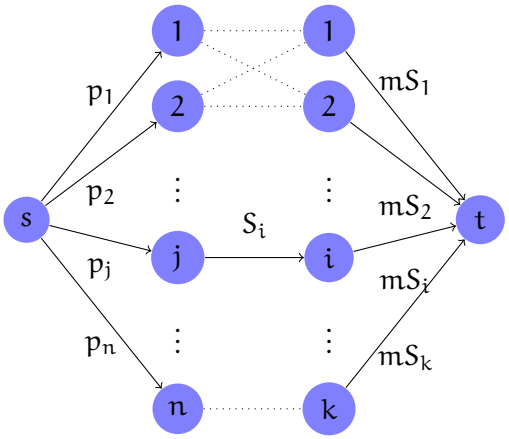


Figura 14.9.: Problema de fluxo para resolver a versão de decisão do problema $P \mid pmtn, r_i \mid L_{max}$.

Com essa abordagem podemos resolver o problema original por busca binária: para cada valor do L_{max} entre 0 e $\bar{L}$ testaremos se existe uma solução tal que cada tarefa executa no intervalo $[r_i, d_i + L_{max}]$. Um limite superior simples é $\bar{L} = \max_i r_i + \sum_i p_i - \min_i d_i$ executando todas tarefas após a liberação da última numa única máquina em ordem arbitrária.

Agendamento de projetos Suponha que temos n projetos, cada um com lucro $p_i \in \mathbb{Z}$, $i \in [n]$, e um grafo de dependências $G = ([n], A)$ sobre os projetos. Caso $(i, j) \in A$, a execução do projeto i é pré-requisito para a execução do projeto j. Um lucro pode ser negativo: neste caso tem uma perda efetiva. Este problema pode ser reduzido para um problema de fluxo máximo s-t: cria um grafo G' com vértices $\{s, t\} \cup [n]$ é

- uma aresta (s, v) para todo $v \in [n]$ com $p_v > 0$, com capacidade p_v,
- uma aresta (v, t) para todo $v \in [n]$ com $p_v < 0$, com capacidade $-p_v$, e
- uma aresta (u, v) para toda dependência $(v, u) \in A$, com capacidade ∞.

Lema 14.17
O valor de um corte $(X, \bar{X})$ em G' é mínimo, sse o lucro total dos projetos $S = X \setminus \{s\}$ é máximo. Além disso um corte mínimo em G' corresponde a uma seleção factível de projetos S.

Prova. Cada corte $(X, \bar{X})$ corresponde com uma seleção de projetos $S = X \setminus \{s\}$. Seja $\bar{S} = [n] \setminus S$. Uma seleção de projetos S é válida, caso para todo projeto $p \in S$, ela contém também todos projetos pré-requisitos de p. O corte correspondente não possui arcos com capacidade ∞. Como o valor do corte $(s, V \setminus \{s\})$ é $\sum_{i,j|p_{ij}>0} p_{ij}$ o corte mínimo é finito, e logo factível, porque não existe um arco entre um projeto selecionado e um projeto não selecionado. O valor de um corte factível é

$$c(X, \bar{X}) = \sum_{a \in F(X,\bar{X})} c_a = \sum_{p \in \bar{S}|p_{ij}>0} p_{ij} - \sum_{p \in S|p_{ij}<0} p_{ij}$$

e nos temos

$$\begin{aligned}\sum_{p \in [n]|p_{ij}>0} p_{ij} - c(X, \bar{X}) &= \sum_{p \in [n]|p_{ij}>0} p_{ij} - \sum_{p \in \bar{S}|p_{ij}>0} p_{ij} + \sum_{p \in S|p_{ij}<0} p_{ij} \\ &= \sum_{p \in S|p_{ij}>0} p_{ij} + \sum_{p \in S|p_{ij}<0} p_{ij} \\ &= \sum_{p \in S} p_{ij}\end{aligned}$$

i.e o lucro total da seleção S. Logo o lucro total é máximo sse o valor do corte é mínimo. ■

14.1.7. Outros problemas de fluxo

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

FLUXO DE MENOR CUSTO

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

Solução Um fluxo s-t f com valor v.

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

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

14.1.8. Exercícios

Exercício 14.1
Mostra como podemos modificar o algoritmo de Dijkstra para encontrar o caminho mais curto entre dois vértices num um grafo para encontrar o caminho com o maior gargalo entre dois vértices. (Dica: Enquanto o algoritmo de Dijkstra procura o caminho com a menor soma de distâncias, estamos procurando o caminho com o maior capacidade mínimo.)

14.2. Emparelhamentos

Dado um grafo não-direcionado $G = (V, A)$, um *emparelhamento* é uma seleção de arestas $M \subseteq A$ 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 $c : A \to \mathbb{Q}$, seja $c(M) = \sum_{e \in M} c_e$ o *valor* do emparelhamento M.

Emparelhamento máximo (EM)

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

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

Objetivo Maximiza $|M|$.

Emparelhamento de peso máximo (EPM)

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

Solução Um emparelhamento $M \subseteq A$.

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

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

Emparelhamento perfeito de peso mínimo (EPPM)

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

Solução Um emparelhamento perfeito $M \subseteq A$, 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 $G = (V, A, c)$ é equivalente com EPM em $-G := (V, A, -c)$ (por quê?). Até EPPM pode ser reduzido para EPM.

Teorema 14.6
EPM e EPPM são problemas equivalentes.

Prova. Seja $G = (V, A, c)$ 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

$$c'_a = \begin{cases} -c_a & \text{caso } a \in A \\ 0 & \text{caso contrário} \end{cases}.$$

Dado um emparelhamento M em G podemos definir um emparelhamento perfeito M' em G': M' inclui todas arestas em M. Além disso, um vértice em V não emparelhado em M será emparelhado com o novo vértice correspondente em V' com uma aresta de custo 0 em M'. Similarmente, os restantes vértices não emparelhados em V' são emparelhados em M' com arestas de custo 0 entre si. Pela construção, o valor de M' é $c'(M') = -c(M)$. Dado um emparelhamento M' em G' podemos obter um emparelhamento M em G com valor $-c(M')$ removendo as arestas que não pertencem a G. Portanto, um EPPM em G' é um EPM em G.
Conversamente, seja $G = (V, A, c)$ uma instância de EPPM. Define $C := 1 + \sum_{a \in A} |c_a|$, novos pesos $c'_e = C - c_e$ e um grafo $G' = (V, A, c')$. Para emparelhamentos M_1 e M_2 em G arbitrários temos

$$c(M_2) - c(M_1) \leq \sum_{\substack{a \in A \\ c_a > 0}} c_a - \sum_{\substack{a \in A \\ c_a < 0}} c_a = \sum_{a \in A} |c_a| < C.$$

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 $G = (V, A, c)$ é

$$\begin{aligned} \textbf{minimiza} \quad & \sum_{a \in A} c_a x_a & & (14.12)\\ \textbf{sujeito a} \quad & \sum_{u \in N(v)} x_{uv} = 1, & \forall v \in V & \\ & x_a \in \mathbb{B}. & & \end{aligned}$$

A formulação do problema do emparelhamento máximo é

$$\begin{aligned} \textbf{maximiza} \quad & \sum_{a \in A} c_a x_a & & (14.13)\\ \textbf{sujeito a} \quad & \sum_{u \in N(v)} x_{uv} \leq 1, & \forall v \in V & \\ & x_a \in \mathbb{B}. & & \end{aligned}$$

Observação 14.4
A matriz de coeficientes de (14.12) e (14.13) 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 14.5
O dual da relaxação linear de (14.12) é

$$\begin{aligned} \text{CIM: } \textbf{maximiza} \quad & \sum_{v \in V} y_v & & (14.14)\\ \textbf{sujeito a} \quad & y_u + y_v \leq c_{uv}, & \forall uv \in A & \\ & y_v \in \mathbb{R}. & & \end{aligned}$$

e o dual da relaxação linear de (14.13)

$$\begin{aligned} \text{MVC: } \textbf{minimiza} \quad & \sum_{v \in V} y_v & & (14.15)\\ \textbf{sujeito a} \quad & y_u + y_v \geq c_{uv}, & \forall uv \in A & \\ & y_v \in \mathbb{R}_+. & & \end{aligned}$$

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 de vértices mínima. Portanto, a observação 14.4 rende no caso não-ponderado:

Teorema 14.7 (Berge, 1951)
Em grafos bi-partidos o tamanho da menor cobertura de vértices é igual ao tamanho do emparelhamento máximo.

Proposição 14.1
Um subconjunto de vértices $I \subseteq V$ de um grafo não-direcionado $G = (V, A)$ é um conjunto independente sse $V \setminus I$ é um cobertura de vértices. Em particular um conjunto independente máximo I corresponde com uma cobertura de vértices mínima $V \setminus I$.

Prova. (Exercício 14.3.) ■ ◊

14.2.1. 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 14.8 (Sack e 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. Pela proposição 14.3 podemos obter uma cobertura mínima via um emparelhamento máximo, que é o complemento de um conjunto independente máximo. Podemos achar o emparelhamento em tempo $O(n^{5/2})$ usando o algoritmo de Hopcroft-Karp, porque o grafo de intersecção é bi-partido (por quê?).

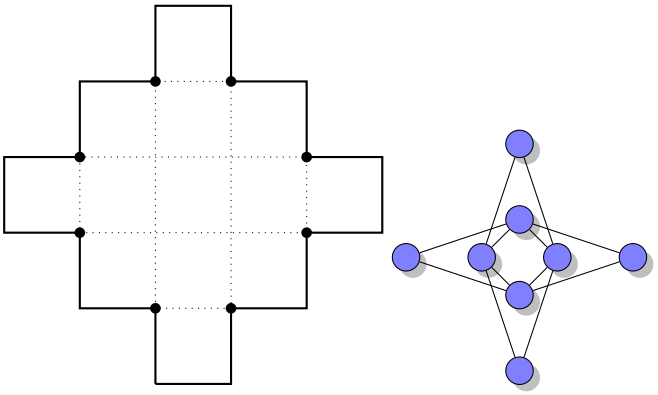


Figura 14.10.: Esquerda: Polígono ortogonal com $n = 8$ vértices de reflexo (pontos), $h = 0$ buracos. As cordas são pontilhadas. Direita: grafo de intersecção.

Problemas de agendamento O problema $1 \mid p_j = p \mid \sum w_j T_j$ é resolvido por um emparelhamento perfeito entre as tarefas e os intervalos de execução $[(i-1)p, ip]$, $i \in [n]$. Podemos resolver ainda $1 \mid p_j = 1, r_j \mid \sum w_j T_j$, observando que sempre existe uma solução com as tarefas executando nos intervalos $[t_i, t_i + 1]$, $i \in [n]$, definido por

$$t_0 = -\infty; \qquad t_i = \max\{t_{i-1} + 1; r_i\}$$

e supondo que $r_1 \leq \cdots \leq r_n$.

14.2.2. 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 14.9
Um EM em grafos bi-partidos pode ser obtido em tempo $O(mn)$.

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

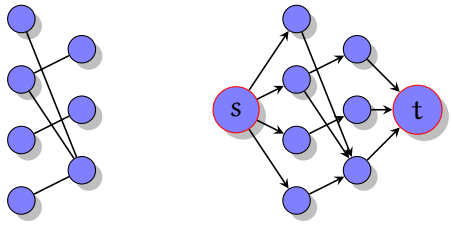


Figura 14.11.: Redução do problema de emparelhamento máximo para o problema do fluxo máximo

Teorema 14.10
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 (14.3)). 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-ponderada combinatorial Um caminho $P = v_1v_2v_3 \ldots v_k$ 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 14.6
Caso temos um caminho $P = v_1v_2v_3 \ldots v_{2k+1}$ 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 14.11 (Hopcroft e Karp (1973))
Seja M^* um emparelhamento máximo e M um emparelhamento arbitrário. O conjunto $M \oplus M^*$ contém pelo menos $k = |M^*| - |M|$ caminhos M-aumentantes disjuntos (de vértices). Um deles possui comprimento menor que $|V|/k - 1$.

Prova. Considere os componentes de G em relação aos arcos $M \oplus M^*$. Cada vértice possui no máximo grau 2. Portanto, os componentes são vértices livres, caminhos simples ou ciclos. Os caminhos e ciclos possuem alternadamente arestas de M e M^*, logo os ciclos tem comprimento par. Os caminhos de comprimento ímpar são ou M-aumentantes, porque para a solução ótima M^* não existem caminhos aumentantes. Ainda temos

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

e portanto $M \oplus M^*$ contém k arcos mais de M^* que de M. Isso mostra que existem pelo menos $|M^*| - |M|$ caminhos M-aumentantes, porque somente os caminhos de comprimento ímpar possuem exatamente um arco mais de M^*. Pelo menos um desses caminhos tem que ter um comprimento (em arcos) menor ou igual que $|V|/k - 1$, senão cada um possui pelo menos $|V|/k + 1$ vértices, i.e. eles contém em total mais que $|V|$ vértices. ■

Corolário 14.2 (Berge (1957))
Um emparelhamento é máximo sse não existe um caminho M-aumentante.

Rascunho de um algoritmo:

Algoritmo 14.3 (Emparelhamento máximo)
Entrada Grafo não-direcionado $G = (V, A)$.

Saída Um emparelhamento máximo M.

$M = \emptyset$
`while (existe um caminho M-aumentante P) do`
  $M := M \oplus P$
`end while`
`return` M

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

Observação 14.7
Um caminho M-aumentante começa num vértice livre em V_1 e termina num vértice livre em V_2. Idéia: Começa uma busca por largura com todos vértices livres em V_1. Segue alternadamente arcos livres em M para encontrar vizinhos em V_2 e arcos em M, para encontrar vizinhos em V_1. A busca pára ao encontrar um vértice livre em V_2 ou após de visitar todos os vértices. Ela tem complexidade $O(m + n)$. ◊

Teorema 14.12
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 14.8
O último teorema é o mesmo que teorema (14.9). ◊

Observação 14.9
Pelo teorema (14.11) sabemos que existem vários caminhos M-alternantes disjuntos (de vértices) e nos podemos aumentar M com todos eles em paralelo. Portanto, estruturamos o algoritmo em fases: cada fase procura um conjunto de caminhos aumentantes disjuntos e aplicá-los para obter um novo emparelhamento. Observe que pelo teorema (14.11) 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 maximal de caminhos M-alternantes disjuntos de menor comprimento.
Podemos encontrar um tal conjunto após uma busca em profundidade usando o DAG (grafo direcionado acíclico) definido pela busca por profundidade. (i) Escolhe um vértice livre em V_2. (ii) Segue os predecessores para achar um caminho aumentante. (iii) Coloca todos vértices em uma fila de deleção. (iv) Processa a fila de deleção: Até que a fila esteja 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 14.18
Em cada fase o comprimento de um caminho aumentante mínimo aumenta por pelo menos dois.

Lema 14.19
O algoritmo termina em no máximo $\sqrt{n}$ fases.

Teorema 14.13
O problema do emparelhamento máximo não-ponderado em grafos bi-partidos pode ser resolvido em tempo $O(m\sqrt{n})$.

Prova. Pelas lemas 14.18 e 14.19 e a observação que toda fase pode ser completada em $O(m)$. ■

Usaremos outro lema para provar os dois lemas acima.

Lema 14.20
Seja M um emparelhamento, P um caminho M-aumentante mínimo, e Q um caminho $M \oplus P$-aumentante. Então $|Q| \geq |P| + 2|P \cap Q|$. ($P \cap Q$ denota as arestas em comum entre P e Q.)

Prova. Caso P e Q não possuem vértices em comum, Q é M-aumentante, $P \cap Q = \emptyset$ e a desigualdade é conseqüência da minimalidade de P.
Caso contrário, P e Q possuem um vértice em comum, e logo também uma aresta, senão $M \oplus P \oplus Q$ possui um vértice de grau dois. $P \oplus Q$ consiste em dois caminhos, e eventualmente um coleção de ciclos. Os dois caminhos são M-aumentantes, pelas seguintes observações:

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

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

3. Nenhum outro vértice de $P \oplus Q$ é livre em relação a M: P só contém dois vértices livres e Q só contém dois vértices livres em $Q \setminus P$.

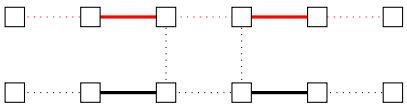
4. Temos dois caminhos M-aumentantes, começando com um vértice livre em Q e terminando com um vértice livre em P. O parte do caminho Q em $Q \setminus P$ é M-alternante, porque as arestas livres em $M \oplus P$ são exatamente as arestas livres em M. O caminho Q entra em P e sai de P com arestas livres,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

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

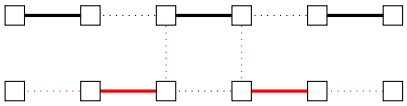
$$|Q| = |P \oplus Q| + 2|P \cap Q| - |P| \geq |P| + 2|P \cap Q|.$$

■

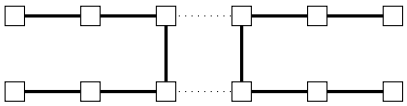
Prova. (do lema 14.18). Seja S o conjunto de caminhos M-aumentantes da fase anterior, e P um caminho aumentante. Caso P é disjunto de todos caminhos em S, ele deve ser mais comprido, porque S é um conjunto máximo de caminhos aumentantes. Caso P possui um vértice em comum com algum



(a) Grafo com emparelhamento M (em negrito) e um caminho M-aumentante P (em vermelho).



(b) O mesmo grafo com emparelhamento $M \oplus P$ (em negrito) e um caminho $M \oplus P$-aumentante Q (em vermelho).



(c) O conjunto de arestas $P \oplus Q$ (em negrito).

Figura 14.12.: Ilustração do lema 14.20.

caminho em S, ele possui também um arco em comum (por quê?) e podemos aplicar lema 14.20. ■

Prova. (do lema 14.19). Seja M^* um emparelhamento máximo e M o emparelhamento obtido após de $\sqrt{n}/2$ fases. O comprimento de qualquer caminho M-aumentante é no mínimo $\sqrt{n}$, pelo lema 14.18. Pelo teorema 14.11 existem pelo menos $|M^*| - |M|$ caminhos M-aumentantes disjuntos de vértices. Mas então $|M^*| - |M| \leq \sqrt{n}$, porque no caso contrário eles possuem mais que n vértices em total. Como o emparelhamento cresce pelo menos um em cada fase, o algoritmo executa no máximo mais $\sqrt{n}$ fases. Portanto, o número total de fases é no máximo $3/2\sqrt{n} = O(\sqrt{n})$. ■

O algoritmo de Hopcroft-Karp é o melhor algoritmo conhecido para encontrar emparelhamentos máximos em grafos bipartidos não-ponderados esparsos[1]. 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 $G = (V, A)$ bi-partido com partição $V_1 \dot\cup V_2$. Podemos representar um emparelhamento usando um vetor `mate`, que contém, para

[1] Feder e Motwani (1991) e Feder e Motwani (1995) propuseram um algoritmo em $O(\sqrt{n}m(2 - \log_n m))$ que é melhor em grafos densos.

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 14.9: ele consiste numa busca por largura até encontrar um ou mais caminhos M-alternantes mínimos e depois uma fase que extrai do DAG definido pela busca um conjunto máximo de caminhos disjuntos (de vértices).
A busca por largura começa com todos vértices livres em V_1. Usamos um vetor H para marcar os arcos que fazem parte do DAG definido pela busca por largura[2] e um vetor m para marcar os vértices visitados.

```
search_paths(M) :=
  for all v ∈ V do m_v := false

  U1 := {v ∈ V1 | v livre}
  for all u ∈ U1 do d_u := 0

  do
    { determina vizinhos em U2 via arestas livres}
    U2 := ∅
    for all u ∈ U1 do
      m_u := true
      for all uv ∈ A, uv ∉ M do
        if not m_v then
          d_v := d_u + 1
          U2 := U2 ∪ v
        end if
      end for
    end for

    { determina vizinhos em U1 via arestas emparelhadas }
    found := false        { pelo menos um caminho encontrado? }
    U1 := ∅
    for all u ∈ U2 do
      m_u := true
      if (u livre) then
        found := true
      else
        v := mate[u]
```

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

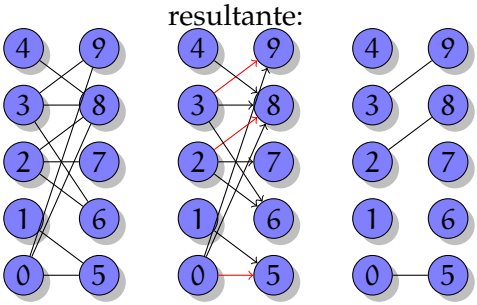
```
          if not m_v then
            d_v := d_u + 1
            U_1 := U_1 ∪ v
          end if
        end for
      end for
    while (not found)
end
```

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:

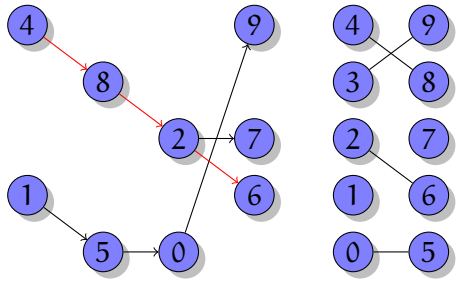
```
extract_paths() :=
  while existe vértice v livre em V_2 do
    aplica um busca em profundidade a partir de v em H
      (procurando um vértice livre em V_1)
    remove todos vértices visitados durante a busca
    caso um caminho alternante P foi encontrado: M := M ⊕ P
  end while
end
```

Exemplo 14.1
Segue um exemplo de aplicação do algoritmo de Hopcroft-Karp.

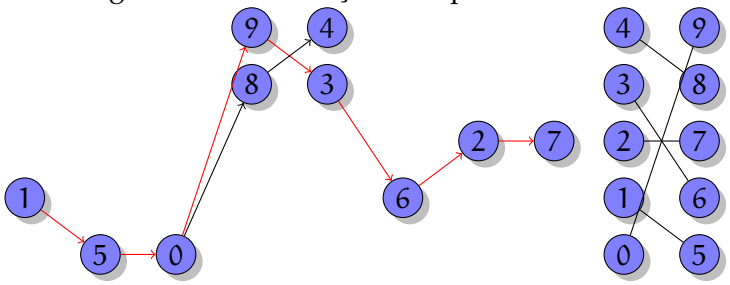
Grafo original, árvore Húngara primeira iteração e emparelhamento resultante:



Árvore Húngara segunda iteração e emparelhamento resultante:



Árvore Húngara terceira iteração e emparelhamento resultante:



◊

Emparelhamentos, coberturas e conjuntos independentes

Proposição 14.2
Seja $G = (S \dot\cup T, A)$ um grafo bipartido e $M \subseteq A$ um emparelhamento em G. Seja R o conjunto de todos vértices livres em S e todos vértices alcançáveis por uma busca na árvore Húngara (i.e. via arestas livres de S para T e arestas do emparelhamento de T para S). Então $(S \setminus R) \cup (T \cap R)$ é uma cobertura de vértices em G.

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

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

Proposição 14.3
Seja $G = (S \dot\cup T, A)$. Caso M é um emparelhamento máximo o conjunto $(S \setminus R) \cup (T \cap R)$ é uma cobertura mínima.

Prova. O tamanho que qualquer emparelhamento M é um limite inferior para o tamanho de qualquer cobertura, porque uma cobertura tem que conter pelo menos um vértice da cada aresta emparelhada. Logo é suficiente demonstrar que $|(S \setminus R) \cup (T \cap R)| = |M|$.

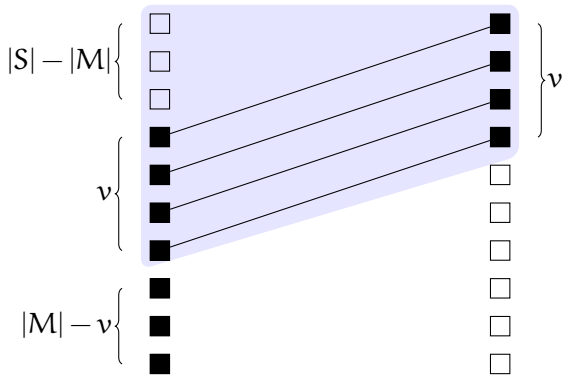


Figura 14.13.: Ilustração da prova da proposição 14.3.

Temos $(S \setminus R) \cup (T \cap R)| = |S \setminus R| + |T \cap R|$ porque S e T são disjuntos. Vamos demonstrar que $|T \cap R| = \nu$ implica $|S \setminus R| = |M| - \nu$.
Supõe $|T \cap R| = \nu$. Como M é máximo não existe caminho M-aumentante, e logo $T \cap R$ contém somente vértices emparelhados. Por isso o número de vértices emparelhados em $S \cap R$ também é ν. Além disso $S \cap R$ contém todos $|S| - |M|$ vértices livres em S. Logo $|S \setminus R| = |S| - (|S| - |M|) - \nu = |M| - \nu$. ■

Observação 14.10
O complemento $V \setminus C$ de uma cobertura C é um conjunto independente (por quê?). Logo um emparelhamento M que define um conjunto R de acordo com a proposição (14.2) corresponde com um conjunto independente $(S \cap R) \cup (T \setminus R)$, e caso M é máximo, o conjunto independente também. ◊

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

Observação 14.11
O caso ponderado pode ser restrito para emparelhamentos perfeitos: caso S e T possuem cardinalidade diferente, podemos adicionar vértices, e depois completar todo grafo com arestas de custo 0. O problema de encontrar um emparelhamento perfeito máximo (ou mínimo) em grafos ponderados é conhecido pelo nome "problema de alocação" (ingl. assignment problem). ◊

Observação 14.12
A redução do teorema 14.9 para um problema de fluxo máximo não se aplica no caso ponderado. Mas, com a simplificação da observação 14.11, podemos

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

O dual do problema 14.15 é a motivação para

Definição 14.3
Um *rotulamento* é uma atribuição $y : V \to \mathbb{R}_+$. Ele é *viável* caso $y_u + y_v \geq c_e$ para todas arestas $e = (u, v)$. (Um rotulamento viável é c-*cobertura de vértices*.) Uma aresta é *apertada* (ingl. tight) caso $y_u + y_v = c_e$. O subgrafo de arestas apertadas é $G_y = (V, A', c)$ com $A' = \{a \in A \mid a \text{ apertada em } y\}$.

Pelo teorema forte de dualidade e o fato que a relaxação linear dos sistemas acima possui uma solução integral (ver observação 14.4) temos

Teorema 14.14 (Egerváry (1931))
Para um grafo bi-partido $G = (S \dot\cup T, A, c)$ com pesos não-negativos $c : A \to \mathbb{Q}_+$ nas arestas, o maior peso de um emparelhamento perfeito é igual ao peso da menor c-cobertura de vértices.

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

---

Algoritmo 14.4 (Emparelhamento de peso máximo)
Entrada Um grafo não-direcionado ponderado $G = (V, E, c)$.

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

$M = \emptyset$
`while (existe um caminho M-aumentante P) do`
  `encontra o caminho M-aumentante mínimo P em` H_M
  `caso` $l(P) \geq 0$`: return M;`

```
    M := M ⊕ P
  end while
  return M
```

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

Observação 14.13
O grafo H_M de um emparelhamento extremo M não possui ciclo (par) negativo. Isso seria uma contradição com a maximalidade de M. Portanto podemos encontrar o caminho mínimo no passo 3 do algoritmo usando o algoritmo de Bellman-Ford em tempo $O(mn)$. Com isso a complexidade do algoritmo é $O(mn^2)$. ◊

Observação 14.14
Lembrando Bellman-Ford: Seja $d_k(t)$ a distância mínima entre s e t com um caminho usando no máximo k arcos ou ∞ caso tal caminho não existe. Temos

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

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

Teorema 14.15
Cada emparelhamento encontrado no algoritmo 14.4 é extremo.

Prova. Por indução sobre $|M|$. Para $M = \emptyset$ o teorema é correto. Seja M um emparelhamento extremo, P o caminho aumentante encontrado pelo algoritmo 14.4 e N um emparelhamento de tamanho $|M|+1$ arbitrário. Como $|N| > |M|$, $M \cup N$ contém uma componente que é um caminho Q M-aumentante (por um argumento similar com aquele da prova do teorema de Hopcroft-Karp 14.11). Sabemos $l(Q) \geq l(P)$ pela minimalidade de P. $N \oplus Q$ é um emparelhamento de cardinalidade $|M|$ (Q é um caminho com arestas em N e M com uma aresta em N a mais), logo $c(N \oplus Q) \leq c(M)$. Com isso temos

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

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

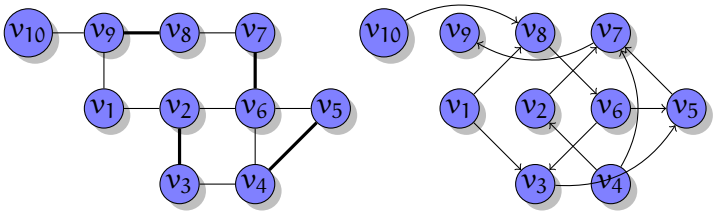


Figura 14.14.: Grafo com emparelhamento e grafo auxiliar.

Proposição 14.4
Caso não existe caminho M-aumentante com comprimento negativo no algoritmo 14.4, M é máximo.

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

Fato 14.1
É possível encontrar o caminho mínimo no passo 3 em tempo $O(m + n \log n)$ usando uma transformação para distâncias positivas e aplicando o algoritmo de Dijkstra. Com isso um algoritmo em tempo $O(n(m + n \log n))$ é possível.

14.2.3. Emparelhamentos em grafos não-bipartidos

O caso não-ponderado Dado um grafo não-direcionado $G = (V, E)$ e um emparelhamento M, podemos simplificar a árvore húngara para um grafo direcionado $D = (V, A)$ com $A = \{(u, v) \mid \exists x \in V : ux \in E, xv \in M\}$. Qualquer passeio M-alternante entre dois vértices livres em G corresponde com um caminho M-alternante em D.
O problema no caso não-bipartido são laços ímpares. No caso bi-partido, todo laço é par e pode ser eliminado sem consequências: de fato o caminho M-alternante mais curto não possui laço. No caso não bi-partido não todo caminho no grafo auxiliar corresponde com um caminho M-alternante no grafo original. O caminho $v_1v_3v_5v_7v_9$ corresponde com o caminho M-alternante $v_1v_2v_3v_4v_5v_6v_7v_8v_9v_{10}$, mas o caminho $v_1v_8c_6v_5v_7v_9$ que corresponde com o passeio $v_1v_9v_8v_7v_6v_4v_5v_6v_7v_8v_0v_{10}$ não é um caminho M-alternante que aumento o emparelhamento. O problema é que o laço ímpar $v_6v_4v_5v_6$ não pode ser eliminado sem consequências.

Tabela 14.2.: Resumo emparelhamentos

	Cardinalidade	Ponderado
Bi-partido	$O(n\sqrt{mn/\log n})$ (Alt et al. 1991) $O(m\sqrt{n}\frac{\log(n^2/m)}{\log n})$ (Feder e 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 e Karzanov 2004; Fremuth-Paeger e Jungnickel 2003)	$O(n^3)$ (Edmonds 1965) $O(mn + n^2 \log n)$ (Gabow 1990)

14.2.4. Notas

Duan et al. (2011) apresentam técnicas de aproximação para emparelhamentos.

14.2.5. Exercícios

Exercício 14.2
É possível somar uma constante $c \in \mathbb{R}$ para todos custos de uma instância do EPM ou EPPM, mantendo a otimalidade da solução?

Exercício 14.3
Prove a proposição 14.1.

15. Algoritmos de aproximação

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

15.1. Problemas, classes e reduções

Definição 15.1
Um *problema de otimização* $\Pi = (\mathcal{P}, \varphi, \mathrm{opt})$ é uma relação binária $\mathcal{P} \subseteq I \times S$ com instâncias $x \in I$ e soluções $y \in S$, junto com

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

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

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

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

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

Convenção 15.1
Escrevemos um problema de otimização na forma

NOME

Instância x

Solução y

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

Com um dado problema de otimização correspondem três problemas:

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

Definição 15.2
Uma relação binária R é *polinomialmente limitada* se

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

Definição 15.3 (Classes de complexidade)
A classe PO consiste dos problemas de otimização tal que existe um algoritmo polinomial A com $\varphi(x, A(x)) = \mathrm{OPT}(x)$ para $x \in I$.
A classe NPO consiste dos problemas de otimização tal que

(i) As instâncias $x \in I$ são reconhecíveis em tempo polinomial.

(ii) A relação $\mathcal{P}$ é polinomialmente limitada.

(iii) Para y arbitrário, polinomialmente limitado: $(x, y) \in \mathcal{P}$ é decidível em tempo polinomial.

(iv) φ é computável em tempo polinomial.

Definição 15.4
Uma *redução preservando a aproximação* entre dois problemas de minimização Π_1 e Π_2 consiste num 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

$$\mathrm{OPT}_{\Pi_2}(x_2) \leq \mathrm{OPT}_{\Pi_1}(x_1) \tag{15.1}$$

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

$$\varphi_{\Pi_1}(x_1, y_1) \leq \varphi_{\Pi_2}(x_2, y_2) \tag{15.2}$$

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

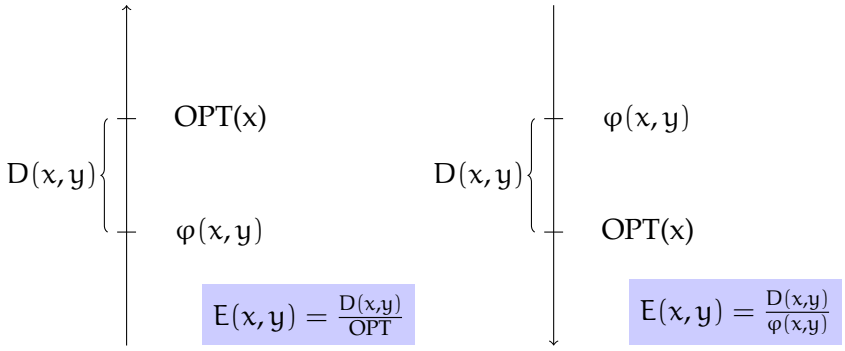
$$\varphi_{\Pi_1}(x_1, y_1) \leq \varphi_{\Pi_2}(x_2, y_2) \leq \alpha \mathrm{OPT}_{\Pi_2}(x_2) \leq \alpha \mathrm{OPT}_{\Pi_1}(x_1).$$

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

15.2. Medidas de qualidade

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

Aproximação relativa



Exemplo 15.1
Coloração de grafos planares e a problema de determinar a árvore geradora e a árvore Steiner de grau mínimo (Fürer e Raghavachari 1994) permitem uma aproximação absoluta, mas não o problema da mochila.
Os problemas da mochila e do caixeiro viajante métrico permitem uma aproximação absoluta constante, mas não o problema do caixeiro viajante. ♢

15.3. Técnicas de aproximação

15.3.1. Algoritmos gulosos

Cobertura de vértices

Algoritmo 15.1 (Cobertura de vértices)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura de vértices $C \subseteq V$.

```
VC-GV(G) :=
  (C,G) := Reduz(G)
  if V = ∅ then
    return C
  else
    escolhe v ∈ V : deg(v) = Δ(G) { grau máximo }
    return C ∪ {v} ∪ VC-GV(G − v)
  end if
```

Proposição 15.1
O algoritmo VC-GV é uma $O(\log |V|)$-aproximação.
Prova. Seja G_i o grafo depois da iteração i e C^* uma cobertura ótima, i.e., $|C^*| = \mathrm{OPT}(G)$.
A cobertura ótima C^* é uma cobertura para G_i também. Logo, a soma dos graus dos vértices em C^* (contando somente arestas em G_i!) ultrapassa o número de arestas em G_i

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

e o grau médio dos vértices em G_i satisfaz

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

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

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

Com isso podemos estimar

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

ou

$$\sum_{0 \le i < \mathrm{OPT}} \Delta(G_i) \ge \|G\|/2,$$

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

$$\mathrm{OPT}\,\lceil \lg \|G\| \rceil \le \mathrm{OPT}\,\lceil 2 \log |G| \rceil = O(\mathrm{OPT} \log |G|)$$

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

Algoritmo 15.2 (Cobertura de vértices)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Um cobertura de vértices $C \subseteq V$.

```
VC-GE(G) :=
  (C, G) := Reduz(G)
  if E = ∅ then
    return C
  else
    escolhe e = {u, v} ∈ E
    return C ∪ {u, v} ∪ VC-GE(G − {u, v})
  end if
```

Proposição 15.2
Algoritmo VC-GE é uma 2-aproximação para VC.

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

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

■

Algoritmo 15.3 (Cobertura de vértices)
Entrada Grafo não-direcionado $G = (V, E)$.

Saída Cobertura de vértices $C \subseteq V$.

```
VC-B(G) :=
  (C, G) := Reduz(G)
```

```
if V = ∅ then
  return C
else
  escolhe v ∈ V : deg(v) = Δ(G) { grau máximo }
  C1 := C ∪ {v} ∪ VC-B(G − v)
  C2 := C ∪ N(v) ∪ VC-B(G − v − N(v))
  if |C1| < |C2| then
    return C1
  else
    return C2
  end if
end if
```

Problema da mochila

KNAPSACK

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

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

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

Observação: O problema da mochila é NP-completo.

Como aproximar?

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

Abordagem

```
K−G(v_i, t_i) :=
  ordene os itens tal que v_i/t_i ≥ v_j/t_j, ∀i < j.
  for i ∈ X do
    if t_i < M then
      S := S ∪ {i}
      M := M − t_i
    end if
```

```
  end for
  return S
```

Aproximação boa?

- Considere

$$v_1 = 1, \ldots, v_{n-1} = 1, v_n = M - 1$$
$$t_1 = 1, \ldots, t_{n-1} = 1, t_n = M = kn \qquad k \in \mathbb{N} \text{ arbitrário}$$

- Então:

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

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

- Taxa de aproximação:

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

- K-G não possui taxa de aproximação fixa!

- Problema: Não escolhemos o item com o maior valor.

Tentativa 2: Modificação

```
K-G'(v_i, t_i) :=
  S_1 := K-G(v_i, t_i)
  v_1 := Σ_{i∈S_1} v_i
  S_2 := {argmax_i v_i}
  v_2 := Σ_{i∈S_2} v_i
  if v_1 > v_2 then
    return S_1
  else
    return S_2
  end if
```

Aproximação boa?

- O algoritmo melhorou?
- Surpresa
  Proposição 15.3
  K-G′ é uma 2-aproximação, i.e. $\mathrm{OPT}(x) < 2\varphi_{\text{K-G}'}(x)$.

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

$$\bar{v_j} = \sum_{1 \le i < j} v_i \le \varphi_{\text{K-G}}(x) \tag{15.3}$$

$$\bar{t_j} = \sum_{1 \le i < j} t_i \le M \tag{15.4}$$

Afirmação: $\mathrm{OPT}(x) < \bar{v_j} + v_j$. Nesse caso

(a) Seja $v_j \le \bar{v_j}$.

$$\mathrm{OPT}(x) < \bar{v_j} + v_j \le 2\bar{v_j} \le 2\varphi_{\text{K-G}}(x) \le 2\varphi_{\text{K-G}'}$$

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

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

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

$$\mathrm{OPT}(x) \le \bar{v_j} + (M - \bar{t_j})\frac{v_j}{t_j} < \bar{v_j} + v_j.$$

■

15.3.2. Aproximações com randomização

Randomização

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

Randomização: Exemplo

SATISFATIBILIDADE MÁXIMA, MAXIMUM SAT

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

Solução Uma atribuição de valores de verdade $a : V \to \mathbb{B}$.

Objetivo Maximiza o número de cláusulas satisfeitas

$$|\{C_i \mid [\![C_i]\!]_a = 1\}|\,.$$

Nossa solução

```
SAT−R(φ) :=
  seja φ = φ(v1,...,vk)
  for all i ∈ [1,k] do
    escolhe vi = 1 com probabilidade 1/2
  end for
```

Observação 15.1
A quantidade $[\![C]\!]_a$ é o valor da cláusula C na atribuição a. ◊

Aproximação?

- Surpresa: Algoritmo é 2-aproximação.

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

$$E[T] = E[\sum_{i\in[n]} [\![C_i]\!]] = \sum_{i\in[n]} E[[\![C_i]\!]] \geq n/2 \geq OPT/2$$

pela linearidade do valor esperado. ■

Outro exemplo

Cobertura de vértices guloso e randomizado.

```
VC−RG(G) :=
  seja w̄ := Σ_{v∈V} deg(v)
  C := ∅
  while E ≠ ∅ do
    escolhe v ∈ V com probabilidade deg(v)/w̄
    C := C ∪ {v}
    G := G − v
  end while
  return C ∪ V
```

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

15.3.3. Programação linear

Técnicas de programação linear são frequentemente usadas em algoritmo de aproximação. Entre eles são o *arredondamento randomizado* e *algoritmos primais-duais*.

Exemplo 15.2 (Arredondamento para cobertura por conjuntos)
Considere o problema de cobertura por conjuntos

$$\begin{aligned} \textbf{minimiza} \quad & \sum_{i\in[n]} w_i x_i, && && (15.5)\\ \textbf{sujeito a} \quad & \sum_{i\in[n]|u\in C_i} x_i \geq 1, && \forall u \in U,\\ & x_i \in \{0,1\}, && \forall i \in [n]. \end{aligned}$$

Seja f_e a frequência de um elemento e, i.e. o número de conjuntos que contém e e f a maior frequência. Um algoritmo de arredondamento simples é dado por

Teorema 15.1
A seleção dos conjuntos com $x_i \geq 1/f$ na relaxação linear de (15.5) é uma f-aproximação do problema de cobertura de conjuntos.

Prova. Como $|\{i \in [n] \mid u \in C_i\}| \leq f$, temos $x_i \geq 1/f$ em média sobre esse conjunto. Logo existe, para cada $u \in U$ um conjunto com $x_i \geq 1/f$ que cobre u e a seleção é uma solução válida. O arrendondamento aumenta o custo por no máximo um fator f, logo temos uma f-aproximação. ■ ◊

15.4. Esquemas de aproximação

Novas considerações

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

Um exemplo: Mochila máxima (Knapsack)

- Problema da mochila (veja página 322):
- Algoritmo MM-PD com programação dinâmica (pág. 112): tempo $O(n \sum_i v_i)$.
- Desvantagem: Pseudo-polinomial.

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

```
MM−PTAS(I,r) :=
  v_max := max_i{v_i}
  t := ⌊log (r−1)/r · v_max/n⌋
  v'_i := ⌊v_i/2^t⌋ para i = 1,...,n
  Define a nova instância I' = ({v'_i},{t_i})
  return MM-PD(I')
```

Teorema 15.2
MM-PTAS é uma r-aproximação em tempo $O(rn^3/(r-1))$.

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

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

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

$$\begin{aligned}
\varphi_{\text{MM-PTAS}}(I,S) = \sum_{i\in S} v_i &\geq \sum_{i\in S} 2^t \left\lfloor v_i/2^t \right\rfloor && \text{definição de } \lfloor\cdot\rfloor \\
&\geq \sum_{i\in S^*} 2^t \left\lfloor v_i/2^t \right\rfloor && \text{otimalidade de MM-PD sobre } v'_i \\
&\geq \sum_{i\in S^*} v_i - 2^t && \text{(A.10)} \\
&= \left(\sum_{i\in S^*} v_i\right) - 2^t|S^*| \\
&\geq \text{OPT}(I) - 2^t n
\end{aligned}$$

Portanto

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

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

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

■

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

- o problema do caixeiro viajante euclidiano com complexidade $O(n^{3000/\epsilon})$ (Arora, 1996);
- o problema do mochila múltiplo com complexidade $O(n^{12(\log 1/\epsilon)/e^8})$ (Chekuri, Kanna, 2000);
- o problema do conjunto independente máximo em grafos com complexidade $O(n^{(4/\pi)(1/\epsilon^2+1)^2(1/\epsilon^2+2)^2})$ (Erlebach, 2001).

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

15.5. Aproximando o problema da árvore de Steiner mínima

Seja $G = (V, A)$ um grafo completo, não-direcionado com custos $c_a \geq 0$ nos arcos. O problema da árvore Steiner mínima (ASM) consiste em achar o subgrafo conexo mínimo que inclui um dado conjunto de *vértices necessários* ou *terminais* $R \subseteq V$. Esse subgrafo sempre é uma árvore (ex. 15.1). 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 15.2
ASM é NP-completo. Para um conjunto fixo de vértices Steiner $V' \subseteq V \setminus R$, a melhor solução é a árvore geradora mínima sobre $R \cup V'$. Portanto a dificuldade é a seleção dos vértices Steiner da solução ótima. ♢

Definição 15.5
Os custos são *métricos* se eles satisfazem a desigualdade triangular, i.e.

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

para qualquer tripla de vértices i, j, k.

Teorema 15.3
Existe uma redução preservando a aproximação de ASM para a versão métrica do problema.

Prova. O fecho métrico de $G = (V, A)$ é um grafo G' completo sobre vértices e com custos $c'_{ij} := d_{ij}$, sendo d_{ij} o comprimento do menor caminho entre i e j em G. Evidentemente $c'_{ij} \leq c_{ij}$ e portanto (15.1) é satisfeita. Para ver que (15.2) 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 15.4
O AGM sobre R é uma 2-aproximação para o problema do ASM.

Prova. Considere a solução ótima S^* de ASM. Duplica todas arestas[1] tal que todo vértice possui grau par. Encontra um ciclo Euleriano nesse grafo.

[1] Isso transforma G num multigrafo.

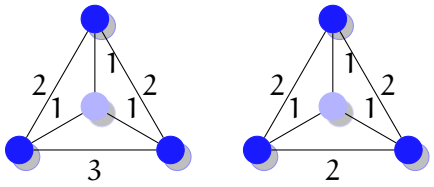


Figura 15.1.: Grafo com fecho métrico.

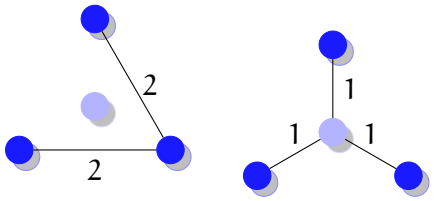


Figura 15.2.: AGM sobre R e melhor solução. ●: vértice em R, ●: vértice Steiner.

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. ■

15.6. Aproximando o PCV

Teorema 15.5
Para qualquer função $\alpha(n)$ computável em tempo polinomial 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.
2. Duplica todas arestas de A.
3. Acha um ciclo Euleriano nesse grafo.
4. Remove vértices duplicados.

Teorema 15.6
O algoritmo acima define uma 2-aproximação.

Prova. A melhor solução do PCV menos uma aresta é uma árvore geradora de G. Portanto $c(A) \leq \text{OPT}$. A solução S obtida pelo algoritmo acima satisfaz $c(S) \leq 2c(A)$ e portanto $c(S) \leq 2\text{OPT}$, pelo mesmo argumento da prova do teorema 15.4. ■

O fator 2 dessa aproximação é resultado do passo 2 que duplica todas arestas para garantir a existência de um ciclo Euleriano. Isso pode ser garantido mais barato: A AGM A possui um número par de vértices com grau ímpar (ver exercício 15.2), e portanto podemos calcular um emparelhamento perfeito mínimo E entre esse vértices. O grafo com arestas $A \cup E$ possui somente vértices com grau par e portanto podemos aplicar os restantes passos nesse grafo.

Teorema 15.7 (Cristofides)
A algoritmo usando um emparelhamento perfeito mínimo no passo 2 é uma $3/2$-aproximação.

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

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

e portanto

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

■

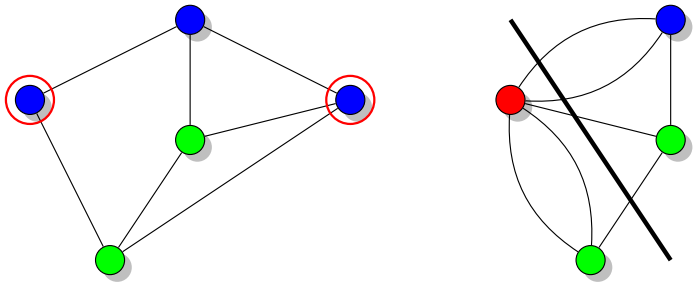


Figura 15.3.: Identificação de dois terminais e um corte no grafo reduzido. Vértices em verde, terminais em azul. O grafo reduzido possui múltiplas arestas entre vértices.

15.7. Aproximando problemas de cortes

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

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

Fato 15.1
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 (Goldschmidt e Hochbaum 1988).

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 15.4 (CI)
Entrada Grafo $G = (V, A, c)$ e terminais $s_1, \ldots, s_k$.

Saída Um corte múltiplo que separa os s_i.

```
Para cada i ∈ [1,k]: Calcula o corte isolante C_i de s_i.
Remove o maior desses cortes e retorne a união dos
   restantes.
```

Teorema 15.8
Algoritmo 15.4 é uma $2 - 2/k$-aproximação.

Prova. Considere o corte mínimo C^*. De acordo com a Fig. 15.4 ele pode ser representado pela união de k cortes que separam os k componentes individualmente:

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

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

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

e ainda $w(C_i) \leq w(C_i^*)$ para os cortes C_i do algoritmo 15.4, porque usamos o corte isolante mínimo de cada componente. Logo, para o corte C retornado pelo algoritmo temos

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

■

A análise do algoritmo é ótimo, como o exemplo da Fig. 15.5 mostra. O menor corte que separa s_i tem peso $2 - \epsilon$, portanto o algoritmo retorne um corte de peso $(2 - \epsilon)k - (2 - \epsilon) = (k - 1)(2 - \epsilon)$, enquanto o menor corte que separa todos terminais é o ciclo interno de peso k.

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

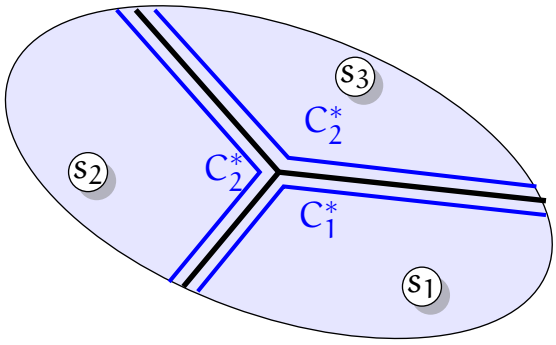


Figura 15.4.: Corte múltiplo e decomposição em cortes isolantes.

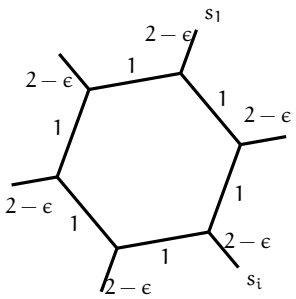


Figura 15.5.: Exemplo de um grafo em que o algoritmo 15.4 retorna uma $2 - 2/k$-aproximação.

Fato 15.2
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 determinando $n-1$ cortes s–t mínimos:

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

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

Algoritmo 15.5 (KCM)
Entrada Grafo $G = (V, A, c)$.

Saida Um k-corte.

```
Calcula uma AGH T em G.
Forma a união dos k−1 cortes mais leves definidos
   por k−1 arestas em T.
```

Teorema 15.9
Algoritmo 15.5 é uma $2 - 2/k$-aproximação.

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

Removendo C^* de G gera componentes $V_1, \ldots, V_k$: Define um grafo sobre esses componentes contraindo os 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

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

■

15.8. Aproximando empacotamento unidimensional

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

EMPACOTAMENTO UNIDIMENSIONAL (MIN-EU) (BIN PACKING)

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

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

Objetivo Minimiza o número de partes ("contêineres") m.

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

Fato 15.3
EU é fortemente NP-completo.

Proposição 15.4
Para um tamanho S fixo EU pode ser resolvido em tempo $O(n^{S^S})$.

Prova. Podemos supor, sem perda de generalidade, que os itens possuem tamanhos $1, 2, \ldots, S-1$. Um padrão de alocação de um contêiner pode ser descrito por uma tupla $(t_1, \ldots, t_{S-1})$ sendo t_i o número de itens de tamanho i. Seja T o conjunto de todos padrões que cabem num contêiner. Como $0 \leq t_i \leq S$ o número total de padrões T é menor que $(S+1)^{S-1} = O(S^S)$. Uma ocupação de m contêineres pode ser descrito por uma tupla $(n_1, \ldots, n_T)$ com n_i sendo o número de contêineres que usam padrão i. O número de contêineres é no máximo n, logo $0 \leq n_i \leq n$ e o número de alocações diferentes é no máximo $(n+1)^T = O(n^T)$. Logo podemos enumerar todas possibilidades em tempo polinomial. ■

Proposição 15.5
Para um m fixo, EU pode ser resolvido em tempo pseudo-polinomial.

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

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

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

Observação 15.3
Com um fator adicional de $O(\log m)$ podemos resolver também MIN-EU, procurando o menor i tal que $B(\underbrace{S, \ldots, S}_{i \text{ vezes}}, 0, \ldots, 0, n)$ é verdadeiro. ◊

A proposição 15.4 pode ser melhorada usando programação dinâmica.

Proposição 15.6
Para um número fixo k de tamanhos diferentes, min-EU pode ser resolvido em tempo $O(n^{2k})$.

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

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

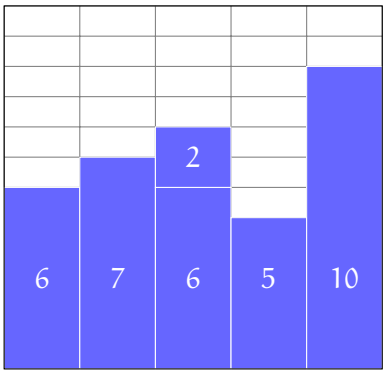
[2]Observe que a disjunção vazia é falsa.

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

Corolário 15.1
Para um tamanho S fixo min-EU pode ser resolvido em tempo $O(n^{2S})$.

Abordagem prática?

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



Aproximação?

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

Prova. Seja B o número de contêineres usadas, $V = \sum_{i \in [n]} s_i$. Como dois contêineres consecutivos contém uma soma > 1, temos $\lfloor B/2 \rfloor < V$ e com $B/2 - 1/2 \leq \lfloor B/2 \rfloor$ ainda $B - 1 < 2V$ ou $B \leq 2V$. Mas precisamos pelo menos $\lceil V \rceil$ contêineres, logo $\lceil V \rceil \leq OPT(x)$. Portanto, $\varphi_{PrC}(x) \leq 2V \leq 2\lceil V \rceil \leq 2OPT(x)$. ■

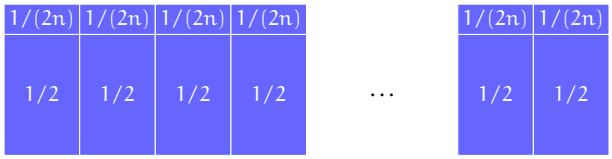
Aproximação melhor?

- Isso é a melhor estimativa possível para este algoritmo!

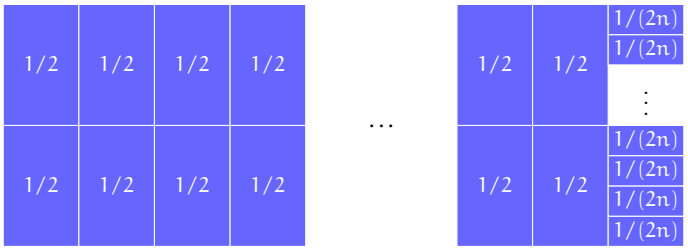
- Considere os $4n$ itens

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

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



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



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

Melhores estratégias

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

$$\varphi_{\mathrm{PiC}}(x) \leq \lceil 1.7\mathrm{OPT}(x) \rceil$$
$$\varphi_{\mathrm{PiCD}}(x) \leq 1.5\mathrm{OPT}(x) + 1$$

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

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

PiCD primeiro vai abrir $|A|$ contêineres com os itens do tipo A e depois $|B|$ contêineres com os itens do tipo B. Temos que analisar o que acontece com os itens em C e D.
Supondo que um contêiner contém somente itens do tipo D, os outros contêineres tem espaço livre menos que $1/3$, senão seria possível distribuir os itens do tipo D para outros contêineres. Portanto, nesse caso

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

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

1) O número de contêineres sem itens tipo D é o mesmo (eles são os últimos distribuídos em não abrem um novo contêiner). Logo é suficiente mostrar
$$\varphi_{\mathrm{PiCD}}(x \setminus D) = \mathrm{OPT}(x \setminus D).$$

2) Os itens tipo A não importam: Sem itens D, nenhum outro item cabe junto com um item do tipo A. Logo:
$$\varphi_{\mathrm{PiCD}}(x \setminus D) = |A| + \varphi_{\mathrm{PiCD}}(x \setminus (A \cup D)).$$

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

■

Garantia ou aproximação melhor?

- Johnson (1973, Tese de doutorado)
$$\varphi_{\mathrm{PiCD}}(x) \leq 11/9\,\mathrm{OPT}(x) + 4$$
- Baker (1985)
$$\varphi_{\mathrm{PiCD}}(x) \leq 11/9\,\mathrm{OPT}(x) + 3$$
- Uma variante de PiCD (Johnson e Garey 1985):
$$\varphi_{\mathrm{PiCDM}}(x) \leq 71/60\,\mathrm{OPT}(x) + 31/6$$

15.8.1. Um esquema de aproximação assintótico para min-EU

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

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

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

Lema 15.1
Para uma instância I e a instância R arredondada temos

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

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

Ideia 2: Descartando itens menores

Lema 15.2
Supõe temos temos um empacotamento para itens de tamanho maior que s_0 em B contêineres. Então existe um empacotamento de todos itens com no máximo

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

contêineres.

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

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

■

Juntando as ideias

Teorema 15.10
Para $\epsilon \in (0,1]$ podemos encontrar um empacotamento usando no máximo $(1+\epsilon)\mathrm{OPT}(I)+1$ contêineres em tempo $O(n^{16/\epsilon^2})$.

Prova. O algoritmo tem dois passos:

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

2. Empacota os itens menores depois.

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

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

grupos. (A primeira desigualdade usa $\lfloor x \rfloor \geq x/2$ para $x \geq 1$. Podemos supor que $n'\epsilon^2/4 \geq 1$, i.e. $n' \geq 4/\epsilon^2$. Caso contrário podemos empacotar os itens em tempo constante usando a proposição 15.6.)
Arredondando essa instância de acordo com lema 15.1 podemos obter uma solução em tempo $O(n^{16/\epsilon^2})$ pela proposição 15.6. Sabemos que $\mathrm{OPT}(I') \geq n' \lceil \epsilon/2\, S \rceil / S \geq n'\epsilon/2$. Logo temos uma solução com no máximo

$$\mathrm{OPT}(I') + \left\lfloor n\epsilon^2/4 \right\rfloor \leq \mathrm{OPT}(I') + n'\epsilon^2/4 \leq (1+\epsilon/2)\mathrm{OPT}(I') \leq (1+\epsilon/2)\mathrm{OPT}(I)$$

contêineres.
O segundo passo, pelo lema 15.2, produz um empacotamento com no máximo

$$\max\left\{(1+\epsilon/2)\mathrm{OPT}(I), \sum_{i\in[n]} s_i/(S-s_0)+1\right\}$$

contêineres, mas

$$\frac{\sum_{i\in[n]} s_i}{S-s_0} \leq \frac{\sum_{i\in[n]} s_i}{S(1-\epsilon/2)} \leq \frac{\mathrm{OPT}(I)}{1-\epsilon/2} \leq (1+\epsilon)\mathrm{OPT}(I).$$

■

15.9. Aproximando problemas de sequênciamento

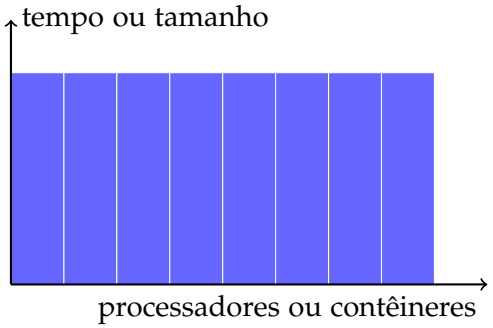
Problemas de sequênciamento recebem nomes da forma

$$\alpha \mid \beta \mid \gamma$$

com campos

Máquina α	
1	Um processador
P	Processadores paralelos
Q	Processadores relacionados
R	Processadores arbitrários
Restrições β	
D_i	Prazo máximo (deadline)
d_i	Prazo previsto (due dates)
r_i	Tempo de liberação (release time)
$p_i = p$	Tempo uniforme p
prec	Precedências
Função objetivo γ	
C_{max}	Maior tempo de término (maximum completion time)
$\sum_i C_i$	Tempo de término total (total completion time)
L_i	Atraso (lateness) $C_i - d_i$
T_i	Tardiness $\max\{L_i, 0\}$

Relação com empacotamento unidimensional:



- Empacotamento unidimensional: Dado C_{max} minimiza o número de processadores.
- $P \parallel C_{max}$: Dado um número de contêineres, minimiza o tamanho dos contêineres.

SEQUENCIAMENTO EM PROCESSORES PARALELOS ($P \parallel C_{max}$)

Entrada O número m de processadores e n tarefas com tempo de execução p_i, $i \in [n]$.

Solução Um *sequenciamento*, definido por uma alocação $M_1 \dot\cup \cdots \dot\cup M_m = [n]$ das tarefas às máquinas.

Objetivo Minimizar o *makespan* (tempo de término) $C_{max} = \max_{j\in[m]} C_j$, com $C_j = \sum_{i\in M_j} p_i$ o tempo de término da máquina j.

Fato 15.4
O problema $P \parallel C_{max}$ é fortemente NP-completo.

Um limite inferior para $C^*_{max} = OPT$ é

$$LB = \max\{\max_{i\in[n]} p_i, \sum_{i\in[n]} p_i/m\}.$$

Uma classe de algoritmos gulosos para este problema são os algoritmos de *sequenciamento em lista* (inglês: list scheduling). Eles processam as tarefas em alguma ordem, e alocam a tarefa atual sempre à máquina de menor tempo de término atual.

Proposição 15.7
Sequenciamento em lista com ordem arbitrária permite uma $2-1/m$-aproximação em tempo $O(n \log n)$.

Prova. Seja C_{max} o resultado do sequenciamento em lista. Considera uma máquina com tempo de término C_{max}. Seja j a última tarefa alocada nessa máquina e C o término da máquina antes de alocar tarefa j. Logo,

$$\begin{aligned} C_{max} = C + p_j &\le \sum_{i\in[j-1]} p_i/m + p_j \le \sum_{i\in[n]} p_i/m - p_j/m + p_j \\ &\le LB + (1-1/m)LB = (2-1/m)LB \le (2-1/m)C^*_{max}. \end{aligned}$$

A primeira desigualdade é correta, porque alocando tarefa j a máquina tem tempo de término mínimo. Usando uma fila de prioridade a máquina com o menor tempo de término pode ser encontrada em tempo $O(\log n)$. ■

Observação 15.4
Pela prova da proposição 15.7 temos

$$\mathrm{LB} \leq C^*_{max} \leq 2\mathrm{LB}.$$

$\Diamond$

O que podemos ganhar com algoritmos off-line? Uma abordagem é ordenar as tarefas por tempo execução não-crescente e aplicar o algoritmo gulos. Essa abordagem é chamada LPT (largest processing time).

Proposição 15.8
LPT é uma $4/3 - m/3$-aproximação em tempo $O(n \log n)$.

Prova. Seja $p_1 \geq p_2 \geq \cdots \geq p_n$ e supõe que isso é o menor contra-exemplo em que o algoritmo retorne $C_{max} > (4/3 - m/3)C^*_{max}$. Não é possível que a alocação do item $j < n$ resulta numa máquina com tempo de término C_{max}, porque $p_1, \ldots, p_j$ seria um contra-exemplo menor (mesmo C_{max}, menor C^*_{max}). Logo a alocação de p_n define o resultado C_{max}.
Caso $p_n \leq C^*_{max}/3$ pela prova da proposição 15.7 temos $C_{max} \leq (4/3 - m/3)C^*_{max}$, uma contradição. Mas caso $p_n > C^*_{max}/3$ todas tarefas possuem tempo de execução pelo menos $C^*_{max}/3$ e no máximo duas podem ser executadas em cada máquina. Logo $C_{max} \leq 2/3C^*_{max}$, outra contradição. ■

15.9.1. Um esquema de aproximação para $P \parallel C_{max}$

Pela observação 15.4 podemos reduzir o $P \parallel C_{max}$ para o empacotamento unidimensional via uma busca binária no intervalo $[\mathrm{LB}, 2\mathrm{LB}]$. Pela proposição 15.5 isso é possível em tempo $O(\log \mathrm{LB}\, mn(2\mathrm{LB}+1)^m)$.
Com mais cuidado a observação permite um esquema de aproximação em tempo polinomial assintótico: similar com o esquema de aproximação para empacotamento unidimensional, vamos *remover elementos menores* e *arredondar* a instância.

Algoritmo 15.6 (Sequencia)
Entrada Uma instância I de $P \parallel C_{max}$, um término máximo C e um parâmetro de qualidade ϵ.

```
Sequencia(I,C,ε):=
  remove as tarefas menores com p_j < εC, j ∈ [n]
  arredonda cada p_j ∈ [εC(1+ε)^i, εC(1+ε)^{i+1}) para algum i
     para p'_j = εC(1+ε)^i
```

```
resolve a instância arredondada com programação
    dinâmica (proposição 15.6)
empacota os itens menores gulosamente, usando novas
    máquinas para manter o término (1+ε)C
```

Lema 15.3
O algoritmo Sequencia gera um sequenciamento que termina em no máximo $(1+\epsilon)C$ em tempo $O(n^{2\lceil \log_{1+\epsilon} 1/\epsilon \rceil})$. Ele não usa mais máquinas que o mínimo necessário para executar as tarefas com término C

Prova. Para cada intervalo válido temos $\epsilon C(1+\epsilon)^i \leq C$, logo o número de intervalos é no máximo $k = \lceil \log_{1+\epsilon} 1/\epsilon \rceil$. O valor k também é um limite para o número de valores p'_j distintos e pela proposição 15.6 o terceiro passo resolve a instância arredondada em tempo $O(n^{2k})$. Essa solução com os itens de tamanho original termina em no máximo $(1+\epsilon)C$, porque $p_j/p'_j < 1+\epsilon$. O número mínimo de máquinas para executar as tarefas em tempo C é o valor $m := \text{min-EU}(C, (p_j)_{j\in[n]})$ do problema de empacotamento unidimensional correspondente. Caso o último passo do algoritmo não usa novas máquinas ele precisa $\leq m$ máquinas, porque a instância arredondada foi resolvida exatamente. Caso contrário, uma tarefa com tempo de execução menor que ϵC não cabe nenhuma máquina, e todas máquinas usadas tem tempo de término mais que C. Logo o empacotamento ótimo com término C tem que usar pelo menos o mesmo número de máquinas. ■

Proposição 15.9
O resultado da busca binária usando o algoritmo Sequencia $C_{max} = \min\{C \in [LB, 2LB] \mid \text{Sequencia}(I, C, \epsilon) \leq m\}$ é no máximo C^*_{max}.

Prova. Com $\text{Sequencia}(I, C, \epsilon) \leq \text{min-EU}(C, (p_i)_{i\in[n]})$ temos

$$\begin{aligned} C_{max} &= \min\{C \in [LB, 2LB] \mid \text{Sequencia}(I, C, \epsilon) \leq m\} \\ &\leq \min\{C \in [LB, 2LB] \mid \text{min-EU}(C, (p_i)_{i\in[n]}) \leq m\} \\ &= C^*_{max} \end{aligned}$$

■

Teorema 15.11
A busca binária usando o algoritmo Sequencia para determinar determina um sequenciamento em tempo $O(n^{2\lceil \log_{1+\epsilon} 1/\epsilon \rceil} \log LB)$ de término máximo $(1+\epsilon)C^*_{max}$.

Prova. Pelo lema 15.3 e proposição 15.9. ■

15.10. Exercícios

Exercício 15.1
Por que um subgrafo conexo de menor custo sempre é uma árvore?

Exercício 15.2
Mostra que o número de vértices com grau ímpar num grafo sempre é par.

Exercício 15.3
Um aluno propõe a seguinte heurística para o empacotamento unidimensional: Ordene os itens em ordem crescente, coloca o item com peso máximo junto com quantas itens de peso mínimo que é possível, e depois continua com o segundo maior item, até todos itens foram colocados em bins. Temos o algoritmo

```
ordene itens em ordem crescente
m := 1;  M := n
while (m < M) do
  abre novo contêiner, coloca v_M, M := M − 1
  while (v_m cabe e m < M) do
    coloca v_m no contêiner atual
    m := m + 1
  end while
end while
```

Qual a qualidade desse algoritmo? É um algoritmo de aproximação? Caso sim, qual a taxa de aproximação dele? Caso não, por quê?

Exercício 15.4
Prof. Rapidez propõe o seguinte pré-processamento para o algoritmo SAT-R de aproximação para MAX-SAT (página 325): Caso a instância contém claúsulas com um único literal, vamos escolher uma delas, definir uma atribuição parcial que satisfazê-la, e eliminar a variável correspondente. Repetindo esse procedimento, obtemos uma instância cujas claúsulas tem 2 ou mais literais. Assim, obtemos $l \geq 2$ na análise do algoritmo, o podemos garantir que $E[X] \geq 3n/4$, i.e. obtemos uma $4/3$-aproximação.
Esta análise é correto ou não?

16. Algoritmos randomizados

Um algoritmo randomizado usa *eventos aleatórios* na sua execução. Modelos computacionais adequadas são máquinas de Turing probabilísticas – mais usadas na área de complexidade – ou máquinas RAM com um comando `random(S)` que retorne um elemento aleatório do conjunto S.
Veja alguns exemplos de probabilidades:

- Probabilidade morrer caindo da cama: $1/2 \times 10^6$ (Roach e 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 $100\,\mathrm{m}^2$).

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

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

[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 (Terrazon 2004).

16.1. Teoria de complexidade

Classes de complexidade

Definição 16.1
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

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

(A probabilidade é sobre todas sequências de bits aleatórios r. Como o algoritmo executa em tempo polinomial no tamanho da entrada $|x|$, o número de bits aleatórios $|r|$ 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 $PP := \bigcup_{\epsilon \in (0,1/2]} R(1/2 + \epsilon, 1/2 + \epsilon)$ (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 16.1 (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? Converter 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 $O(d)$):

```
identico(p,q) :=
  Seleciona um número aleatório r no intervalo [1,100d].
  Caso p(r) = q(r) retorne ``sim''.
  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 $co - RP$. ◊

Observação 16.1
É uma pergunta em aberto se o teste de identidade pertence a P. ◊

16.1.1. Amplificação de probabilidades

Caso não estamos satisfeitos com a probabilidade de $1/100$ no exemplo acima, podemos repetir o algoritmo k vezes, e responder "sim" somente se todas k repetições responderam "sim". A probabilidade erradamente responder "não" para polinômios idênticos agora é $(1/100)^k$, i.e. ela diminui exponencialmente com o número de repetições.
Essa técnica é uma *amplificação* da probabilidade de obter a solução correta. Ela pode ser aplicada para melhorar a qualidade de algoritmos em todas classes "Monte Carlo". Com um número constante de repetições, obtemos uma probabilidade baixa nas classes RP, $co - RP$ 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 16.6.

Teorema 16.1
$R(,,1) = R(\text{¿},1)$ para $0 < \alpha, \beta < 1$.

Prova. Sem perda de generalidade seja $\alpha < \beta$. Claramente $R(\text{¿},1) \subseteq R(,,1)$. Supõe que A é um algoritmo que testemunha $L \in R(,,1)$. Execute A no máximo k vezes, respondendo "sim" caso A responde "sim" em alguma iteração e "não" caso contrário. Chama esse algoritmo A'. Caso $x \notin L$ temos $Pr(A'(x) = \text{"não"}) = 1$. Caso $x \in L$ temos $Pr(A'(x) = \text{"sim"}) \geq 1 - (1-\alpha)^k$, logo para $k \geq \ln(1-\beta)/\ln(1-\alpha)$, $Pr(A'(x) = \text{"sim"}) \geq \beta$. ■

Corolário 16.1
$RP = R(,,1)$ para $0 < \alpha < 1$.

Teorema 16.2
$\mathsf{R}(,,) = \mathsf{R}(_{\epsilon,\epsilon})$ para $1/2 < \alpha, \beta$.

Prova. Sem perda de generalidade seja $\alpha < \beta$. Claramente $\mathsf{R}(_{\epsilon,\epsilon}) \subseteq \mathsf{R}(,,)$. Supõe que A é um algoritmo que testemunha $L \in \mathsf{R}(,,)$. Executa A k vezes, responde "sim" caso a maioria de respostas obtidas foi "sim", e "não" caso contrário. Chama esse algoritmo A'. Para $x \in L$ temos

$$\Pr(A'(x) = \text{“sim”}) = \Pr(A(x) = \text{“sim”} \geq \lfloor k/2 \rfloor + 1 \text{ vezes}) \geq 1 - e^{-2k(\alpha - 1/2)^2}$$

e para $k \geq \ln(\beta - 1)/2(\alpha - 1/2)^2$ temos $\Pr(A'(x) = \text{“sim”}) \geq \beta$. Similarmente, para $x \notin L$ temos $\Pr(A'(x) = \text{“não”}) \geq \beta$. Logo $L \in \mathsf{R}(_{\epsilon,\epsilon})$. ■

Corolário 16.2
$\mathsf{BPP} = \mathsf{R}(,,)$ para $1/2 < \alpha$.

Observação 16.2
Os resultados acima são válidos ainda caso o erro dimiui polinomialmente com o tamanho da instância, i.e. $\alpha, \beta \geq n^{-c}$ no caso do teorema 16.1 e $\alpha, \beta \geq 1/2 + n^{-c}$ no caso do teorema 16.2 para um constante c (ver por exemplo Arora e Barak (2009)). ◊

16.1.2. Relação entre as classes

Duas caracterizações alternativas de ZPP

Definição 16.2
Um algoritmo A é *honesto* se

i) ele responde ou "sim", ou "não" ou "não sei",

ii) $\Pr(A(x) = \text{não sei}) \leq 1/2$, e

iii) no caso ele responde, ele não erra, i.e., para x tal que $A(x) \neq$ "não sei" temos $A(x) =$ "sim" $\iff x \in L$.

Uma linguagem é honesta caso ela possui um algoritmo honesto. Com isso também podemos falar da classe das linguagens honestas.

Teorema 16.3
ZPP é a classe das linguagens honestas.

Lema 16.1
Caso $L \in \mathsf{ZPP}$ existe um algoritmo um algoritmo honesto para L.

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

```
if A1(x) = ‘‘não’’ e A2(x) = ‘‘não’’ then
  return ‘‘não’’
else if A1(x) = ‘‘não’’ e A2(x) = ‘‘sim’’ then
  return ‘‘não sei’’
else if A1(x) = ‘‘sim’’ e A2(x) = ‘‘não’’ then
  { caso impossível }
else if A1(x) = ‘‘sim’’ e A2(x) = ‘‘sim’’ then
  return ‘‘sim’’
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 $x \notin L$, $\Pr(A_1(x) = \text{“não”} \wedge A_2(x) = \text{“sim”}) \leq 1 \times 1/2 = 1/2$. ■

Lema 16.2
Caso L possui um algoritmo honesto $L \in \mathsf{RP}$ e $L \in \mathsf{co}-\mathsf{RP}$.

Prova. Seja A um algoritmo honesto. 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. ■

Definição 16.3
Um algoritmo A é *sem falha* se ele sempre responde “sim” ou “não” corretamente em *tempo polinomial esperado*. Com isso podemos também falar de linguagens sem falha e a classe das linguagens sem falha.

Teorema 16.4
ZPP é a classe das linguagens sem falha.

Lema 16.3
Caso $L \in \mathsf{ZPP}$ existe um algoritmo sem falha para L.

Prova. Sabemos que existe um algoritmo honesto para L. Repete o algoritmo honesto até encontrar um “sim” ou “não”. Como o algoritmo honesto executa em tempo polinomial $p(n)$, o tempo esperado desse algoritmo ainda é polinomial:

$$\sum_{k>0} k 2^{-k} p(n) \leq 2p(n)$$

■

Lema 16.4
Caso L possui um algoritmo A sem falha, $L \in RP$ e $L \in co-RP$.

Prova. Caso A tem tempo esperado $p(n)$ executa ele para um tempo $2p(n)$. Caso o algoritmo responde, temos a resposta certa. Caso contrário, responde "sim". Pela desigualdade de Markov temos uma resposta com probabilidade $Pr(T \geq 2p(n)) \leq p(n)/2p(n) = 1/2$. Isso mostra que existe um algoritmo honesto para L, e pelo lema 16.2 $L \in RP$. O argumento para $L \in co-RP$ é similar. ■

Mais relações

Teorema 16.5
$RP \subseteq NP$ e $co-RP \subseteq co-NP$

Prova. Supõe que temos um algoritmo em RP para algum problema L. Podemos, não-deterministicamente, gerar todas sequências r de bits aleatórios e responder "sim" caso alguma execução encontra "sim". O algoritmo é correto, porque caso para um $x \notin L$, não existe uma sequência aleatória r tal que o algoritmo responde "sim". A prova do segundo caso é similar. ■

Teorema 16.6
$RP \subseteq BPP$ e $co-RP \subseteq BPP$.

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

```
if A(x) = ‘‘não’’ e A(x) = ‘‘não’’ then
  return ‘‘não’’
else
  return ‘‘sim’’
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 $x \in L$,

$$Pr(A'(x) = \text{“sim”}) = 1 - Pr(A'(x) = \text{“não”}) = 1 - Pr(A(x) = \text{“não”} \wedge A(x) = \text{“não”})$$
$$\geq 1 - 1/2 \times 1/2 = 3/4 > 2/3.$$

(Observe que para k repetições de A obtemos $Pr(A'(x) = \text{“sim”}) \geq 1 - 1/2^k$, i.e., o erro diminui exponencialmente com o número de repetições.) O argumento para $co-RP$ é similar. ■

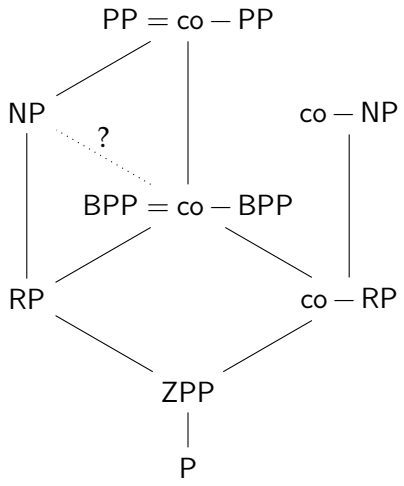


Figura 16.1.: Relações entre classes de complexidade para algoritmos randomizados.

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 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 aleatória 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.

16.2. Seleção

O algoritmo determinístico para selecionar o k-ésimo elemento de uma sequencia não ordenada $x_1, \ldots, x_n$ discutido na seção 6.3.3 (página 142) pode ser simplificado usando randomização: escolheremos um elemento pivô $m = x_i$ aleatório. Com isso o algoritmo 6.10 fica mais simples:

Algoritmo 16.1 (Seleção randomizada)
Entrada Números $x_1, \ldots, x_n$, posição k.

Saída O k-ésimo maior número.

```
S(k,{x_1,...,x_n}) :=
  if n ≤ 1
    calcula e retorne o k-ésimo elemento
  end if
  m := x_i para um i ∈ [n] aleatória
  L := {x_i | x_i < m, 1 ≤ i ≤ n}
  R := {x_i | x_i ≥ m, 1 ≤ i ≤ n}
  i := |L| + 1
  if i = k then
    return m
  else if i > k then
    return S(k, L)
  else
    return S(k − i, R)
  end if
```

Para determinar a complexidade podemos observar que com probabilidade $1/n$ temos $|L| = i$ e $|R| = n - i$ e o caso pessimista é uma chamada recursiva com $\max\{i, n-i\}$ elementos. Logo, com custo cn para particionar o conjunto e os testes temos

$$\begin{aligned} T(n) &\leq \sum_{i\in[0,n]} 1/n T(\max\{n-i,i\}) + cn \\ &= 1/n\left(\sum_{i\in[0,\lfloor n/2\rfloor]} T(n-i) + \sum_{i\in[\lceil n/2\rceil,n]} T(i)\right) + cn \\ &2/n \sum_{i\in[0,\lfloor n/2\rfloor]} T(n-i) + cn \end{aligned}$$

Separando o termo $T(n)$ do lado direito obtemos

$$\begin{aligned} (1-2/n)T(n) &\leq 2/n \sum_{i\in[1,\lfloor n/2\rfloor]} T(n-i) + cn \\ \iff T(n) &\leq \frac{2}{n-2}\left(\sum_{i\in[1,\lfloor n/2\rfloor]} T(n-i) + cn^2/2\right). \end{aligned}$$

Provaremos por indução que $T(n) \leq c'n$ para uma constante c'. Para um $n \leq n_0$ o problema pode ser claramente resolvido em tempo constante (por exemplo em $O(n_0 \log n_0)$ via ordenação). Logo, supõe que $T(i) \leq c'i$ para

$i < n$. Demonstraremos que $T(n) \leq c'n$. Temos

$$\begin{aligned} T(n) &\leq \frac{2}{n-2}\left(\sum_{i\in[1,\lfloor n/2\rfloor]} T(n-i) + cn^2/2\right) \\ &\leq \frac{2c'}{n-2}\left(\sum_{i\in[1,\lfloor n/2\rfloor]} n - i + cn^2/2c'\right) \\ &= \frac{2c'}{n-2}\left((2n - \lfloor n/2\rfloor - 1)\lfloor n/2\rfloor/2 + cn^2/2c'\right) \end{aligned}$$

e com $2n - \lfloor n/2\rfloor - 1 \leq 3/2n$

$$\begin{aligned} &\leq \frac{c'}{n-2}(3/4n^2 + cn^2/c') \\ &= c'n\frac{(3/4 + c/c')n}{n-2} \end{aligned}$$

Para $n \geq n_0 := 16$ temos $n/(n-2) \leq 8/7$ e com um $c' > 8c$ temos

$$T(n) \leq c'n(3/4 + 1/8)8/7 \leq c'n.$$

16.3. 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:

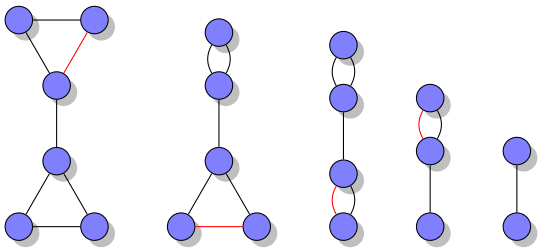
- Calcular a árvore de Gomory-Hu: a aresta de menor peso define o corte mínimo.
- Calcular o corte mínimo (via fluxo máximo) entre um vértice fixo $s \in V$ e todos outros vértices: o menor corte encontrado é o corte mínimo.

Custo em ambos casos: $O(n)$ aplicações de um algoritmo de fluxo máximo, i.e. $O(mn^2)$ usando o algoritmo de Orlin.

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.

```
cmr(G) :=
  while G possui mais que dois vértices
    escolhe uma aresta {u,v} aleatoriamente
    identifica u e v em G
  end while
  return o corte definido pelos dois vértices em G
```

Exemplo 16.2
Uma sequencia de contrações (das arestas vermelhas).



◊

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

Lema 16.5
A probabilidade que os k arestas de um corte mínimo sobrevivem $n - n'$ contrações (de n para n' vértices) é $\Omega((n'/n)^2)$.

Prova. Como o corte mínimo é k, cada vértice possui grau pelo menos k, e portanto o número de arestas após da iteração $0 \leq i < n - n'$ 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 pelo menos $k/(k(n-i)/2) = 2/(n-i)$. Logo, a probabilidade do corte sobreviver todas iterações

é pelo menos

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

■

Teorema 16.7
Dado um corte mínimo C de tamanho k, a probabilidade do algoritmo cmr retornar C é $\Omega(n^{-2})$.

Prova. Caso o grafo possui n vértices, o algoritmo termina em $n-2$ iterações: podemos aplicar o lema acima com $n' = 2$. ■

Observação 16.3
O que acontece se repetimos o algoritmo algumas vezes? Seja C_i uma variável que indica se o corte mínimo foi encontrado na repetição i. Temos $\Pr(C_i = 1) \ge 2n^{-2}$ e portanto $\Pr(C_i = 0) \le 1-2n^{-2}$. Para kn^2 repetições, vamos encontrar $C = \sum C_i$ cortes mínimos com probabilidade

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

Para $k = \log n$ obtemos $\Pr(C \ge 1) \ge 1-n^{-2}$. ♢

Logo, ao repetir o algoritmo $n^2 \log n$ vezes e retornar o menor corte encontrado, achamos o corte mínimo com probabilidade razoável. Se a implementação realiza uma contração em tempo $O(n)$ 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. Define $f(n) = \left\lceil 1 + n/\sqrt{2} \right\rceil$.

```
cmr2(G) :=
  if (G possui menos que 6 vértices)
    determina o corte mínimo C por exaustão
    return C
  else
    n' := f(n)
    seja G1 o resultado de n-n' contrações em G
    seja G2 o resultado de n-n' contrações em G
    C1 := cmr2(G1)
    C2 := cmr2(G2)
    return o menor dos dois cortes C1 e C2
  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 16.6
A probabilidade de um corte mínimo sobreviver $n - f(n)$ contrações é pelo menos $1/2$.

Prova. Pelo lema 16.5 a probabilidade é pelo menos

$$\frac{f(n)(f(n)-1)}{n(n-1)} \geq \frac{(1+n/\sqrt{2})(n/\sqrt{2})}{n(n-1)} = \frac{\sqrt{2}+n}{2(n-1)} \geq \frac{n}{2n} = \frac{1}{2}.$$

■

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

$$\begin{aligned}
\Pr(\text{o corte sobrevive em } G_1) &\geq 1/2\, P(f(n)) \\
\Pr(\text{o corte sobrevive em } G_2) &\geq 1/2\, P(f(n)) \\
\Pr(\text{o corte não sobrevive em } G_1 \text{ nem } G_2) &\leq (1 - 1/2P(f(n)))^2 \\
P(n) = \Pr(\text{o corte sobrevive em } G_1 \text{ ou } G_2) &\geq 1 - (1 - 1/2P(f(n)))^2 \\
&= P(f(n)) - 1/4P(f(n))^2
\end{aligned}$$

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

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

e depois $R(k) = 4/Q(k) - 1$ com base $R(0) = 3$ para obter

$$\frac{4}{R(k+1)+1} = \frac{4}{R(k)+1} - \frac{4}{(R(k)+1)^2} \iff R(k+1) = R(k) + 1 + 1/R(k).$$

$R(k)$ satisfaz

$$k < R(k) < k + H_{k-1} + 3$$

Prova. Por indução. Para $k = 1$ temos $1 < R(1) = 13/3 < 1 + H_0 + 3 = 5$. Caso a HI está satisfeito, temos

$$\begin{aligned} R(k+1) &= R(k) + 1 + 1/R(k) > R(k) + 1 > k + 1 \\ R(k+1) &= R(k) + 1 + 1/R(k) < k + H_{k-1} + 3 + 1 + 1/k = (k+1) + H_k + 3 \end{aligned}$$

■

Logo, $R(k) = k + \Theta(\log k)$, e com isso $Q(k) = \Theta(1/k)$ e finalmente $P(n) = \Theta(1/\log n)$.
Para determinar a complexidade do algoritmo cmr2 observe que temos $O(\log n)$ níveis de recursão e cada contração pode ser feita em tempo $O(n^2)$, portanto

$$T_n = 2T(f(n)) + O(n^2).$$

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

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

16.4. Teste de primalidade

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

Teorema 16.8 (Hadamard (1896), 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 aleatório no intervalo $[2, n]$ ser primo assintoticamente é somente $1/\ln n$. Então para encontrar um número primo, temos que testar se n é primo mesmo. Observe que isso não é igual a fatoração de n. De fato, temos testes randomizados (e determinísticos) em tempo polinomial, enquanto não sabemos fatorar nesse tempo. Uma abordagem simples é testar todos os divisores:

```
Primo1(n) :=
  for i = 2,3,5,7,...,⌊√n⌋ do
    if i|n return ''Não''
  end for
  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 16.9 (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 16.4
Para $a, b \in \mathbb{Z}$ denotamos com (a, b) o maior divisor em comum (MDC) de a e b. No caso $(a, b) = 1$, a e b são números *coprimos*.

Teorema 16.10 (Divisão modulo p)
Caso p é 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 $1 \leq a < p$)

$$a^{p-1} \equiv 1 \mod p. \tag{16.1}$$

Um teste melhor então é

```
Primo2(n) :=
  seleciona a ∈ [1,n−1] aleatoriamente
  if (a,n) ≠ 1 return ‘‘Não’’
  if a^{n−1} ≡ 1 return ‘‘Sim’’
  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 16.5
Um número composto n que satisfaz $a^{n-1} \equiv 1 \mod n$ 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 16.11 (Alford et al. (1994))
Seja $C(n)$ o número de números Carmichael até n. Assintoticamente temos $C(n) > n^{2/7}$.

Exemplo 16.3
$C(n)$ 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 (16.1) 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 16.12 (Raiz modular)
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 ímpar. 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 (16.12) e a minimalidade de i (que exclui o caso $\equiv 1$). Por isso:

Definição 16.6
Um número n é um *pseudo-primo forte com base* a caso

$$\text{Ou } a^u \equiv 1 \mod p \text{ ou existe um menor } i \in [0, t-1] \text{ tal que } (a^u)^{2^i} \equiv -1 \tag{16.2}$$

```
Primo3(n) :=
  seleciona a ∈ [1,n−1] aleatoriamente
  if (a,n) ≠ 1 return ''Não''
```

```
seja n − 1 = 2^t u
if a^u ≡ 1 return ''Sim''
if (a^u)^{2^i} ≡ −1 para um i ∈ [0, t − 1] return ''Sim''
return ''Não''
```

Teorema 16.13 (Monier (1980) e Rabin (1980))
Caso n e composto e ímpar, mais que $3/4$ dos $a \in [1, n-1]$ com $(a, n) = 1$ não satisfazem o critério (16.2) acima.

Portanto com k testes, a probabilidade de falhar $\Pr(\text{Sim} \mid n \text{ composto}) \leq (1/4)^k = 2^{-2k}$. De fato a probabilidade é menor:

Teorema 16.14 (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 16.4
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 pelo menos $1/4$ dos a com $(a, n) = 1$. De fato, para o menor testemunho $w(n)$ de um número n ser composto temos

$$\text{Se o HGR é verdade: } w(n) < 2\log^2 n \tag{16.3}$$

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.

16.5. Exercícios

Exercício 16.1
Encontre um primo p e um valor b tal que a identidade do teorema 16.10 não é correta.

Exercício 16.2
Encontre um número p não primo tal que a identidade do teorema 16.12 não é correta.

Parte V.

Teoria de complexidade

17. Do algoritmo ao problema

17.1. Introdução

Motivação

- Análise e projeto: Foca em *algoritmos*.
- Teoria de complexidade: Foca em *problemas*.
- Qual a complexidade intrínsica de problemas?
- *Classes de complexidade* agrupam problemas.
- Interesse particular: Relação entre as classes.

Abstrações: Alfabetos, linguagens

- Seja Σ um *alfabeto* finito de símbolos.
- Codificação: Entradas e saídas de um algoritmo são *palavras* sobre o alfabeto $\omega \in \Sigma^*$.
- Tipos de problemas:
  - Problema construtivo: Função $\Sigma^* \to \Sigma^*$ Alternativa permitindo várias soluções: relação $R \subseteq \Sigma^* \times \Sigma^*$.
  - Problema de decisão: Função $\Sigma^* \to \{S, N\}$ Equivalente: conjunto $L \subseteq \Sigma^*$ (uma *linguagem*).
  - Problema de otimização: Tratado como problema de decisão com a pergunta "Existe solução $\geq k$?" ou "Existe solução $\leq k$?".
- Frequentemente: Alfabeto $\Sigma = \{0, 1\}$ com codificação adequada.

Convenção 17.1
Sem perda de generalidade suporemos $\Sigma = \{0, 1\}$.

Definição 17.1
Uma *linguagem* é um conjunto de palavras sobre um alfabeto: $L \subseteq \Sigma^*$.

Modelo de computação: Máquina de Turing

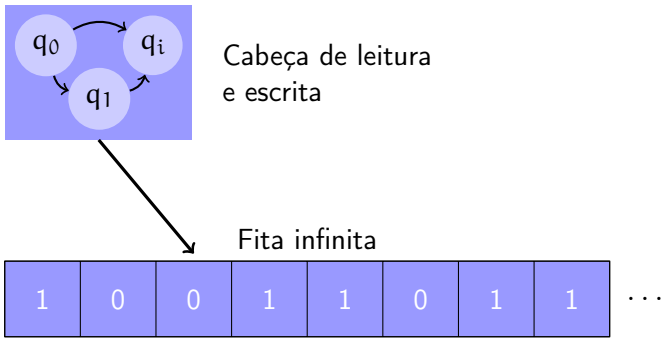
- Foco: Estudar as limitações da complexidade, não os algoritmos.
- Portanto: Modelo que facilite o estudo teórica, não a implementação.
- Solução: Máquina de Turing.

> But I was completely convinced only by Turing's paper. (Gödel em uma carta para Kreisel, em maio de 1968, falando sobre uma definição da computação.).

Figura 17.1.: Alan Mathison Turing (*1912, +1954)

> Computing is normally done by writing certain symbols on paper. "We may suppose this paper is divided into squares like a child's arithmetic book. In elementary arithmetic the two-dimensional character of the paper is sometimes used. But such a use is always avoidable, and I think that it will be agreed that the two-dimensional character of paper is no essential of computation. I assume then that the computation is carried out on one-dimensional paper, i.e. on a tape divided into squares. I shall also suppose that the number of symbols which may be printed is finite. If we were to allow an infinity of symbols, then there would be symbols differing to an arbitrarily small extent. The effect of this restriction of the number of symbols is not very serious. It is always possible to use sequences of symbols in the place of single symbols. Thus an Arabic numeral such as 17 or *999999999999999* is normally treated as a single symbol. Similarly in any European language words are treated as single symbols (Chinese, however, attempts to have an enumerable infinity of symbols). The differences from our point of view between the single and compound symbols is that the compound symbols, if they are too lengthy, cannot be observed at one glance. This is in accordance with experience. We cannot tell at a glance whether *9999999999999999* and *999999999999999* are the same. (Turing 1936).

Máquina de Turing



Máquina de Turing (MT)

$$M = (Q, \Sigma, \Gamma, \delta)$$

- Alfabeto de entrada Σ (sem branco $\sqcup$)
- Conjunto de estados Q entre eles três estados particulares:
  - Um estado inicial $q_0 \in Q$, um que aceita $q_a \in Q$ e um que rejeita $q_r \in Q$.
- Alfabeto de fita $\Gamma \supseteq \Sigma$ (inclusive $\in \Gamma$)
- Regras $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$, escritas da forma

$$q, a \to q'a'D$$

(com $q, q' \in Q$, $a, a' \in \Sigma$ e $D \in \{L, R\}$).

Máquina de Turing: Operação

- Início da computação:
  - No estado inicial q_0 com cabeça na posição mais esquerda,
  - com entrada $w \in \Sigma^*$ escrita na esquerda da fita, resto da fita em branco.
- Computação: No estado q lendo um símbolo a aplica uma regra $qa \to q'a'D$ (um L na primeira posição não tem efeito) até
  - não encontrar uma regra: a computação termina, ou

- entrar no estado q_a: a computação termina e *aceita*, ou
- entrar no estado q_r: a computação termina e *rejeita*.
- Outra possibilidade: a computação não termina.

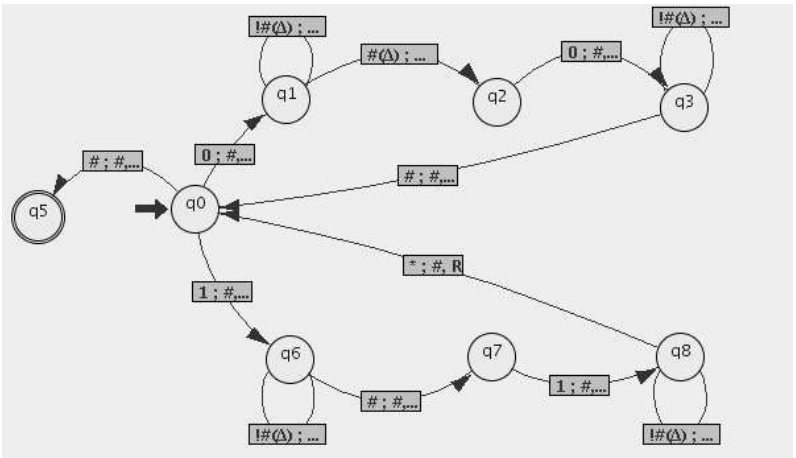
Exemplo 17.1 (Decidir ww^R)

Tabela da transição

Seja $\Sigma = \{0, 1\}$. Uma máquina de Turing que reconhece a linguagem $\{ww^R \mid w \in \Sigma^*\}$ é

q0	0	#	R	q1
q1	#(Δ)	Δ	L	q2
q1	!#(Δ)	Δ	R	q1
q2	0	#	L	q3
q3	!#(Δ)	Δ	L	q3
q3	#	#	R	q0
q0	#	#	R	q5
q0	1	#	R	q6
q6	!#(Δ)	Δ	R	q6
q6	#	#	L	q7
q7	1	#	L	q8
q8	!#(Δ)	Δ	L	q8
q8	*	#	R	q0

Notação gráfica



(convenções e abreviações do Turing machine simulator; veja página da disciplina). ◊

Máquinas não-determinísticas

- Observe: Num estado q lendo símbolo a temos exatamente uma regra da forma $qa \to q'a'D$.
- Portanto a máquina de Turing é *determinística* (MTD).
- Caso mais que uma regra que se aplica em cada estado q e símbolo a a máquina é *não-determinística* (MTND).
- A função de transição nesse caso é

$$\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

Linguagem de uma MTD

- O conjunto de palavras que uma MTD M aceita é *a linguagem reconhecida* de M.

$$L(M) = \{w \in \Sigma^* \mid M \text{ aceita } w\}$$

- Uma linguagem tal que existe um MTD M que reconhece ela é *Turing-reconhecível por* M.

$$L \text{ Turing-reconhecível} \iff \exists M : \ L = L(M)$$

- Observe que uma MTD não precisa parar sempre. Se uma MTD sempre para, ela *decide* a sua linguagem.
- Uma linguagem Turing-reconhecível por uma MTD M que sempre para é *Turing-decidível*.

$$L \text{ Turing-decidível} \iff \exists M : \ L = L(M) \text{ e } M \text{ sempre para}$$

Observação 17.1
Para representar problemas sobre números inteiros, ou estruturas de dados mais avançados como grafos, temo que codificar a entrada como palavara em Σ^*. Escrevemos $\langle x \rangle$ para codificação do objeto x. ◊

Linguagem de uma MTND

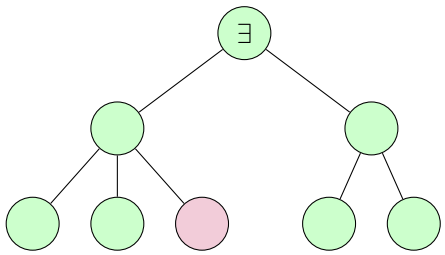
- Conjunto de linguagens Turing-reconhecíveis: linguagens *recursivamente enumeráveis* ou *computavelmente enumeráveis*.
- Conjunto de linguagens Turing-decidíveis: linguagens *recursivas* ou *computáveis*.
- Para uma máquina não-determinística temos que modificar a definição: ela precisa somente um estado que aceita e
  - ela reconhece a linguagem

    $$L(M) = \{w \in \Sigma^* \mid \text{existe uma computação tal que } M \text{ aceita } w\}$$

  - ela decide uma linguagem se todas computações sempre param.

Máquina de Turing não-determinística

- Resposta *sim*, se existe uma computação que responde *sim*.



Robustez da definição

- A definição de uma MTD é *computacionalmente robusto*: as seguintes definições alternativas decidem as mesmas linguagens:
  (a) Uma MT com $k > 1$ fitas (cada uma com cabeça própria),
  (b) uma MT com fita duplamente infinita,
  (c) uma MT com alfabeto restrito $\Sigma = \{0, 1\}$,
  (d) uma MTND e
  (e) uma MT com fita de duas ou mais dimensões.
- O poder computacional também é equivalente com vários outros modelos de computação (p.ex. cálculo lambda, máquina RAM, autômato celular): Tese de Church-Turing.

Prova. (Rascunho.)

(a) Seja a_{ij} o símbolo na posição j da fita i e l_i o índice do último símbolo usado na fita i. A máquina com uma única fita representa os k fitas da forma

$$\#a_{11} \dots a_{1l_1} \# \dots \# a_{k1} \dots a_{kl_k}.$$

Para representar as posições das cabeças, usamos símbolos com uma marca $\dot{a}$. Uma MTD simula a máquina com $k > 1$ fitas em dois passos: (i) Determina os símbolos abaixo das cabeças em um passo. (ii) executa as operações da k cabeças. Caso uma cabeça ultrapassa a direita da representação, a MTD estende-la, copiando todos símbolos da direita mais uma para direita.

(b) Seja a_i o símbolo na posição i da fita, com $i \in \mathbb{Z}$. A máquina com uma única fita meia-infinita representa essa fita por

$$a_0 \langle a_{-1} a_1 \rangle \langle a_{-2} a_2 \rangle \dots.$$

com símbolos novos em $\Sigma \cup \Sigma^2$. Os estados da máquina são $Q \times \{S, I\}$, registrando além da estado da máquina simulada, se o simulação trabalho no parte superior ou inferior. A simulação possui dois conjuntos de regras para transições em estados (q, S) e estados (q, I) e um conjunto de regras para passar de cima para baixo e vice versa.

(c) Cada símbolo é representado por uma sequência em $\{0,1\}^w$ de comprimento $w = \lceil \log |\Sigma| \rceil$. Na simulação a MTD primeiro leia os w símbolos atuais, calcula e escreve a representação do novo símbolo e depois se movimenta w posições para esquerda ou direita.

(d) A MTD simula uma MTN listando todas as execuções possíveis sistematicamente.

(e) Usando uma representação linear da fita, por exemplo linha por linha no caso de duas dimensões, similar com o caso 1, a máquina pode ser simulado calculando o novo índice da cabeça.

■

Observação 17.2
Uma modelo conveniente com $k > 1$ fitas é usar a primeira fita como *fita de entrada* e permitir somente leitura; uma outra fita das $k - 1$ restantes e usado como *fita da saída* caso a MT calcula uma função. ◊

Proposição 17.1
Se uma MTD $M = (Q, \Sigma, \Gamma, \delta)$ com $\Sigma = \{0, 1\}$ decide uma linguagem L em tempo $t(n)$, com $t(n)$ tempo-construtível, então existe uma MT com alfabeto $\Gamma' = \{0, 1, \}$ que decide L em tempo $4 \log |\Gamma| t(n)$.

Prova. Para construir uma MT que trabalha com Γ' vamos codificar cada símbolo em Γ por um string de $\log |\Gamma|$ bits. Em cada passo, a simulação lê $\log |\Gamma|$ símbolos a partir da posição atual, calcula o novo símbolo e estado usando as regras de M, escreve os novos $\log |\Gamma|$ símbolos, e movimenta a cabeção $\log |\Gamma|$ posições para direita ou esquerda. Para essa simulação, a nova máquina precisa armazenar o estado Q e no máximo $\log |\Gamma|$ símbolos na cabeça, e ainda precisa um contador de 1 até $\log |\Gamma|$. Isso é possível com $O(Q \times |\Gamma|^2)$ estados e em menos que $4 \log |\Gamma| t(n)$ passos (ler, escrever, movimentar e talvez alguns passos para o controle). ■

Proposição 17.2
Se uma MTD $M = (Q, \Sigma, \Gamma, \delta)$ com $\Sigma = \{0, 1\}$ e com $k > 1$ fitas decide uma linguagem L em tempo $t(n) \geq n$, com $t(n)$ tempo-construtível, então existe uma MT com única fita que decide L em tempo $5kt(n)^2$.

Prova. Vamos construir uma MT M' que simula M com uma única fita. M' armazena o conteúdo das k fitas de M de forma intercalada nas posições $1 + ki$, para $i \in [k]$. Para representar as posições das cabeças, usamos símbolos com uma marca $\dot{a}$. M' não altera as primeiras n posições da sua fita que contém a entrada, mas copia-las para a posição $n + 1$ na codificação acima.
Para simular um passo de M, M' escaneia a fita de esquerda para direita para determinar as posições das cabeças e os símbolos correspondentes. Depois M' usa as regras de transição de M para determinar o novo símbolo, estado, e o movimento da cabeça. Um segunda scan da direita para esquerda atualiza a fita de acordo.
M' produz as mesma saída que M. Como a maior posição visitada por M é $t(n)$, a maior posição visitada por M' é no máximo $kt(n) + 2n \leq (k + 2)t(n) \leq 2kt(n)$. Portanto, para cada passo de M, M' precisa no máximo $5kt(n)$ passos ($4kt(n)$ para escanear, e alguns para o controle). ■

Máquinas universais Uma fato importante é que existem máquinas de Turing *universais*. Uma máquina universal é capaz simular a execução de qualquer outra máquina de Turing M, dado uma representação $\langle M, x \rangle$ de M e da entrada x. A codificação de M consiste em uma lista de todas entradas é saídas da função de transição de M. É conveniente usar uma codificação com duas características: (i) Cada string em $\{0, 1\}^*$ representa alguma MT.

Caso a codificação é inválida, ele representa uma MT default. (ii) Cada MT possui um número infinito de representações. Isso é possível permitindo uma sequência de 1's no final da representação de M.

Teorema 17.1 (Arora e Barak (2009, Th. 1.9)**)**
Existe uma MTD U, tal que para cada $i, x \in \{0,1\}^*$ temos $U(i,x) = M_i(x)$ com M_i a MTD representada por i. Caso M_i para com entrada x em T passos, $U(i,x)$ para em $cT \log T$ passos, com C uma constante que é independente de $|x|$, e depende somente do tamanho do alfabeto, o número de fitas a número de estados de M_i.

Prova. Provaremos uma versão simplificada com tempo de simulação cT^2. Primeiro vamos construir uma MT U' com 5 fitas. Uma fita é a fita de entrada, uma fita representa a fita da máquina de Turing M simulada, uma fita contém a descrição de M, uma fita contém o estado atual de M, e a última fita é a fita de saída.
Para simular um passo de M, U' escaneia a fita com as regras de transição e o estado atual de M, para achar a regra de transição a ser aplicada, e depois executa essa regra. O trabalho para fazer isso é um número constante c' de passos por passo de M.
Para achar uma MTD U com uma única fita, podemos aplicar proposição 17.2. O número de passos de U então é limitado por cT^2 com $c = 25c'$. ■

Observação 17.3
Uma simulação de MTND é possível com os mesmos limites. ♢

Exemplo 17.2 (Máquina universal)
Considere a máquina $M = (\{u,d\},\{a,b\},\{a,b,\},\delta)$ com

$$\delta = \{ua \to ubL, ub \to uaL, u \to dbR, da \to uR, db \to daR, d \to uaL\}.$$

Essa máquina é universal? Ver http://www.wolframscience.com/prizes/tm23. Aparentemente o problema foi resolvido em outubro de 2007. ♢

Computabilidade e complexidade

Decibilidade versus complexidade

- Qual é o poder computacional?
- Surpreendentemente (?), vários problemas não são decidíveis.

- Exemplo: O "Entscheidungsproblem" de Hilbert, o problema de parada, etc.
- A equivalência dos modelos significa que o modelo concreto não importa?
- Sim para computabilidade, não para complexidade!

Exemplo de um modelo diferente: A máquina de RAM.

A máquina RAM

A *máquina RAM* (random access machine) é o modelo padrão para análise de algoritmos. Ela possui

- um processador com um ou mais registros, e com apontador de instruções,
- uma memória infinita de números inteiros e
- instruções elementares (controle,transferência inclusive endereçamento indireto,aritmética).

A máquina RAM

Existem RAMs com diferentes tipos de instruções aritméticas

- *SRAM*: somente sucessor
- *RAM*: adição e subtração
- *MRAM*: multiplicação e divisão

e com diferentes tipos de custos

- Custo *uniforme*: cada operação em $O(1)$
- Custo *logarítmico*: proporcional ao número de bits dos operandos

Exemplos de simulação

Teorema 17.2 (Leeuwen (1990))

$$\begin{aligned}
\mathrm{m-tapes} &\leq 1-\mathrm{tape}(\mathit{time}\ \mathrm{k}n^2\ \&\ \mathit{space}\ \mathit{Lin}) \\
\mathrm{m-tapes} &\leq 2-\mathrm{tape}(\mathit{time}\ \mathrm{k}n \log n\ \&\ \mathit{space}\ \mathit{Lin}) \\
\mathrm{SRAM-UTIME} &\leq \mathsf{T}(\mathit{time}\ n^2 \log n) \\
\mathrm{RAM-UTIME} &\leq \mathsf{T}(\mathit{time}\ n^3) \\
\mathrm{MRAM-UTIME} &\leq \mathsf{T}(\mathit{time}\ \mathit{Exp}) \\
\mathrm{SRAM-LTIME} &\leq \mathsf{T}(\mathit{time}\ n^2) \\
\mathrm{RAM-LTIME} &\leq \mathsf{T}(\mathit{time}\ n^2) \\
\mathrm{MRAM-LTIME} &\leq \mathsf{T}(\mathit{time}\ \mathit{Poly})
\end{aligned}$$

Robustez da complexidade

Tese estendida de Church-Turing
Qualquer modelo de computação universal é equivalente à máquina de Turing com

- custo adicional de tempo no máximo polinomial
- custo adicional de espaço no máximo constante

- Equivalência definido por simulação mutual.
- Verdadeiro para *quase* todos modelos conhecidos:

  Maquina de Turing, cálculo lambda, máquina RAM, máquina pontador, circuitos lógicos, autômatos celulares (Conway), avaliação de templates em C++, computador billiard, ...

- Computador quântico?

Consequência: Shor's trilemma

Ou

- a tese estendida de Church-Turing é errada, ou
- a física quântica atual é errada, ou
- existe um algoritmo de fatoração clássico rápido.

18. Classes de complexidade

18.1. Definições básicas

Complexidade pessimista

- Recursos básicos: *tempo* e *espaço*.
- A *complexidade de tempo (pessimista)* é uma função

$$t : \mathbb{N} \to \mathbb{N}$$

tal que $t(n)$ é o número máximo de passos para entradas de tamanho n.

- A *complexidade de espaço (pessimista)* é uma função

$$s : \mathbb{N} \to \mathbb{N}$$

tal que $s(n)$ é o número máximo de posições usadas para entradas de tamanho n.

- Uma MTND tem complexidades de tempo $t(n)$ ou espaço $s(n)$, se essas funções são limites superiores para todas computações possíveis de tamanho n.

Funções construtíveis

- No que segue, consideraremos somente funções t e s que são *tempo-construtíveis* e *espaço-construtíveis*.

  Definição 18.1
  Uma função $t(n)$ é tempo-construtível caso existe uma MTD com complexidade de tempo $t(n)$ que precisa $t(n)$ passos para alguma entrada de tamanho n. Uma função $s(n)$ é espaço-construtível caso existe uma MTD com complexidade de espaço $s(n)$ que precisa $s(n)$ posições para alguma entrada de tamanho n.

- Exemplos: $n^k, \log n, 2^n, n!, \ldots$.

Observação 18.1
A restrição para funções tempo- ou espaço-construtíveis exclui funções não-computáveis ou difíceis de computar e assim permite uma simulação eficiente por outras MT. Existem funções que não são espaço-construtíveis; um exemplo simples é uma função que não é computável; um outro exemplo é $\lceil \log \log n \rceil$. ◊

Classes de complexidade fundamentais

- Uma *classe de complexidade* é um conjunto de linguagens.
- Classes fundamentais: Para $t, s : \mathbb{N} \to \mathbb{N}$ e um problema $L \subseteq \Sigma^*$
  - $L \in \mathsf{DTIME}[t(n)]$ se existe uma máquina Turing determinística tal que aceita L com complexidade de tempo $t(n)$.
  - $L \in \mathsf{NTIME}[t(n)]$ se existe uma máquina Turing não-determinística que aceita L com complexidade de tempo $t(n)$.
  - $L \in \mathsf{DSPACE}[s(n)]$ se existe uma máquina Turing determinística que aceita L com complexidade de espaço $s(n)$.
  - $L \in \mathsf{NSPACE}[s(n)]$ se existe uma máquina Turing não-determinística que aceita L com complexidade de espaço $s(n)$.

Hierarquia básica

- Observação

$$\mathsf{DTIME}[F(n)] \subseteq \mathsf{NTIME}[F(n)] \subseteq \mathsf{DSPACE}[F(n)] \subseteq \mathsf{NSPACE}[F(n)]$$

- Definições conhecidas:

$$\mathsf{P} = \bigcup_{k \geq 0} \mathsf{DTIME}[n^k]; \quad \mathsf{NP} = \bigcup_{k \geq 0} \mathsf{NTIME}[n^k]$$

- Definições similares para espaço:

$$\mathsf{PSPACE} = \bigcup_{k \geq 0} \mathsf{DSPACE}[n^k]; \quad \mathsf{NSPACE} = \bigcup_{k \geq 0} \mathsf{NSPACE}[n^k]$$

- Com a observação acima, temos

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{DSPACE} \subseteq \mathsf{NSPACE}\,.$$

Prova. (Da observação.) Como uma máquina não-determinística é uma extensão da uma máquina determinística, temos obviamente $\mathsf{DTIME}[F(n)] \subseteq \mathsf{NTIME}[F(n)]$ e $\mathsf{DSPACE}[F(n)] \subseteq \mathsf{NSPACE}[F(n)]$. A inclusão $\mathsf{NTIME}[F(n)] \subseteq \mathsf{DSPACE}[F(n)]$ segue, porque todas computações que precisam menos que $F(n)$ passos, precisam menos que $F(n)$ espaço também. ■

Classes de complexidade

Zoológico de complexidade

18.2. Hierarquias básicas

Aceleração

Teorema 18.1
Podemos comprimir ou acelerar computações por um fator constante. Para todo $c > 0$ no caso de espaço temos

$$L \in \mathsf{DSPACE}[s(n)] \Rightarrow L \in \mathsf{DSPACE}[cs(n)]$$
$$L \in \mathsf{NSPACE}[s(n)] \Rightarrow L \in \mathsf{NSPACE}[cs(n)]$$

e no caso do tempo, para máquinas de Turing com $k > 1$ fitas e $t(n) = \omega(n)$

$$L \in \mathsf{DTIME}[s(n)] \Rightarrow L \in \mathsf{DTIME}[cs(n)]$$
$$L \in \mathsf{NTIME}[s(n)] \Rightarrow L \in \mathsf{NTIME}[cs(n)]$$

Prova. (Rascunho.) A ideia é construir uma MT M' que simula uma MT M executando m passos em um passo. M' inicialmente copia a entrada para uma outra fita, codificando cada m símbolos em um símbolo em tempo $n + \lceil n/m \rceil$. Depois, em cada passo da simulação, M' leia os símbolos na esquerda e direta e na posição atual em tempo 4. Depois ela calcula os novos estados no controle finito, e escreve os três símbolos novos em tempo 4. Logo, cada m passos podem ser simulados em 8 passos de tempo. A simulação total precisa tempo

$$n + \lceil n/m \rceil + \lceil 8t(n)/m \rceil \leq n + n/m + 8t(n)/m + 2 \leq 3n + 8t(n)/m,$$

que para $cm \geq 16 \Leftrightarrow 8/m \leq c/2$ e n suficientemente grande não ultrapassa $ct(n)$. O número finito de palavras que não satisfazem esse limite superior é reconhecido diretamente no controle finito. ■

Hierarquia de tempo (1)

- É possível que a decisão de todos problemas tem um limite superior (em termos de tempo ou espaço)? Não.

Teorema 18.2
Para $t(n)$ e $s(n)$ total e recursivo, existe um linguagem L tal que $L \notin \mathsf{DTIME}[t(n)]$ ou $L \notin \mathsf{DSPACE}[s(n)]$, respectivamente.

Prova. (Rascunho). Por diagonalização. As máquinas de Turing são enumeráveis: seja $M_1, M_2, \ldots$ uma enumeração deles e seja $x_1, x_2, \ldots$ uma enumeração das palavras em Σ^*. Define

$$L = \{x_i \mid M_i \text{ não aceita } x_i \text{ em tempo } t(|x_i|)\}.$$

Essa linguagem é decidível: Uma MT primeiramente calcula $t(|x_i|)$ (que é possível porque $t(n)$ é recursivo e total.). Depois com entrada x_i, ela determina i e a máquina M_i correspondente e simula M_i $t(|x_i|)$ passos. Se M_i aceita, ela rejeita, senão ela aceita.
Essa linguagem não pertence a $\mathsf{DTIME}[t(n)]$. Prova por contradição: Seja $L = L(M_i)$. Então $x_i \in L$? Caso sim, M_i não aceita em tempo $t(|x_i|)$, uma contradição. Caso não, M_i não aceita em tempo $t(|x_i|)$, e portanto $x_i \in L$, outra contradição. ■

Hierarquia de tempo (2)
Além disso, as hierarquias de tempo são "razoavelmente densos":

Teorema 18.3 (Hartmanis,Stearns, 1965)
Para f, g com g tempo-construtível e $f \log f = o(g)$ temos

$$\mathsf{DTIME}(f) \subsetneq \mathsf{DTIME}(g).$$

Para funções f, g, com $g(n) \geq \log_2 n$ espaço-construtível e $f = o(g)$ temos

$$\mathsf{DSPACE}(f) \subsetneq \mathsf{DSPACE}(g).$$

Prova. (Rascunho.) Para provar o segundo parte (que é mais fácil) temos que mostrar que existe uma linguagem $L \subseteq \Sigma^*$ tal que $L \in \mathsf{DSPACE}[g]$ mas $L \notin \mathsf{DSPACE}[f]$. Vamos construir uma MT M sobre alfabeto de entrada $\Sigma = \{0, 1\}$ tal que $L(M)$ satisfaz essas característica. A ideia básica é diagonalização: com entrada w simula a máquina M_w sobre w e garante nunca reconhecer a mesma linguagem que M_w caso ela é limitada por f.
Para realizer essa ideia:

(i) Temos que garantir que M precisa não mais que $g(n)$ espaço. Portanto, M começa de marcar $g(|w|)$ espaço no começo (isso é possível porque g é espaço-construtível). Caso a simulação ultrapassa o espaço marcado, M rejeita.

(ii) Nos temos que garantir que M pode simular todas máquinas que tem limite de espaço $f(n)$. Isso tem duas problemas (a) M possui um alfabeto de fita fixo, mas a máquina simulada pode ter mais símbolos de fita. Portanto, a simulação precisa um fator c_1 de espaço a mais. (b) Por definição, para $f \in o(g)$ é suficiente que $f \leq cg$ a partir de um $n > n_0$. Logo para entradas $|w| \leq n_0$ o espaço $g(n)$ pode ser insuficiente para simular qualquer máquina que precisa espaço $f(n)$. Esses problemas podem ser resolvidos usando uma enumeração de MT (com alfabeto Σ) tal que cada máquina possui codificações de comprimento arbitrário (por exemplo permitindo $\langle M \rangle 10^n$).

(iii) Além disso, temos que garantir que a simulação para. Portanto M usa um contador com $O(\log f(n))$ espaço, e rejeita caso a simulação ultrapassa $c_2^{f(n)}$ passos; c_2 depende das características da máquina simulada (número de estados, etc.): com isso podemos escolher uma máquina

Com essas preparações, com entrada w, M construa M_w, verifica que M_w é uma codificação de uma MT e depois simula M_w com entrada w. M rejeita se M_w aceita e aceita se M_w rejeita.
Não existe uma MT que, em espaço $f(n)$ reconhece $L(M)$: Supõe que M' seria máquina com essa característica. Com entrada $w = \langle M' \rangle 01^n$ para n suficientemente grande M consegue de simular M' e portanto, se $w \in M'$ então $w \notin M$ e se $w \notin M'$ então $w \in M$, uma contradição.
A ideia da prova do primeiro parte é essencialmente a mesma. O fator de $\log f$ surge, porque para simular um MT para um número de passos, é necessário contar o número de passos até $f(n)$ em $\log f(n)$ bits. Com uma simulação mais eficiente (que não é conhecida) seria possível de obter um teorema mais forte. ■

Espaço polinomial

Teorema 18.4 (Savitch)
Para cada função espaço-construtível $s(n) \geq \log_2 n$

$$\mathsf{NSPACE}[s(n)] \subseteq \mathsf{DSPACE}[s(n)^2]$$

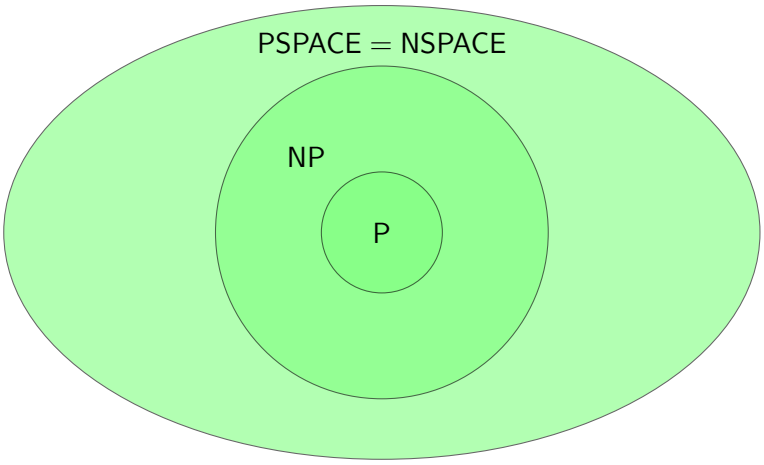
- Corolário: $\mathsf{DSPACE} = \mathsf{NSPACE}$

Figura 18.1.: Walter J. Savitch (*1943)

- Não-determinismo ajuda pouco para espaço!

Prova. (Rascunho.) Caso $L \in \mathsf{NSPACE}[s(n)]$ o tempo é limitado por um $c^{s(n)}$. A construção do Savitch procura deterministicamente uma transição do estado inicial para um estado final com menos que $c^{s(n)}$ passos. A abordagem e por divisão e conquista: Para saber se existe uma transição $A \Rightarrow B$ com menos de 2^i passos, vamos determinar se existem transições $A \Rightarrow I$ e $I \Rightarrow B$ que passam por um estado intermediário I, cada um com 2^{i-1} passos. Testando isso todas configurações I que precisam espaço menos que $s(n)$. A altura da árvore de busca resultante é $O(s(n))$ e o espaço necessário em cada nível também e $O(s(n))$, resultando em $O(s(n)^2)$ espaço total.
A função tem que ter $s(n) \geq \log_2 n$, por que para a simulação precisamos também gravar a posição de cabeça de entrada em cada nível, que precisa $\log_2 n$ bits. ■

Espaço polinomial (2)



18.3. Exercícios

Exercício 18.1
Dado uma máquina de Turing com oráculo para o problema de parada, é possível calcular a função do "castor ocupado" (ingl. busy beaver)?

19. Teoria de NP-completude

19.1. Caracterizações e problemas em NP

A hierarquia de Chomsky classifica linguagens em termos de autômatos e gramáticas que aceitam ou produzem elas:

Linguagem	Nome	Tipo	Autômato	Gramática
Regular	REG	3	Autômato finito (determinístico)	Regular
Livre de contexto	CFL	2	Autômato de pilha (não-determinístico)	Livre de contexto
Sensitiva ao contexto	CSL	1	MT linearmente limitada (não-determinístico)	Sensitiva ao contexto
Recursivamente enumerável	RE	0	MT	Sistema semi-Thue (sem restrição)

O seguinte teorema relaciona a hierarquia de Chomsky com as classes de complexidade (sem prova, referências em (*Complexity zoo*), Atallah (1999, Th. 25.6) e Sipser (2006, Th. 7.16)).

Teorema 19.1 (Complexidade das linguagens da hierarquia de Chomsky)

$$\begin{aligned}
\mathsf{REG} &= \mathsf{DSPACE}[O(1)] = \mathsf{DSPACE}[o(\log\log n)] \\
\mathsf{REG} &\subseteq \mathsf{DTIME}[n] \\
\mathsf{CFL} &\subseteq \mathsf{DSPACE}[n^3] \\
\mathsf{CSL} &= \mathsf{NSPACE}[n]
\end{aligned}$$

Normalmente, nosso interesse são soluções, não decisões: Ao seguir vamos definir P e NP em termos de soluções. As perguntas centrais como $\mathsf{P} \neq \mathsf{NP}$ acabam de ter respostas equivalentes.

P e NP em termos de busca

- A computação de uma solução pode ser vista como função $\Sigma^* \to \Sigma^*$
- Exemplo: Problema SAT construtivo: Uma solução é uma atribuição.
- Definição alternativa: Uma computação é uma relação $R \subseteq \Sigma^* \times \Sigma^*$.

- Vantagem: Permite mais que uma solução para cada entrada.
- Intuição: Uma relação é a especificação de um problema de busca: para entrada x queremos achar alguma solução y tal que $(x, y) \in R$.
- Nosso interesse são soluções que podem ser "escritas" em tempo polinomial:
  Definição 19.1
  Uma relação binária R é *polinomialmente limitada* se

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

P e NP em termos de busca

- A definição de P e NP é como classes de problemas de decisão.
- A linguagem correspondente com uma relação R é

$$L_R = \{x \mid \exists y : (x, y) \in R\}$$

- A classe P: Linguagens L_R tal que existe uma MTD que, com entrada $x \in L_R$, em tempo polinomial, busque $(x, y) \in R$ ou responda, que não tem.
- Essa definição do P às vezes é chamado FP ou PF.
- A classe NP: Linguagens L_R tal que existe MTD que, com entrada (x, y), decide se $(x, y) \in R$ em tempo polinomial. y se chama um *certificado*.

A restrição para problemas de decisão facilita o tratamento teórico, mas não é importante para a tratabilidade de problemas.

Teorema 19.2 ((Trevisan 2010, Th. 2),(Arora e Barak 2009, Th. 2.18)**)**
Para cada problema de busca definido por uma relação polinomialmente limitada R, existe um problema de decisão tal que, caso o problema de decisão pode ser resolvido em tempo $t(n)$, o problema de busca pode ser resolvido em tempo $n^{O(1)}t(n)$. Em particular, $P = NP$ sse cada problema de busca possui solução em tempo polinomial.

Prova. Para a relação R, considera o problema de decidir, para entrada x, w, se existe um z tal que $(x, w \circ z) \in R$. Supõe que temos um algoritmo A que resolve este problema em tempo $t(n)$. Então podemos construir o seguinte algoritmo, para entrada x

```
if A(x, ε) = não then
  return ''não existe solução''
end if
w := ε
while (x, w) ∉ R do
  if A(x, w ∘ 0) = sim then
    w := w ∘ 0
  else
    w := w ∘ 1
  end if
end while
return w
```

É simples de ver que este algoritmo acha uma solução para o problema de busca, caso existe uma, construindo-a símbolo por símbolo. Como R é polinomialmente limitado, uma solução possui no máximo um número polinomial de símbolos, e por isso o algoritmo o número de chamadas de A é não mais que polinomial no tamanho da entrada. ■

Exemplos de problemas em NP

$$\text{CLIQUE} = \{\langle G, k\rangle \mid \text{Grafo não-direcionado } G \text{ com clique de tamanho } k\}$$
$$\text{SAT} = \{\langle \phi\rangle \mid \phi \text{ fórmula satisfatível da lógica proposicional em FNC}\}$$
$$\text{TSP} = \{\langle M, b\rangle \mid \text{Matriz simétrica de distâncias } M \text{ que tem TSP-cíclo} \leq b\}$$
$$\text{COMPOSITE} = \{\langle n\rangle \mid n = m_1 m_2 \text{ com } m_1, m_2 > 1\}$$

19.2. Reduções

Reduções

Definição 19.2 (Redução em tempo polinomial)
Uma (many-one) *redução* entre duas linguagens L, L' com alfabetos Σ e Σ' é um função total $f : \Sigma^* \to \Sigma'^*$ tal que $x \in L \iff f(x) \in L'$. Se f é computável em tempo polinomial, se chama uma *redução em tempo polinomial*; escrevemos $L \leq_P L'$.

Definição 19.3 (Problemas difíceis e completos)
Dado uma classe de problemas C e um tipo de redução $\leq$, um problema L é C-$\leq$-*difícil*, se $L' \leq L$ para todos $L' \in C$. Um problema L que é C-$\leq$-difícil é *completo*, se $L \in C$.

- Motivo: Estudar a complexidade *relativa* de problemas; achar problemas "difíceis" para separar classes.
- Do interesse particular: A separação de P e NP. Denotamos a classe de problemas NP-completos NPC.

Características de $\leq_P$

Proposição 19.1 (Fecho para baixo)
Se $A \leq_P B$ e $B \in P$ então $A \in P$.

Proposição 19.2 (Transitividade)
$\leq_P$ é transitivo, i.e. se $A \leq_P B$ e $B \leq_P C$ então $A \leq_P C$.

Prova. (Fecho para baixo.) Uma instância $w \in A$ pode ser reduzido em tempo polinomial para $w' \in B$. Depois podemos simular B com entrada w' em tempo polinomial. Como a composição de polinômios é um polinômio, $A \in P$.
(Transitividade.) Com o mesmo argumento podemos reduzir $w \in A$ primeiro para $w' \in B$ e depois para $w'' \in C$, tudo em tempo polinomial. ■

O problema de parada

- O problema da parada

$$\text{HALT} = \{\langle M, w\rangle \mid \text{MT } M \text{ para com entrada } w\}$$

  não é decidível.

- Qual o caso com

Parada limitada (ingl. bounded halt)

Instância MT M, entrada w e um número n (em codificação unária).

Questão M para em n passos?

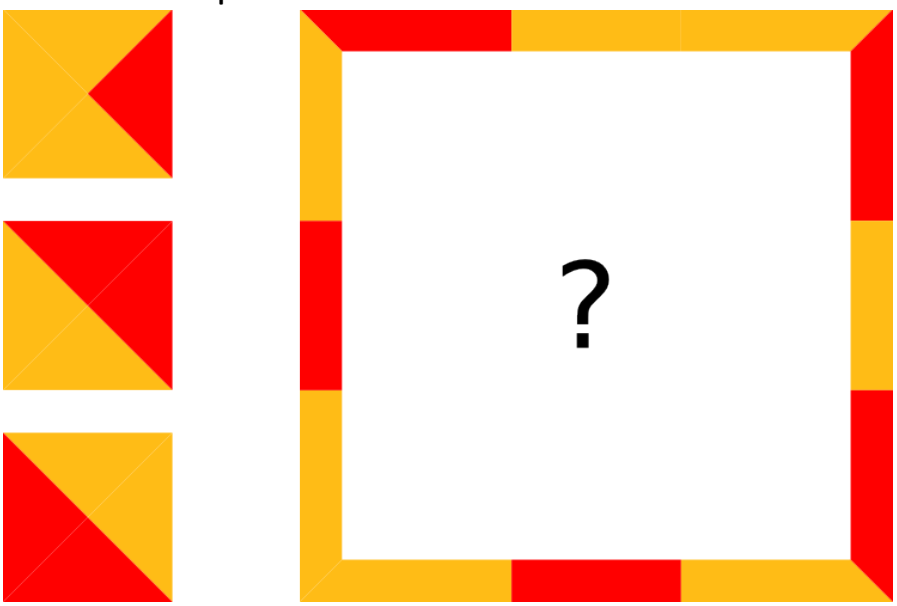
Teorema 19.3

$$\mathsf{BHALT} = \{\langle M, w, 1^n\rangle \mid \text{MT } M \text{ para com entrada } w \text{ em } n \text{ passos}\}$$

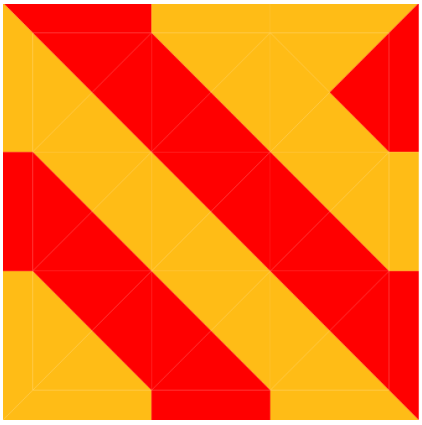
é NP-completo.

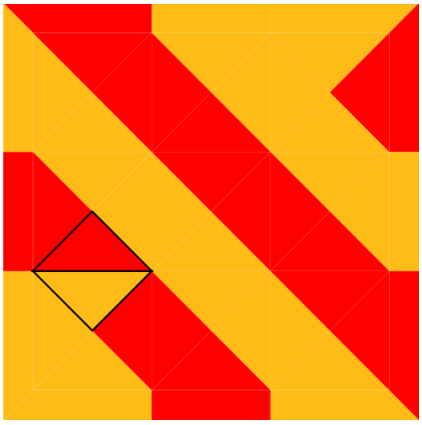
Prova. BHALT $\in$ NP porque podemos verificar uma execução em n passos em tempo polinomial. Observe que a codificação de uma execução em limitada polinomialmente em termos da entrada $\langle M, w, 1^n\rangle$ pela codificação de n em unário. Logo é suficiente de mostrar que qualquer problema em NP pode ser reduzido para BHALT.
Para alguma linguagem $L \in$ NP, seja M uma MTND com $L = L(M)$ que aceita L em tempo n^k. Podemos reduzir uma entrada $w \in L$ em tempo polinomial para $w' = \langle M, w, 1^{n^k}\rangle$, temos $w \in L \Leftrightarrow w' \in$ BHALT. Logo $L \leq_P$ BHALT. ■

Ladrilhar: Exemplo

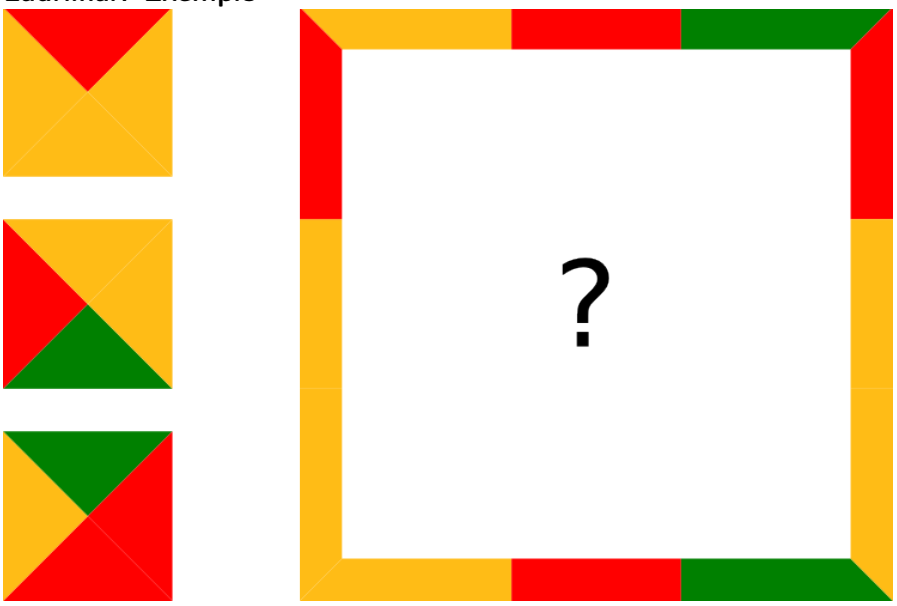


Ladrilhar: Solução





Ladrilhar: Exemplo



Ladrilhar: O problema

- Para um conjunto finito de cores C, o *tipo* de um ladrilho é uma função

$$t : \{N, W, S, E\} \to C.$$

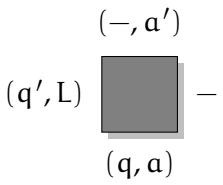
LADRILHAMENTO

Instância Tipos de ladrilhos $t_1, \ldots, t_k$ e uma grade de tamanho $n \times n$ com cores nas bordas. (Cada ladrilho pode ser representado por quatro símbolos para as cores; a grade consiste em n^2 ladrilhos em branco e $4n$ cores, logo uma instância tem tamanho $O(k + n^2)$.)

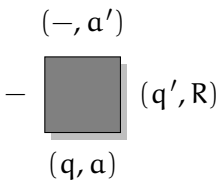
Questão Existe um ladrilhamento da grade tal que todas cores casam (sem girar os ladrilhos)?

Teorema 19.4 (Levin)
Ladrilhamento é NP-completo.

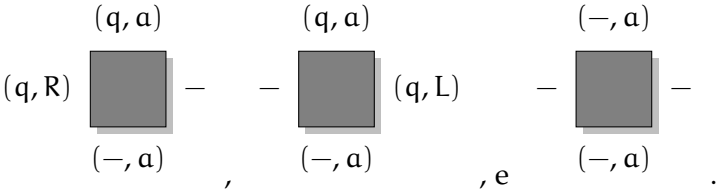
Prova. O problema é em NP, porque dado um conjunto de tipos de ladrilhos e um ladrilhamento, podemos verificar as restrições das cores em tempo polinomial.
Vamos reduzir qualquer problema em $L \in NP$ para LADRILHAMENTO. Seja $L = L(M)$ para alguma MTND e seja k tal que M precisa tempo n^k. Para entrada w, vamos construir uma instância de LADRILHAMENTO do tamanho $(|w|^k)^2$. Ideia: os cores dos cantos de sul e de norte codificam um símbolo da fita, a informação se a cabeça está presente, e o estado da máquina. Os cores dos cantos oeste e este codificam informação adicional para mover a cabeça. O canto sul da grade será colorida conforme o estado inicial da máquina, o canto norte com o estado final e vamos projetor as ladrilhas de tal forma que ladrilhar uma linha (de sul para o norte) e somente possível, se as cores no sul e norte representam configurações sucessores.
Vamos usar as cores $(Q \cup \{-\}) \times \Gamma$ na direção norte/sul e $(Q \times \{L, R\}) \cup \{-\}$ na direção oeste/este. Para uma regra $q, a \to q', a', L$ os ladrilhos tem a forma



e para $q, a \to q', a', R$



Além disso, tem ladrilhos



, , e .

As cores no sul da grade representam a configuração inicial

$$(q_0, a_1)(-, a_2) \cdots (-, a_n)(-,) \cdots (-,)$$

as cores no norte a configuração final (supondo que a máquina limpa a fita depois, que sempre é possível)

$$(q_a, -)(-,) \cdots (-,)$$

e as cores dos lados oeste e este todos são $-$. Pela construção uma computação da MT que aceita corresponde com um ladrilhamento e vice versa. A construção do grade e das tipos de ladrilhos pode ser computado por uma máquina de Turing em tempo polinomial. ■

Resultado intermediário

- Primeiros problemas em NPC: Para uma separação é "só" provar que Ladrilhamento $\notin$ P ou BHALT $\notin$ P.
- Infelizmente: a prova é difícil, mesmo que a maioria das pesquisadores acredita P $\neq$ NP.
- Outro valor: Para provar que um problema L $\in$ NPC, é suficiente de mostrar que, por exemplo

$$\text{Ladrilhamento} \leq_P L.$$

Proposição 19.3
Se A $\subseteq$ B e A é fechado para baixo em relação à redução $\leq$ e L e B-$\leq$-completo então

$$L \in A \iff A = B.$$

Exemplo: O problema SAT

SAT

Instância Fórmula proposicional em forma normal conjuntiva $\Phi(x_1, \ldots, x_n$

Questão Tem uma atribuição $a_1, \ldots, a_n \in \mathbb{B}$ que satisfaz Φ?

Teorema 19.5 (Cook)
SAT é NP-completo.

Prova (1)

Objetivo: Provar Ladrilhamento $\leq_P$ SAT.
Seja

$N_{x,y,c}$ variável "o norte da posição x, y tem cor c"
S, W, E analogamente

$$L_{i,x,y} := N_{x,y,t_i(N)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(N)}} \neg N_{x,y,c}$$
$$\wedge W_{x,y,t_i(W)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(W)}} \neg W_{x,y,c}$$
$$\wedge S_{x,y,t_i(S)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(S)}} \neg S_{x,y,c}$$
$$\wedge E_{x,y,t_i(E)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(E)}} \neg E_{x,y,c}$$

Prova (2)

Sejam $c_{x,y}$ as cores na bordas. Seja ϕ a conjunção de

$$\bigwedge_{x\in[1,n]}\bigwedge_{y\in[1,n]}\bigvee_{c\in[1,k]} L_{c,x,y} \quad \text{Toda posição tem um ladrilho}$$

$$\bigwedge_{x\in[1,n]} S_{x,1,c_{x,1}} \wedge N_{x,n,c_{x,n}} \quad \text{Cores corretas nas bordas N,S}$$

$$\bigwedge_{y\in[1,n]} W_{1,y,c_{1,y}} \wedge E_{n,y,c_{n,y}} \quad \text{Cores corretas nas bordas W,E}$$

$$\bigwedge_{x\in[1,n[}\bigwedge_{y\in[1,n]} E_{x,y,c} \Rightarrow W_{x+1,y,c} \quad \text{Correspondência E-W}$$

$$\bigwedge_{x\in]1,n]}\bigwedge_{y\in[1,n]} W_{x,y,c} \Rightarrow E_{x-1,y,c} \quad \text{Correspondência W-E}$$

$$\bigwedge_{x\in[1,n[}\bigwedge_{y\in[1,n[} N_{x,y,c} \Rightarrow S_{x,y+1,c} \quad \text{Correspondência N-S}$$

$$\bigwedge_{x\in[1,n[}\bigwedge_{y\in]1,n]} S_{x,y,c} \Rightarrow N_{x,y-1,c} \quad \text{Correspondência S-N}$$

Prova (3)

- O número de variáveis e o tamanho de ϕ é polinomial em n, k; ϕ pode ser computado em tempo polinomial para uma instância de LADRILHAMENTO.
- Portanto, SAT é NP-difícil.
- SAT $\in$ NP, porque para fórmula ϕ e atribuição a, podemos verificar $a \models \phi$ em tempo polinomial.

O significado do P = NP

Kurt Gödel 1958: Uma carta para John von Neumann

Obviamente, podemos construir uma máquina de Turing, que decide, para cada fórmula F da lógica de predicados de primeira ordem e cada número natural n, se F tem uma prova do tamanho n (tamanho = número de símbolos). Seja $\Phi(F, n)$ o número de passos que a máquina precisa para isso, e seja $\Psi(n) = \max_F \Phi(F, n)$. A questão é como $\Phi(n)$ cresce para uma máquina ótima. É possível provar que $\Phi(n) \geq kn$. Se existisse uma máquina com $\Phi(n) \sim kn$ (ou pelo menos $\Phi(n) \sim kn^2$), isso teria consequências da maior

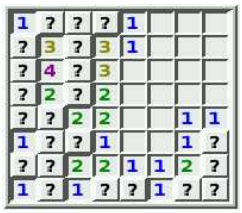
importância. Assim, seria óbvio, apesar da indecibilidade do Entscheidungsproblem, poderia-se substituir completamente o raciocínio do matemático em questões de sim-ou-não por máquinas. (Sipser 1992)
Em original a carta diz

> Man kann offenbar leicht eine Turingmaschine konstruieren, welche von jeder Formel F des engeren Funktionenkalküls u. jeder natürl. Zahl n zu entscheiden gestattet, ob F einen Beweis der Länge n hat [Länge = Anzahl der Symbole]. Sei $\Psi(F, n)$ die Anzahl der Schritte, die die Maschine dazu benötigt u. sei $\phi(n) = \max_F \Psi(F, n)$. Die Frage ist, wie rasch $\phi(n)$ für eine optimale Maschine wächst. Man kann zeigen $\phi(n) \geq k \cdot n$. Wenn es wirklich eine Maschine mit $\phi(n) \sim k \cdot n$ (oder auch nur $\sim k \cdot n^2$) gäbe, hätte das Folgerungen von der grössten Tragweite. Es würde nämlich offenbar bedeuten, dass man trotz der Unlösbarkeit des Entscheidungsproblems die Denkarbeit des Mathematikers bei ja-oder-nein Fragen vollständig durch Maschinen ersetzen könnte.

O significado do $P = NP$

- Centenas de problemas NP-completos conhecidos seriam tratável.
- Todos os problemas cujas soluções são reconhecidas facilmente (polinomial), teriam uma solução fácil.
- Por exemplo na inteligência artificial: planejamento, reconhecimento de linguagens naturais, visão, talvez também composição de música, escrever ficção.
- A criptografia conhecida, baseada em complexidade, seria impossível.

> I have heard it said, with a straight face, that a proof of $P = NP$ would be important because it would airlines schedule their flight better, or shipping companies pack more boxes in their trucks! (Aaronson 2005)



Mais um problema NP-completo

MINESWEEPER CONSISTÊNCIA

Instância Uma matriz de tamanho $b \times b$ cada campo ou livre, ou com um número ou escondido.

Decisão A matriz é consistente (tem uma configuração dos campos escondidos, que podem ser "bombas" ou livres tal que os números são corretos)?

O mundo agora

- O milagre da NP-completude
  - Qualquer problema em NP tem uma redução polinomial para SAT!
  - Por que não usar só SAT? (soluções em 1.3^n)?

Teorema 19.6 (Ladner (1975))
- Se $P \neq NP$, existe uma linguagem $L \in NP$ que nem é NP-completo nem em P.

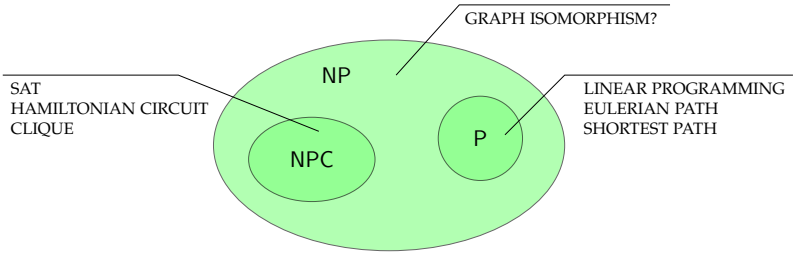


Figura 19.1.: Stephen Arthur Cook (*1939)

Muitos se interessavam

Woeginger's página sobre P vs. NP

Figura 19.2.: Leonid Levin (*1948)

19.3. Exercícios

Exercício 19.1
Mostra que a versão de decisão do seguinte problema é NP-completo: A entrada é uma instância do problema do caixeiro viajante e uma solução ótima do problema. Dado uma nova cidade e a distâncias correspondentes encontra a nova solução ótima.

20. Fora de NP

Classes fora de P–NP

$$\begin{aligned}
\mathsf{L} &= \mathsf{DSPACE}[\log n] \\
\mathsf{NL} &= \mathsf{NSPACE}[\log n] \\
\mathsf{EXPTIME} &= \bigcup_{k>0} \mathsf{DTIME}[2^{n^k}] \\
\mathsf{NEXPTIME} &= \bigcup_{k>0} \mathsf{NTIME}[2^{n^k}] \\
\mathsf{EXPSPACE} &= \bigcup_{k>0} \mathsf{DSPACE}[2^{n^k}] \\
\mathsf{NEXPSPACE} &= \bigcup_{k>0} \mathsf{NSPACE}[2^{n^k}]
\end{aligned}$$

Co-classes

Definição 20.1 (Co-classes)
Para uma linguagem L, a *linguagem complementar* é $\overline{L} = \Sigma^* \setminus L$. Para uma classe de complexidade C, a *co-classe* $\mathrm{co}-C = \{\overline{L} \mid L \in C\}$ e a classe das linguagens complementares.

Proposição 20.1
$P = \mathrm{co}-P$.

- Qual problema pertence à NP?

$$\overline{\mathsf{CLIQUE}}, \overline{\mathsf{SAT}}, \overline{\mathsf{TSP}}, \overline{\mathsf{COMPOSITE}}.$$

Prova. Seja $L \in P$. Logo existe um MTD M tal que $L = L(M)$ em tempo n^k. Podemos facilmente construir uma MTD que rejeita se M aceita e aceita se M rejeita. ■

Não sabemos se $\overline{\mathsf{CLIQUE}}, \overline{\mathsf{SAT}}, \overline{\mathsf{TSP}}$ pertencem à NP. Em 2002 foi provado, que $\overline{\mathsf{COMPOSITE}} \in P$ (Agrawal et al. 2004). Observe que se aplicas so para o *teste* se um número e primo ou não. O problema de fatorização é mais complicado.

A classe co-NP

- A definição da classe NP é unilateral. Por exemplo, considere

  TAUT

  Instância Fórmula proposicional em forma normal disjuntiva φ.

  Decisão φ é uma tautologia (todas as atribuições satisfazem φ)?

- Uma prova sucinta para esse problema não é conhecido, então *suponhamos* que $\mathsf{TAUT} \notin \mathsf{NP}$.

- Em outras palavras, NP parece de não ser fechado sobre a complementação:

$$\text{co-NP} \neq \text{NP?}$$

Proposição 20.2
Se $L \in \mathsf{NPC}$ então $L \in \text{co-NP} \iff \mathsf{NP} = \text{co-NP}$.

Proposição 20.3
TAUT é co-NP-completo.

Prova. (Proposição 20.2.) Seja $L \in \mathsf{NPC}$. ($\rightarrow$): Seja $L \in \text{co-NP}$. Se $L' \in \mathsf{NP}$, temos $L' \leq_P L \in \text{co-NP}$, logo $\mathsf{NP} \subseteq \text{co-NP}$. Se $L' \in \text{co-NP}$, então $\overline{L'} \in \mathsf{NP}$ e $\overline{L'} \leq_P L \in \text{co-NP}$, logo $\overline{L'} \in \text{co-NP}$ e $L' \in \mathsf{NP}$. ($\leftarrow$): Como $L \in \mathsf{NPC} \subseteq \mathsf{NP}$, e $\mathsf{NP} = \text{co-NP}$, também $L \in \text{co-NP}$. ■

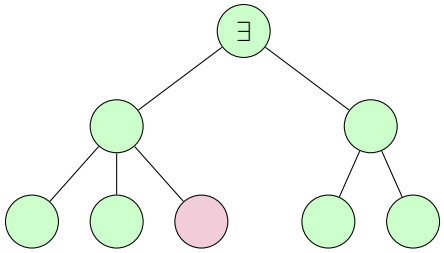
Prova. (Proposição 20.3, rascunho.) $\mathsf{TAUT} \in \text{co-NP}$, porque uma MT com um estado universal pode testar todas atribuições das variáveis proposicionais e aceita se todas são verdadeiras.
Para provar a completude, temos que provar, que toda linguagem $\overline{L} \in \text{co-NP} \leq_P \mathsf{TAUT}$. A prova é uma modificação da prova do teorema de Cook: Com entrada $w \in \overline{L}$ produzimos uma fórmula φ_w usando o método de Cook. Temos

$$\begin{aligned} w \in \overline{L} &\iff \varphi_w \text{ satisfatível} && \text{pela def. de } \varphi_w \\ w \in L &\iff \varphi_w \text{ insatisfatível} && \text{negação da afirmação} \\ &\iff \neg\varphi_w \text{ é tautologia} \end{aligned}$$

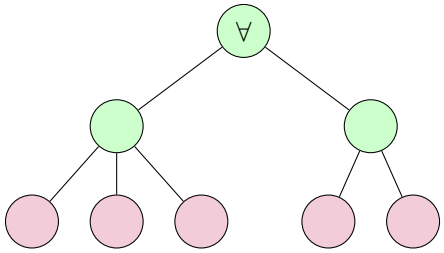
■

A classe co-NP

- NP: Resposta *sim*, se existe uma computação que responde *sim*.

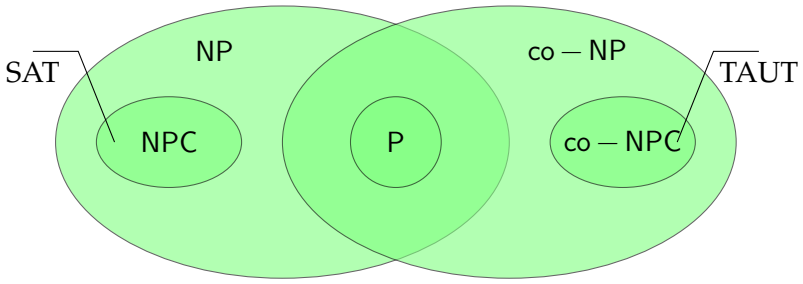


- Ou: Dado um certificado, *verificável* em tempo polinomial.
- co-NP: Resposta *sim*, se todas as computações respondem *sim*



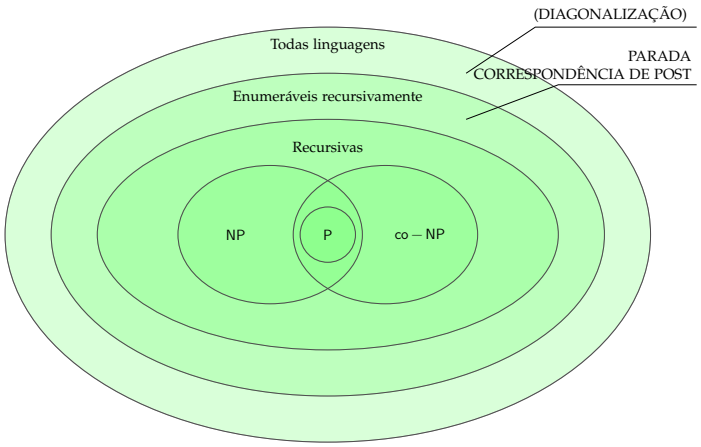
- Ou: Dado um "falsificado", *falsificável* em tempo polinomial.

O mundo da complexidade ao redor do P



20.1. De P até PSPACE

O mundo inteiro (2)



Problemas PSPACE-completos

- Não sabemos, se NP = PSPACE ou até P = PSPACE
- Como resolver isso? Da mesma forma que a questão P = NP: busque problemas PSPACE-completos (relativo a $\leq_P$).
- Considere

FORMULAS BOOLEANAS QUANTIFICADAS (INGL. QUANTIFIED BOOLEAN FORMULAS, QBF)

Instância Uma sentença booleana

$$\Phi := (Q_1 x_1)(Q_2 x_2) \cdots (Q_n x_n)[\varphi(x_1, x_2, \ldots, x_n)]$$

com $Q_i \in \{\forall, \exists\}$.

Decisão Φ é verdadeira?

- Exemplo:

$$(\forall x_1)(\exists x_2)(\forall x_3)(x_1 \vee x_3 \equiv x_2)$$

Teorema 20.1
QBF é PSPACE-completo.

Prova. (Rascunho.) É fácil de provar que QBF $\in$ PSPACE: Podemos verificar recursivamente que a sentença é verdadeira: Para uma fórmula $Qx_1\varphi(x_1, \ldots)$ com $Q \in \{\forall, \exists\}$ vamos aplicar o algoritmos para os casos $\varphi(0)$ e $\varphi(1)$.

Para provar a completude, temos que mostrar que toda linguagem $L \in$ PSPACE pode ser reduzido para QBF. Assume que existe uma MT que reconhece $L = L(M)$ em espaço n^k e seja $w \in L$. A ideia principal é construir uma fórmula $\phi_{q,s,t}$ que é verdadeira caso existe uma transição do estado q para s em no máximo t passos. Com isso podemos testar $\phi_{q_0,q_f,2^{cf(n)}}$ com $2^{cf(n)}$ sendo o número máximo de estados para entradas de tamanho n.
Um estado pode ser codificado por um string de $|w|^k$ bits. Para $\phi_{q,r,1}$ podemos usar basicamente a mesma fórmula do teorema de Cook. Para $t > 1$ a fórmula

$$\phi_{q,s,t} = \exists r(\phi_{q,r,t/2} \wedge \phi_{r,s,t/2})$$

é verdadeiro caso existe uma transição com estado intermediário r. Essa fórmula infelizmente tem t símbolos (que é demais para $2^{cf(n)}$), mas a fórmula

$$\phi_{q,s,t} = \exists r \forall (a,b) \in \{(q,r),(r,s)\}(\phi_{a,b,t/2})$$

evite a ocorrência dupla de ϕ a tem comprimento polinomial. ■

Outro exemplo

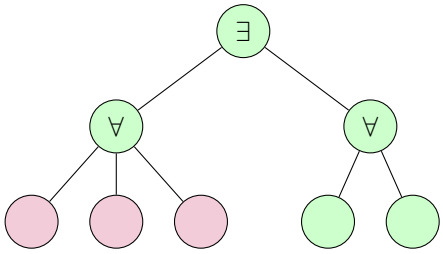
PALAVRA EM LINGUAGEM SENSÍVEL AO CONTEXTO

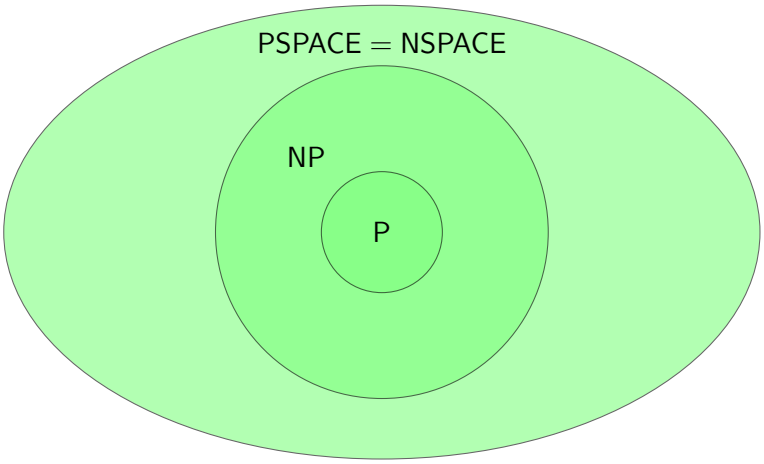
Instância Gramática Γ sensível ao contexto, palavra w.

Decisão $w \in L(\Gamma)$?

Mais quantificações

O que acontece, se nós permitimos mais quantificações?





A hierarquia polinomial

- Estendemos relações para aridade $i+1$. Uma relação $R \subseteq (\Sigma^*)^{i+1}$ é *limitada polinomial*, se

$$\forall(x, y_1, \ldots, y_i) \in R \, \exists p \in \text{poly} \, \forall i \, |y_i| \leq p(|x|)$$

- **Definição**: Σ_i é a classe das linguagens L, tal que existe uma relação de aridade $i+1$ que pode ser reconhecida em tempo polinomial, e

$$x \in L \iff \exists y_1 \forall y_2 \cdots Q_i : (x, y_1, \ldots, y_i) \in R$$

- **Definição**: Π_i é a classe das linguagens L, tal que existe uma relação de aridade $i+1$ que pode ser reconhecida em tempo polinomial, e

$$x \in L \iff \forall y_1 \exists y_2 \cdots Q_i : (x, y_1, \ldots, y_i) \in R$$

- As classes Σ_i e Π_i formam a *hierarquia polinomial*.
- Observação: $\Sigma_1 = \text{NP}$, $\Pi_1 = \text{co-NP}$.

Quantificações restritas ou não

- Conjunto das classes com quantificações restritas:

$$\text{PH} = \bigcup_{k \geq 0} \Sigma_k$$

- Classe das linguagens reconhecidas por um máquina de Turing com alternações sem limite: APTIME
- As máquinas correspondentes são *máquinas de Turing com alternação* com tempo $t(n)$: ATIME$[t(n)]$.

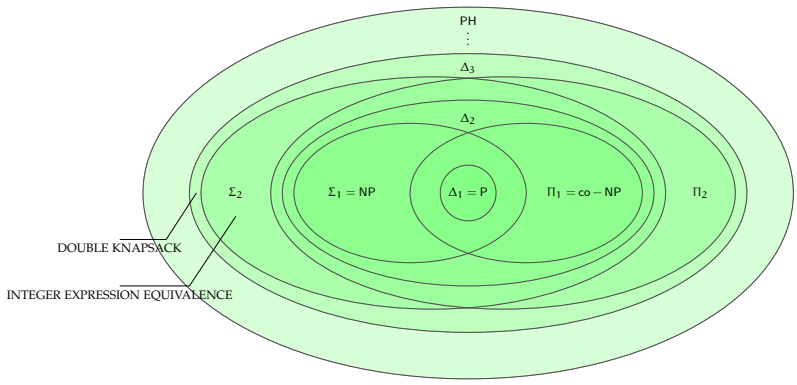
Teorema 20.2 (Chandra, Kozen, Stockmeyer)
Para $t(n) \geq n$

$$\mathsf{ATIME}[t(n)] \subseteq \mathsf{DSPACE}[t(n)] \subseteq \bigcup_{c>0} \mathsf{ATIME}[ct(n)^2].$$

Corolário 20.1
ATIME $=$ PSPACE

- Esta caracterização facilita entender por que QBF é PSPACE-completo

A hierarquia polinomial



Mais exemplos da classe PSPACE

- Observação: Uma questão com alternação é típica para resolver jogos.
- Ganhar um jogo em um passo: "Existe um passo tal que possa ganhar?"
- Ganhar um jogo em dois passos: "Existe um passo, tal que para todos os passos do adversário, existe um passo tal que possa ganhar?"
- Ganhar um jogo:

$$\exists p_1\, \forall p_2\, \exists p_3\, \forall p_4 \cdots \exists p_{2k+1} :$$

$p_1, p_2, p_3, \ldots, p_{2k+1}$é uma sequência de passos para ganhar.

- Portanto, vários jogos são PSPACE-completos: Generalized Hex, generalized Geography, ...

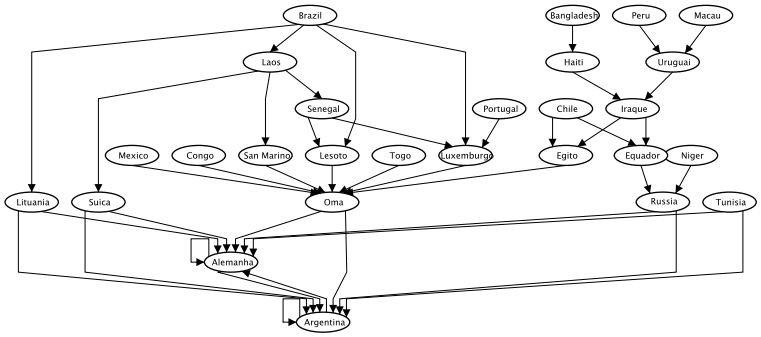
Mais exemplos da classe PSPACE (2)

Jogo de geografia para dois jogadores

> Alternadamente cada jogador diz o nome de um país. Cada nome tem que começar com a última letra do nome anterior. O primeiro jogador que não é capaz de dizer um novo país, perde.

Peru...

Mais exemplos da classe PSPACE (3)



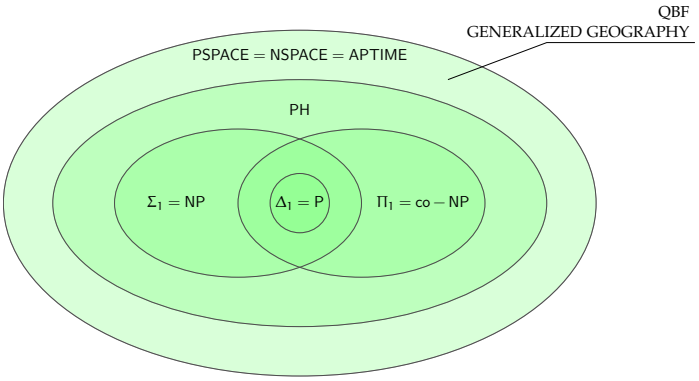
GEOGRAFIA GENERALIZADA (INGL. GENERALIZED GEOGRAPHY)

Instância Um grafo $G = (V, E)$ e um nó $v_0 \in V$

Decisão Jogando "geografia" com este grafo, o primeiro jogador pode ganhar com certeza?

Teorema 20.3
Geografia generalizada é PSPACE-completo.

O mundo até PSPACE



20.2. De PSPACE até ELEMENTAR

Problemas intratáveis demonstráveis

- Agora, consideramos os seguintes classes

$$EXP = DTIME[2^{n^{O(1)}}] = \bigcup_{k \geq 0} DTIME[2^{n^k}]$$

$$NEXP = NTIME[2^{n^{O(1)}}] = \bigcup_{k \geq 0} NTIME[2^{n^k}]$$

$$EXPSPACE = DSPACE[2^{n^{O(1)}}] = \bigcup_{k \geq 0} DSPACE[2^{n^k}]$$

- Estas classes são as primeiras demonstravelmente separadas de P.
- Consequência: Uma linguagem completa em EXP não é tratável.
- Exemplo de um problema EXP-completo:

Xadrez generalizada (ingl. Generalized chess)

Instância Uma configuração de xadrez com tabuleiro de tamanho $n \times n$.

Decisão Branco pode forçar o ganho?

Problemas ainda mais intratáveis

- As classes $k-\mathsf{EXP}$, $k-\mathsf{NEXP}$ e $k-\mathsf{EXPSPACE}$ tem k níveis de exponenciação!
- Por exemplo, considere a torre de dois de altura três: $2^{2^{2^k}}$

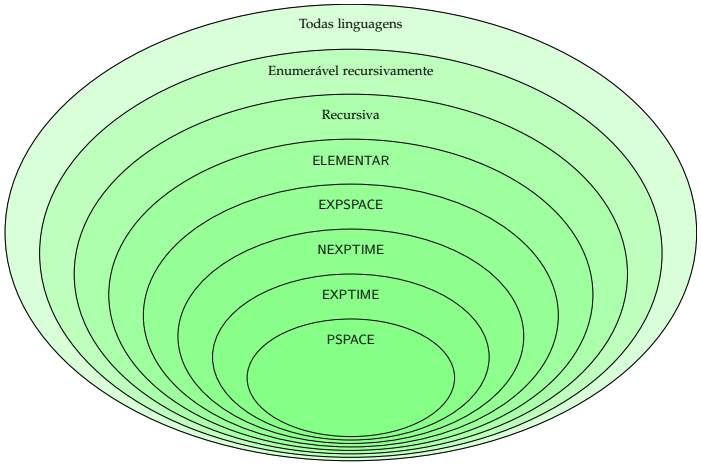
0	1	2	3	4
4	16	65536	$\approx 1.16 \times 10^{77}$	$\approx 2 \times 10^{19728}$

- Problemas desse tipo são *bem* intratáveis

$$\mathsf{ELEMENTAR} = \bigcup_{k \geq 0} k-\mathsf{EXP}$$

- Mas tem ainda problemas decidíveis fora desta classe!

O mundo até ELEMENTAR



Um corte final: Expressões regulares

- Uma expressão regular é
  - 0 ou 1 (denota o conjunto $L(0) = \{0\}$ e $L(1) = \{1\}$).
  - $e \circ f$, se $\circ$ é um operador, e e, f são expressões regulares.
- Operadores possíveis: $\cup$, $\cdot$, ^{2}, *, $\neg$.

- Decisão: Dadas as expressões regulares e, f, $L(e) \neq L(f)$?

Expressões regulares com	Completo para
$\cup, \cdot$	NP
$\cup, \cdot, ^*$	PSPACE
$\cup, \cdot, ^2$	NEXP
$\cup, \cdot, ^2, ^*$	EXPSPACE
$\cup, \cdot, \neg$	Fora do ELEMENTAR!

- O tempo do último problema de decisão cresce ao menos como uma torre de altura $\lg n$.

20.3. Exercícios

Exercício 20.1
Considera a seguinte prova que o problema de isomorfismo de grafos (GI) é PSPACE-completo:

> The equivalence problem for regular expressions was shown to be PSPACE-complete by Meyer e Stockmeyer (1972). Booth (1978) has shown that isomorphism of finite automata is equivalent to graph isomorphism. Taking these two results together with the equivalence of regular expressions, right-linear grammars, and finite automata (see Hopcroft et al. (2006) for example), shows that graph isomorphism is PSPACE-complete. (Delacorte 2007)

Sabendo que GI pertence a NP isso implicaria PSPACE = NP. Encontra o erro na prova.

21. Complexidade de circuitos

(As notas seguem Arora/Barak.)

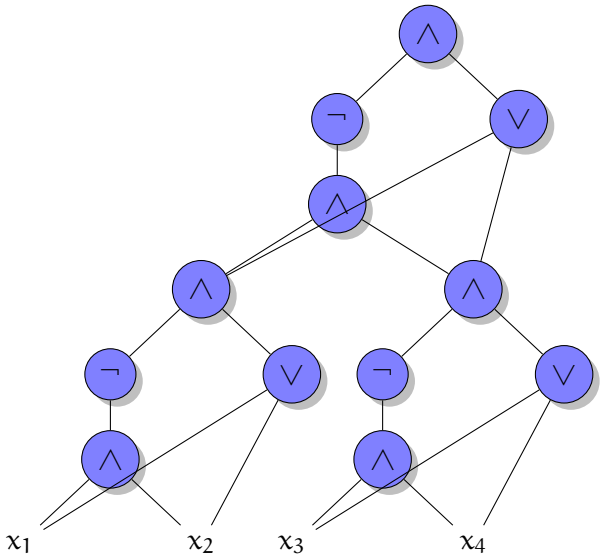
Um modelo alternativo de computação são *circuitos booleanos*. Circuitos tem a vantagem de ser matemáticamente mais simples que máquinas de Turing: o modelo de "execução" é a propagação dos valores de entrada na direção da saída, e nenhum elemento do circuito é alterado. Isso fez pesquisadores esperar que encontrar limitantes inferiores para complexidade de circuitos é mais simples. (Uma esperança que não se realizou até hoje.)

Definição 21.1 (Circuito booleano)
Para cada $n \in \mathbb{N}$ um *circuito* com n entradas e uma saída é um grafo direcionado acíclico $C = (V, E)$. O grau de entrada de um vértice se chama o seu *fan-in*, o grau de saída o *fan-out*. O grafo possui n *fontes* (vértices com fan-in 0) e um *destino* (com fan-out 0). Os restantes vértices são as *portas lógicas* rotulados com $\mathtt{land}$, $\vee$ ou $\neg$. Os vértices rotulados com $\wedge$ ou $\vee$ possuem *fan-in* 2 e os vértices rotulados com $\neg$ fan-in 1. O *tamanho* $|C|$ de C é igual ao número de vértices de C.
Para um circuito C e entrada $x \in \{0,1\}^n$ a saída correspondente é definida por $C(x) = v(d)$, com d o vértice destino é $v(d)$ é definido recursivamente por

$$v(d) = \begin{cases} v(e_1) \wedge v(e_2) & \text{caso } d \text{ é rotulado com } \wedge \text{ e } (e_1, d), (e_2, d) \in E \\ v(e_1) \vee v(e_2) & \text{caso } d \text{ é rotulado com } \vee \text{ e } (e_1, d), (e_2, d) \in E \\ \neg v(e) & \text{caso } d \text{ é rotulado com } \wedge \text{ e } (e, d) \in E \\ x_i & \text{case } d \text{ é a } i\text{-gésima entrada} \end{cases}$$

Observação 21.1
A definição permite um fan-out arbitráriamente grande. Um fan-in $k > 2$ pode ser implementado por uma cadeia de $k-1$ portas com fan-in 2. $\Diamond$

Exemplo 21.1
A função booleana simétrica $S_{1,3}(x_1, \ldots, x_4)$ é realizado pelo circuito



◊

Para estudar a complexidade assintótica, um único circuito não é suficiente, porque o número de entradas é fixo: temos que estudar famílias de circuitos.

Definição 21.2
Para uma função $t : \mathbb{N} \to \mathbb{N}$, uma *família de circuitos* de tamanho $t(n)$ é uma sequencia $\{C_n\}_{n\in\mathbb{N}}$ de circuitos booleanos. O circuito C_n tem n entradas e uma saída e tamanho $|C_n| \leq t(n)$. Uma linguagem L pertence à classe de complexidade $\mathsf{SIZE}(t(n))$ caso existe um família de circuitos de tamanho $t(n)$ $\{C_n\}_{n\in\mathbb{N}}$ tal que para todo $x \in \{0,1\}^*$ temos $x \in L$ sse $C_{|x|}(x) = 1$.

Proposição 21.1
Cada função $f : \{0,1\}^n \to \{0,1\}$ pode ser calculada por um circuito de tamanho $O(n2^n)$.

Prova. Para toda atribuição $a \in \{0,1\}^n$ existe uma claúsula (maxterm) C_a tal que $C_a(a) = 0$ e $C_a(a') = 1$ para toda atribuição $a' \neq a$. Podemos construir uma fórmula φ em forma normal conjuntivo que é a conjunção de todas claúsulas C_a para $a \in \{0,1\}^n$ tal que $f(a) = 0$:

$$\varphi(x) = \bigwedge_{a:f(a)=0} C_a(x)$$

Não é difícil verificar que $\varphi(x) = f(x)$. Uma implementação por um circuito booleano precisa no máximo $n + (n-1+n)2^n + 2^n - 1 + 1 = n + 2n2^n = O(n2^n)$ portas lógicas. ■

Portanto, a classe $\mathsf{SIZE}(2n2^n)$ contém todas funções booleanas computáveis. Para o estudo de complexidade essa classe é poderoso demais. Ela contém até funções não computáveis como o problema de parada (por quê?). Isso é o motivo para investigar quais funções booleanas são computáveis com menos portas lógicas.

Definição 21.3 (Circuitos com um número polinomial de portas)
A classe de complexidade $\mathsf{P/poly}$ contém todas linguagens decidíveis por familias de circuitos de tamanho polinomial, i.e.,

$$\mathsf{P/poly} = \bigcup_{k>0} \mathsf{SIZE}(n^k)$$

O seguinte lema estabelece que tudo que pode ser decidido por uma MT, pode ser decido por uma família de circuitos booleanos de tamanho não mais que o quadrado do tempo de execução da MT.

Lema 21.1
Para uma função $t : \mathbb{N} \to \mathbb{N}$ tal que $L \in \mathsf{DTIME}(t(n))$ temos $L \in \mathsf{SIZE}(O(t(n)^2))$.

Prova. (Rascunho.) Seja $L \in P$ alguma linguagem e M a MTD correspondente. Cada execução possível de M pode ser representado por uma tableau $T = (t_{ij})$ de tamanho $t(n) \times t(n)$. Cada célula t_{ij} contém um símbolo da fita em Γ ou ainda um símbolo em $\Gamma \times Q$ representando adicionalmente a posição e o estado da cabeça. O conteúdo da célula t_{ij} depende somente do conteúdo das células $t_{i-1,j-1}$, $t_{i-1,j}$ e $t_{i-1,j+1}$. Seja $s_{ijk} \in \{0,1\}$ um valor booleano que é verdadeira caso a célula $t_{i,j}$ contém o símbolo k (com $k = |\Gamma \cup \Gamma \times Q|$). Para cada regra da MTD que define o valor de t_{ij} como s_0, dado valores $t_{i-1,j-1} = s_1$, $t_{i-1,j} = s_2$ e $t_{i-1,j+1} = s_3$, vamos adicionar um circuito $s_{i-1,j-1,s_1} \wedge s_{i-1,j,s_2} \wedge s_{i-1,j+1,s_3}$ à célula t_{ij}. O novo símbolo em i,j é k, caso existe alguma transição desse tipo, i.e,

$$s_{ijk} = \bigvee_{t_{ij}=k | t_{i-1,j-1}=s_1, t_{i-1,j}=s_2, t_{i-1,j+1}=s_3} s_{i-1,j-1,s_1} \wedge s_{i-1,j,s_2} \wedge s_{i-1,j+1,s_3}$$

Para terminar a construção do circuito, temos que definir as entradas de acordo com o estado inicial da máquina, e definir uma saída do circuito. Isso é possível usando uma MTD modificada M' que antes de aceitar posiciona a cabeça na primeira célula da fita e escreve o símbolo 0 nessa posição. Com isso a saída do circuito é $s_{t(n),1,(0,q_a)}$.
O número de portas lógicas necessárias para implementar o circuito é no máximo $k(k^3 - 1 + 2k^3) = O(k^4)$ por célula, e portanto $O(t(n)^2)$. ■

Corolário 21.1
$\mathsf{P} \subseteq \mathsf{P/poly}$

Prova. Caso $L \in \mathsf{P}$, $L \in \mathsf{DTIME}(n^k)$ para algum $k \in \mathbb{N}$. Pelo lema 21.1 temos $L \in \mathsf{SIZE}(O(n^{2k}))$ e portanto $L \in \mathsf{P/poly}$. ■

Existe uma separação entre P e P/poly? Sim.

Proposição 21.2
Cada linguagem $L \subseteq \{0,1\}^*$ *unária* ($L \subseteq \{1\}^*$) pertence a P/poly.

Prova. Para $1^n \in L$ escolhe $\bigwedge_i x_i$, senão o circuito constante 0. ■

Uma linguagem que não pertence a P mas a P/poly é a versão unária do problema de parada

$$\text{UHALT} = \{1^n \mid \text{a representação binária de } n \text{ representa uma par } \langle M, x\rangle \text{ tal que } M(x) \text{ para}\}.$$

O último resultado mostre que P/poly é ainda uma classe poderosa, que contém linguagens indecidíveis. O problema com a definição da classe é que ela supõe somente a existência de uma família de circuitos que resolve o problema para cada tamanho de entrada n. Os circuitos para cada n podem ser bem diferentes: de fato pode ser difícil atualmente construir os circuitos para um dado n. Por isso, uma ideia é só permitir as famílias de circuitos construtíveis:

Definição 21.4 (Circuitos P-uniformes)
Uma família $\{C_n\}$ de circuitos é P-uniforme caso existe um MTD M que em tempo polinomial calcula uma descrição de C_n para entrada 1^n.

Porém, com esse restrição o "poder" da classe disparece:

Teorema 21.1
Uma linguagem L é decidível por uma família P-uniforme de circuitos sse $L \in \mathsf{P}$.

Prova. (Rascunho.) Caso L é decidível por uma família P-uniforme de circuitos podemos construir uma MTD que para entrada w primeira calcula $C_{|w|}$ em tempo polinomial e depois avalia $C_{|w|}(w)$.
Para provar a outra direção podemos usar a construção do lema 21.1 observando que a construção é possível em tempo polinomial em n. ■

Máquinas de Turing com conselho Uma caracterização alternativa de $\mathsf{P/poly}$ é como MT *com conselho*.

Definição 21.5 (Máquina de Turing com conselho)
Para funções $T, a : \mathbb{N} \to \mathbb{N}$ a classe de complexidade $\mathsf{DTIME}(t(n))/a(n)$ decidíveis por máquinas de Turing em tempo $t(n)$ com conselho de $a(n)$ bits contém todas linguagens L tal que existe uma sequencia $\{\alpha_n\}_{n\in\mathbb{N}}$ de strings com $\alpha_n \in \{0,1\}^{a(n)}$ é uma MT M tal que

$$x \in L \iff M(x, \alpha_{|x|}) = 1$$

e M precisa para a entrada $(x, \alpha_{|x|})$ no máximo $O(t(n))$ passos.

Teorema 21.2 (Caracterização alternativa de P/poly)

$$\mathsf{P/poly} = \bigcup_{k,l>0} \mathsf{DTIME}(n^k)/n^l$$

Prova. (Rascunho.)
Caso $L \in \mathsf{P/poly}$ temos uma família $\{C_n\}_{n\in\mathbb{N}}$ de tamanho polinomial que decide L. A descrição de C_n serve como conselho para uma MTD M que simplesmente para entrada x avalia $C_n(x)$.
Caso L é decidível por uma MTD M com conselho $\{\alpha_n\}_{n\in\mathbb{N}}$ de tamanho polinomial $a(n)$, podemos usar a construção do lema 21.1 para construir, para cada tamanho n, um circuito D_n tal que $D_n(x, \alpha) = M(x, \alpha)$. Com isso podemos também construir um circuito C_n tal que $C_n(x) = D_n(x, \alpha_{|x|})$: C_n simplesmente tem as entradas α_n "hard-wired", sem aumentar a complexidade do circuito. ■

Um limitante inferior para uma classe restrita

Definição 21.6 (Classes AC^d e AC)
Para cada d, a classe AC^d contém todas linguagens que decidíveis para uma família de circuitos $\{C_n\}_{n\in\mathbb{N}}$ de tamanho polynomial, com fan-in ilimitado e profundidade $O(\log^d n)$. A classe AC é $\bigcup_{k>0} \mathsf{AC}^d$.

Lema 21.2 (Lema de troca de Håstad (switching lemma))
Dado uma função que possui uma representação por uma fórmula em k-DNF (forma normal disjuntiva), e uma restrição randômica ρ que atribuição a t entradas de f valores randômicos, temos, para cada $s \geq 2$

$$\Pr_\rho[f|_\rho \text{ não possui } s\text{-CNF}] \leq \left(\frac{(n-t)k^{10}}{n}\right)^{s/2}$$

Teorema 21.3
Seja $\oplus(x_1, \ldots, x_n) = \sum_{i \in [n]} x_i \mod 2$ a função de paridade. $\oplus \notin AC^0$.

Prova. Seja C algum circuito em AC^0. Podemos supor a seguinte forma normal:

- C tem fan-out 1: caso contrário podemos introduzir cópias de subcircuitos, mantendo um tamanho polinomial e a mesma profundidade (constante).
- C tem todas negações nas entradas ou equivalente temos $2n$ entradas $x_i, \neg x_i$, $1 \leq i \leq n$.
- Os níveis de C alternadamente são conjunções e disjunções: como a fan-in é ilimitado dá para juntar cascatas de operações do mesmo tipo.
- O último nível são conjunções com fan-in 1.

Sejam n^b o tamanho e d a profundidade desse circuito. A ideia da prova é: (i) converter os últimos dois níveis em FNC para FND ou em FND para FNC (ii) juntar dois níveis com a mesma operação aumentando o fan-in do circuito e diminuindo a profundidade por um (iii) repetir passos (i) e (ii) $d-2$ vezes, (iv) aplicar o lema 21.6 $d-2$ para argumentar que isso com alta probabilidade é possível (v) argumentar que o circuito restrito resultante não pode obtido por alguma restrição da função da paridade.
A i-gésima restrição vai resultar num circuito de tamanho n_i com fan-in k_i no último nível. Temos $n_0 = n$ e vamos restringir $n_i - \sqrt{n_i}$ variáveis na $i+1$-ésima restrição, i.e., $n_i = n^{1/2^i}$, mantendo um fan-in no máximo $k_i = 10b2^i$. Supõe essas restrições são satisfeitas após da i-gésima restrição e o penúltimo nível contém disjunções. Os últimos dois nível representam fórmulas em k_i-DNF. Pelo lema 21.6 então existe com probabilidade ao menos

$$1 - \left(\frac{k_i^{10}}{\sqrt{n_i}}\right)^{k_{i+1}/2} \geq 1 - \left(\frac{1}{10n^b}\right)$$

para n suficientemente grande, uma k_{i+1}-CNF que representa a mesma função. Agora existem dois níveis de conjunções que podemos unir reduzindo a profundidade do circuito por um. Caso o penúltimo nível consiste em conjunções uma transformação similar é possível.
O lema é aplicada para cada um dos n^b portas lógicas no máximo um vez, e a probabilidade de falhar e $\leq 1/10n^b$, portanto a probabilidade de falhar nas $d-2$ reduções de um nível é $\leq 1/10$, i.e., com probabilidade $9/10$ existe um

circuito com as características desejadas. Este circuito resultando tem fan-in k_{d_2} no último nível e portanto é uma k_{d-2}-FNC ou k_{d-2}-FND. Portanto, fixando no máximo k_{d-2} variáveis (zerando todas variáveis de uma claúsula, por examplo), obtemos uma função constante. Mas a função da paridade nunca é constante para uma restrição de menos que n variáveis. Portanto o circuito não pode ser a representação de $\oplus$. ■

A. Conceitos matemáticos

Nessa seção vamos definir alguns conceitos básicos da matemática.

Definição A.1
Para um conjunto C e *fecho de Kleene* C^* denota o conjunto de todos sequências sobre C.

A.1. Funções comuns

$\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$ e $\mathbb{R}$ denotam os conjuntos dos números naturais sem 0, inteiros, racionais e reais, respectivamente. Escrevemos também $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$, para qualquer conjunto C, $C_+ := \{x \in C | x > 0\}$ e $C_- := \{x \in C \mid x < 0\}$. Por exemplo

$$\mathbb{R}_+ = \{x \in \mathbb{R} \mid x > 0\}.^{1}$$

Para um conjunto finito S, $\mathcal{P}(S)$ denota o conjunto de todos subconjuntos de S.

Definição A.2 (Valor absoluto)
O valor absoluta $|\cdot|$ é definido por

$$|x| = \begin{cases} x & \text{se } x \geq 0 \\ -x & \text{se } x < 0 \end{cases}$$

Proposição A.1 (Regras para valores absolutos)

$$|-x| = |x| \tag{A.1}$$
$$x \leq |x| \tag{A.2}$$
$$|x + y| \leq |x| + |y| \quad \text{Desigualdade triangular} \tag{A.3}$$
$$|xy| = |x||y| \tag{A.4}$$

Prova. (i) Se $-x > 0$ temos $x < 0$, logo $|-x| = -x$ e $|x| = -x$. O casos restantes podem ser analisadas analogamente. (ii) Analise de casos. (iii)

[1] Alguns autores usam $\mathbb{R}^+$.

Para $x+y<0$: $|x+y|=-(x+y)=(-x)+(-y)\leq|-x|+|-y|=|x|+|y|$. Para $x+y\geq 0$: $|x+y|=x+y\leq|x|+|y|$. (iv) Para $xy\geq 0$: Se $x=0$ temos $|xy|=0=|x||y|$, se $x>0$ temos $y>0$ e $|xy|=xy=|x||y|$, se $x<0$ temos $y<0$ e $|xy|=xy=(-|x|)(-|y|)=|x||y|$. Caso $xy<0$ similar. ■

Corolário A.1

$$\left|\sum_{i\in[n]} x_i\right| \leq \sum_{i\in[n]} |x_i| \tag{A.5}$$

$$\left|\prod_{i\in[n]} x_i\right| = \prod_{i\in[n]} |x_i| \tag{A.6}$$

$$\tag{A.7}$$

Prova. Prova com indução sobre n. ■

Proposição A.2 (Regras para o máximo)
Para $a_i, b_i \in \mathbb{R}$

$$\max_i a_i + b_i \leq \max_i a_i + \max_i b_i \tag{A.8}$$

Prova. Seja $a_k + b_k = \max_i a_i + b_i$. Logo

$$\max_i a_i + b_i = a_k + b_k \leq \left(\max_i a_i\right) + b_i \leq \max_i a_i + \max_i b_i.$$

■

Definição A.3 (Pisos e tetos)
Para $x \in \mathbb{R}$ o *piso* $\lfloor x \rfloor$ é o maior número inteiro menor que x e o *teto* $\lceil x \rceil$ é o menor número inteiro maior que x. Formalmente

$$\lfloor x \rfloor = \max\{y \in \mathbb{Z} \mid y \leq x\}$$
$$\lceil x \rceil = \min\{y \in \mathbb{Z} \mid y \geq x\}$$

O *parte fracionário* de x é $\{x\} = x - \lfloor x \rfloor$.

Observe que o parte fracionário sempre é positivo, por exemplo $\{-0.3\} = 0.7$.

Proposição A.3 (Regras para pisos e tetos)
Pisos e tetos satisfazem

$$x \leq \lceil x \rceil < x+1 \tag{A.9}$$
$$x-1 < \lfloor x \rfloor \leq x \tag{A.10}$$

Definição A.4
O *fatorial* é a função

$$n! : \mathbb{N} \to \mathbb{N} : n \mapsto \prod_{1 \le i \le n} i.$$

Temos a seguinte aproximação do fatorial (fórmula de Stirling)

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(1/n)) \tag{A.11}$$

Uma estimativa menos precisa pode ser obtida por

$$e^n = \sum_{i \ge 0} \frac{n^i}{i!} > \frac{n^n}{n!}$$

que implica

$$(n/e)^n \le n! \le n^n.$$

Revisão: Logaritmos

$\log_a 1 = 0$		(A.12)
$a^{\log_a n} = n$	por definição	(A.13)
$\log_a nm = \log_a n + \log_a m$	propriedade do produto	(A.14)
$\log_a n/m = \log_a n - \log_a m$	propriedade da divisão	(A.15)
$\log_a n^m = m \log_a n$	propriedade da potência	(A.16)
$\log_a n = \log_c n / \log_c a$	mudança de base	(A.17)
$\log_b a = 1/\log_a b$		(A.18)
$a^{\log_c b} = b^{\log_c a}$	expoentes	(A.19)

Os números harmônicos

$$H_n = \sum_{i \in [n]} 1/i$$

ocorrem frequentemente na análise de algoritmos.

Proposição A.4 (Números harmônicos)

$$\ln n < H_n < 1 + \ln n.$$

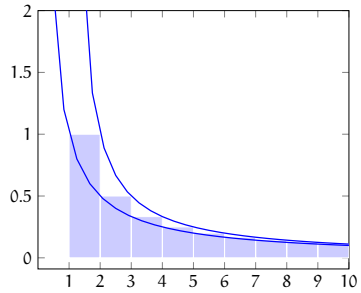


Figura A.1.: Cota inferior e superior dos números harmônicos.

Prova. Resultado da observação que

$$\int_1^{n+1} \frac{1}{x}\,dx < H_n < 1 + \int_2^{n+1} \frac{1}{x-1}\,dx$$

(veja figura A.1) e o fato que $\int 1/x = \ln x$:

$$\ln n \le \ln(n+1) = \int_1^{n+1} \frac{1}{x} < H_n$$
$$H_n < 1 + \int_2^{n+1} \frac{1}{x-1} = 1 + \ln n$$

■

Uma analise detalhada rende

Proposição A.5 (Números harmônicos)

$$H_n = \ln n + \gamma + (2n)^{-1} + O(n^{-2})$$

com $\gamma \approx 0.5772156649$ a *constante de Euler-Mascheroni*.

Definição A.5
O *logaritmo iterado* é

$$\log^* n = \begin{cases} 0, & \text{se } n \le 1, \\ 1 + \log^*(\log n), & \text{caso contrário,} \end{cases}$$

O logaritmo iterado é uma função que cresce extremamente lento; para valores práticos de n, $\log^* n$ não ultrapassa 5.

Definição A.6
Uma função f é *convexa* se ela satisfaz a desigualdade de Jensen

$$f\big(\Theta x + (1-\Theta)y\big) \leq \Theta f(x) + (1-\Theta)f(y). \tag{A.20}$$

Similarmente uma função f é *concava* caso $-f$ é convexo, i.e., ela satisfaz

$$f(\Theta x + (1-\Theta)y) \geq \Theta f(x) + (1-\Theta)f(y). \tag{A.21}$$

Exemplo A.1
Exemplos de funções convexas são x^{2k}, $1/x$. Exemplos de funções concavas são $\log x$, $\sqrt{x}$. ◊

Proposição A.6
Para $\sum_{i\in[n]} \Theta_i = 1$ e pontos x_i, $i \in [n]$ uma função convexa satisfaz

$$f\big(\sum_{i\in[n]} \Theta_i x_i\big) \leq \sum_{i\in[n]} \Theta_i f(x_i) \tag{A.22}$$

e uma função concava

$$f\big(\sum_{i\in[n]} \Theta_i x_i\big) \geq \sum_{i\in[n]} \Theta_i f(x_i) \tag{A.23}$$

Prova. Provaremos somente o caso convexo por indução, o caso concavo sendo similar. Para $n = 1$ a desigualdade é trivial, para $n = 2$ ela é válida por definição. Para $n > 2$ define $\bar{\Theta} = \sum_{i\in[2,n]} \Theta_i$ tal que $\Theta + \bar{\Theta} = 1$. Com isso temos

$$f\big(\sum_{i\in[n]} \Theta_i x_i\big) = f\big(\Theta_1 x_1 + \sum_{i\in[2,n]} \Theta_i x_i\big) = f(\Theta_1 x_1 + \bar{\Theta} y)$$

onde $y = \sum_{j\in[2,n]} (\Theta_j/\bar{\Theta}) x_j$, logo

$$\begin{aligned} f\big(\sum_{i\in[n]} \Theta_i x_i\big) &\leq \Theta_1 f(x_1) + \bar{\Theta} f(y) \\ &= \Theta_1 f(x_1) + \bar{\Theta} f\big(\sum_{j\in[2,n]} (\Theta_j/\bar{\Theta}) x_j\big) \\ &\leq \Theta_1 f(x_1) + \bar{\Theta} \sum_{j\in[2,n]} (\Theta_j/\bar{\Theta}) f(x_j) = \sum_{i\in[n]} \Theta_i x_i \end{aligned}$$

■

A.2. Somas

Revisão: Notação Somatório
Para k uma constante arbitrária temos

$$\sum_{i\in[n]} ka_i = k\sum_{i\in[n]} a_i \qquad \text{Distributividade} \tag{A.24}$$

$$\sum_{i\in[n]}\sum_{j\in[m]} a_i b_j = \left(\sum_{i\in[n]} a_i\right)\left(\sum_{j\in[m]} b_j\right) \qquad \text{Distributividade generalizada} \tag{A.25}$$

$$\sum_{i\in[n]} (a_i + b_i) = \sum_{i\in[n]} a_i + \sum_{i\in[n]} b_i \qquad \text{Associatividade} \tag{A.26}$$

$$\sum_{i=1}^{p} a_i + \sum_{i=p+1}^{n} a_i = \sum_{i\in[n]} a_i \tag{A.27}$$

$$\sum_{i=0}^{n} a_{p-i} = \sum_{i=p-n}^{p} a_i \tag{A.28}$$

A última regra é um caso particular de troca de índice (ou comutação) para somas. Para um conjunto finito C e uma permutação dos números inteiros π temos

$$\sum_{i\in C} a_i = \sum_{\pi(i)\in C} a_{\pi(i)}.$$

No exemplo da regra acima, temos $C = [0, n]$ e $\pi(i) = p - i$ e logo

$$\sum_{0\le i\le n} a_{p-i} = \sum_{0\le p-i\le n} a_{p-(i-p)} = \sum_{p-n\le i\le p} a_i.$$

Parte da análise de algoritmos se faz usando somatórios, pois laços *while* e *for* em geral podem ser representados por somatórios. Como exemplo, considere o seguinte problema. Dadas duas matrizes A e B, faça um algoritmo que copie a matriz triangular inferior de B para A.

Algoritmo A.1 (CopiaMTI)
Entrada Matrizes quadráticos A e B e dimensão n.

Saída Matriz A com a matriz triangular inferior copiada de B.

```
for i := 1 to n do
```

```
    for j := 1 to i do
      A_ij = B_ij
    end for
  end for
```

Uma análise simples deste algoritmo seria:

$$\sum_{i\in[n]}\sum_{j=1}^{i} 1 = \sum_{i\in[n]} i = n(n+1)/2 = O(n^2)$$

Séries

$$\sum_{i\in[n]} i = n(n+1)/2 \quad \text{Série aritmética} \tag{A.29}$$

$$\sum_{0\le i\le n} x^i = \frac{x^{n+1}-1}{x-1} \quad \text{Série geométrica, para } x \neq 1 \tag{A.30}$$

se $|x| < 1$ então

$$\sum_{k\ge 0} x^k = \frac{1}{1-x} \quad \text{Série geométrica infinitamente decrescente} \tag{A.31}$$

Série geométrica com limites arbitrários:

$$\sum_{a\le i\le b} x^i = \frac{x^{b+1}-x^a}{x-1} \quad \text{para } x \neq 1 \tag{A.32}$$

Logo

$$f(b) = \sum_{a\le i\le b} x^i = \begin{cases} \Theta(x^b) & \text{para } x > 1 \\ \Theta(1) & \text{para } x < 1 \end{cases} \tag{A.33}$$

Séries

$$\sum_{i=1}^{n} 2^i = 2^{n+1} - 2 \tag{A.34}$$

$$\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{A.35}$$

$$\sum_{i=0}^{n} i2^i = 2 + (n-1)2^{n+1} \tag{A.36}$$

Mais geral para alguma sequência f_i temos

$$\begin{aligned}\sum_{i\in[n]} if_i &= \sum_{i\in[n]}\sum_{j\in[i]} f_i = \sum_{j\in[i]\le n} f_i = \sum_{j\in[n]}\sum_{j\le i\le n} f_i \\ &= \sum_{j\in[n]} \left(\sum_{i\in[n]} f_i - \sum_{1\le i<j} f_i \right).\end{aligned}$$

Uma aplicação: para $x \neq 1$

$$\begin{aligned}\sum_{i\in[n]} ix^i &= \sum_{j\in[n]} \left(\sum_{i\in[n]} x^i - \sum_{1\le i<j} x^i \right) = \sum_{j\in[n]} \left(\frac{x^{n+1}-x}{x-1} - \frac{x^j - x}{x-1} \right) \\ &= \frac{1}{x-1} \sum_{j\in[n]} \left(x^{n+1} - x^j \right) \\ &= \frac{1}{x-1} \left(nx^{n+1} - \frac{x^{n+1}-x}{x-1} \right) = \frac{x}{(x-1)^2} \left(x^n(nx-n-1)+1 \right).\end{aligned}$$

Em particular para $x = 1/2$ obtemos

$$\sum_{i\in[n]} i2^{-i} = 2(2(2^{-1} - 2^{-n-1}) - n2^{-n-1}) = 2((1-2^{-n}) - n2^{-n-1}) = 2 - 2^{-n}(n+2). \tag{A.37}$$

Assintoticamente

$$f(n) = \sum_{i\in[n]} ix^i = \begin{cases} \Theta(nx^n) & \text{caso } x > 1 \\ \Theta(1) & \text{caso } x < 1 \end{cases}. \tag{A.38}$$

A.3. Indução

Revisão: Indução matemática

- Importante para provar resultados envolvendo inteiros.
- Seja P(n) uma propriedade relativa aos inteiros.
  - Se P(n) é verdadeira para n=1 e
  - se P(k) verdadeira implica que P(k+1) é verdadeira
  - então P(n) é verdadeira para todo inteiro $n \geq 1$.

Revisão: Indução matemática

- Para aplicarmos indução matemática deve-se:
  - Passo inicial: verificar se P(n) é verdadeira para a base n_0.
  - Hipótese: assumir P(n) válida.
  - Prova: provar que P(n) é válida para qualquer valor de $n \geq n_0$.
- Se os passos acima forem verificados, conclui-se que P(n) é valida para qualquer valor de $n \geq n_0$

Indução matemática: exercícios

- Mostre que $n! \leq n^n$.
- Mostre que $\frac{1}{\log_a(c)} = \log_c(a)$.
- Demonstre a propriedade dos expoentes.
- Encontre uma fórmula alternativa para

$$\sum_{i \in [n]} 2i - 1$$

  e prove seu resultado via indução matemática.
- Use indução matemática para provar que

$$\sum_{0 \leq i < n} q^i = \frac{q^n - 1}{q - 1}.$$

- Resolva os exercícios do capítulo 1.

A.4. Limites

Definição A.7 (Limites)
Para $f : \mathbb{N} \to \mathbb{R}$ o limite de n para ∞ é definido por

$$\lim_{n\to\infty} f(n) = c \iff \exists c\, \forall \epsilon > 0\, \exists n_0\, \forall n > n_0\, |f(n) - c| < \epsilon. \tag{A.39}$$

Caso não existe um $c \in \mathbb{R}$ a função é *divergente*. Uma forma especial de divergência é quando a função ultrapasse qualquer número real,

$$\lim_{n\to\infty} f(n) = \infty \iff \forall c\, \exists n_0\, \forall n > n_0\, f(n) > c \tag{A.40}$$

Também temos

$$\liminf_{n\to\infty} f(n) = \lim_{n\to\infty} \left(\inf_{m\geq n} f(m) \right)$$
$$\limsup_{n\to\infty} f(n) = \lim_{n\to\infty} \left(\sup_{m\geq n} f(m) \right)$$

Lema A.1 (Definição alternativa do limite)
É possível substituir $<$ com $\leq$ na definição do limite.

$$\lim_{n\to\infty} f(n) = c \iff \forall \epsilon > 0\, \exists n_0\, \forall n > n_0\, |f(n) - c| \leq \epsilon$$

Prova. $\Rightarrow$ é obvio. Para $\Leftarrow$, escolhe $\epsilon' = \epsilon/2 < \epsilon$. ■

A.5. Grafos

Seja $[D]^k$ o conjunto de todos subconjuntos de tamanho k de D.
Um *grafo* (ou grafo não-direcionado) é um par $G = (V, E)$ de *vértices* (ou nós ou pontos) V e *arestas* (ou arcos ou linhas) E tal que $E \subseteq [V]^2$. Com $|G|$ e $\|G\|$ denotamos o número de vértices e arestas, respectivamente. Dois vértices u, v são *adjacentes*, se $\{u, v\} \in E$, duas arestas e, f são adjacentes, se $e \cap f \neq \emptyset$. Para um vértice v, a *vizinhança* (de vértices) $N(v)$ é o conjunto de todas vértices adjacentes com ele, e a vizinhança (de arestas) $E(v)$ é o conjunto de todas arestas adjacentes com ele. O *grau* de um vértice v é o número de vizinhos $\delta(v) = |N(v)| = |E(v)|$.
Um *caminho* de comprimento k é um grafo $C = (\{v_0, \ldots, v_k\}, \{\{v_i, v_{i+1}\} \mid 0 \leq i < k\})$ com todo v_i diferente. Um ciclo de comprimento $k+1$ é um caminho com a aresta adicional $\{v_n, v_0\}$. O caminho com comprimento k é denotado com P^k, o ciclo de comprimento k com C^k.
Um grafo G é *conexo* se para todo par de vértices u, v existe um caminho entre eles em G.
Um *subgrafo* de G é um grafo $G' = (V', E')$ tal que $V' \subseteq V$ e $E' \subseteq E$, escrito $G' \subseteq G$. Caso G' contém todas arestas entre vértices em V' (i.e. $E' = E \cap [V']^2$) ela é um *subgrafo induzido* de V' em G, escrito $G' = G[V']$.
Um *grafo direcionado* é um par $G = (V, E)$ de vértices V e arestas $E \subseteq V^2$. Cada aresta $e = (u, v)$ tem um *começo* u e um *termino* v.

B. Soluções dos exercícios

Solução do exercício 1.2.

Prova de 1.6:
"⇒": Seja $f \in O(g)$. Como $s(n) = \sup_{m \geq n} f(m)/g(m)$ é não-crescente e maior ou igual que 0, é suficiente mostrar que existe um n tal que $s(n) < \infty$. Por definição do O temos $c > 0$ e n_0 tal que $\forall n > n_0\, f \leq cg$. Logo $\forall n > n_0 \sup_{m \geq n} f(m)/g(m) \leq c$.
"⇐": Seja $\limsup_{n \to \infty} f(n)/g(n) < \infty$. Então

$$\exists c > 0 \exists n_0 \forall n > n_0 (\sup_{m \geq n} f(m)/g(m)) < c.$$

Isso implica, que para o mesmo n_0, $\forall n > n_0\, f < cg$ e logo $f \in O(g)$.
Prova de 1.7:
"⇒": Seja $f \in o(g)$, i.e. para todo $c > 0$ temos um n_0 tal que $\forall n > n_0\, f \leq cg$. Logo $\forall n > n_0\, f(n)/g(n) \leq c$, que justifique $\lim_{n \to \infty} f(n)/g(n) = 0$ (veja lema A.1).
"⇐": Seja $\lim_{n \to \infty} f(n)/g(n) = 0$, i.e. para todo $c > 0$ existe um n_0 tal que $\forall n > n_0\, f(n)/g(n) < c$ pela definição do limite. Logo $\forall n > n_0\, f \leq cg$, tal que $f \in o(g)$.
Prova de 1.8:
"⇒": Seja $f \in \Omega(g)$. Como $i(n) = \inf_{m \geq n} f(m)/g(m)$ é não-decrescente, é suficiente mostrar, que existe um n tal que $i(n) > 0$. Pela definição de Ω existem $c > 0$ e n_0 tal que $\forall n > n_0\, f \geq cg$. Logo $\forall n > n_0\, f(n)/g(n) \geq c > 0$, i.e. $i(n_0 + 1) > 0$.
"⇐": Suponha $\liminf_{n \to \infty} f(n)/g(n) = l > 0$. Vamos considerar os casos $l < \infty$ e $l = \infty$ separadamente.
Caso $l < \infty$: Escolhe, por exemplo, $c = l/2$. Pela definição do limite existe n_0 tal que $\forall n > n_0\, |l - f/g| \leq l/2$. Logo $f \geq l/2g$ (f/g aproxima l por baixo) e $f \in \Omega(g)$.
Caso $l = \infty$, $i(n)$ não tem limite superior, i.e. $(\forall c > 0)\exists n_0\, i(n_0) > c$. Como $i(n)$ é não-decrescente isso implica $(\forall c > 0)\exists n_0 (\forall n > n_0)\, i(n) > c$. Portanto $\forall n > n_0 f > cg$ e $f \in \omega(g) \subseteq \Omega(g)$.

Prova de 1.9:

$$\begin{aligned} & f \in \omega(g) \\ \iff & (\forall c > 0)\exists n_0(\forall n > n_0) : f \geq cg \\ \iff & (\forall c > 0)\exists n_0(\forall n > n_0) : f(n)/g(n) \geq c \\ \iff & f(n)/g(n) \text{ não possui limite} \end{aligned}$$

Solução do exercício 1.3.
Prova de 1.10: Escolhe $c = 1$, $n_0 = 0$.
Prova de 1.11: Se $g \in cO(f)$, temos $g = cg'$ e existem $c' > 0$ e n_0 tal que $\forall n > n_0\ g' \leq c'f$. Portanto $\forall n > n_0\ g = cg' \leq cc'f$ e com cc' e n_0 temos $g \in O(f)$.
Prova de 1.12: Para $g \in O(f) + O(f)$ temos $g = h + h'$ com $c > 0$ e n_0 tal que $\forall n > n_0\ h \leq cf$ e $c' > 0$ e n_0' tal que $\forall n > n_0\ h' \leq c'f$. Logo para $n > \max(n_0, n_0')$ temos $g = h + h' \leq (c + c')f$.
Prova de 1.13: Para $g \in O(O(f))$ temos $g \leq ch$ com $h \leq c'f$ a partir de índices n_0 e n_0', e logo $g \leq cc'h$ a partir de $\max(n_0, n_0')$.
Prova de 1.14: $h = f'g'$ com $f' \leq c_f f$ e $g' \leq c_g g$ tal que $h = f'g' \leq c_f c_g fg$.
Prova de 1.15: Para $h \in O(fg)$ temos $c > 0$ e n_0 tal que $\forall n > n_0\ h \leq cfg$. Temos que mostrar, que h pode ser escrito como $h = fg'$ com $g' \in O(g)$. Seja

$$g'(n) = \begin{cases} h(n)/f(n), & \text{se } f(n) \neq 0, \\ c\,g(n), & \text{caso contrário,} \end{cases}$$

Verifique-se que $h = fg'$ por análise de casos. Com isso, temos também $g' = h/f \leq cfg/f = cg$ nos casos $f(n) \neq 0$ e $g' = cg \leq cg$ caso contrário.

Solução do exercício 1.4.

(a) Temos as equivalências

$$\begin{aligned} f \preceq g \iff & f \in O(g) \\ \iff & \exists c\ \exists n_0\ \forall n > n_0\ f \leq cg \\ \iff & \exists c'\ \exists n_0\ \forall n > n_0\ g \geq c'f \qquad \text{com } c' = 1/c \\ \iff & g \in \Omega(f) \end{aligned}$$

(b) A reflexividade e transitividade são fáceis de verificar. No exemplo do $\preceq$, $f \preceq f$, porque $\forall n f(n) \leq f(n)$ e $f \preceq g$, $g \preceq h$ garante que a partir de um n_0 temos $f \leq cg$ e $g \leq c'h$ e logo $f \leq (cc')h$ também. Caso $f \preceq g$ e $g \preceq f$ temos com item (a) $f \succeq g$ e logo $f \asymp g$ pela definição de Θ.

(c) Temos as equivalências

$$\begin{aligned} f \prec g &\iff f \in o(g) \\ &\iff \forall c\, \exists n_0\, \forall n > n_0\, f \leq cg \\ &\iff \forall c'\, \exists n_0\, \forall n > n_0\, g \geq c'f \qquad \text{com } c' = 1/c \\ &\iff g \in \omega(f) \end{aligned}$$

(d) O argumento é essencialmente o mesmo que no item (a).

(e) Como Θ é definido pela intersecção de O e Ω, a sua reflexividade e transitividade é uma consequência da reflexividade e transitividade do O e Ω. A simetria é uma consequência direta do item (a).

Solução do exercício 1.5.
Prova de 1.21 e 1.22: Suponha $f \succ g$ e $f \preceq g$. Então existe um c tal que a partir de um n_0 temos que $f = cg$ (usa as definições). Mas então $f \not\succ g$ é uma contradição. A segunda característica pode ser provada com um argumento semelhante.
Para provar as três afirmações restantes considere o par de funções n e $e^{n \sin(n)}$. Verifique-se que nenhuma relação $\prec$, $\preceq$, $\succ$, $\succeq$ ou $\asymp$ é verdadeira.

Solução do exercício 1.6.
As características correspondentes são

$$f = \Omega(f) \tag{B.1}$$
$$c\Omega(f) = \Omega(f) \tag{B.2}$$
$$\Omega(f) + \Omega(f) = \Omega(f) \tag{B.3}$$
$$\Omega(\Omega(f)) = \Omega(f) \tag{B.4}$$
$$\Omega(f)\Omega(g) = \Omega(fg) \tag{B.5}$$
$$\Omega(fg) = f\Omega(g) \tag{B.6}$$

Todas as características se aplicam para Ω também. As provas são modificações simples das provas das características 1.10 até 1.15 com $\leq$ substituído por $\geq$.
Prova.
Prova de B.1: Escolhe $c = 1$, $n_0 = 0$.
Prova de B.2: Se $g \in c\Omega(f)$, temos $g = cg'$ e existem $c' > 0$ e n_0 tal que $\forall n > n_0\, g' \geq c'f$. Portanto $\forall n > n_0\, g = cg' \geq cc'f$ e com cc' e n_0 temos $g \in \Omega(f)$.

Prova de B.3: Para $g \in \Omega(f) + \Omega(f)$ temos $g = h + h'$ com $c > 0$ e n_0 tal que $\forall n > n_0\ h \geq cf$ e $c' > 0$ e n_0' tal que $\forall n > n_0\ h' \geq c'f$. Logo para $n > \max(n_0, n_0')$ temos $g = h + h' \geq (c + c')f$.
Prova de B.4: Para $g \in \Omega(\Omega(f))$ temos $g \geq ch$ com $h \geq c'f$ a partir de índices n_0 e n_0', e logo $g \geq cc'h$ a partir de $\max(n_0, n_0')$.
Prova de B.5: $h = f'g'$ com $f' \geq c_f f$ e $g' \geq c_g g$ tal que $h = f'g' \geq c_f c_g fg$.
Prova de B.6: $h \geq cfg$. Escrevendo $h = fg'$ temos que mostrar $g' \in \Omega(g)$. Mas $g' = h/f \geq cfg/f = cg$. ■

Solução do exercício 1.7.
"$\Leftarrow$":
Seja $f + c \in O(g)$, logo existem c' e n_0 tal que $\forall n > n_0\ f + c \leq c'g$. Portanto $f \leq f + c \leq c'g$ também, e temos $f \in O(g)$.
"$\Rightarrow$":
Essa direção no caso geral não é válida. Um contra-exemplo simples é $0 \in O(0)$ mas $0 + c \notin O(0)$. O problema é que a função g pode ser 0 um número infinito de vezes. Assim f tem que ser 0 nesses pontos também, mas $f + c$ não é. Mas com a restrição que $g \in \Omega(1)$, temos uma prova:
Seja $f \in O(g)$ logo existem c' e n_0' tal que $\forall n > n_0'\ f \leq c'g$. Como $g \in \Omega(1)$ também existem c'' e n_0'' tal que $\forall n > n_0''\ g \geq c''$. Logo para $n > \max(n_0', n_0'')$

$$f + c \leq c'g + c \leq c'g + \frac{c}{c''}g = (c' + \frac{c}{c''})g.$$

Solução do exercício 1.8.

(a) Para $n \geq 2$ temos $\log 1 + n \leq \log 2n = \log 2 + \log n \leq 2 \log n$.

(b) Seja $f \in \log O(n^2)$, i.e. $f = \log g$ com g tal que $\exists n_0, c\ \forall n > n_0\ g \leq cn^2$. Então $f = \log g \leq \log cn^2 = \log c + 2 \log n \leq 3 \log n$ para $n > \max(c, n_0)$.

(c) Temos que mostrar que existem c e n_0 tal que $\forall n > n_0\ \log\log n \leq c \log n$. Como $\log n \leq n$ para todos $n \geq 1$ a inequação acima está correto com $c = 1$.

Solução do exercício 1.9.
Para provar $f_n = O(n)$ temos que provar que existe um c tal que $f_n \leq cn$ a partir um ponto n_0. É importante que a constante c é a mesma para todo n. Na verificação do professor Veloz a constante c muda implicitamente, e

por isso ela não é válida. Ele tem que provar que $f_n \leq cn$ para algum c fixo. Uma tentativa leva a

$$\begin{aligned} f_n &= 2f_{n-1} \\ &\leq 2cn \\ &\not\leq cn \quad \text{Perdido!} \end{aligned}$$

que mostra que essa prova não funciona.

Solução do exercício 1.10.
E simples ver que $f \in \hat{o}(g)$ implica $f \in o(g)$. Para mostrar a outra direção suponha que $f \in o(g)$. Temos que mostrar que $\forall c > 0 : \exists n_0$ tal que $f < cg$. Escolhe um c. Como $f \in o(g)$ sabemos que existe um n_0 tal que $f \leq c/2g$ para $n > n_0$. Se $g \neq 0$ para $n > n_0'$ então $c/2g < g$ também. Logo $f \leq c/2g < cg$ para $n > \max(n_0, n_0')$.

Solução do exercício 1.11.
Primeira verifique-se que Φ satisfaz $\Phi + 1 = \Phi^2$.
Prova que $f_n \in O(\Phi^n)$ com indução que $f_n \leq c\Phi^n$. Base: $f_0 = 0 \leq c$ e $f_1 = 1 \leq c\Phi$ para $c \geq 1/\Phi \approx 0.62$. Passo:

$$f_n = f_{n-1} + f_{n-2} \leq c\Phi^{n-1} + c\Phi^{n-2} = (c\Phi + c)\Phi^{n-2} \leq c\Phi^n$$

caso $c\Phi + c \leq c\Phi^2$.
Prova que $f_n \in \Omega(\Phi^n)$ com indução que $f_n \geq c\Phi^n$. Base: Vamos escolher $n_0 = 1$. $f_1 = 1 \geq c\Phi$ e $f_2 = 1 \geq c\Phi^2$ caso $c \leq \Phi^{-2} \approx 0.38$. Passo:

$$f_n = f_{n-1} + f_{n-2} \geq c\Phi^{n-1} + c\Phi^{n-2} = (c\Phi + c)\Phi^{n-2} \leq c\Phi^n$$

caso $c\Phi + c \geq c\Phi^2$.

Solução do exercício Toscani e Veloso (2005, p. 2.3).

(a) $3n + 7 \leq 5n + 2 \iff 5 \leq 2n \iff 2.5 \leq n$ (equação linear)

(b) $5n + 7 \leq 3n^2 + 1 \iff 0 \leq 3n^2 - 5n - 6 \iff 5/6 + \sqrt{97}/6 \leq n$ (equação quadrática)

(c) $5\log_2 n + 7 \leq 5n + 1 \iff 7^5 + 2^7 - 2 \leq 2^{5n} \iff 16933 \leq 2^{5n} \iff 2.809\ldots \leq n$

(d) Veja item (b)

(e) $52^n + 3 \geq 3n^2 + 5n \iff n \geq 2^n \geq (3n^2 + 5n - 3)/5 \Leftarrow 2^n \geq n^2$.

(f) $n^2 3^n \geq n^3 2^n + 1 \Leftarrow n^2 3^n \geq n^3 2^{n+1} \iff 2\log_2 n + n\log_2 3 \geq 3\log_2 n + (n+1)\log_2 2 \iff n\log_2 3 \geq \log_2 n + (n+1) \Leftarrow n(\log_2 3 - 1)/2 \geq \log_2 n$

Solução do exercício Toscani e Veloso (2005, p. 2.9).
Com $f \in \Theta(n^r)$ e $g \in \Theta(n^s)$ temos

$$c_1 n^r \leq f \leq c_2 n^r; \quad d_1 n^s \leq g \leq d_2 n^s \qquad \text{a partir de um } n_0$$

(para constantes c_1, c_2, d_1, d_2.) Logo

$$\begin{aligned} & d_1 f^q \leq g \circ f \leq d_2 f^q \\ & \Rightarrow d_1 (c_1 n^p)^q \leq g \leq d_2 (c_2 n^p)^q \\ & \Rightarrow f_1 c_1^q n^{p+q} \leq g \leq d_2 c_2^q n^{p+q} \\ & \Rightarrow g \in \Theta(n^{p+q}) \end{aligned}$$

Solução do exercício 1.18.
Como $\log n = (1/\epsilon)\log n^\epsilon \leq (1/\epsilon) n^\epsilon$, usando $c = 1/\epsilon$ e $n_0 = 1$ satisfaz a condição necessária da definição de O.

Solução do exercício 2.1.

$$C_p[\text{Alg1}] = \sum_{i \in [n]} \sum_{j=1}^{2^{i-1}} c = \frac{c}{2} \cdot \sum_{i \in [n]} 2^i = c \cdot 2^n - c = O(2^n)$$

Os detalhes da resolução do algoritmo abaixo foram suprimidos. Resolva com detalhes e confira se a complexidade final corresponde à encontrada na análise abaixo.

$$\begin{aligned} C_p[\text{Alg2}] &= \sum_{i \in [n]} \sum_{\substack{j \in [2]^i \\ j \text{ ímpar}}} j^2 \leq \sum_{i \in [n]} \sum_{j \in [2]^i} j^2 \\ &= O(\sum_{i \in [n]} (2^i)^3) \\ &= O(\sum_{i \in [n]} 8^i) = \frac{8^{n+1} - 8}{7} \leq 8^{n+1} = O(8^n) \end{aligned}$$

$$
\begin{aligned}
C_p[\text{Alg3}] &= \sum_{i\in[n]} \sum_{j=i}^{n} 2^i = \sum_{i\in[n]} 2^i \cdot (n-i+1) \\
&= \sum_{i\in[n]} (n2^i - i2^i + 2^i) = \sum_{i\in[n]} n.2^i - \sum_{i\in[n]} i.2^i + \sum_{i\in[n]} 2^i \\
&= n \cdot (2^{n+1} - 2) - (2 + (n-1) \cdot 2^{n+1}) + (2^{n+1} - 2) \\
&= n2^{n+1} - 2n - 2 - n2^{n+1} + 2^{n+1} + 2^{n+1} - 2 \\
&= 2^{n+2} - 2n - 4 = O(2^n)
\end{aligned}
$$

$$
\begin{aligned}
C_p[\text{Alg4}] &= \sum_{i\in[n]} \sum_{j=1}^{i} 2^j = \sum_{i\in[n]} (2^{i+1} - 2) \\
&= 2 \sum_{i\in[n]} 2^i - \sum_{i\in[n]} 2 = 2 \cdot (2^{n+1} - 2) - 2n \\
&= 4 \cdot 2^n - 4 - 2n = O(2^n)
\end{aligned}
$$

$$
\begin{aligned}
C_p[\text{Alg5}] &= \sum_{i\in[n]} \sum_{j=i}^{n} 2^j = \sum_{i\in[n]} \left(\sum_{j=1}^{n} 2^j - \sum_{j=1}^{i-1} 2^j \right) \\
&= \sum_{i\in[n]} \left(2^{n+1} - 2 - (2^{i-1+1} - 2) \right) = \sum_{i\in[n]} \left(2 \cdot 2^n - 2 - 2^i + 2 \right) \\
&= 2 \sum_{i\in[n]} 2^n - \sum_{i\in[n]} 2^i = 2 \cdot n2^n - (2^{n+1} - 2) \\
&= 2 \cdot n2^n - 2 \cdot 2^n + 2 = O(n2^n)
\end{aligned}
$$

Solução do exercício 2.2.
O problema é o mesmo da prova do exercício 1.9: Na prova a constante c muda implicitamente. Para provar $T_n = O(n)$ temos que provar $T_n \leq cn$ para c fixo. Essa prova vira

$$
\begin{aligned}
T_n &= n - 1 + 2/n \sum_{0\leq i<n} O(i) \\
&\leq n - 1 + 2c/n \sum_{0\leq i<n} i \\
&= n - 1 + c(n-1) = cn + (n - 1 - c) \\
&\not\leq cn \qquad \text{Não funciona para } n > c + 1
\end{aligned}
$$

Solução do exercício 2.3.
Uma solução simples é manter um máximo M e o segundo maior elemento m no mesmo tempo:

```
M := ∞
m := ∞
for i = 1,...,n do
  if a_i > M then
    m := M
    M := a_i
  else if a_i > m do
    m := a_i
  end if
end for
return m
```

O número de comparações é ao máximo dois por iteração, e esse limite ocorre numa sequência crescendo $1, 2, \ldots, n$. Portanto, a complexidade pessimista é $2n = \Theta(n)$. Existem outras soluções que encontram o segundo maior elemento com somente $n + \log_2 n$ comparações.

Solução do exercício 2.4.
Uma abordagem simples com busca exaustiva é

```
m := ∑_{i∈[n]} a_i
for C ⊆ [1,n] do
  m' := |∑_{i∈C} a_i − ∑_{i∉C} a_i|
  if m' < m then
    m := m'
  end if
end for
```

O algoritmo tem complexidade pessimista $c_p = O(n) + O(2^n nc) = O(n2^n)$.

Solução do exercício 2.5.
Para um dado n temos sempre $n - \lfloor n/2 \rfloor$ atualizações. Logo, o número médio de atualizações é a mesma.

Solução do exercício 2.6.
Seja $A, A_1, \ldots, A_n$ as variáveis aleatórias que denotam o número total de atualizações, e o número de atualizações devido a posição i, respectivamente.

Com a distribuição uniforme temos $E[A_i] = 1/6$ e pela linearidade

$$E[A] = E\left[\sum_{i\in[n]} A_i\right] = n/6.$$

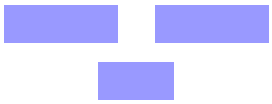
Com o mesmo argumento a segunda distribuição leva a $E[A_i] = 1/10$ é $E[A] = n/10$ finalmente.

Solução do exercício 2.7.
Cada chave em nível $i \in [1, k]$ precisa i comparações e a árvore tem $\sum_{i\in[k]} 2^{i-1} = 2^k - 1$ nós e folhas em total. Para o número de comparações C temos

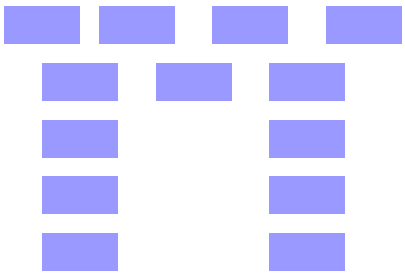
$$E[C] = \sum_{i\in[k]} \Pr(C = i)i = \sum_{i\in[k]} \frac{2^{i-1}}{2^{k-1}} i = 2^{-k} \sum_{i\in[k]} 2^i i = 2(k-1) + 2^{1-k}.$$

Solução do exercício 4.1.
Um contra-exemplo para "começa mais cedo", "termina mais tarde" e "maior duração":

Um contra-exemplo para "menor duração" e "maior número de conflitos"



Um contra-exemplo para "menor número de conflitos", "sobreposição menor"



Solução do exercício 4.2.
O seguinte algoritmo resolva o problema:

Algoritmo B.1 (Subsequência)
Entrada Sequência $S' = s'_1 \dots s'_m$ e $S = s_1 \dots s_n$.

Saída `true`, se $S' \subseteq S$ (S' é uma subsequência de S)

```
if m > n then
  return false
end if
i := 1
for j := 1,...,n do
  if s'_i = s_j then
    i := i + 1
    if i > m then
      return true
    end if
  end if
end for
return false
```

e tem complexidade $O(n)$. A corretude resulta de observação que para cada subsequência possível temos outra subsequência que escolhe o elemento mais esquerda em S. Portanto, podemos sempre escolher gulosamente o primeiro elemento da sequência maior.

Solução do exercício 4.3.
O seguinte algoritmo resolve o problema:

Algoritmo B.2 (Bases)
Entrada Uma sequência de posições x_i de n cidades, $i \in [n]$.

Saída Uma sequência mínima de posições b_i de bases.

```
Sejam S = x'_1 ... x'_n as posições em ordem crescente
B = ϵ
while S ≠ ϵ do
  Seja S = x'S'
  B := B, (x' + 4)   { aumenta a sequência B }
  Remove todos os elementos x ≤ x' + 8 de S
end while
```

O algoritmo tem complexidade $O(n)$ porque o laço tem ao máximo n iterações. Prova de corretude: Seja b_i as posições do algoritmo guloso acima, e b'_i as posições de alguma outra solução. Afirmação: $b_i \geq b'_i$. Portanto, a solução gulosa não contém mais bases que alguma outra solução. Prova da afirmação com indução: A base $b_1 \geq b'_1$ é correto porque toda solução tem que alimentar a primeira casa e o algoritmo guloso escolhe a última posição possível. Passo: Seja $b_i \geq b'_i$ e sejam h, h' as posições da próximas casas sem base. O algoritmo guloso escolha $h+4$, mas como $b_i \geq b'_i$ e $h \geq h'$ temos $b'_{i+1} \leq h'+4$ porque h' precisa uma base. Logo, $x_{i+1} = h+4 \geq h'+4 \geq b'_{i+1}$.

Solução do exercício 5.3.
Seja $B(c_1, \ldots, c_K, i)$ uma variável que indica que existe um empacotamento dos itens $i, \ldots, n$ nos K contêineres com capacidades $c_1, \ldots, c_K$. Logo a solução do problema de decisão é $B(T, \ldots, T, 1)$. As variáveis B podem ser calculados com programação dinâmica usando a recorrência

$$B(c_1, \ldots, c_K, i) = \begin{cases} \bigvee_{j \in [k] | c_j \geq t_i} B(c_1, \ldots, c_j - s_i, \ldots, c_K, i+1), & \text{caso } i \leq n, \\ \text{falso}, & \text{caso contrário,} \end{cases}$$

em tempo $O(T^K K n)$ e espaço $O(T^K n)$. (Observe que a disjunção vazio é falso por definição.)

Solução do exercício 6.1.

1. $T(n) = 9T(n/3) + n$

$$\begin{aligned} T(n) &= \sum_{0 \leq i < \log_3 n} 9^i (n/3^i) + \Theta(9^{\log_3 n}) \\ &= n \sum_{0 \leq i < \log_3 n} 3^i + n^2 \\ &= n \frac{3^{\log_3 n} - 1}{2} + n^2 = \Theta(n^2) \end{aligned}$$

2. $T(n) = 2T(n/2) + n \log n$

$$\begin{aligned} T(n) &= \sum_{0 \le i < \log n} 2^i (n/2^i) \log_2 n/2^i + \Theta(2^{\log_2 n}) \\ &= n \sum_{0 \le i < \log n} \log_2 n - i + \Theta(n) \\ &= n \log_2^2 n - \frac{n \log_2 n (\log_2 n - 1)}{2} + \Theta(n) \\ &= O(n \log_2^2 n) \end{aligned}$$

Solução do exercício 6.3.
Se usarmos a hipótese $T(n) \le cn^{\log_2 3}$, não conseguiremos finalizar a prova pois permanecerá um fator adicional que não podemos remover da equação. Caso este fator adicional for menor em ordem que a complexidade que queremos provar, podemos usar uma hipótese mais forte como apresentado abaixo.
Hipótese: $T(n) \le cn^{\log_2 3} - dn$

$$\begin{aligned} T(n) &\le 3(c(n/2)^{\log_2 3} - d(n/2)) + bn \\ &\le 3cn^{\log_2 3}/(2^{\log_2 3}) - 3d(n/2) + bn \\ &\le cn^{\log_2 3} - 3d(n/2) + bn \\ &\le cn^{\log_2 3} - dn \end{aligned}$$

A desigualdade acima é verdadeira para $-3d(n/2) + bn \le -dn$, ou seja, para $d \le -2b$: $T(n) \in O(n^{\log_2 3} - dn) \in O(n^{\log_2 3})$.

Solução do exercício 6.4.
A recorrência correspondente é

$$T(n) = 7T(n/2) + \Theta(n^2).$$

Analisando com a árvore de recorrência, obtemos 7^i problemas em cada nível, cada um com tamanho $n/2^i$ e custo $c(n/2^i)^2 = cn^2/4^i$ e altura $h = \lceil \log_2 n \rceil$ (com $h+1$ níveis) que leva a soma

$$\begin{aligned} T(n) &\le \sum_{0 \le i \le h} cn^2 (7/4)^i + 7^{h+1} \\ &= (4/3)cn^2((7/4)^{h+1} - 1) + 7^{h+1} \qquad \text{com } 4^{h+1} \ge 4n^2 \\ &\le (7c/3 + 1)7^h - (4/3)cn^2 \qquad \text{com } 7^h \le 7\,7^{\log_2 n} \\ &\le (49c/3 + 1)n^{\log_2 7} = O(n^{\log_2 7}). \end{aligned}$$

Para aplicar o método de substituição, podemos estimar $T(n) \leq an^c - bn^2$ com $c = \log_2 7$ que resulta em

$$\begin{aligned} T(n) &= 7T(n/2) + dn^2 \\ &\leq 7a/2^c n^c - 7b/4n^2 + dn^2 \\ &= an^c - bn^2 + (d - 3b/4)n^2 \end{aligned}$$

que é satisfeito para $d - 3/4 \leq 0 \Leftrightarrow b \geq (4/3)d$.
Para aplicar o teorema Mestre, é suficiente de verificar que com $\Theta(n^2) = O(n^{\log_2 7} - \epsilon)$ se aplica o caso 1, e portanto a complexidade é $\Theta(n^{\log_2 7})$.

Solução do exercício 6.4.

O algoritmo obedece a recorrência

$$T(n) = T(n/5) + T(7n/10 + 6) + O(n)$$

(sem considerar o teto). Na aplicação da árvore de recorrência, enfrentamos dois problemas: (i) Os ramos tem comprimento diferente, porque os sub-problemas tem tamanho diferente (portanto o teorema Mestre não se aplica nesse caso). (ii) O tamanho $7n/10 + 6$ do segundo subproblema leva a somas difíceis.
Por isso, vamos estimar o custo da árvore da seguinte forma: (i) Temos que garantir, que o segundo subproblema sempre é menor: $7n/10 + 6 < n$. Isso é satisfeito para $n > 20$. (ii) Vamos substituir o sub-problema $7n/10 + 6$ com a cota superior $(7 + \epsilon)n/10$ para um $\epsilon > 0$ pequeno. Isso é satisfeito para $n \geq 60/\epsilon$. (iii) Sejam $c_1 := 1/5$, $c_2 := (7 + \epsilon)/10$ e $c := c_1 + c_2$. Então a árvore tem custo $c^i n$ no nível i e no ramo mais longo (que corresponde a c_2) uma altura de $h = \left\lceil \log_{c_2} 20/n \right\rceil$. Portanto, obtemos uma cota superior para o custo da árvore

$$\begin{aligned} T(n) &\leq n \sum_{0 \leq i \leq h} c^i + F(n) \\ &\leq n \sum_{0 \leq i < \infty} c^i + F(n) && \text{porque } c < 1 \\ &= 10n/(1 - \epsilon) + F(n) \end{aligned}$$

com o número de folhas $F(n)$. Caso $F(n) = O(n)$ obtemos a estimativa desejada $T(n) = O(n)$. Observe que a estimativa

$$F(n) = 2^{h+1} \leq 42^{\log_{c_2} 20} n^{\log_{1/c_2} 2} = \Omega(n^{1.94})$$

não serve! Como as folhas satisfazem a recorrência

$$F(n) \leq \begin{cases} F(\lceil n/5 \rceil) + F(\lfloor 7n/10 + 6 \rfloor) & \text{se } n > 20 \\ O(1) & \text{se } n \leq 20 \end{cases}$$

$F(n) \leq cn$ pode ser verificado com substituição (resolvido no livro do Cormen). O teorema Mestre não se aplica nesta recorrência.

Bibliografia

Aaronson, S. (mar. de 2005). "NP-complete problems and physical reality". Em: *ACM SIGACT News* (ver p. 397).

Adelson-Velskii, G. e E. M. Landis (1962). "An algorithm for the organization of information (in Russian)". Em: *Proceedings of the USSR Academy of Sciences*. Vol. 146, pp. 263–266 (ver p. 211).

Agrawal, M., N. Kayal e N. Saxena (2004). "PRIMES is in P". Em: *Annals of Mathematics* 160.2, pp. 781–793 (ver pp. 365, 399).

Aho, A. V., J. E. Hopcroft e J. D. Ullman (1974). *The Design and Analysis of Computer Algorithms*. Addison-Wesley (ver p. 212).

Akra, M. e L. Bazzi (1998). "On the Solution of Linear Recurrence Equations". Em: *Computational Optimization and Applications* 10, pp. 195–210 (ver p. 137).

Alford, W. R., A. Granville e C. Pomerance (1994). "There are infinitely many Carmichael numbers". Em: *Annals Math.* 140 (ver p. 364).

Alon, N., Z. Galil, O. Margalit e M. Naor (1992). "Witnesses for Boolean Matrix Multiplication and for Shortest Paths". Em: *FOCS* (ver p. 118).

Alt, H., N. Blum, K. Mehlhorn e M. Paul (1991). "Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m \log n})$". Em: *Information Processing Letters* 37, pp. 237–240 (ver p. 316).

Andrews, J. e J. A. Sethian (2007). "Fast marching methods for the continuous traveling salesman problem". Em: *Proc. Natl. Acad. Sci. USA* 104.4. DOI: 10.1073/pnas.0609910104 (ver p. 259).

Aragon, C. R. e R. Seidel (1989). "Randomized Search Trees". Em: *Proc. 30th Symp. Foundations of Computer Science*. IEEE Computer Society Press. DOI: 10.1109/SFCS.1989.63531 (ver p. 198).

Arora, S. e B. Barak (2009). *Computational Complexity: A Modern Approach*. Cambridge University Press (ver pp. 352, 377, 388).

Atallah, M. J., ed. (1999). *Algorithms and theory of computation handbook*. CRC Press (ver p. 387).

Baker, B. S. (1985). "A new proof for the first fit decreasing bin packing algorithm". Em: *J. Alg.* 6, pp. 49–70. DOI: 10.1016/0196-6774(85)90018-5 (ver p. 340).

Bellman, R. (1962). "Dynamic Programming Treatment of the Travelling Salesman Problem". Em: *J. ACM* 9.1, pp. 61–63 (ver p. 118).

Berge, C. (1957). "Two theorems in graph theory". Em: *Proc. National Acad. Science* 43, pp. 842–844 (ver p. 305).

Black Jr., J. R. e C. U. Martel (1998). *Designing Fast Graph Data Structures: An Experimental Approach*. Rel. téc. Department of Computer Science, University of California, Davis (ver p. 273).

Bluestein, L. I. (1970). "A linear filtering approach to the computation of the discrete Fourier transform". Em: *IEEE Trans. Audio Electroacoust* AU-18.451 (ver p. 144).

Booth, K. S. (1978). "Isomorphism testing for graphs, semigroups, and finite automata are polynomially equivalent problems". Em: *SIAM J. Comput.* 7.3, pp. 273–279. DOI: 10.1137/0207023 (ver p. 409).

Brodal, G. S., R. Fagerberg e R. Jacob (2001). *Cache Oblivious Search Trees via Binary Trees of Small Height*. Rel. téc. RS-01-36. BRICS (ver p. 259).

Broder, A. e M. Mitzenmacher (2003). "Network applications of Bloom filter: A survey". Em: *Internet Mathematics* 1.4, pp. 485–509 (ver p. 270).

Buluç, A., J. T. Fineman, M. Frigo, J. R. Gilbert e C. E. Leiserson (2009). "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks". Em: *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pp. 233–244. DOI: 10.1145/1583991.1584053 (ver p. 190).

Chan, T. M. (2007). "More Algorithms for All-Pairs Shortest Paths in Weighted Graphs". Em: *STOC'07* (ver p. 118).

Chazelle, B. (2000). "A Minimum Spanning Tree Algorithm with Inverse-Ackermann Type Complexity". Em: *Journal ACM* 47, pp. 1028–1047 (ver p. 258).

Clausen, J. (1999). *Branch and Bound Algorithms – Principles and examples* (ver p. 162).

Complexity zoo. Online (ver p. 387).

Coppersmith, D. e S. Winograd (1987). "Matrix multiplication via arithmetic progressions". Em: *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pp. 1–6 (ver p. 69).

Cormen, T. H., C. E. Leiserson, R. L. Rivest e C. Stein (2009). *Introduction to Algorithms*. 3rd. The MIT Press (ver p. 259).

Cowen, L., R. Cowen e A. Steinberg (2008). "Totally Greedy Coin Sets and Greedy Obstructions". Em: *The Electronic Journal of Combinatorics* 15 (ver p. 98).

Damgård, I., P. Landrock e C. Pomerance (1993). "Average case error estimates for the strong probable prime test". Em: *Mathematics of computation* 61.203, pp. 177–194 (ver p. 365).

Dean, B. C., M. X. Goemans e N. Immorlica (2006). “Finite termination of "augmenting path"algorithms in the presence of irrational problem data”. Em: *ESA'06: Proceedings of the 14th conference on Annual European Symposium*. Zurich, Switzerland: Springer-Verlag, pp. 268–279. DOI: http://dx.doi.org/10.1007/11841036_26 (ver p. 282).

Delacorte, M. (2007). *Graph Isomorphism is PSPACE-complete*. arXiv: 0708.4075v1 [cs.CC] (ver p. 409).

Dementiev, R., L. Kettner, J. Mehnert e P. Sanders (2004). “Engineering a Sorted List Data Structure for 32 Bit Keys”. Em: *Workshop on Algorithm Engineering & Experiments*, pp. 142–151 (ver p. 259).

Diestel, R. (2005). *Graph theory*. 3rd. Springer (ver p. 22).

Du, D.-Z. e K.-I. Ko, eds. (1997). *Advances in Algorithms, Languages, and Complexity - In Honor of Ronald V. Book*. Kluwer (ver p. 50).

Duan, R., S. Pettie e H.-H. Su (2011). “Scaling algorithms for approximate and exact maximum weight matching”. Em: *CoRR* abs/1112.0790 (ver p. 316).

Edmonds, J. (1965). “Paths, Trees, and Flowers”. Em: *Canad. J. Math* 17, pp. 449–467 (ver p. 316).

Edmonds, J. e R. Karp (1972). “Theoretical improvements in algorithmic efficiency for network flow problems”. Em: *JACM* 19.2, pp. 248–264 (ver p. 284).

Egerváry, J. (1931). “Matrixok kombinatorius tulajdonságairól (On combinatorial properties of matrices)”. Em: *Matematikai és Fizikai Lapok* 38, pp. 16–28 (ver p. 313).

Feder, T. e R. Motwani (1991). “Clique partitions, graph compression and speeding-up algorithms”. Em: *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing (23rd STOC)*, pp. 123–133 (ver p. 308).

– (1995). “Clique partitions, graph compression and speeding-up algorithms”. Em: *Journal of Computer and System Sciences* 51, pp. 261–272 (ver pp. 308, 316).

Ford, L. R. e D. R. Fulkerson (1956). “Maximal flow through a network”. Em: *Canadian Journal of Mathematics* 8, pp. 399–404 (ver p. 280).

Frank, A. (2004). *On Kuhn's Hungarian Method – A tribute from Hungary*. Rel. téc. Egerváry Research Group on Combinatorial Optimization.

Fremuth-Paeger, C. e D. Jungnickel (2003). “Balanced network flows VIII: a revised theory of phase-ordered algorithms and the $O(\sqrt{n}m \log(n^2/m)/\log n$ bound for the nonbipartite cardinality matching problem”. Em: *Networks* 41, pp. 137–142 (ver p. 316).

Fürer, M. (2007). “Faster Integer Multiplication”. Em: *Proc. 29th STOC*, pp. 57–66. DOI: 10.1145/1250790.1250800 (ver p. 148).

Fürer, M. e B. Raghavachari (1994). "Approximating the minimu-degree steiner tree to within one of optimal". Em: *Journal of Algorithms* (ver p. 319).

Gabow, H. N. (1990). "Data structures for weighted matching and nearest common ancestors with linking". Em: *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 434–443 (ver p. 316).

Goel, A., M. Kapralov e S. Khanna (2010). "Perfect Matchings in $O(n \log n)$ Time in Regular Bipartite Graphs". Em: *STOC 2010* (ver p. 308).

Goldberg, A. V. e A. V. Karzanov (2004). "Maximum skew-symmetric flows and matchings". Em: *Mathematical Programming A* 100, pp. 537–568 (ver p. 316).

Goldschmidt, O. e D. S. Hochbaum (1988). "Polynomial Algorithm for the k-Cut Problem". Em: *Proc. 29th FOCS*, pp. 444–451 (ver p. 332).

Graham, R. L., D. E. Knuth e O. Patashnik (1988). *Concrete Mathematics: a foundation for computer science*. Addison-Wesley (ver pp. 173, 174).

Grcar, J. F. (2011). "How ordinary elimination became Gaussian elimination". Em: *Historia Mathematica* 38, pp. 163–218. DOI: 10.1016/j.hm.2010.06.003 (ver p. 15).

Gurevich, Y. e S. Shelah (1987). "Expected computation time for Hamiltonian Path Problem". Em: *SIAM J. on Computing* 16.3, pp. 486–502 (ver p. 50).

Habgood, K. e I. Arel (2010). "Revisiting Cramer's rule for solving dense linear systems". Em: *Proceedings of the 2010 Spring Simulation Multiconference*. DOI: 10.1145/1878537.1878623 (ver p. 29).

Hadamard, J. (1896). "Sur la distribution des zéros de la fonction zeta(s) et ses conséquences arithmétiques". Em: *Bull. Soc. math. France* 24, pp. 199–220 (ver p. 362).

Haeupler, B., S. Sen e R. E. Tarjan (2009). "Heaps simplified". Em: *(Preprint)*. arXiv:0903.0116 (ver pp. 230, 238).

Hartmanis, J. e R. E. Stearns (1965). "On the Computational Complexity of Algorithms". Em: *Transactions of the American Mathematical Society* 117, pp. 285–306 (ver p. 29).

Hibbard, T. N. (1962). "Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting". Em: *JACM* 9.1, pp. 16–17 (ver p. 197).

Hirschberg, D. S. (1975). "A linear space algorithm for computing maximal common subsequences". Em: *Comm. of the ACM* 18.6, pp. 341–343 (ver p. 108).

Hoare, C. A. R. (1962). "Quicksort". Em: *Computer Journal* 5, pp. 10–15 (ver p. 56).

Hopcroft, J. E. e R. Karp (1973). "An $n^{5/2}$ algorithm for maximum matching in bipartite graphs". Em: *SIAM J. Comput.* 2, pp. 225–231 (ver p. 304).

Hopcroft, J. E., R. Motwani e J. D. Ullman (2006). *Introduction to Automata Theory, Languages, and Computation*. 3rd. Addison Wesley (ver p. 409).
Johnson, D. S. (1973). "Near-optimal bin packing algorithms". Tese de doutoramento. Massachusetts Institute of Technology. Dept. of Mathematics. URL: http://hdl.handle.net/1721.1/57819 (ver p. 340).
Johnson, D. S. e M. R. Garey (1985). "A 71/60 theorem for bin packing". Em: *J. Complex.* 1.1, pp. 65–106. DOI: 10.1016/0885-064X(85)90022-6 (ver p. 340).
Jones, M. J. e J. M. Rehg (1998). *Statistical Color Models with Application to Skin Detection*. Rel. téc. CRL 98/11. Cambridge Research Laboratory (ver p. 295).
Kaltofen, E. e G. Villard (2004). "On the complexity of computing determinants". Em: *Computational complexity* 13, pp. 91–130 (ver p. 29).
Kaplan, H. e U. Zwick (2009). "A simpler implementation and analysis of Chazelle's soft heaps". Em: *SODA '09: Proceedings of the Nineteenth Annual ACM -SIAM Symposium on Discrete Algorithms*. New York, New York: Society for Industrial e Applied Mathematics, pp. 477–485 (ver p. 258).
Karatsuba, A. A. e Y. Ofman (1962). "Multiplication of Many-Digital Numbers by Automatic Computers". Em: *Doklady Akad. Nauk SSSR* 145.2. Translation in Soviet Physics-Doklady 7 (1963), pp. 595–596, pp. 293–294 (ver pp. 129, 148).
Kilby, P., J. Slaney, S. Thiebaux e T. Walsh. "Estimating search tree size'". Em: *National Conference on Artificial Intelligence (AAAI 2006)*. Ed. por Y. Gil e R. Mooney, pp. 1014–1019. URL: www.cse.unsw.edu.au/~tw/kstwaaai06.pdf (ver p. 175).
Kleinberg, J. e E. Tardos (2005). *Algorithm design*. Addison-Wesley (ver pp. 98, 99).
Knuth, D. E. *Dancing Links*. URL: http://www-cs-faculty.stanford.edu/~uno/preprints.html (ver p. 190).
– (1975). "Estimating the efficiency of backtrack programs". Em: *Mathematics of Computation* 29.129, pp. 121–136 (ver p. 161).
– (1998). *The art of computer programming*. 2nd. Vol. III, Sorting and searching. Addison-Wesley (ver pp. 61, 198).
Knuth, D. E. (1976). "Big omicron and big omega and big theta". Em: *SIGACT News*.
Kuhn, H. W. (1955). "The Hungarian Method for the assignment problem". Em: *Naval Re4search Logistic Quarterly* 2, pp. 83–97 (ver p. 316).
Kullmann, O. (2008). *Fundaments of Branching Heuristics: Theory and Examples*. Rel. téc. CSR 7-2008. Swansea University. URL: http://www.swan.ac.uk/compsci/research/reports/2008/CSR7-2008.pdf (ver p. 175).

Ladner, R. (1975). "On the structure of polynomial time reducibility". Em: *Journal of the ACM*. URL: http://weblog.fortnow.com/2005/09/favorite-theorems-np-incomplete-sets.html (ver p. 398).

Land, A. H. e A. G. Doig (1960). "An automatic method of solving discrete programming problems". Em: *Econometrica* 28.3, pp. 497–520. DOI: 10.2307/1910129 (ver p. 162).

Leeuwen, J. van, ed. (1990). *Handbook of theoretical computer science*. Vol. A: Algorithms and complexity. MIT Press. URL: http://www.amazon.com/Handbook-Theoretical-Computer-Science-Vol/dp/0262720140 (ver p. 379).

Leighton, T. (1996). Manuscript, MIT (ver p. 137).

Levenshtein, V. I. (1966). "Binary codes capable of correcting deletions, insertions, and reversals". Em: *Soviet Physics Doklady*, pp. 707–710 (ver p. 110).

Levin, L. (1973). "Universal'ny e pereborny e zadachi (Universal Search Problems: in Russian)". Em: *Problemy Peredachi Informatsii* 9.3, pp. 265–266 (ver p. 32).

Li, J. e J. Peebles (2015). "Replacing Mark Bits with Randomness in Fibonacci Heaps". Em: *Int. Coloq. Automata, Languages, and Progr.* Ed. por M. Halldórsson, K. Iwama, N. Kobayashi e B. Speckmann. Vol. 9134. LNCS, pp. 886–897 (ver p. 259).

Liskov, B. e S. Zilles (1974). "Programming with abstract data types". Em: *Proceedings of the ACM SIGPLAN symposium on Very high level languages*. Vol. 9. 4. DOI: 10.1145/800233.807045 (ver p. 179).

Lloyd, S. (2002). "Computational capacity of the universe". Em: *Physical Review Letters* 88.23. http://focus.aps.org/story/v9/st27 (ver p. 14).

Magazine, M., G.L.Nemhauser e L.E.Trotter (1975). "When the greedy solution solves a class of Knapsack problems". Em: *Operations research* 23.2, pp. 207–217 (ver p. 98).

Meyer, A. R. e L. J. Stockmeyer (1972). "The equivalence problem for regular expression with squaring requires exponential time". Em: *Proc. 12th IEEE Symposium on Switching and Automata Theory*, pp. 125–129 (ver pp. 13, 409).

Monien, B. e E. Speckenmeyer (1985). "Solving satisfiability in less than 2^n steps." Em: *Discrete Appl. Math.* 10, pp. 287–295 (ver p. 160).

Monier, L. (1980). "Evaluation and comparison of two efficient probabilistic primality testing algorithms". Em: *Theoret. Comp. Sci.* 12, pp. 97–108 (ver p. 365).

Munkres, J. (1957). "Algorithms for the assignment and transporation problems". Em: *J. Soc. Indust. Appl. Math* 5.1, pp. 32–38 (ver p. 316).

Noshita, K. (1985). "A theorem on the expected complexity of Dijkstra's shortest path algorithm". Em: *Journal of Algorithms* 6, pp. 400–408 (ver pp. 66, 218).
Park, J.-S., M. Penner e V. K. Prasanna (2004). "Optimizing Graph Algorithms for Improved Cache Performance". Em: *IEEE Trans. Par. Distr. Syst.* 15.9, pp. 769–782 (ver p. 273).
Pearson, D. (2005). "A polynomial time algorithm for the change-making problem". Em: *Operations Research Letters* 33.3, pp. 231–234 (ver p. 98).
Pugh, W. (jun. de 1990). "Skip lists: a probabilistic alternative to balanced trees". Em: *Communications of the ACM* 33.6, pp. 668–676. DOI: 10.1145/78973.78977 (ver p. 191).
Rabin, M. O. (1980). "Probabilistic algorithm for primality testing". Em: *J. Number Theory* 12, pp. 128–138 (ver p. 365).
Roach, E. e V. Pieper (2007). "Die Welt in Zahlen". Em: *Brand eins* 3 (ver p. 349).
Roura, S. (2001). "Improved Master Theorems for Divide-and-Conquer Recurrences". Em: *Journal of the ACM* 48.2, pp. 170–205 (ver p. 137).
Saad, Y. (2003). *Iterative methods for sparse linear systems*. second edition. Philadelphia, PA: SIAM (ver p. 190).
Sack, J. e J. Urrutia, eds. (2000). *Handbook of computational geometry*. Elsevier (ver p. 302).
Schönhage, A. e V. Strassen (1971). "Schnelle Multiplikation grosser Zahlen". Em: *Computing* 7, pp. 281–292 (ver p. 148).
Schrijver, A. (2003). *Combinatorial optimization. Polyhedra and efficiency.* Vol. A. Springer (ver p. 289).
Sedgewick, R. *Algorithms for the masses*. URL: http://www.cs.princeton.edu/~rs/talks/AlgsMasses.pdf (acedido em 14/03/2011) (ver p. 205).
– (1992). *Algorithmen in C++*. Addison-Wesley (ver p. 198).
Sethian, J. A. (1999). *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, Fluid Mechanics, Computer Vision and Materials Science*. Cambridge Monographs on Applied and Computational Mathematics. Cambridge University Press (ver p. 259).
Sipser, M. (1992). "The history and status of the P versus NP question". Em: *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*, pp. 603–619 (ver p. 397).
– (2006). *Introduction to the theory of computation*. Thomson (ver p. 387).
Strassen, V. (1969). "Gaussian Elimination is not Optimal". Em: *Numer. Math* 13, pp. 354–356 (ver pp. 69, 139).
Terrazon (2004). *Soft Errors in Electronic Memory – A White Paper*. Rel. téc. Terrazon Semiconductor (ver p. 349).

Toscani, L. V. e P. A. S. Veloso (2005). *Complexidade de Algoritmos*. 2a. Editora Sagra Luzzatto. URL: http://www.inf.ufrgs.br/~laira/ (ver pp. 24, 27, 45, 61, 79, 435, 436).

Trevisan, L. (2010). P *and* NP. http://lucatrevisan.wordpress.com/2010/04/24/cs254-lecture-2-p-and-np (ver p. 388).

Turing, A. M. (1936). "On computable numbers with an application to the Entscheidungsproblem". Em: *Proc. London MathŜoc.* 2.42, pp. 230–265 (ver p. 370).

Vallée Poussin, C.-J. de la (1896). "Recherches analytiques la théorie des nombres premiers". Em: *Ann. Soc. scient. Bruxelles* 20, pp. 183–256 (ver p. 362).

Vitányi, P. M. B. e L. Meertens (1985). "Big Omega Versus the Wild Functions". Em: *SIGACT News* 16.4 (ver p. 29).

Vitter, J. e P. Flajolet (1990). "Handbook of theoretical computer science". Em: ed. por J. van Leeuwen. Vol. A: Algorithms and complexity. MIT Press. Cap. Average-case analysis of algorithms and data structures. URL: http://www.amazon.com/Handbook-Theoretical-Computer-Science-Vol/dp/0262720140 (ver p. 51).

Wang, J. (1997). "Advances in Algorithms, Languages, and Complexity - In Honor of Ronald V. Book". Em: Kluwer. Cap. Average-Case Intractable NP Problems (ver p. 50).

Wilkinson Microwave Anisotropy Probe (2010). Online. URL: http://map.gsfc.nasa.gov (acedido em 13/03/2011) (ver p. 14).

Williams, V. V. e R. Williams (2010). *Triangle detection versus matrix multiplication: a study of truly subcubic reducibility* (ver p. 118).

Williams, V. V. *Breaking the Coppersmith-Winograd barrier*. URL: www.cs.berkeley.edu/~virgi/matrixmult.pdf (ver p. 69).

Zadeh, N. (1972). "Theoretical Efficiency of the Edmonds-Karp Algorithm for Computing Maximal Flows". Em: *J. ACM* 19.1, pp. 184–192 (ver p. 284).

Zwick, U. (1995). "The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate". Em: *Theoretical Computer Science* 148.1, pp. 165 –170. DOI: DOI:10.1016/0304-3975(95)00022-O (ver p. 283).

Índice