

Algoritmos e complexidade

Notas de aula

Marcus Ritt
Luciana S. Buriol

com contribuições de
Edson Prestes

3 de Dezembro de 2010

Conteúdo

I. Análise de algoritmos	7
1. Introdução e conceitos básicos	9
1.1. Notação assintótica	21
1.2. Notas	27
1.3. Exercícios	27
2. Análise de complexidade	31
2.1. Introdução	31
2.2. Complexidade pessimista	35
2.2.1. Metodologia de análise de complexidade	35
2.2.2. Exemplos	41
2.3. Complexidade média	47
2.4. Outros tipos de análise	61
2.4.1. Análise agregada	61
2.4.2. Análise amortizada	62
2.5. Notas	63
2.6. Exercícios	64
II. Projeto de algoritmos	69
3. Introdução	71
4. Algoritmos gulosos	73
4.1. Introdução	73
4.2. Algoritmos em grafos	77
4.2.1. Árvores espalhadas mínimas	77
4.2.2. Caminhos mais curtos	82
4.3. Algoritmos de seqüenciamento	83
4.4. Tópicos	87
4.5. Notas	92
4.6. Exercícios	92

5. Programação dinâmica	93
5.1. Introdução	93
5.2. Comparação de sequências	97
5.2.1. Subsequência Comum Mais Longa	97
5.2.2. Similaridade entre strings	102
5.3. Problema da Mochila	106
5.4. Multiplicação de Cadeias de Matrizes	107
5.5. Tópicos	111
5.5.1. Algoritmo de Floyd-Warshall	111
5.5.2. Caixeiro viajante	113
5.5.3. Árvore de busca binária ótima	114
5.5.4. Caminho mais longo	118
5.6. Exercícios	119
6. Divisão e conquista	121
6.1. Introdução	121
6.2. Resolver recorrências	123
6.2.1. Método da substituição	124
6.2.2. Método da árvore de recursão	128
6.2.3. Método Mestre	131
6.2.4. Um novo método Mestre	136
6.3. Tópicos	138
6.4. Exercícios	139
7. Backtracking	141
8. Algoritmos de aproximação	155
8.1. Introdução	155
8.2. Aproximações com randomização	165
8.3. Aproximações gulosas	166
8.4. Esquemas de aproximação	172
8.5. Exercícios	175
III. Algoritmos	177
9. Algoritmos em grafos	179
9.1. Fluxos em redes	180
9.1.1. Algoritmo de Ford-Fulkerson	181
9.1.2. O algoritmo de Edmonds-Karp	185
9.1.3. Variações do problema	187

9.1.4. Aplicações	191
9.1.5. Outros problemas de fluxo	194
9.2. Emparelhamentos	196
9.2.1. Aplicações	199
9.2.2. Grafos bi-partidos	199
9.2.3. Emparelhamentos em grafos não-bipartidos	209
9.2.4. Exercícios	210
10. Algoritmos de aproximação	211
10.1. Aproximação para o problema da árvore de Steiner mínima	211
10.2. Aproximações para o PCV	213
10.3. Algoritmos de aproximação para cortes	214
10.4. Exercícios	218
IV. Teoria de complexidade	219
11. Do algoritmo ao problema	221
11.1. Introdução	221
12. Classes de complexidade	233
12.1. Definições básicas	233
12.2. Hierarquias básicas	235
12.3. Exercícios	239
13. Teoria de NP-completude	241
13.1. Caracterizações e problemas em NP	241
13.2. Reduções	243
13.3. Exercícios	253
14. Fora do NP	255
14.1. De P até PSPACE	257
14.2. De PSPACE até ELEMENTAR	262
14.3. Exercícios	264
15. Complexidade de circuitos	267
A. Conceitos matemáticos	275
A.1. Funções comuns	275
A.2. Somatório	279
A.3. Indução	281
A.4. Limites	283

Conteúdo

A.5. Probabilidade discreta	283
A.6. Grafos	285
B. Soluções dos exercícios	287

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 3538 do 2010-12-03, compilada em 3 de Dezembro de 2010. A obra está licenciada sob uma [Licença Creative Commons](#) (Atribuição-Uso Não-Comercial-Não a obras derivadas 2.5 Brasil).

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.

1. Introdução e conceitos básicos

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, correção, 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 [54, Corolário 2.1] (no momento é suficiente saber que isso significa que o tempo para resolver o problema cresce ao 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

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

◊

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 de 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	$\approx 2 \times 10^{102}$

1. Introdução e conceitos básicos

(Os dados correspondem ao consenso científico no momento; obviamente novos descobrimentos podem mudar *Wilkinson Microwave Anisotropy Probe* [73], Lloyd [52])

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

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

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

$$\det(A) = \sum_{\sigma \in S_n} \left(\text{sgn}(\sigma) \prod_{1 \leq i \leq n} a_{i,\sigma(i)} \right).$$

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

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

(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

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

cuja solução é

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

e como $\sum_{1 \leq i < n} 1/i!$ aproxima e temos $n! \leq T_n \leq n!(1 + (A + M)e)$ 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. 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 de Gauss)

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

Saída A em forma triangular superior.

```
1  eliminação gauss( $a \in \mathbb{R}^{n \times n}$ )=
2  for  $i := 1, \dots, n$  do { elimina coluna  $i$  }
3    for  $j := i+1, \dots, n$  do { elimina linha  $j$  }
4      for  $k := n, \dots, i$  do
5         $a_{jk} := a_{jk} - a_{ik}a_{ji}/a_{ii}$ 
6      end for
7    end for
8  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 $n! \sum_{1 \leq i < n} 1/i! = \lfloor n!(e-1) \rfloor$.

1. Introdução e conceitos básicos

obtendo

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

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

Logo temos um algoritmo que determina a solução com

$$\sum_{1 \leq i \leq n} 3(n-i+1)(n-i) = n^3 - n$$

operações de ponto flutuante, que é (exceto valores de n bem pequenos) consideravelmente melhor que os resultados com $n!$ operações acima².

Observe que esse método também fornece o determinante da 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 μ s

²O 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	$N + \log_3 10 \approx N + 2.1$

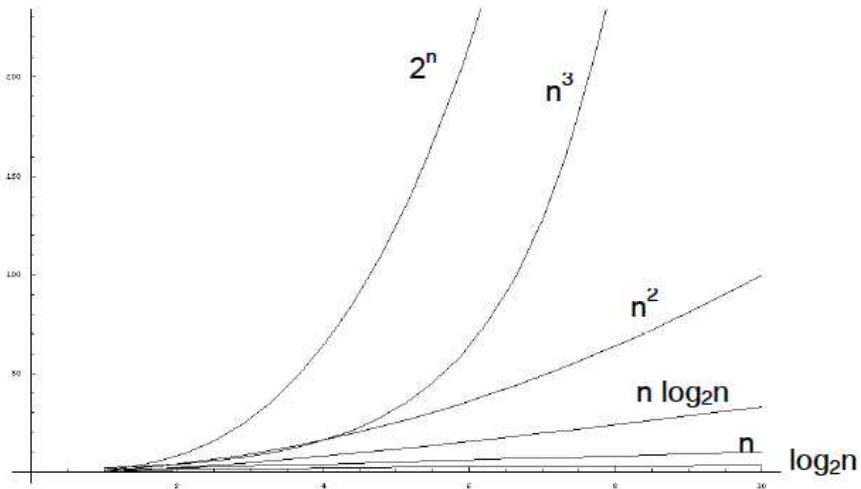
Exemplo 1.4

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 rápida?

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

1. Introdução e conceitos básicos



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.

1. Introdução e conceitos básicos

Complexidade de algoritmos

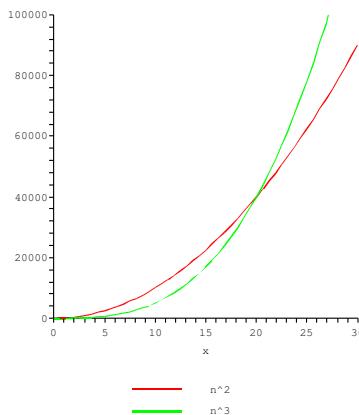
- Considere dois algoritmos A e B com tempo de execução $O(n^2)$ e $O(n^3)$, respectivamente. Qual deles é o mais eficiente ?
- Considere dois programas A e B com tempos de execução $100n^2$ milisegundos,e $5n^3$ milisegundos, respectivamente, qual é o mais eficiente?

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	$\frac{50(10^6) \log 10^6 OP}{10^7 OP/s} = 30s$	$\frac{50(10^6) \log 10^6 OP}{10^9 OP/s} = 0.3s$

◊

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 : Freqüentemente 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.

◊

1. Introdução e conceitos básicos

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, \dots, 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 [18, 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 saíntes.

CICLO HAMILTONIANO

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

Decisão Existe um ciclo hamiltoniano, i.e. um caminho v_1, v_2, \dots, 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 $c : \mathbb{N} \rightarrow \mathbb{R}^+$, 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.

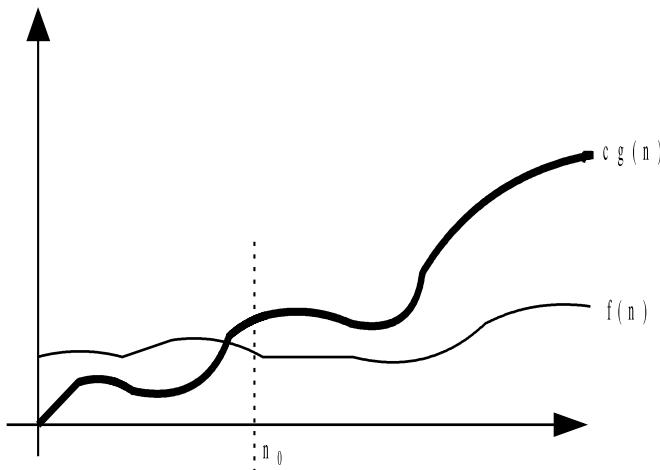
Notação assintótica: O

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

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

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 valores³.

Caso $f \in O(g)$ com constante $c = 1$, digamos que g é uma *cota assintótica superior* de f [66, 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

$$n^2 \in O(n^3 \log_2 n) \quad c = 1; n_0 = 2$$

$$32n \in O(n^3) \quad c = 32; n_0 = 1$$

$$10^n n^2 \notin O(n2^n) \quad \text{porque } 10^n n^2 \leq cn2^n \iff 5^n n \leq c$$

$$n \log_2 n \in O(n \log_{10} n) \quad c = 4; n_0 = 1$$

◇

³Mais correto (mas menos confortável) seria escrever $\lambda n.4n^2 = O(\lambda n.n^2)$

O : Exemplos**Proposição 1.1**

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

$$|p(n)| \in O(n^m) \quad (1.1)$$

Prova.

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

■

Notação assintótica: Outras classes

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

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

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

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

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

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

- Funções que crescem igual à $g(n)$

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)) \quad (1.5)$$

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 equivalentes, substituindo $<$ para \leq e $>$ para \geq (veja exercício 1.10).

1. Introdução e conceitos básicos

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

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

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

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

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

Prova. Exercício. ■

Convenção 1.2

Escrevemos f, g, \dots para funções $f(n), g(n), \dots$ caso não tem ambigüedade 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 direito 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

$$f = O(f) \quad (1.10)$$

$$cO(f) = O(f) \quad (1.11)$$

$$O(f) + O(f) = O(f) \quad (1.12)$$

$$O(O(f)) = O(f) \quad (1.13)$$

$$O(f)O(g) = O(fg) \quad (1.14)$$

$$O(fg) = fO(g) \quad (1.15)$$

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” [66, 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)

$$f \prec g \iff f \in o(g) \quad (1.16)$$

$$f \preceq g \iff f \in O(g) \quad (1.17)$$

$$f \succ g \iff f \in \omega(g) \quad (1.18)$$

$$f \succeq g \iff f \in \Omega(g) \quad (1.19)$$

$$f \asymp g \iff f \in \Theta(g) \quad (1.20)$$

Essas relações são chamadas “notação de Vinogradov”⁴.

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 $[\mathbb{N} \rightarrow \mathbb{R}^+]$

1. $f \preceq g \iff g \succeq f$,
2. \preceq e \succeq são ordenações parciais (reflexivas, transitivas e anti-simétricas em relação de \asymp),
3. $f \prec g \iff g \succ f$,
4. \prec e \succ são transitivas,
5. \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

$$\begin{aligned} f &\in \Theta(f) \\ f \in \Theta(g) &\Rightarrow g \in \Theta(f) \\ f \in \Theta(g) \wedge g &\in \Theta(h) \Rightarrow f \in \Theta(h) \end{aligned}$$

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

⁴Uma 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

$$f \succ g \Rightarrow f \not\geq g \quad (1.21)$$

$$f \prec g \Rightarrow f \not\leq g \quad (1.22)$$

mas as seguintes afirmações *não* são verdadeiras:

$$f \not\geq g \Rightarrow f \succ g$$

$$f \not\leq g \Rightarrow f \prec g$$

$$f \prec g \vee f \asymp g \vee f \succ g \quad (\text{Tricotomia})$$

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 [70].

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 [35].

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 [41].

Nossa discussão da regra de Cramer usa dois métodos naivos para calcular determinantes. Habgood e Arel [34] 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 287.)

Exercício 1.1

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

1. Introdução e conceitos básicos

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 aplique) para Ω .

Exercício 1.7

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

$$f \in O(g) \iff f + c \in O(g) \quad (1.23)$$

Exercício 1.8

Prove ou mostre um contra-exemplo.

1. $\log(1+n) = O(\log n)$
2. $\log O(n^2) = O(\log n)$
3. $\log \log n = O(\log n)$

Exercício 1.9

Considere a função definida pela recorrência

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

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

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

(denotado com \hat{o} para diferenciar da definição o) é equivalente com a definição 1.2 para funções $g(n)$ que são diferentes de 0 a partir de um n_0 .

Exercício 1.11

Mostre que os 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:

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

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, analise a complexidade mutual.

$$\begin{aligned} &n^3, n^3 \log^{1/3} n / \log^{1/3} n, n^3 \log^{1/2} \log n / \log^{1/2} n, n^3 \log^{1/2}, \\ &n^3 \log^{5/7} \log n / \log^{5/7} n, n^3 \log^2 \log n / \log n, n^3 \log^{1/2} \log n / \log n, \\ &n^3 \log n, n^3 \log^{5/4} \log n / \log^{5/4} n, n^3 \log^3 \log n / \log^2 n \end{aligned}$$

Exercício 1.16

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

Exercício 1.17

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

1. Introdução e conceitos básicos

2. 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 $f \in \Omega(g)$.
3. 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 [51])

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

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

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

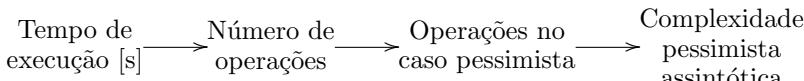
Para analisar a eficiência de algoritmos faz pouco sentido medir os recursos gastos em computadores específicos, porque devido a diferentes conjuntos de instruções, arquiteturas e desempenho dos processadores, as medidas são difíceis de comparar. Portanto, usamos um *modelo* de uma máquina que reflete 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 às vezes está chamado *tese de Church-Turing estendida*.

O plano

Uma hierarquia de abstrações:



Custos de execuções

- Seja E o conjunto de sequências de operações fundamentais.

2. Análise de complexidade

- Para um algoritmo A , com entradas D seja

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

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

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

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

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

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

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

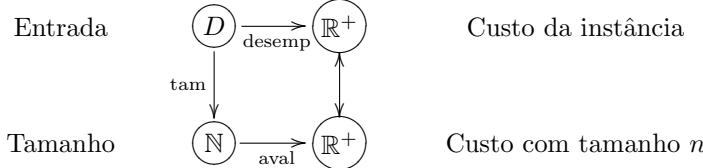
- O objetivo é definir uma função

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

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 freqüentemente não é conhecida, difícil de determinar, ou é diferente em aplicações diferentes.

Definição alternativa

- A complexidade pessimista é definida como

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

2. Análise de complexidade

- 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, \dots, a_n de números em \mathbb{N} .

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

```
1  for i:=1 to n do
2      if (ai = 1) then
3          return true
4      end if
5  end for
6  return false
```

Para analisar o algoritmo, podemos escolher, por exemplo, as operações básicas $O = \{for, if, return\}$ 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

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

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

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

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

$$\text{exec}[\text{Busca1}](d) : D \rightarrow E :$$

$$(a_1, \dots, a_n) \mapsto \begin{cases} (\text{for}, \text{if})^i \text{return} & \text{caso existe } i = \min\{j \mid a_j = 1\} \\ (\text{for}, \text{if})^n \text{return} & \text{caso contrário} \end{cases}$$

Com essas definições temos também a função de desempenho

$$\text{desemp}[\text{Busca1}](n) = \text{custo} \circ \text{exec}[\text{Busca1}] :$$

$$(a_1, \dots, a_n) \mapsto \begin{cases} 2i + 1 & \text{caso existe } i = \min\{j \mid a_j = 1\} \\ 2n + 1 & \text{caso contrário} \end{cases}$$

Agora podemos aplicar a definição da complexidade pessimista para obter

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

Observe que C_p^{\leq} é monotônica, e portanto $C_p^{\leq} = C_p^{\leq}$.

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, $C_o[\text{Busca1}](n) = 2 = O(1)$.

◊

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

2. Análise de complexidade

1. Atribuição: $v := e$
2. Sequência: $c_1; c_2$
3. Condicional: se b então c_1 senão c_2
4. Iteração definida: para i de j até m faça c
5. Iteração indefinida: enquanto b faça c

A forma “se b então c_1 ” vamos tratar como abreviação de “se b então c_1 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

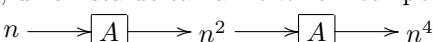
A seguir vamos entender implicitamente todas operações sobre funções *pontualmente*, i.e. para alguma operação \circ , e funções f, g com $\text{dom}(f) = \text{dom}(g)$ temos

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

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 $\Theta(n^2)$.

- A sequência $A; A$, mesmo sendo composta pelos dois algoritmos com $\Theta(n^2)$ *individualmente*, tem complexidade $\Theta(n^4)$.

- Portanto, vamos diferenciar 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 $c_p[S] = \Theta(n^2 + n) = \Theta(n^2)$. \diamond

2. Análise de complexidade

Exemplo 2.3

Considere uma partição das entradas do tamanho n tal que $\{d \in D \mid tam(d) = n\} = D_1(n) \dot{\cup} D_2(n)$ 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

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

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 $O(1)$

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

- 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:

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

- Freqüentemente $c_p[\leftarrow_e]$ 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 $v := \text{ordene}(w)$. Com complexidade pessimista para a ordenação da lista $c_p[\text{ordene}(w)] = O(n^2)$ e complexidade $c_p[\leftarrow_e] = O(n)$ para a transferência, temos $c_p[v := \text{ordene}(w)] = O(n^2) + O(n) = O(n^2)$. \diamond

Iteração definida

Seja $C = \text{para } i \text{ de } j \text{ até } m \text{ faça } c$

- O número de iterações é fixo, mas j e m dependem da entrada d .
- Seja $N(n) = \max_d \{m(d) - j(d) + 1 \mid \text{tam}(d) \leq n\}$ 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 $C = \text{enquanto } b \text{ faça } c$

- 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 $h(n) = \max\{H(d) \mid \text{tam}(d) \leq n\}$ o número máximo de iterações com entradas até tamanho n .
- Em analogia com a iteração definida temos uma sequência

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

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.

2. Análise de complexidade

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 $C = \text{se } b \text{ então } c_1 \text{ senão } c_2$

- 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 c_1 obtemos com $c_p[\text{skip}] = 0$

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

Exemplo 2.5

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

Exemplo

- 1 se $\text{hd}(u) = 0$ então
- 2 $v := \text{ordena}(u)$
- 3 senão
- 4 $s := \text{soma}(u)$

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

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

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

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

◊

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, \dots, a_n de números inteiros.

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

```

1  for i:=1 to n
2    { Inv:  $a_{n-i+2} \leq \dots \leq a_n$  são os  $i - 1$  maiores elementos }
3    for j:=1 to n-i
4      if  $a_j > a_{j+1}$  then
5        swap  $a_j, a_{j+1}$ 
6      end if
7    end for
8  end for

```

Bubblesort: Complexidade

- A medida comum para algoritmos de ordenação: o número de comparações (de chaves).

2. Análise de complexidade

- Qual a complexidade pessimista?

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

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

◊

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, \dots, a_n de números inteiros.

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

```
1  for i:=2 to n do
2    { invariante:  $a_1, \dots, a_{i-1}$  ordenado }
3    { coloca item i na posição correta }
4    c:=a_i
5    j:=i;
6    while c < a_{j-1} and j > 1 do
7      a_j:=a_{j-1}
8      j:=j - 1
9    end while
10   a_j:=c
11 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 [66, cap. 3.3].)

Máximo**Algoritmo 2.4 (Máximo)****Entrada** Uma sequência de números a_1, \dots, a_n com $n > 0$.**Saída** O máximo $m = \max_i a_i$.

```

1   $m := a_1$ 
2  for  $i := 2, \dots, n$  do
3      if  $a_i > m$  then
4           $m := a_i$ 
5      end if
6  end for
7  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 $\Theta(n)$. \diamond

Exemplo 2.9 (Busca seqüencial)

O segundo algoritmo que queremos estudar é a busca seqüencial.

Busca seqüencial

Algoritmo 2.5 (Busca seqüencial)

Entrada Uma sequência de números a_1, \dots, 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.

```

1  for i := 1, ..., n do
2      if  $a_i = c$  then
3          return i
4      end if
5  end for
6  return  $\infty$ 
```

Busca seqüencial

- 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 $\Theta(n)$.

◊

Counting-Sort

Algoritmo 2.6 (Counting-Sort)

Entrada Um inteiro k , uma sequência de números a_1, \dots, a_n e uma sequência de contadores c_1, \dots, c_n .

Saída Uma sequência ordenada de números b_1, \dots, b_n .

```

1  for i := 1, ..., k do
2       $c_i := 0$ 
3  end for
4  for i := 1, ..., n do
5       $c_{a_i} := c_{a_i} + 1$ 
6  end for
7  for i := 2, ..., k do
```

```

8       $c_i := c_i + c_{i-1}$ 
9  end for
10 for  $i := n, \dots, 1$  do
11    $b_{c_{a_i}} := a_i$ 
12    $c_{a_i} := c_{a_i} - 1$ 
13 end for
14 return  $b_1, \dots, b_n$ 

```

Loteria Esportiva

Algoritmo 2.7 (Loteria Esportiva)

Entrada Um vetor de inteiros $r[1, \dots, n]$ com o resultado e uma matriz $A_{13 \times n}$ com as apostas dos n jogadores.

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

```

1   $i := 1$ 
2  while  $i \leq n$  do
3     $p_i := 0$ 
4    for  $j$  de  $1, \dots, 13$  do
5      if  $A_{i,j} = r_j$  then  $p_i := p_i + 1$ 
6    end for
7     $i := i + 1$ 
8  end while
9  for  $i$  de  $1, \dots, n$  do
10   if  $p_i = 13$  then print(Apostador  $i$  é ganhador!)
11 end for
12 return  $p$ 

```

Exemplo 2.10 (Busca Binária)

Busca Binária

Algoritmo 2.8 (Busca Binária)

Entrada Um inteiro x e uma sequência ordenada $S = a_1, a_2, \dots, a_n$ de números.

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

```

1  i := 1
2  f := n
3  m :=  $\left\lfloor \frac{f-i}{2} \right\rfloor + i$ 
4  while i ≤ f do
5      if am = x then return m
6      if am < x then f := m - 1
7      else i := m + 1
8      m :=  $\left\lfloor \frac{f-i}{2} \right\rfloor + i$ 
9  end while
10 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

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

◊

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

$$c_{ik} = \sum_{1 \leq j \leq n} a_{ij} b_{jk} \quad 1 \leq i \leq m; 1 \leq k \leq o.$$

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

```

1   for i := 1, ..., m do
2       for k := 1, ..., o do
3           cik := 0
4           for j := 1, ..., n do
5               cik := cik + aijbjk
6           end for
7       end for
8   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 $\Theta(n^3)$ operações¹.

◊

2.3. Complexidade média

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

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

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

2. Análise de complexidade

- 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?

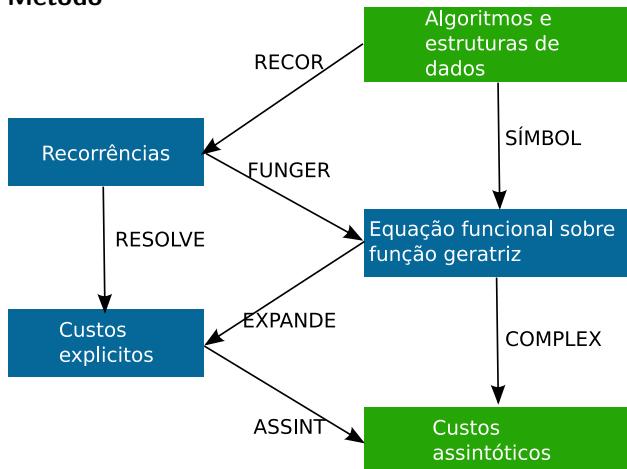
- Possibilidade: Problemas intratáveis viram tratáveis?
- Exemplos de tempo esperado:
 - CAMINHO HAMILTONIANO: linear!
 - PARADA NÃO-DETERMINÍSTICO EM k PASSOS: fica NP-completo.

(Resultados citados: [33, 19] (Caminho Hamiltoniano), [72] (Parada em k passos).)

Criptografia

- Alguns métodos da Criptografia dependem da existência de “funções sem volta” (inglês: one-way functions).
- Uma função sem volta $f : \{0,1\}^* \rightarrow \{0,1\}^*$ 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



[71]

Exemplo 2.12 (Busca seqüencial)

(Continuando exemplo 2.9.)

Busca seqüencial

- 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

$$c_n[BS](n) = \frac{1}{n+1} \left(n + \sum_{1 \leq i \leq n} i \right)$$

iterações e uma complexidade média de $\Theta(n)$.

◊

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 $D_n = \{(a_1, \dots, a_n) \mid a_1 \geq 1, \dots, a_n \geq 1\}$. 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

$$\Pr[(a_1, \dots, a_n)] = \prod_{1 \leq i \leq n} 2^{-a_i}$$

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

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

Temos $\text{desemp}[A] = 2p + 1$ (veja os resultados do exemplo 2.1). Para i estar na primeira posição com elemento 1 as posições $1, \dots, i-1$ 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 $\Pr[p = n] = 2^{1-n}$.

2. Análise de complexidade

A complexidade média calcula-se como

$$\begin{aligned}
 c_p[A](n) &= E[\text{desemp}[A]] = E[2p + 1] = 2E[p] + 1 \\
 E[p] &= \sum_{i \geq 0} \Pr[p = i]i = 2^{-n}n + \sum_{1 \leq i \leq n} 2^{-n}n \\
 &= 2^{-n}n + 2 - 2^{-n}(n+2) = 2 - 2^{1-n} \quad (\text{A.33}) \\
 c_p[A](n) &= 5 - 2^{2-n} = O(1)
 \end{aligned}$$

A seguinte tabela mostra os custos médios para $1 \leq n \leq 9$

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

$$\sum_{1 \leq j \leq i} j/i = (i+1)/2$$

comparações em média.

- Logo o número esperado de comparações é

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

◊

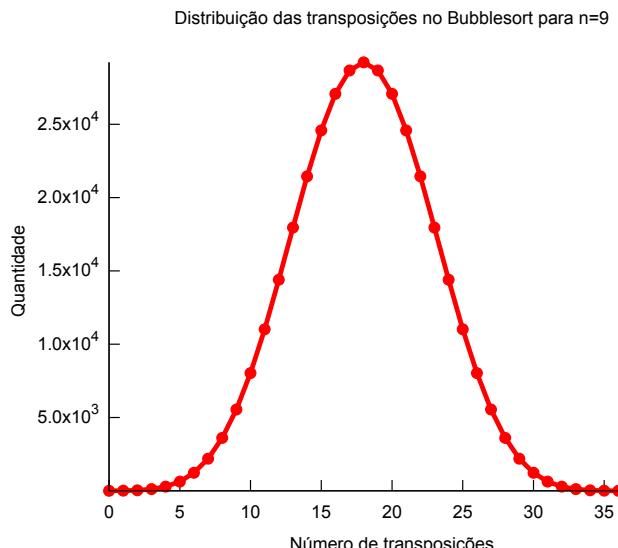
Exemplo 2.15 (Bubblesort)

(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.

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

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

2. Análise de complexidade

- Freqüentemente o número de inversões facilita a análise de algoritmos de ordenação.
- Em particular para algoritmos com transposições de elementos adjacentes:
$$\#\text{Transposições} = \#\text{ Inversões}$$

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:

$$35124; \quad 42153$$

- 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

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

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

◊

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, \dots, b_n .
- b_i é o número de elementos na esquerda de i que são maiores que i .
- Exemplo: Para 53142
$$\begin{array}{ccccc} b_1 & b_2 & b_3 & b_4 & b_5 \\ 2 & 3 & 1 & 1 & 0 \end{array}$$
- 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
$$\begin{array}{ccccc} b_1 & b_2 & b_3 & b_4 & b_5 \\ 3 & 1 & 2 & 1 & 0 \end{array}$$
- 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 $X = \sum_{1 \leq i \leq n} X_i$.
- Portanto, o número esperado de máximos locais é

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

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

**Exemplo 2.17 (Quicksort)**

Nessa seção vamos analisar é Quicksort, um algoritmo de ordenação que foi inventado pelo C.A.R. Hoare em 1960 [38].

Quicksort

- Exemplo: o método Quicksort de ordenação por comparação.
- Quicksort usa divisão e conquista.
- Idéia:
 - 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, \dots, 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 $i \in]m, r]$.

```

1  escolhe um pivô  $a_p$ 
2  troca  $a_p$  e  $a_r$ 
3  i := l - 1 { último índice menor que pivô }
4  for j:=1 to r - 1 do
5    if  $a_j \leq a_r$  then
6      i := i + 1
7      troca  $a_i$  e  $a_j$ 
8    end if
9  end for
10 troca  $a_{i+1}$  e  $a_r$ 
11 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

$$c_p[\text{Partition}] = n - 1 = \Theta(n) \quad (\text{Primeiro elemento})$$

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

Quicksort

Algoritmo 2.11 (Quicksort)

Entrada Índices l, r e um vetor a com elementos a_l, \dots, a_r .

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

2. Análise de complexidade

```

1  if  $l < r$  then
2      m := Partition(l, r, a);
3      Quicksort(l, m - 1, a);
4      Quicksort(m + 1, r, a);
5  end if

```

$$\begin{aligned}
\text{desemp}[QS](a_l, \dots, a_r) &= \text{desemp}[P](a_l, \dots, a_r) \\
&\quad + \text{desemp}[QS](a_l, \dots, a_{m-1}) + \text{desemp}[QS](a_{m+1}, \dots, a_r) \\
\text{desemp}[QS](a_l, \dots, a_r) &= 0 \quad \text{se } l \geq r
\end{aligned}$$

Complexidade pessimista

- Qual a complexidade pessimista?

- Para entrada $d = (a_1, \dots, a_n)$, sejam $d_l = (a_l, \dots, a_{m-1})$ e $d_r = (a_{m+1}, \dots, a_r)$

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

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

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 partiçao das entradas com tamanho n tal que para $d \in D_n^i$ temos $|d_l| = i-1$ (e conseqüentemente $|d_r| = n-i$). Então

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

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 $i = 1$

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

- Caso $i = n/2$

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

- 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 $[1, n]$).

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

2. Análise de complexidade

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

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

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 [44], p. 119] tem mais detalhes).

Se denotamos o desempenho com $T_n = c_m[QS](n)$, obtemos a recorrência

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

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

1. 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).

2. Escolhe o maior dos dois primeiros elementos². Temos $\Pr[X = i] = 2(i-1)/(n(n-1))$.

²Supomos para análise que todos elementos são diferentes. Um algoritmo prático tem que considerar o caso que um ou mais elementos são iguais [66, p. 72].

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

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

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

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

2. Análise de complexidade

Logo temos as probabilidades

Pivô i	1	2	3
$\Pr[X = i]$	0	$1/3$	$2/3$

◊

Resolver a equação

- A solução da recorrência

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

é $T_n = \Theta(n \ln n)$.

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

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

multiplicando por n obtemos

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

a mesma equação para $n - 1$ é

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

subtraindo a segunda da primeira obtemos

$$\begin{aligned} nT_n - (n - 1)T_{n-1} &= 2n + 2T_{n-1} \quad \text{para } n > 0, \text{ verificando } n = 1 \\ nT_n &= (n + 1)T_{n-1} + 2n \end{aligned}$$

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

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

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

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

e portanto

$$\begin{aligned} T_n &= 2(n+1) \sum_{1 \leq i \leq n} \frac{1}{i+1} \\ &= 2(n+1)\left(H_n - \frac{n}{n+1}\right) \\ &= \Theta(n \ln n) \end{aligned}$$

◊

2.4. Outros tipos 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 direcionado estruturado como uma sequência das listas de adjacências de seus nós.

Saída Posição i vetor de distâncias (número de arcos) de cada nó ao origem.

```

1  for cada vértice  $u \in V - \{s\}$  do
2       $c_u := \text{BRANCO}$ 
3       $d_u = \infty$ 
4  end for
5   $c_s := \text{CINZA}$ 
```

2. Análise de complexidade

```

6    $d_s := 0$ 
7    $Q := \emptyset$ 
8   Enqueue(Q, s)
9   while  $Q \neq \emptyset$ 
10    u := Dequeue(Q)
11    for cada  $v \in Adj(u)$ 
12      if  $c_v = \text{BRANCO}$ 
13        then  $c_v = \text{CINZA}$ 
14         $d_v = d_u + 1$ 
15        Enqueue(Q, v)
16      end if
17    end for
18     $c_u = \text{PRETO}$ 
19 end while

```

Uma análise simples observa que o laço *while* nas linhas 9–19 tem no máximo $|V|$ iterações, uma para cada nó do grafo. O laço *for* interior nas linhas 11–17 tem d_u iterações, sendo d_u o grau de saída do nó u . No caso pessimista $d_u = |V| - 1$, portanto esta análise resulta numa complexidade pessimista $O(|V|^2)$.

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 $\sum_{u \in V} d_u \leq |E|$. Portanto essa análise resulta numa complexidade pessimista $O(|V| + |E|)$. \diamond

2.4.2. Análise amortizada

Exemplo 2.20

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

Outra forma de análise amortizada, é usando uma *função potencial* φ , que associa a cada estado de uma estrutura de dados um valor positivo (a “poupança”). O custo amortizado de uma operação que transforma uma estrutura

e_1 em uma estrutura e_2 e $c - \varphi(e_1) + \varphi(e_2)$, com c o custo de operação. No exemplo do contador, podemos usar como $\varphi(i)$ o número de bits na representação binário de i . Agora, se temos um estado e_1

$$\underbrace{11\cdots 1}_{p \text{ bits um}} \ 0 \ \underbrace{\cdots}_{q \text{ bits um}}$$

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

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

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

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

◊

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

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

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

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

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

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

2.5. Notas

O algoritmo 2.9 para multiplicação de matrizes não é o melhor possível: Temos o algoritmo de Strassen que precisa somente $n^{\log_2 7} \approx n^{2.807}$ multiplicações (o algoritmo está detalhado no capítulo 6.3) e o algoritmo de Coppersmith-Winograd com $n^{2.376}$ multiplicações [13]. Ambas são pouco usadas na prática porque o 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). A conjectura atual é que existe um algoritmo (ótimo) de $O(n^2)$.

2.6. Exercícios

(Soluções a partir da página 292.)

Exercício 2.1

Qual a complexidade pessimista dos seguintes algoritmos?

Algoritmo 2.13 (Alg1)

Entrada Um problema de tamanho n .

```
1  for i := 1...n do
2    for j := 1...2i
3      operações constantes
4      j := j + 1 // iterações com valores ímpares de j
5    end for
6  end for
```

Algoritmo 2.14 (Alg2)

Entrada Um problema de tamanho n .

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

Algoritmo 2.15 (Alg3)

Entrada Um problema de tamanho n .

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

```
5 end for
```

Algoritmo 2.16 (Alg4)**Entrada** Um problema de tamanho n .

```
1 for i := 1 ... n do
2     j := 1
3     while j ≤ i do
4         operações com complexidade  $O(2^j)$ 
5         j := j + 1
6     end for
7 end for
```

Algoritmo 2.17 (Alg5)**Entrada** Um problema de tamanho n .

```
1 for i := 1 ... n do
2     j := i
3     while j ≤ n do
4         operações com complexidade  $O(2^j)$ 
5         j := j + 1
6     end for
7 end for
```

Exercício 2.2

Tentando resolver a recorrência

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

que ocorre na análise do Quicksort (veja exemplo 2.17), o aluno J. Rapidez

2. Análise de complexidade

chegou no seguinte resultado: Supondo que $T_n = O(n)$ obtemos

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

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, \dots, 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, \dots, 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

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

Exercício 2.6

Algoritmo 2.18 (Count6)

Entrada Uma sequência a_1, \dots, a_n com $a_i \in [1, 6]$.

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

```

1    $k := 0$ 
2   for  $i = 1, \dots, n$  do
3       if  $a_i = 6$  then
4            $k := k + 1$ 
5       end if
6   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 $P[1] = 1/2$, $P[2] = P[3] = P[4] = P[5] = P[6] = 1/10$?

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. 60)

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

resolve as recorrências

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

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

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).

2. Análise de complexidade

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 $O(1)$.

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 [66, cap. 5.1.3].)

Algoritmos gulosos

- Algoritmos gulosos se aplicam a problemas de otimização.
- Idéia 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 freqüentemente são apropriados para processamento online.

Trocar moedas

TROCA MÍNIMA

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

Solução Números c_1, \dots, c_n tal que $s = \sum_{1 \leq i \leq n} c_i v_i$

Objetivo Minimizar o número de unidades $\sum_{1 \leq i \leq n} c_i$.

A abordagem gulosa

```

1  for  $i := 1, \dots, n$  do
2     $c_i := \lfloor s/v_i \rfloor$ 
3     $s := s - c_i v_i$ 
4  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? \diamond

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, \dots, 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: freqüentemente 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)

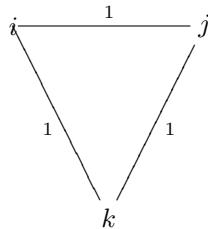
Um problema de otimização tem *subestrutura ótima* se uma solução ótima (mínima ou máxima) do problema consiste em soluções ótimas das subproblemas.

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).

4. Algoritmos gulosos

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 ijk , mas o subcaminho kj não é o subcaminho mais longo entre k e j . \diamond

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 .

```

1  resolve( $I$ ) := 
2     $S^* := \text{nil}$                                 { melhor solução }
3    for todos subproblemas  $I'$  de  $I$  do
4       $S' := \text{resolve}(I')$ 
5      estende  $S'$  para uma solução  $S$  de  $I$ 
6      if  $S'$  é a melhor solução then
7         $S^* := S$ 
8      end if
9    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.

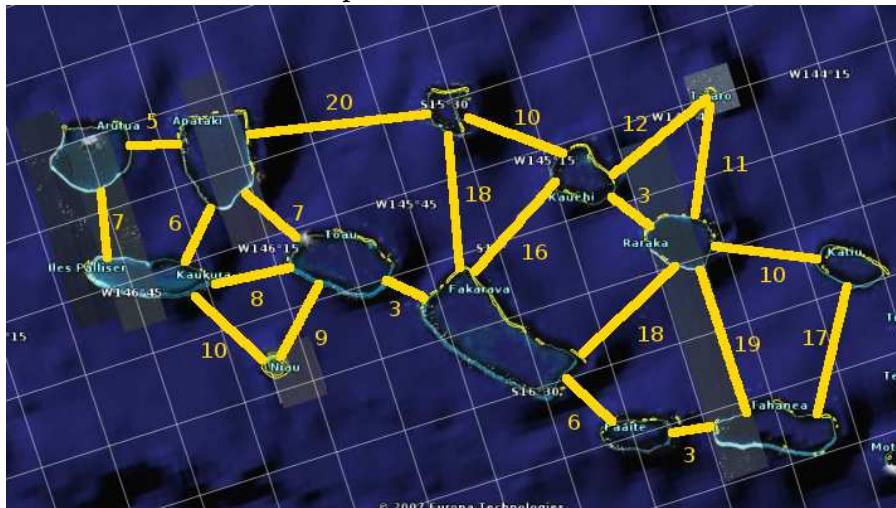
Uma subestrutura ótima é uma condição necessária para um algoritmo guloso ou de programação dinâmica ser ótima, mas ela não é suficiente.

4.2. Algoritmos em grafos

4.2.1. Árvores espalhadas mínimas

Motivação

Pontes para Polinésia Francesa!



Árvores espalhadas mínimas (AEM)

ÁRVORE ESPALHADA MÍNIMA (AEM)

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

Solução Um subgrafo $H = (V_H, E_H)$ conexo.

Objetivo Minimiza os custos $\sum_{e \in E_H} c(e)$.

- 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

Resolver AEM

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

Lema 4.2 (Corte)

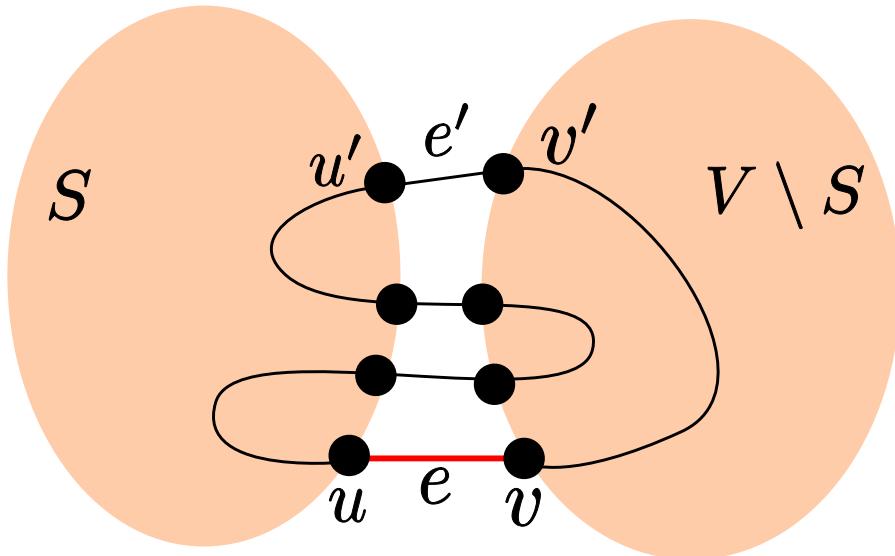
Considere um *corte* $V = S \cup V \setminus S$ (com $S \neq \emptyset$, $V \setminus S \neq \emptyset$). O arco mínimo entre S e $V \setminus S$ faz parte de qualquer AEM.

Prova. Na prova vamos supor, que todos pesos são diferentes.

Suponha um corte S tal que o arco mínimo $e = \{u, v\}$ entre S e $V \setminus S$ não faz parte de um AEM T . Em T existe um caminho de u para v que contém ao menos um arco e' que cruza o corte. Nossa afirmação: Podemos substituir e' com e , em contradição com a minimalidade do T .

Prova da afirmação: Se substituirmos e' por e obtemos um grafo T' . Como e é mínimo, ele custa menos. O novo grafo é conexo, porque para cada par de nós ou temos um caminho já em T que não faz uso de 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.

■

Prova**Resolver AEM**

- A característica do corte possibilita dois algoritmos simples:
 1. 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*.
 2. Começa sem arcos, e repetidamente adicione o arco com custo mínimo que não produz um ciclo: *algoritmo de Kruskal*.

AEM: Algoritmo de Prim**Algoritmo 4.2 (AEM-Prim)**

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

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

4. Algoritmos gulosos

```

1  $V' := \{v\}$  para um  $v \in V$ 
2  $E_T := \emptyset$ 
3 while  $V' \neq V$  do
4     escolhe  $e = \{u, v\}$  com custo mínimo
5         entre  $V'$  e  $V \setminus V'$  (com  $u \in V'$ )
6          $V' := V' \cup \{v\}$ 
7          $E_T := E_T \cup \{e\}$ 
8 end while

```

AEM: Algoritmo de Kruskal

Algoritmo 4.3 (AEM-Kruskal)

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

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

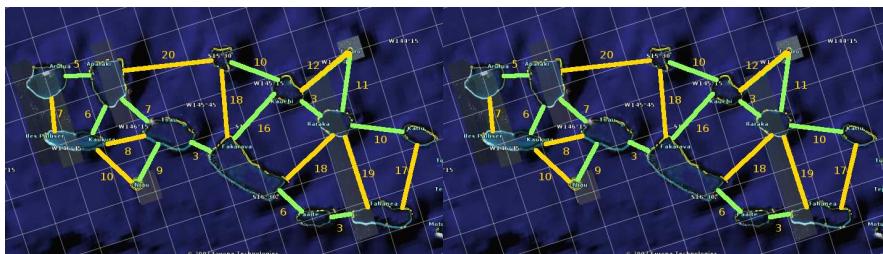
```

1  $E_T := \emptyset$ 
2 while  $(V, E_T)$  não é conexo do
3     escolha  $e$  com custo mínimo que não produz círculo
4      $E_T := E_T \cup \{e\}$ 
5 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 $Q.\text{atualiza}(e, c')$ modifica a chave de e para v' caso v' é menor que a chave atual.
- Ambas operações podem ser implementadas com custo $O(\log n)$.

AEM: Algoritmo de Prim

Algoritmo 4.4 (AEM-Prim)

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

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

```

1   $E_T := \emptyset$ 
2   $Q := \{((v, u), c(v, u)) \mid u \in N(v)\}$  { nós alcancáveis }
3   $Q := Q \cup \{((u, u), \infty) \mid u \in V \setminus N(v) \setminus \{v\}\}$  { nós restantes }
4  while  $Q \neq \emptyset$  do
5     $((u, v), c) := Q.\min()$  {  $(u, v)$  é arco mínimo }
6    for  $w \in N(v)$  do
7       $Q.\text{atualiza}((v, w), c(v, w))$ 
8    end for
9     $E_T := E_T \cup \{(u, v)\}$ 
10   end while

```

4. Algoritmos gulosos

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.
- $c_p[\text{AEM-PRIM}] = O(n \log n + m \log n) = O(m \log n)$

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 freqüente: 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 $G = (\{v_1, \dots, v_n\}, E)$ com função de custos $c : E \rightarrow \mathbb{R}^+$ e um nó inicial $s \in V$.

Solução Uma atribuição $d : V \rightarrow \mathbb{R}^+$ 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 seqüenciamento**Seqüenciamento de intervalos**

Considere o seguinte problema

SEQÜENCIAMENTO DE INTERVALOS

Instância Um conjunto de intervalos $S = \{[c_i, f_i], 1 \leq i \leq n\}$, 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 (Seqüenciamento de intervalos)**

Entrada Um conjunto de intervalos $S = \{[c_i, f_i] \mid 1 \leq i \leq n\}$, cada com começo c_i e fim f_i tal que $c_i < f_i$.

Saída Um conjunto máximo de intervalos compatíveis C .

```

1    $C := \emptyset$ 
2   while  $S \neq \emptyset$  do
3       Seja  $[c, f] \in S$  com  $f$  mínimo
4        $C := C \cup [c, f]$ 
5        $S := S \setminus \{i \in S \mid i \cap [c, f] \neq \emptyset\}$ 
6   end while
7   return  $C$ 

```

Seja $C = ([c_1, f_1], \dots, [c_n, f_n])$ o resultado do algoritmo SEQÜENCIAMENTO DE INTERVALOS e $O = ([c'_1, f'_1], \dots, [c'_m, f'_m])$ seqüenciamento máximo, ambos em ordem crescente.

Proposição 4.2

Para todo $1 \leq i \leq n$ 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 vai escolher entre os intervalos que começam depois f_i o com terminação mínima. O próximo intervalo $[c'_{i+1}, m'_{i+1}]$ do seqüenciamento 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 seqüenciamento do algoritmo guloso é ótimo.

Prova. Suponha que o algoritmo guloso retorna menos intervalos C que um seqüenciamento ótimo O . Pela proposição 4.2, o último (n -ésimo) 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, uma 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 intervalo 1 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 seqüenciamento de intervalos define um grafo não-direcionado $G = (V, E)$ com

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

- 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.

4. Algoritmos gulosos

Variação do problema

Considere uma variação de SEQÜENCIAMENTO DE INTERVALOS:

PARTICIONAMENTO DE INTERVALOS

Instância Um conjunto de intervalos $S = \{[c_i, f_i], 1 \leq i \leq n\}$, 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 $S = \{[c_i, f_i], 1 \leq i \leq n\}$, 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.

- 1 Ordene S em ordem de começo crescente .
- 2 **for** $i := 1, \dots, n$ **do**
- 3 Exclui rótulos de intervalos
- 4 precedentes conflitantes
- 5 Atribui ao intervalo i o número
- 6 inteiro mínimo ≥ 1 que sobra

Corretude

- Com profundidade d o algoritmo precisa ao menos d rótulos.
- De fato ele precisa exatamente d rótulos. Por quê?
- 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 $c : V \rightarrow C$ 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)

GATCCCTCCGCTCTGGGGAGGCAGCGCTGGCGGCCGG ...

com 1666bp¹.

¹ “bp” é a abreviação do inglês “base pair” (par de bases)

4. Algoritmos gulosos

- 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 freqüências, use códigos de diferente comprimento

$$\begin{array}{cccc} A & G & T & C \\ \hline .18 & .30 & .16 & .36 \end{array}$$

Códigos: Exemplos

- Tentativa 1

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

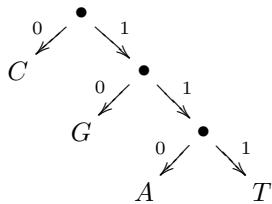
- Desvantagem: Ambíguo! $01 = TA$ ou $01 = G$?
- Tentativa 2

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

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

Códigos livre de prefixos

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



Cada código livre de prefixo pode ser representado usando uma árvore binária: Começando com a raiz, o sub-árvore da esquerda representa os códigos que começam com 0 e a sub-árvore da direita representa os códigos que começam com 1. Esse processo 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 de 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.

```

1  if |C| = 0 then
2      return Nil
3  end if
4  if C = {ε} then
5      return Leaf
6  end if
7  Escreve C = 0C1 ∪ 1C2
8  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 freqüência f é

$$-\log_2 f,$$

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

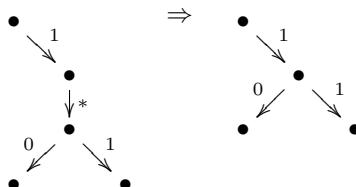
$$A = 1/3, B = 2/3; \quad H \approx 0.92b$$

mas o código ótimo precisa ao menos 1b por símbolo.

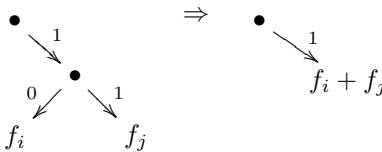
- Nossa exemplo: $H \approx 1.92$.

Como achar o melhor código?

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



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



Algoritmo

Algoritmo 4.8 (Huffman)

Entrada Um alfabeto de símbolos S com uma freqüência f_s para cada $s \in S$.

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

```

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

Exemplo 4.4

Saccharomyces cerevisiae

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

MSITNGTSRSVSAMGHPAVERYTPGHIVCVGTHKVEVV...

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 freqüências

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

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

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

4.5. Notas

O algoritmo guloso para sistemas de moedas Magazine, Nemhauser e Trotter [53] dão critérios necessários e suficientes para uma solução gulosa do problema de troca ser ótima. Dado um sistema de moedas, Pearson [56] 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. [15] estudam sistemas de moedas totalmente gulosos.

4.6. Exercícios

(Soluções a partir da página 295.)

Exercício 4.1 (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: [43]).

Exercício 4.2 (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: [43]).

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	\dots
#	1	1	2	3	5	8	13	\dots

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

```
1 fib(n) :=  
2   if n ≤ 1 then  
3     return 1  
4   else  
5     return fib(n - 1) + fib(n - 2)  
6   end if  
7 end
```

Qual a complexidade dessa implementação? Temos a recorrência de tempo

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

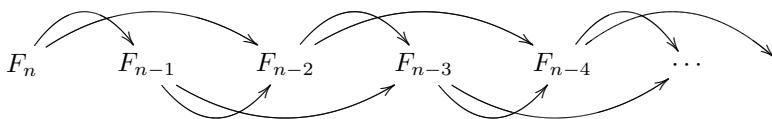
5. Programação dinâmica

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

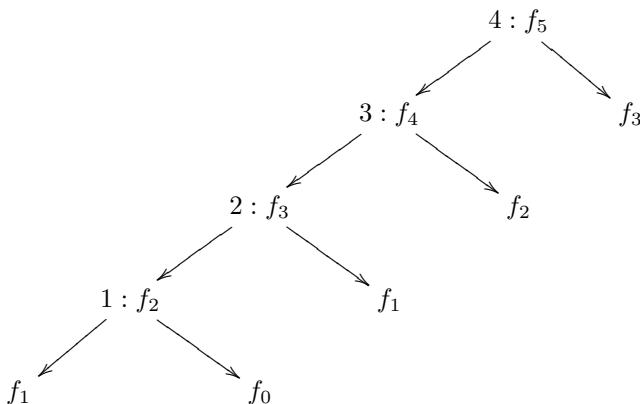


mas somente um número polinomial de valores diferentes! Idéia: usar uma *cache*!

```

1   $f_0 := 1$ 
2   $f_1 := 1$ 
3   $f_i := \perp$  para  $i \geq 2$ 
4
5  fib( $n$ ) :=
6    if  $f_n = \perp$  then
7       $f_n := \text{fib}(n-1) + \text{fib}(n-2)$ 
8    end if
9    return  $f_n$ 
10 end
  
```

Exemplo de uma execução:



	f_5	f_4	f_3	f_2	f_1	f_0
Inicial	\perp	\perp	\perp	\perp	1	1
1	\perp	\perp	\perp	2	1	1
2	\perp	\perp	3	2	1	1
3	\perp	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

```

1 fib(n) :=
2   f0 := 1
3   f1 := 1
4   for i ∈ [2, n] do
5     fi := fi-1 + fi-2
6   end for
7   return fn
8 end

```

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

```

1 fib(n) :=
2   f := 1
3   g := 1
4   for i ∈ [2, n] do
5     { invariante: f = Fi-2 ∧ g = Fi-1 }
6     g := f + g
7     f := g - f
8     { invariante: f = Fi-1 ∧ g = Fi }
9   end for

```

A idéia de armazenar valores intermediários usados freqüentemente numa computação recursiva é uma das idéias 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*).

Programação Dinâmica (PD)

1. Para aplicar PD o problema deve apresentar **subestrutura ótima** (uma

5. Programação dinâmica

solução ótima para um problema contém soluções ótimas de seus subproblemas) e **superposição de subproblemas** (reutiliza soluções de subproblemas).

2. Pode ser aplicada a problemas NP-completos e polinomiais.
3. Em alguns casos o algoritmo direto tem complexidade exponencial, enquanto que o algoritmo desenvolvido por PD é polinomial.
4. Às vezes a complexidade continua exponencial, mas de ordem mais baixa.
5. É útil quando não é fácil chegar a uma sequência ótima de decisões sem testar todas as sequências possíveis para então escolher a melhor.
6. Reduz o número total de sequências viáveis, descartando aquelas que sabidamente não podem resultar em sequências ótimas.

Idéias básicas da PD

- Objetiva construir uma resposta ótima através da combinação das respostas obtidas para subproblemas
- Inicialmente a entrada é decomposta em partes mínimas e resolvidas de forma ascendente (bottom-up)
- A cada passo os resultados parciais são combinados resultando respostas para subproblemas maiores, até obter a resposta para o problema original
- A decomposição é feita uma única vez, e os casos menores são tratados antes dos maiores
- Este método é chamado **ascendente**, ao contrário dos métodos recursivos que são métodos **descendentes**.

Passos do desenvolvimento de um algoritmo de PD

1. Caracterizar a estrutura de uma solução ótima
2. Definir recursivamente o valor de uma solução ótima
3. Calcular o valor de uma solução ótima em um processo ascendente
4. Construir uma solução ótima a partir de informações calculadas

5.2. Comparação de sequências

5.2.1. Subsequência Comum Mais Longa

Subsequência Comum Mais Longa

- Dada uma sequência $X = \langle x_1, x_2, \dots, x_m \rangle$, uma outra sequência $Z = \langle z_1, z_2, \dots, z_k \rangle$, é uma subsequência de X se existe uma sequência estritamente crescente $\langle i_1, i_2, \dots, i_k \rangle$ de índices de X tais que, para todo $j = 1, 2, \dots, k$, temos $x_{i_j} = z_j$.
- Exemplo: $Z = \langle B, C, D, B \rangle$ é uma subsequência de $X = \langle A, B, C, D, A, B \rangle$.
- Dadas duas sequências X e Y , dizemos que uma sequência Z é uma subsequência comum de X e Y se Z é uma subsequência tanto de X quanto de Y .
- Exemplo: $Z = \langle B, C, A, B \rangle$ é uma subsequência comum *mais longa* de $X = \langle A, B, C, B, D, A, B \rangle$ e $Y = \langle B, D, C, A, B, A \rangle$

Subsequência Comum Mais Longa

- Definição do Problema da SCML

SCML

Instância Duas seqüencias $X = \langle x_1, x_2, \dots, x_m \rangle$ e $Y = \langle y_1, y_2, \dots, y_n \rangle$.

Solução Uma subsequência comum Z de X, Y .

Objetivo Maximizar o comprimento de Z .

- Exemplo de Aplicação: comparar dois DNAs

$$X = ACCGGTCGAGTG$$

$$Y = GTCGTTCGGAATGCCGTTGCTCTGTAAA$$

- Os caracteres devem aparecer na mesma ordem, mas não precisam ser necessariamente consecutivos.

Teorema: Subestrutura Ótima de uma SCML

Sejam as sequências $X = \langle x_1, x_2, \dots, x_m \rangle$ e $Y = \langle y_1, y_2, \dots, y_n \rangle$, e seja $Z = \langle z_1, z_2, \dots, z_k \rangle$ qualquer SCML de X e Y

- Se $x_m = y_n$, então $z_k = x_m = y_n$ e Z_{k-1} é uma SCML de X_{m-1} e Y_{n-1}
- Se $x_m \neq y_n$, então $z_k \neq x_m$ implica que Z é uma SCML de X_{m-1} e Y
- Se $x_m \neq y_n$, então $z_k \neq y_m$ implica que Z é uma SCML de X e Y_{n-1}

Notação: Se $Z = \langle z_1, z_2, \dots, z_n \rangle$, para $0 \leq k \leq n$, $Z_k = \langle z_1, z_2, \dots, z_k \rangle$

Denotando com $S(X, Y)$ a subsequência mais longa entre X e Y , isso leva ao definição recursiva

$$S(X, Y) = \begin{cases} S(X', Y') + 1 & \text{se } X = X'c, Y = Y'c \\ \max\{S(X, Y'), S(X', Y)\} & \text{se } X = X'c_1, Y = Y'c_2 \text{ e } c_1 \neq c_2 \\ 0 & \text{se } X = \epsilon \text{ ou } Y = \epsilon \end{cases}$$

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.

Com memoização ou armazenamento de valores intermediários, podemos reduzir o tempo e espaço para $O(nm)$:

SCML

Algoritmo 5.1 (SCML)

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 .

```

1  m := comprimento(X)
2  n := comprimento(Y)
3  for i := 0 to m do c[i, 0] := 0;
4  for j := 1 to n do c[0, j] := 0;
5  for i := 1 to m do
6      for j := 1 to n do
7          if xi = yj then

```

```

8            $c[i, j] := c[i - 1, j - 1] + 1$ 
9       else
10       $c[i, j] := \max(c[i, j - 1], c[i - 1, j])$ 
11   end if
12 end for
13 return  $c[m, n]$ 

```

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

Exemplo 5.1

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

◊

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 é menor que o comprimento de uma coluna, podemos manter duas linhas

5. Programação dinâmica

e calcular os valores linha por linha. Caso as colunas são menores, procedemos das mesma 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.

```
1 m := comprimento[X]
2 n := comprimento[Y]
3 for i := 0 to m do c[i, 0] := 0;
4 for j := 1 to n do c[0, j] := 0;
5 for i := 1 to m do
6   for j := 1 to n do
7     if  $x_i = y_j$  then
8       c[i, j] := c[i - 1, j - 1] + 1
9       b[i, j] := ↗
10    else if c[i - 1, j]  $\geq$  c[i, j - 1] then
11      c[i, j] := c[i - 1, j]
12      b[i, j] := ↑
13    else
14      c[i, j] := c[i, j - 1]
15      b[i, j] := ←
16 return c e b
```

Exemplo

	.	B	D	C	A	B	A
.	0	0	0	0	0	0	0
A	0	$\uparrow 0$	$\uparrow 0$	$\uparrow 0$	$\nwarrow 1$	$\leftarrow 1$	$\nwarrow 1$
B	0	$\nwarrow 1$	$\leftarrow 1$	$\leftarrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$
C	0	$\uparrow 1$	$\uparrow 1$	$\nwarrow 2$	$\leftarrow 2$	$\uparrow 2$	$\uparrow 2$
B	0	$\nwarrow 1$	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\leftarrow 3$
D	0	$\uparrow 1$	$\nwarrow 2$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\uparrow 3$
A	0	$\uparrow 1$	$\uparrow 2$	$\uparrow 2$	$\nwarrow 3$	$\uparrow 3$	$\nwarrow 4$
B	0	$\nwarrow 1$	$\uparrow 2$	$\uparrow 2$	$\uparrow 3$	$\nwarrow 4$	$\uparrow 4$

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 [37], 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 .

```

1  if i = 0 or j = 0 then return
2  if b[i,j] =  $\nwarrow$  then
3      Print-SCML(b, X, i 1, j 1)
4      print  $x_i$ 
5  else if b[i,j] =  $\uparrow$  then
6      Print-SCML(b, X, i 1, j)
7  else
8      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 caracter, a deleção de um caracter ou a substituição de um caracter 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 caracter diferente (inclusive a “falta” de um caracter $-$) 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 caracteres 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:

$$d(s, t) = \begin{cases} \max(|s|, |t|) & \text{se } |s| = 0 \text{ ou } |t| = 0 \\ \min(d(s', t) + 1, d(s, t') + 1, d(s', t') + [c_1 \neq c_2]) & \text{se } s = s'c_1 \text{ e } t = t'c_2 \end{cases}$$

Essa distância está conhecida como *distância de Levenshtein* [50]. Uma implementação direta é

Distância de Edição

Algoritmo 5.4 (Distância)

Entrada Dois strings s, t e seus respectivos tamanhos n e m medidos em número de caracteres.

Saída A distância mínima entre s e t .

```

1  distância (s , t , n ,m):=
2      if (n=0) return m
3      if (m=0) return n
4      if (sn = tm) then
5          sol0 = distância (s , t , n - 1 ,m - 1)
6      else
7          sol0 = distância (s , t , n - 1 ,m - 1) + 1
8      end if
9      sol1 = distância (s , t , n ,m - 1) + 1
10     sol2 = distância (s , t , n - 1 ,m) + 1
11     return min(sol0,sol1,sol2)

```

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

Distância de Edição

Algoritmo 5.5 (PD-distância)

Entrada Dois strings s e t , e n e m , seus respectivos tamanhos medidos em número de caracteres.

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 $s[1 \dots i]$ e $t[1 \dots j]$.

```

1  PD–distância (s , t ,n ,m):=
2      for i := 0, . . . , n do mi,0 := i
3      for i := 1, . . . , m do m0,i := i
4      for i := 1, . . . , n do
5          for j := 1, . . . , m do
6              if (si = tj) then
7                  sol0 := mi-1,j-1
8              else
9                  sol0 := mi-1,j-1 + 1
10             end if

```

5. Programação dinâmica

```

11       $sol_1 := m_{i,j-1} + 1$ 
12       $sol_2 := m_{i-1,j} + 1$ 
13       $m_{i,j} := \min(sol_0, sol_1, sol_2);$ 
14      end for
15      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

Distância entre textos

Algoritmo 5.6 (PD-distância)

Entrada Dois strings s e t , e n e m , seus respectivos tamanhos medidos em número de caracteres.

Saída A distância mínima entre s e t e uma matriz $P = (p_{i,j})$ que armazena a sequência de operações.

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 $s[1 \dots i]$ e $t[1 \dots j]$.

```

1 PD-distância (s ,t ,n ,m):=
2   for  $i := 0, \dots, n$  do  $m_{i,0} = i; p_{i,0} := -1$ 
3   for  $i := 1, \dots, m$  do  $m_{0,i} = i; p_{0,i} := -1$ 

```

```

4   for  $i := 1, \dots, n$  do
5     for  $j := 1, \dots, m$  do
6       if ( $s_i = t_j$ ) then
7          $sol_0 = m_{i-1,j-1}$ 
8       else
9          $sol_0 = m_{i-1,j-1} + 1$ 
10      end if
11       $sol_1 := m_{i,j-1} + 1$ 
12       $sol_2 := m_{i-1,j} + 1$ 
13       $m_{i,j} := \min(sol_0, sol_1, sol_2);$ 
14       $p_{i,j} := \min\{i \mid sol_i = m_{ij}\}$ 
15    end for
16  return  $m_{i,j}$ 

```

Reconstrução da Sequência de Operações

Algoritmo 5.7 (PD-operações)

Entrada Uma matriz $P = (p_{ij})$ de tamanho $n \times m$ com marcação de operações, strings s, t , posições i e j .

Saída Uma sequência a_1, a_2, a_x de operações executadas.

```

1 PD–operações (P, s , t , i , j ):=
2   case
3      $p_{i,j} = -1:$ 
4       return
5      $p_{i,j} = 0:$ 
6       PD–operações (s ,t ,i – 1 ,j – 1)
7       print ( 'M' )
8      $p_{i,j} = 1:$ 
9       PD–operações (s ,t ,i ,j – 1)
10      print ( 'I' )
11      $p_{i,j} = 2:$ 
12       PD–operações (s ,t ,i – 1 ,j )
13       print ( 'D' )
14   end case

```

5. Programação dinâmica

O algoritmo possui complexidade de tempo e espaço $O(mn)$, sendo que o espaço são duas matrizes 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 com valores v_i e peso w_i , $1 \leq i \leq n$, e um limite de peso da mochila W .

Solução Um subconjunto $S \subseteq [1, n]$ 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.

Idéia: 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 idéia 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 $M(n, W)$. 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

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

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

$$\begin{aligned} &M_1(M_2(M_3M_4)) \\ &M_1((M_2M_3)M_4) \\ &(M_1M_2)(M_3M_4) \\ &(M_1(M_2M_3))M_4 \\ &((M_1M_2)M_3)M_4 \end{aligned}$$

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

5. Programação dinâmica

Dependendo do tamanho das 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 possíveis¹.

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 $T(1) = 1$.

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

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.

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

Multiplicação de Cadeias de Matrizes

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

Algoritmo Multi-Mat-1

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

Algoritmo 5.8 (Multi-Mat-1)

Entrada Cadeia de matrizes (A_1, A_2, \dots, A_n) e suas respectivas dimensões b_i , $0 \leq i \leq n$. A matrix A_i tem dimensão $b_{i-1} \times b_i$.

Saída Número mínimo de multiplicações.

```

1  for i:=1 to n do mi,j := 0
2  for u:=1 to n-1 do {diagonais superiores}
3    for i:=1 to n-u do {posição na diagonal}
4      j:=i+u           {u = j-i}
5      mi,j := ∞
6      for k:=i to j-1 do
7        c := mi,k + mk+1,j + bi-1 · bk · bj
8        if c < mi,j then mi,j := c
9      end for
10   end for
11 end for
12 return m1,n
```

Considerações para a Análise

- O tamanho da entrada se refere ao número de matrizes a serem multiplicadas
- As operações aritméticas sobre os naturais são consideradas operações fundamentais

Análise de Complexidade do Algoritmo

$$\begin{aligned}
 C_p &= \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} (1 + \sum_{k=i}^{j-1} 4) = \sum_{u=1}^{n-1} \sum_{i=1}^{n-u} (1 + 4u) \\
 &= \sum_{u=1}^{n-1} (n-u)(1+4u) = \sum_{u=1}^{n-1} n + 4nu - u - 4u^2 = O(n^3)
 \end{aligned}$$

Análise de Complexidade do Algoritmo

$$C_p[\text{Inicialização}] = O(n)$$

$$C_p[\text{Iteração}] = O(n^3)$$

$$C_p[\text{Finalização}] = O(1)$$

$$C_p[\text{Algoritmo}] = O(n) + O(n^3) + O(1) = O(n^3)$$

Algoritmo Multi-Mat-2

Retorna o número mínimo de multiplicações e a parentização respectiva para multiplicar a cadeia de matrizes passada como parâmetro.

Algoritmo 5.9 (Multi-Mat-1)

Entrada Cadeia de matrizes (A_1, A_2, \dots, A_n) e suas respectivas dimensões armazenadas no vetor b .

Saída Número mínimo de multiplicações.

```

1  for i:=1 to n do  $m_{i,j} := 0$  {inicializa diagonal principal}
2  for d:=1 to n-1 do { para todas diagonais superiores}
3    for i:=1 to n-u do { para cada posição na diagonal}
4      j:=i+u           {u=j-i}
5       $m_{i,j} := \infty$ 
6      for k:=i to j do
7        c:=  $m_{i,k} + m_{k+1,j} + b_{i-1} \cdot b_k \cdot b_j$ 
8        if c <  $m_{i,j}$  then
9           $m_{i,j} := c$ 
10          $P_{i,j} = k$ 
11       end for
12     end for
13   end for
14   return  $m_{1,n}, p$ 

```

Algoritmo Print-Parentização

Algoritmo 5.10 (Print-Parentização)

Entrada Matriz P , índices i e j .

Saída Impressão da parentização entre os índices i e j .

```

1  if  $i = j$  then
2    print " $A_i$ "
3  else
4    print "("
5    Print-Parentização( $P, i, P_{i,j}$ )
6    Print-Parentização( $P, P_{i,j} + 1, j$ )
7    print ")"
8 end if

```

5.5. Tópicos

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.

Algoritmo de Floyd-Warshall

- Conteúdo disponível na seção 25.2 do Cormen, e Exemplo 5.2.4 (Laird&Veloso, 2^a edição).
- 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

Subcaminhos de caminhos mais curtos são caminhos mais curtos.

- Lema 24.1 (Cormen): Dado um grafo orientado ponderado $G = (V, E)$, com função peso $w : E \rightarrow R$, seja $p = (v_1, v_2, \dots, v_k)$ um caminho mais

5. Programação dinâmica

curto do vértice v_1 até o vértice v_k e, para quaisquer i e j tais que $1 \leq i \leq j \leq k$, seja $p_{ij} = (v_i, v_{i+1}, \dots, v_j)$ o subcaminho p desde o vértice v_i até o vértice v_j . Então, p_{ij} é um caminho mais curto de v_i até v_j .

Algoritmo de Floyd-Warshall

Algoritmo 5.11 (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 .

```
1   $D^0 := D$ 
2  for  $k := 1$  to  $n$ 
3      for  $i := 1$  to  $n$ 
4          for  $j := 1$  to  $n$ 
5               $d_{ij}^k := \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$ 
6  return  $D^n$ 
```

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

Excuso 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 sub-cúbico para a multiplicação de matrizes em semi-aneis 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 [11], Williams e Williams [74].

Outra observação é que o algoritmo de Floyd-Warshall somento calcula as distâncias entre todos pares de vértices. Para determinar os caminhos mais curtos, veja por exemplo [5]. \diamond

Exemplo

5.5.2. Caixeiro viajante

O problema de caixeiro viajante é um exemplo em que a programação dinâmica ajuda reduzir um trabalho exponencial. Esperadamente, o algoritmo final ainda é exponencial (o problema é NP-completo), mas significadamente menor.

PROBLEMA DO CAIXEIRO VIAJANTE

Instância Um grafo $G=(V,E)$ com pesos d (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 $\sum_{1 \leq i < n} d_{\{i, i+1\}} + d_{\{n, 1\}}$.

O algoritmo é baseado na seguinte idéia (proposta por Bellman [9]). Seja v_1, v_2, \dots 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, \dots que passa por todos vértices exceto v_1 e volta. Da mesma forma a última sub-solução tem o caminho v_3, v_4, \dots que passa por todos vértices exceto v_1, v_2 e volta, como sub-solução. Essa soluções têm sub-estrutura ótima, porque qualquer outro caminho menor pode ser substituído para o caminho atual.

5. Programação dinâmica

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 minizar sobre todas continuações possíveis. Isso leva à recorrência

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

Se ordenamos todos os sub-conjuntos dos vértices $[1, n]$ 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 uma subconjunto de $[1, n]$ como número entre 0 e $2^n - 1$. Nessa caso, a ordem natural já respeita a ordem \subseteq entre os conjuntos, e podemos substituir um teste $v \in V_j$ com $2^v \& j = 2^v$ e a operação $V_j \setminus \{v\}$ com $j - 2^v$.

```
1   for i ∈ [1, n] do  $T_{i,0} := d_{i1}$  { base }
2   for j ∈ [1,  $2^n - 1$ ] do
3       for i ∈ [1, n] do
4            $T_{i,j} := \min_{2^k \& j=2^k} d_{ik} + T_{i,j-2^k}$  { tempo  $O(n)$  ! }
5       end for
6   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 do espaço é $O(n 2^n)$. Essa é atualmente o melhor algoritmo exato conhecido para o problema do caixeiro viajante (veja também xkcd.com/399).

5.5.3. Árvore de busca binária ótima

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 5.2

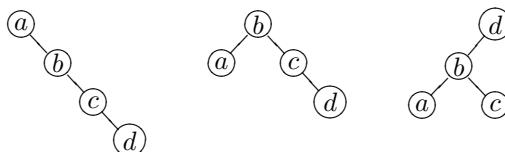
Considere a sequência ordenada $a < b < c < d$ e as probabilidades

Exemplo

	Elemento	a	b	c	d
Pr		0.2	0.1	0.6	0.1

qual seria uma árvore ótima? Alguns exemplos

Árvore correspondente



que têm 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. \diamond

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

1. uma sequência ordenada $a_1 < a_2 < \dots < a_n$ de n chaves e
2. probabilidades

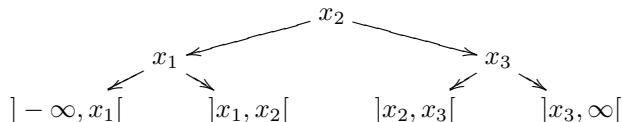
$$\Pr[c < a_1], \Pr[c = a_1], \Pr[a_1 < c < a_2], \dots \\ \dots, \Pr[a_{n-1} < c < a_n], \Pr[c = a_n], \Pr[a_n < c]$$

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

5. Programação dinâmica

(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 $a_{n+1} = \infty$. Com isso, obtemos a complexidade média de busca

$$c_M = \sum_{1 \leq i \leq n} \Pr[c = a_i] \text{prof}(a_i) + \sum_{0 \leq i \leq n} \Pr[a_i < c < a_{i+1}] (\text{prof}([a_i, a_{i+1}]) - 1);$$

ela depende da árvore concreta.

Como achar a árvore ótima? A observação crucial é a seguinte: *Uma das chaves deve ser a raiz e as duas sub-árvores da esquerda e da direita devem ser árvores ótimas pelas sub-sequências correspondentes.*

Para expressar essa observação numa equação, vamos denotar com $c_M(e, d)$ a complexidade média de uma busca numa sub-árvore ótima para os elementos a_e, \dots, a_d . Para a complexidade da árvore inteira, definido acima, temos $c_M = c_M(1, n)$. Da mesma forma, obtemos

$$c_M(e, d) = \sum_{e \leq i \leq d} \Pr[c = a_i] \text{prof}(a_i) + \sum_{e-1 \leq i \leq d} \Pr[a_i < c < a_{i+1}] (\text{prof}([a_i, a_{i+1}]) - 1)$$

Árvore de Busca Binária Ótima

Supondo que a_r é a raiz desse sub-árvore, essa complexidade pode ser escrito

como

$$\begin{aligned}
 c_M(e, d) &= \Pr[c = a_r] \\
 &\quad + \sum_{e \leq i < r} \Pr[c = a_i] \text{prof}(a_i) + \sum_{e-1 \leq i < r} \Pr[a_i < c < a_{i+1}] (\text{prof}(]a_i, a_{i+1}[) - 1) \\
 &\quad + \sum_{r < i \leq d} \Pr[c = a_i] \text{prof}(a_i) + \sum_{r \leq i \leq d} \Pr[a_i < c < a_{i+1}] (\text{prof}(]a_i, a_{i+1}[) - 1) \\
 &= \left(\sum_{e-1 \leq i \leq d} \Pr[a_i < c < a_{i+1}] + \sum_{e \leq i \leq d} \Pr[c = a_i] \right) \\
 &\quad + c_M(e, r-1) + c_M(r+1, d) \\
 &= \Pr[a_{e-1} < c < a_{d+1}] + c_M(e, r-1) + c_M(r+1, d)
 \end{aligned}$$

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 $d > e$:

$$c_M(e, d) = \begin{cases} \min_{e \leq r \leq d} \Pr[a_{e-1} < c < a_{d+1}] + c_M(e, r-1) + c_M(r+1, d) & \text{caso } e \leq d \\ 0 & \text{caso } e > d \end{cases} \quad (5.1)$$

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 $1 \leq i \leq n$ e $i-1 \leq j \leq n$.
- r_{ij} : raiz da árvore ótima para as chaves a_i até a_j , para $1 \leq i \leq j \leq n$.
- p_{ij} : $\Pr[a_{i-1} < c < a_{j+1}]$

Árvore de Busca Binária Ótima

Algoritmo 5.12 (ABB-ÓTIMA)

Entrada Probabilidades $p_i = \Pr[c = a_i]$ e $q_i = \Pr[a_i < c < a_{i+1}]$.

Saída Vetores c e r como descrita acima.

```

1   for i:=1 to n+1 do
2      $p_{i(i-1)} := q_{i-1}$ 
3      $c_{i(i-1)} := 0$ 
4   end for
5
6   for d:=0 to n-1 do { para todas diagonais }
7     for i:=1 to n-d do { da chave  $i$  }
8        $j := d+i$            { até chave  $j$  }
9        $p_{ij} := p_{i(j-1)} + p_j + q_j$ 
10       $c_{ij} := \infty$ 
11      for r:=i to j do
12         $c := p_{ij} + c_{i(r-1)} + c_{(r+1)j}$ 
13        if  $c < c_{ij}$  then
14           $c_{ij} := c$ 
15           $r_{ij} := r$ 
16      end for
17    end for
18  end for

```

i/j	1	2	3	\dots	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 $O(n^3)$.

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

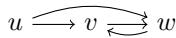


Figura 5.1.: Esquerda: O subcaminho uv de uvw não é o caminho mais longo entre u e v .

$w, u \cdots v$ 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 computada em tempo $O(n)$, que resulta numa complexidade total de $O(n^3 2^n)$.

Um corolário é que caso o grafo é aciclico, o caminho mais longo pode ser calculado em tempo polinomial.

5.6. Exercícios

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 aciclicos possui uma subestrutura ótima? Justifique. Caso sim, propõe um algoritmo de programação dinâmica que resolve o problema.

6. Divisão e conquista

6.1. Introdução

Método de Divisão e Conquista

- **Dividir** o problema original em um determinado número de subproblemas independentes
- **Conquistar** os subproblemas, resolvendo-os recursivamente até obter o caso base.
- **Combinar** as soluções dadas aos subproblemas, a fim de formar a solução do problema original.

Recorrências

- O tempo de execução dos algoritmos recursivos pode ser descrito por uma recorrência.
- Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seu valor em entradas menores.

Divisão e conquista

Algoritmo 6.1 (DC)

Entrada Uma instância I de tamanho n .

```
1  if  $n = 1$  then
2      return Solução direta
3  else
4      Divide  $I$  em sub instâncias  $I_1, \dots, I_k$ ,  $k > 0$ 
5          com tamanhos  $n_i < n$ .
6      Resolve recursivamente:  $I_1, \dots, I_k$ .
7      Resolve  $I$  usando sub soluções  $DC(I_1), \dots, DC(I_k)$ .
8  end if
```

6. Divisão e conquista

Recursão natural

- Seja $d(n)$ o tempo para a divisão.
- Seja $s(n)$ o tempo para computar a solução final.
- Podemos somar: $f(n) = d(n) + s(n)$ e obtemos

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

Recursão natural: caso balanceado

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

Mergesort

Algoritmo 6.2 (Mergesort)

Entrada Índices p, r e um vetor A com elementos A_p, \dots, A_r

Saída A com elementos em ordem não-decrescente, i.e. para $i < j$ temos $A_i \leq A_j$.

```
1  if p < r then
2      q = ⌊(p+r)/2⌋
3      MergeSort(A,p,q);
4      MergeSort(A,q+1,r);
5      Merge(A,p,q,r)
6  end if
```

Recorrências simplificadas

Formalmente, a equação de recorrência do Mergesort é

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

Em vez de

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

Para simplificar a equação, sem afetar a análise de complexidade correspondente, em geral:

- supõe-se argumentos inteiros para funções (omitindo pisos e tetos)
- omite-se a condição limite da recorrência

Recorrências: caso do Mergesort

A equação de recorrência do Mergesort é

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

Sendo que:

- $T(n)$ representa o tempo da chamada recursiva da função para um problema de tamanho n .
- $2T(\frac{n}{2})$ indica que, a cada iteração, duas chamadas recursivas ($2T$) serão executadas para entradas de tamanho $\frac{n}{2}$.
- Os resultados das duas chamadas recursivas serão combinados (*merged*) com um algoritmo com complexidade de pior caso $\Theta(n)$.

6.2. Resolver recorrências

Métodos para resolver recorrências

Existem vários métodos para resolver recorrências:

- Método da substituição
- Método de árvore de recursão
- Método mestre

6.2.1. Método da substituição

Método da substituição

- O método da substituição envolve duas etapas:
 1. pressupõe-se um limite hipotético.
 2. usa-se indução matemática para provar que a suposição está correta.
- Aplica-se este método em casos que é fácil pressupor a forma de resposta.
- Pode ser usado para estabelecer limites superiores ou inferiores.

Mergesort usando o método da substituição

Supõe-se que a recorrência

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n$$

tem limite superior igual a $n \log n$, ou seja, $T(n) = O(n \log n)$. Devemos provar que $T(n) \leq cn \log n$ para uma escolha apropriada da constante $c > 0$.

$$\begin{aligned} T(n) &\leq 2\left(c\left\lfloor \frac{n}{2} \right\rfloor \log\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n\right) \\ &\leq cn \log n/2 + n = cn \log n - cn + n \\ &\leq cn \log n \end{aligned}$$

para $c \geq 1$.

A expressão na equação

$$c \cdot n \log n - \underbrace{c \cdot n + n}_{\text{resíduo}}$$

se chama *resíduo*. O objetivo na prova é mostrar, que o resíduo é negativo.

Como fazer um bom palpite

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

- Usa-se o resultado do método de árvore de recursão como limite hipotético para o método da substituição.

Exemplo 6.1

Vamos procurar o máximo de uma sequência por divisão é conquista:

Algoritmo 6.3 (Máximo)

Entrada Uma sequência a e dois índices l, r tal que a_l, \dots, a_{r-1} é definido.

Saída $\max_{l \leq i < r} a_i$

- 1 $m_1 := M(a, l, \lfloor (l+r)/2 \rfloor)$
- 2 $m_2 := M(a, \lfloor (l+r)/2 \rfloor, r)$

Isso leva a recorrência

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

◊

Algumas sutilezas nas resoluções

- Considere a recorrência $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$. Prove que $T(n) = O(n)$
- Considere a recorrência $T(n) = 2T(\lfloor n/2 \rfloor) + n$. É possível provar que $T(n) = O(n)$?
- Considere a recorrência $T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \log n$. Prove que $T(n) = O(\log n \log \log n)$

Proposta de exercícios: 4.1-1, 4.1-2, 4.1-5 e 4.1-6 do Cormen (pág. 54 da edição traduzida).

Substituição direta

- Supomos que a recorrência tem a forma

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

6. Divisão e conquista

- Para resolver ele podemos substituir

$$\begin{aligned}T(n) &= f(n) + T(n-1) = f(n) + f(n-1) + T(n-1) + f(n-2) \\&= \dots = \sum_{1 \leq i \leq n} f(i)\end{aligned}$$

(Os \dots substituem uma prova por indução.)

É simples de generalizar isso para

$$\begin{aligned}T(n) &= T(n/c) + f(n) \\&= f(n) + f(n/c) + f(n/c^2) + \dots \\&= \sum_{0 \leq i \leq \log_c n} f(n/c^i)\end{aligned}$$

Exemplo 6.2

Na aula sobre a complexidade média do algoritmo Quicksort (veja página 57), encontramos uma recorrência da forma

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

cuja solução é $\sum_{1 \leq i \leq n} i = n(n-1)/2$. \diamond

Substituição direta

- Da mesma forma podemos resolver recorrências como

$$T(n) = \begin{cases} f(1) & \text{caso } n = 1 \\ T(n/2) + f(n) & \text{caso contrário} \end{cases}$$

- substituindo

$$\begin{aligned}T(n) &= f(n) + T(n/2) = f(n) + f(n/2) + T(n/4) \\&= \dots = \sum_{0 \leq i \leq \log_2 n} f(n/2^i)\end{aligned}$$

Exemplo 6.3

Ainda na aula sobre a complexidade média do algoritmo Quicksort (veja página 57), encontramos outra recorrência da forma

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

Para colocá-la na forma acima, vamos dividir primeiro por n para obter

$$T(n)/n = 1 + T(n/2)/(n/2)$$

e depois substituir $A(n) = T(n)/n$, que leva à recorrência

$$A(n) = 1 + A(n/2)$$

cuja solução é $\sum_{0 \leq i \leq \log_2 n} 1 = \log_2 n$. Portanto temos a solução $T(n) = n \log_2 n$.

Observe que a análise não considera constantes: qualquer função $cn \log_2 n$ também satisfaz a recorrência. A solução exata é determinada pela base; uma alternativa é concluir que $T(n) = \Theta(n \log_2 n)$. \diamond

Também podemos generalizar isso. Para

$$f(n) = 2f(n/2)$$

é simples de ver que

$$f(n) = 2f(n/2) = 2^2 f(n/4) = \dots = 2^{\log_2 n} n = n$$

(observe que toda função $f(n) = cn$ satisfaz a recorrência).

Generalizando obtemos

$$\begin{aligned} f(n) &= cf(n/2) = \dots = c^{\log_2 n} = n^{\log_2 c} \\ f(n) &= c_1 f(n/c_2) = \dots = c_1^{\log_{c_2} n} = n^{\log_{c_2} c_1} \end{aligned}$$

O caso mais complicado é o caso combinado

$$T(n) = c_1 T(n/c_2) + f(n).$$

O nosso objetivo é nos livrar da constante c_1 . Se dividirmos por $c_1^{\log_{c_2} n}$ obtemos

$$\frac{T(n)}{c_1^{\log_{c_2} n}} = \frac{T(n/c_2)}{c_1^{\log_{c_2} n/c_2}} + \frac{f(n)}{c_1^{\log_{c_2} n}}$$

e substituindo $A(n) = T(n)/c_1^{\log_{c_2} n}$ temos

$$A(n) = A(n/c_2) + \frac{f(n)}{c_1^{\log_{c_2} n}}$$

6. Divisão e conquista

uma forma que sabemos resolver:

$$A(n) = \sum_{0 \leq i \leq \log_{c_2} n} \frac{f(n/c_2^i)}{c_1^{\log_{c_2} n / c_2^i}} = \frac{1}{c_1^{\log_{c_2} n}} \sum_{0 \leq i \leq \log_{c_2} n} f(n/c_2^i) c_1^i$$

Após de resolver essa recorrência, podemos re-substituir para obter a solução de $T(n)$.

Exemplo 6.4 (Multiplicação de números inteiros)

A multiplicação de números binários (veja exercício 6.3) gera a recorrência

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

Dividindo por $3^{\log_2 n}$ obtemos

$$\frac{T(n)}{3^{\log_2 n}} = \frac{T(n/2)}{3^{\log_2 n/2}} + \frac{cn}{3^{\log_2 n}}$$

e substituindo $A(n) = T(n)/3^{\log_2 n}$ temos

$$\begin{aligned} A(n) &= A(n/2) + \frac{cn}{3^{\log_2 n}} \\ &= c \sum_{0 \leq i \leq \log_2 n} \frac{n/2^i}{3^{\log_2 n/2^i}} \\ &= \frac{cn}{3^{\log_2 n}} \sum_{0 \leq i \leq \log_2 n} \left(\frac{3}{2}\right)^i \\ &= \frac{cn}{3^{\log_2 n}} \frac{(3/2)^{\log_2 n+1}}{1/2} = 3c \end{aligned}$$

e portanto

$$T(n) = A(n)3^{\log_2 n} = 3cn^{\log_2 3} = \Theta(n^{1.58})$$

◇

6.2.2. 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.

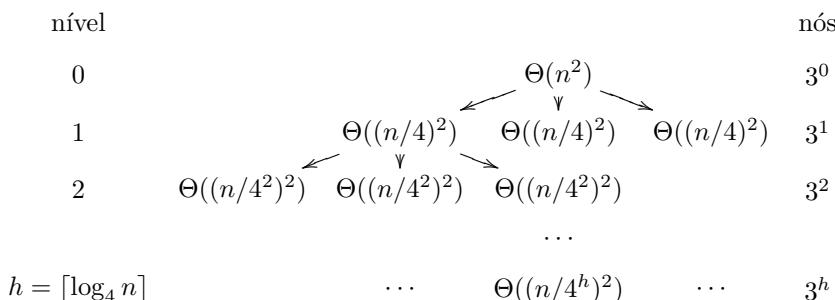
- Numa árvore de recursão cada nó representa o custo de um único sub-problema da respectiva chamada recursiva

- 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 $i = \log_4 n = \frac{\log_2 n}{2}$.
- Quantos níveis tem a árvore? A árvore tem $\log_4 n + 1$ níveis ($0, 1, 2, 3, \dots, \log_4 n$).
- Quantos nós têm cada nível? 3^i .
- Qual o tamanho do problema em cada nível? $\frac{n}{4^i}$.
- Qual o custo de cada nível i da árvore? $3^i c(\frac{n}{4^i})^2$.
- 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 $\Theta(n^2)$.

6. Divisão e conquista

- 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 $(\frac{3d}{16} + c) \leq d$, ou seja, para valores de d tais que $d \geq \frac{16}{13}c$

Exemplo 6.5

Outro exemplo é a recorrência $T(n) = 3T(n/2) + cn$ da multiplicação de números binários. Temos $\log_2 n$ níveis na árvore, 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 portanto 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

$$T(n) = T(n/3) + T(2n/3) + O(n)$$

Proposta de exercícios: 4.2-1, 4.2-2 e 4.2-3 do Cormen [14].

6.2.3. Método Mestre**Método Mestre**

Para aplicar o método mestre deve ter a recorrência na seguinte forma:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

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

Proposta de exercícios: 6.1 e 6.2.

Algoritmo Potenciação**Algoritmo 6.4 (Potenciação-Trivial (PT))**

Entrada Uma base $a \in \mathbb{R}$ e um exponente $n \in \mathbb{N}$.

Saída A potência a^n .

```

1  if n = 0
2    return 1
3  else
4    return PT(a, n - 1) × a
5 end if

```

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 6.5 (Potenciação-Npotência2 (P2))

Entrada Uma base $a \in \mathbb{R}$ e um exponente $n \in \mathbb{N}$.

Saída A potência a^n .

```

1  if n = 1 then
2    return a
3  else
4    x := P2(a, n ÷ 2)
5    return x × x
6 end if

```

Complexidade da potenciação-Npotência2

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 0 \\ T(\lfloor \frac{n}{2} \rfloor) + c & \text{se } n > 0 \end{cases}$$

A complexidade dessa recorrência é logarítmica, ou seja, $T(n) \in O(\log n)$

Busca Binária

Algoritmo 6.6 (Busca-Binária(i,f,x,S))

Entrada Um inteiro x , índices i e f e uma sequência $S = a_1, a_2, \dots, a_n$ de números ordenados.

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

```

1  if  $i = f$  then
2    if  $a_i = x$  return  $i$ 
3    else return  $\infty$ 
4  end if
5   $m := \left\lfloor \frac{f-i}{2} \right\rfloor + i$ 
6  if  $x < a_m$  then
7    return Busca Binária( $i, m - 1$ )
8  else
9    return Busca Binária( $m + 1, f$ )
10 end if

```

Complexidade da Busca-Binária

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T(\lfloor \frac{n}{2} \rfloor) + c & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é logarítmica, ou seja, $T(n) \in O(\log n)$

Quicksort

Algoritmo 6.7 (Quicksort)

Entrada Índices l, r e um vetor a com elementos a_l, \dots, a_r .

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

```

1  if  $l < r$  then
2    m := Partition(l, r, a);

```

```

3     Quicksort( $l, m - 1, a$ );
4     Quicksort( $m + 1, r, a$ );
5 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

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{se } n > 1 \end{cases}$$

A complexidade dessa recorrência é $T(n) \in O(n \log n)$

Complexidade do Quicksort com Particionamento Balanceado

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + \Theta(n) & \text{se } n > 1 \end{cases}$$

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 6.6 (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

$$T(n) = \sum_{0 \leq i < \log_2 n} (n \log n - in) + \Theta(n) = \Theta(n \log^2 n)$$

◊

Exemplo 6.7 (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

$$\begin{aligned} T(n) &= \sum_{0 \leq i < \log_2 n} (n/(\log n - i)) + \Theta(n) \\ &= \sum_{1 \leq j \leq \log n} n/j + \Theta(n) = nH_n + \Theta(n) = \Theta(n \log \log n) \end{aligned}$$

◊

Prova do Teorema Master

- Consideremos o caso simplificado em que $n = 1, b, b^2, \dots$, ou seja, n é uma potência exata de dois e assim não precisamos nos preocupar com tetos e pisos.

Prova do Teorema Master

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:

$$\begin{cases} T(n) = \Theta(1) & \text{se } n = 1 \\ T(n) = aT\left(\frac{n}{b}\right) + f(n) & \text{se } n = b^i \end{cases}$$

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 Master

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

$$g(n) = \sum_{j=0}^{\log_b(n-1)} a^j \cdot f(n/b^j)$$

6. Divisão e conquista

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 Master

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

$$\begin{cases} T(n) = \Theta(1) & \text{se } n = 1 \\ T(n) = aT\left(\frac{n}{b}\right) + f(n) & \text{se } n = b^i \end{cases}$$

onde i é um inteiro positivo. Então $T(n)$ pode ser limitado assintoticamente para potências exatas de b como a seguir:

Prova do Método 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 $T(n) = \Theta(f(n))$

6.2.4. Um novo método Mestre

O método Master se aplica somente no caso que a árvore de recursão está balanceado. O método de Akra-Bazzi [4] é uma generalização do método Master, que serve também em casos não平衡ados¹. O que segue é uma versão generalizada de Leighton [49].

Teorema 6.1 (Método Akra-Bazzi e Leighton)

Dado a recorrência

$$T(x) = \begin{cases} \Theta(1) & \text{se } x \leq x_0 \\ \sum_{1 \leq i \leq k} a_i T(b_i x + h_i(x)) + g(x) & \text{caso contrário} \end{cases}$$

com constantes $a_i > 0$, $0 < b_i < 1$ e funções g , h , que satisfazem

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

¹Uma abordagem similar foi proposta por [57].

para um $\epsilon > 0$ e a constante x_0 e suficientemente grande² 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 $\sum_{1 \leq i \leq k} a_i b_i^p = 1$.

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

$$T(x) = \begin{cases} \Theta(1) & \text{se } x \leq x_0 \\ \sum_{1 \leq i \leq k} a_i T(\lceil b_i x \rceil) + g(x) & \text{caso contrário} \end{cases}$$

demonstrando que obtemos a mesma solução aplicando tetos. ◊

Exemplo 6.8

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

$$\begin{aligned} T(n) &\in \Theta\left(n^p + \left(1 + \int_1^n c_1 u / u^{p+1} du\right)\right) = \Theta\left(n^p \left(1 + c_1 \int_1^n u^{-p} du\right)\right) \\ &= \Theta\left(n^p \left(1 + \frac{c_1}{1-p} n^{1-p}\right)\right) \\ &= \Theta\left(n^p + \frac{c_1}{1-p} n\right) = \Theta(n) \end{aligned}$$

◊

Exemplo 6.9

Considere $T(n) = 2T(n/2) + n \log n$ do exemplo 6.6 (que não pode ser resolvido pelo teorema Master). $2(1/2)^p = 1$ define $p = 1$ e temos

$$\begin{aligned} T(n) &\in \Theta\left(n + \left(1 + \int_1^n \log u / u du\right)\right) = \Theta\left(n \left(1 + [\log^2(u)/2]_1^n\right)\right) \\ &= \Theta\left(n \left(1 + \log^2(n)/2\right)\right) \\ &= \Theta(n \log^2(n)) \end{aligned}$$

²As condições exatas são definidas em Leighton [49].

**Exemplo 6.10**

Considere $T(n) = 2T(n/2) + n/\log n$ do exemplo 6.7 (que não pode ser resolvido pelo teorema Master). Novamente $p = 1$ e temos

$$\begin{aligned} T(n) &\in \Theta\left(n + \left(1 + \int_1^n 1/(u \log u) du\right)\right) = \Theta\left(n(1 + [\log \log u]_1^n)\right) \\ &= \Theta(n(1 + \log \log n)) \\ &= \Theta(n \log(\log n)) \end{aligned}$$



6.3. Tópicos

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. Mencionamos que existem algoritmos mais eficientes. A idéia do algoritmo de Strassen [65] 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 obtém-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 $T(n) = O(n^{\log_2 7}) = O(n^{2.81})$.

6.4. Exercícios

(Soluções a partir da página 297.)

Exercício 6.1

Resolva os seguintes recorrências

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(2n/3) + 1$
3. $T(n) = 3T(n/4) + n \log n$
4. $T(n) = 2T(n/2) + n \log n$
5. $T(n) = 4T(n/2) + n^2 \lg n$
6. $T(n) = T(n-1) + \log n$
7. $T(n) = 2T(n/2) + n/\log n$

6. Divisão e conquista

$$8. T(n) = 3T(n/2) + n \log n$$

Exercício 6.2

Aplique o teorema mestre nas seguintes recorrências:

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(2n/3) + 1$
3. $T(n) = 3T(n/4) + n \log n$
4. $T(n) = 2T(n/2) + n \log n$

Exercício 6.3

Descreva o funcionamento dos problemas abaixo, extraia as recorrências dos algoritmos, analise a complexidade pelos três métodos vistos em aula.

1. Produto de número binários (Exemplo 5.3.8 de Toscani e Veloso [66]).
2. Algoritmo de Strassen para multiplicação de matrizes (Cormen 28.2 e capítulo 6.3).
3. Encontrar o k-ésimo elemento de uma lista não ordenada (Cormen 9.3).

Exercício 6.4

A recorrência na análise do algoritmo de Strassen leva em conta somente multiplicações. Determina e resolve a recorrência das multiplicações e adições.

7. 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).

Backtracking: Árvore de busca

- Seja uma solução dado por um vetor (s_1, s_2, \dots, s_n) com $s_i \in S_i$.
- Queremos somente soluções que satisfazem uma propriedade $P_n(s_1, \dots, s_n)$.
- Idéia: Refinar a busca de força bruta, aproveitando restrições cedo
- Define propriedades $P_i(s_1, \dots, s_i)$ tal que

$$P_{i+1}(s_1, \dots, s_{i+1}) \Rightarrow P_i(s_1, \dots, s_i)$$

Backtracking: Árvore de busca

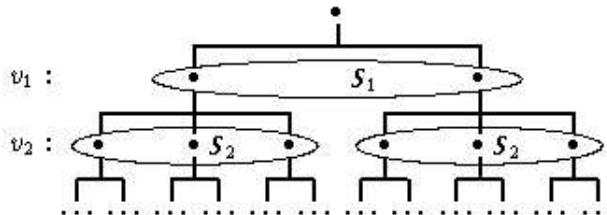
- A árvore de busca $T = (V, A, r)$ é definido por

$$\begin{aligned} V &= \{(s_1, \dots, s_i) \mid P_i(s_1, \dots, s_i)\} \\ A &= \{(v_1, v_2) \in V \mid v_1 = (s_1, \dots, s_i), v_2 = (s_1, \dots, s_{i+1})\} \\ r &= () \end{aligned}$$

- Backtracking busca nessa árvore em profundidade.

7. Backtracking

- Observe que é suficiente manter o caminho da raiz até o nodo atual na memória.

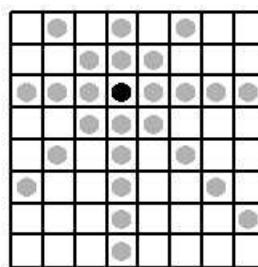


O problema das n -rainhas

PROBLEMA DAS n -RAINHAS

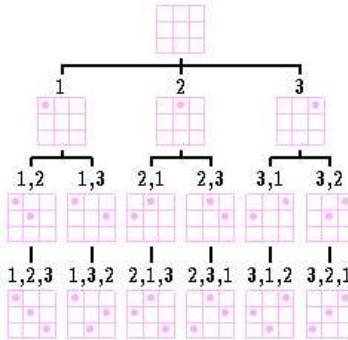
Instância Um tablado de xadrez de dimensão $n \times n$, e n rainhas.

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)

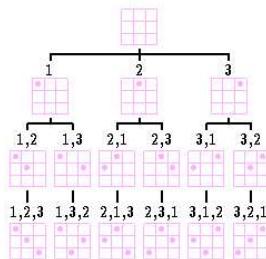
O que representam as folhas da árvore de busca para este problema?



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

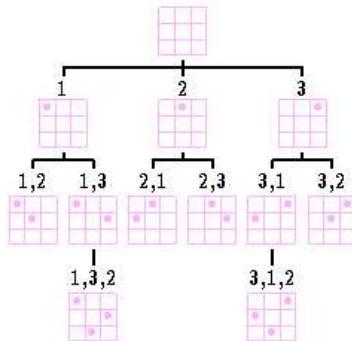
$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(1/n)) \quad (7.1)$$



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)

7. Backtracking



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

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.

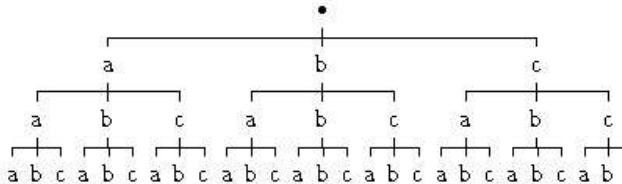
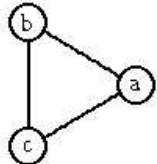
PROBLEMA DO CAIXEIRO VIAJANTE

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 $\sum_{1 \leq i < n} p_{\{i,i+1\}} + p_{\{n,1\}}$.

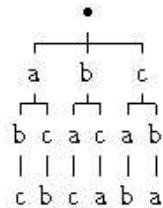
O problema do caixeiro viajante



O problema do caixeiro viajante

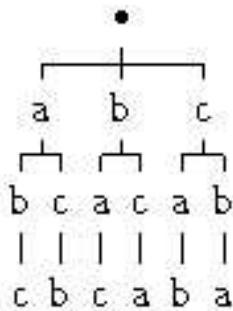
- Para cada chamada recursiva existem diversos vértices que podem ser selecionados
- Vértices ainda não selecionados são os candidatos possíveis
- A busca exaustiva é gerada caso nenhuma restrição for imposta
- Todas as permutações de cidades geram as soluções factíveis ($P_n = n(n-1)(n-2)\dots 1 = n!$)
- Este problema têm solução n^{2^n} usando programação dinâmica.
- Mas: para resolver em PD é necessário n^{2^n} de memória!

7. Backtracking



O problema do caixeiro viajante

- Alguma idéia de como diminuir ainda mais o espaço de busca?



Vantagens x Desvantagens e Características Marcantes

Vantagens

- Fácil implementação
- Linguagens da área de programação lógica (prolog, op5) trazem mecanismos embutidos para a implementação de backtracking

Desvantagens

- Tem natureza combinatória. A busca exaustiva pode ser evitada com o teste de restrições, mas o resultado final sempre é combinatório

Característica Marcante

- Backtracking = “retornar pelo caminho”: constroem o conjunto de soluções ao retornarem das chamadas recursivas.

Problema de Enumeração de conjuntos

ENUMERAÇÃO DE CONJUNTOS

Instância Um conjunto de n itens $S = a_1, a_2, a_3, \dots, a_n$.

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, \dots, 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-polynomial) 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 $c : V \rightarrow C$ 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 $|C|$.

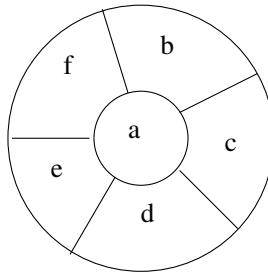
- Coloração de grafos de intervalo é um problema polinomial.
- Para um grafo qualquer este problema é NP-completo.
- Dois números são interessantes nesse contexto:
 - O *número de clique* $\omega(G)$: O tamanho máximo de uma clique que se encontra como sub-grafo de G .
 - O *número cromático* $\chi(G)$: 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

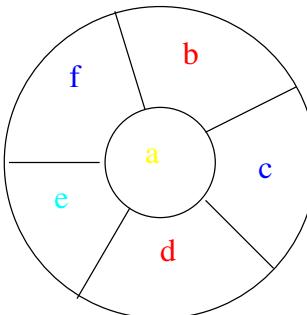
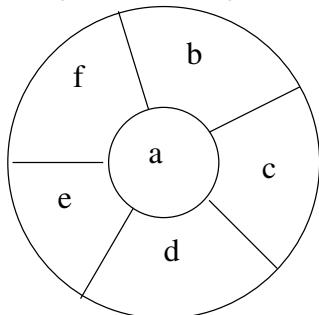
- A coloração de mapas é uma abstração do problema de colorir vértices.
- Projete um algoritmo de backtracking para colorir um grafo planar (mapa).
- Um grafo planar é aquele que pode ser representado em um plano sem qualquer intersecção entre arestas.
- Algoritmo $O(n^n)$, supondo o caso em que cada 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 7.1 (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.

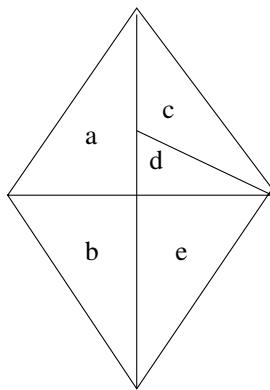
Para um vértice $v \in V$, vamos definir o conjunto de cores adjacentes $C(v) := \{c(u) \mid \{u, v\} \in A\}$.

```

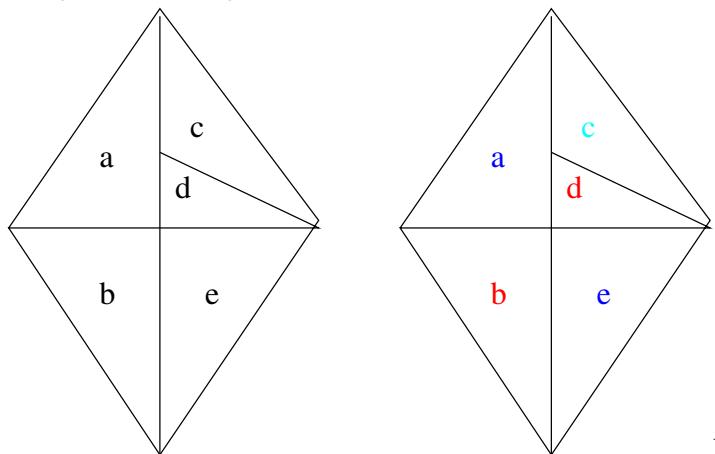
1  return bt - coloring(1,1)
2
3  boolean bt - coloring( $v, c_v$ ) :=
4      if  $v > n$ 
5          return true
6
7      if  $c \notin C(v)$  then {  $v$  colorível com  $c_v$ ? }
8           $c(v) := c_v$ 
9          for  $c_u \in [1, 4]$  do
10             if bt - coloring( $v + 1, c_u$ )
11                 return true
12             end for
13         else
14             return false
15         end if
16     end

```

De quantas cores precisamos?



De quantas cores precisamos?



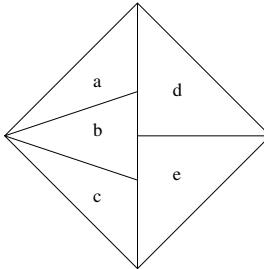
de 3 cores!

Precisamos

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.

7. Backtracking

- Metaheurísticas: heurísticas guiadas por heurísticas (*meta*=além + *heuris* = encontrar).
 - Ex: Algoritmos genéticos, Busca Tabu, GRASP (*greedy randomized adaptive search procedure*), Simulated annealing, etc.

8. Algoritmos de aproximação

8.1. Introdução

Vertex cover

Considere

COBERTURA POR VÉRTICES (INGL. VERTEX COVER)

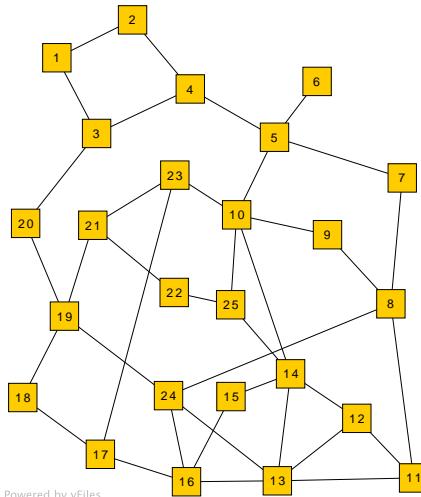
Instância Grafo não-direcionado $G = (V, E)$.

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?



8. Algoritmos de aproximação

Simplificando o problema

- Vértice de grau 1: Usa o vizinho.
- Vértice de grau 2 num triângulo: Usa os dois vizinhos.

Algoritmo 8.1 (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 .

```
1 Reduz( $G$ ) :=  
2   while (alguma regra em baixo se aplica) do  
3     Regra 1:  
4       if  $\exists u \in V : \deg(u) = 1$  then  
5         seja  $\{u, v\} \in E$   
6          $C := C \cup \{v\}$   
7          $G := G - \{u, v\}$   
8       end if  
9     Regra 2:  
10    if  $\exists u \in V : \deg(u) = 2$  then  
11      seja  $\{u, v\}, \{u, w\} \in E$   
12      if  $\{v, w\} \in E$  then  
13         $C := C \cup \{v, w\}$   
14         $G := G - \{u, v, w\}$   
15      end if  
16    end while  
17    return ( $C, G$ )
```

Uma solução exata com busca exhaustiva:

Árvore de busca

Algoritmo 8.2 (Árvore de busca)

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

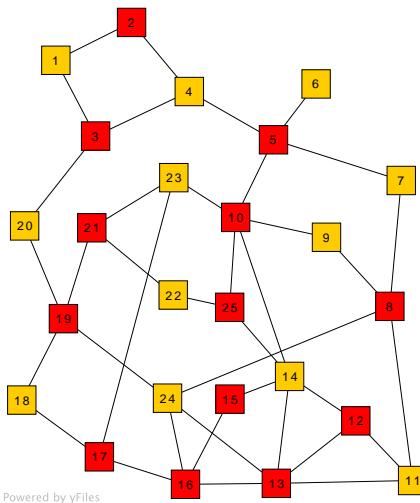
Saída Cobertura por vértices $S \subseteq V$ mínima.

```

1 minVertexCover(G):=
2   if  $E = \emptyset$  return  $\emptyset$ 
3   escolhe  $\{u, v\} \in E$ 
4    $C_1 := \text{minVertexCover}(G - u) \cup \{u\}$ 
5    $C_2 := \text{minVertexCover}(G - v) \cup \{v\}$ 
6   if  $(|C_1| < |C_2|)$  then
7     return  $C_1$ 
8   else
9     return  $C_2$ 
10 end if

```

Solução ótima?



Problemas de otimização

Definição 8.1

Um *problema de otimização* é uma relação binária $\mathcal{P} \subseteq I \times S$ com instâncias $x \in I$ e soluções $y \in S$, tal que $(x, y) \in \mathcal{P}$ e com

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

8. Algoritmos de aproximação

- um objetivo: Encontrar mínimo ou máximo $\text{OPT}(x) = \text{opt}\{\phi(x, y) \mid (x, y) \in \mathcal{P}\}$.

Tipo de problemas

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

Convenção

Escrevemos um problema de otimização na forma

Nome

Instância x

Solução y

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

Classes de complexidade

- PO: Problemas de otimização com algoritmo polinomial.
- NPO: Problemas de otimização tal que
 1. Instâncias reconhecíveis em tempo polinomial.
 2. A relação \mathcal{P} é polinomialmente limitada.
 3. Para y arbitrário, polinomialmente limitado: $(x, y) \in \mathcal{P}$ decidível em tempo polinomial.
 4. φ é computável em tempo polinomial.
- NPO contém todos problemas de otimização, que satisfazem critérios mínimos de tratabilidade.

Lembrança (veja definição 13.1): \mathcal{P} é polinomialmente limitada, se para soluções $(x, y) \in \mathcal{P}$ temos $|y| \leq p(|x|)$, para um polinômio p .

Motivação

- Para vários problemas não conhecemos um algoritmo eficiente.
- No caso dos problemas NP-completos: solução eficiente é pouco provável.
- O quer fazer?
- Idéia: Para problemas da otimização, não busca o ótimo.
- Uma solução “quase” ótima também ajuda.

O que é “quase”? Aproximação absoluta

- A solução encontrada difere da solução ótima ao máximo um *valor* constante.
- Erro absoluto:

$$D(x, y) = |\text{OPT}(x) - \varphi(x, y)|$$
- Aproximação absoluta: Algoritmo garante um y tal que $D(x, y) \leq k$.
- Exemplos: Coloração de grafos, árvore geradora e árvore Steiner de grau mínimo [28]
- Contra-exemplo: Knapsack.

O que é “quase”? Aproximação relativa

- A solução encontrada difere da solução ótima ao máximo um *fator* constante.
- Erro relativo:

$$E(x, y) = D(x, y) / \max\{\text{OPT}(x), \varphi(x, y)\}.$$

- Algoritmo ϵ -aproximativo ($\epsilon \in [0, 1]$): Fornece solução y tal que $E(x, y) \leq \epsilon$ para todo x .
- Soluções com $\epsilon \approx 0$ são ótimas.
- Soluções com $\epsilon \approx 1$ são péssimas.

8. Algoritmos de aproximação

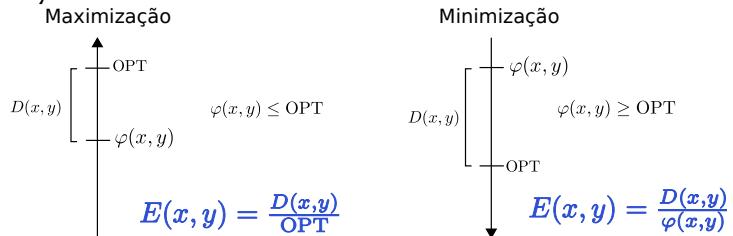
Aproximação relativa: Taxa de aproximação

- Definição alternativa
- Taxa de aproximação $R(x, y) = 1/(1 - E(x, y)) \geq 1$.
- Com taxa r , o algoritmo é r -aproximativo.
- (Não tem perigo de confusão com o erro relativo, porque $r \in [1, \infty]$.)

Aproximação relativa: Exemplos

- Exemplo: Knapsack, Caixeiro viajante métrico.
- Contra-exemplo: Caixeiro viajante [69].
- Classe correspondente APX: r -aproximativo em tempo polinomial.

Aproximação relativa



Exemplo: Cobertura por vértices gulosa

Algoritmo 8.3 (Cobertura por vértices)

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

Saída Cobertura por vértices $C \subseteq V$.

```
1  VC GV(G) :=  
2  (C, G) := Reduz(G)  
3  if V = ∅ then  
4      return C  
5  else
```

```

6      escolhe  $v \in V : \deg(v) = \Delta(G)$  { grau máximo }
7      return  $C \cup \{v\} \cup \text{VC-GV}(G - v)$ 
8  end if

```

Proposição 8.1

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

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

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

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

e o grau médio dos vértices em G_i satisfaz

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

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

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

Com isso podemos estimar

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

ou

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

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

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

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

Exemplo: Buscar uma Cobertura por vértices

Algoritmo 8.4 (Cobertura por vértices)

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

Saída Cobertura por vértices $C \subseteq V$.

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

Aproximação: Motivação

- Queremos uma aproximação
- Quais soluções são aproximações boas?
- Problemas:
 - Tempo polinomial desejada
 - Aproximação desejada (“heurística com garantia”)
 - Freqüentemente: a análise e o problema. Intuição:
 - Simples verificar se um conjunto é uma cobertura.
 - Difícil verificar a minimalidade.

Exemplo: Outra abordagem

Algoritmo 8.5 (Cobertura por vértices)

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

Saída Cobertura por vértices $C \subseteq V$.

```

1  VC GE( $G$ ) := 
2    ( $C, G$ ) := Reduz( $G$ )
3    if  $E = \emptyset$  then
4      return  $C$ 
5    else
6      escolhe  $e = \{u, v\} \in E$ 
7      return  $C \cup \{u, v\} \cup \text{VC-GE}(G - \{u, v\})$ 
8    end if

```

Proposição 8.2

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

Prova. Cada cobertura contém ao menos um dos dois vértices escolhidos, i.e.

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

■

Técnicas de aproximação

- Aproximações gulosas
- Randomização
- Busca local
- Programação linear
- Programação dinâmica
- Algoritmos seqüenciais (“online”), p.ex. para particionamento.

Exemplo 8.1

SEQUENCIAMENTO EM PROCESSORES PARALELOS

8. Algoritmos de aproximação

Entrada m processadores, n tarefas com tempo de execução l_i , $1 \leq i \leq n$.

Solução Um *sequenciamento* $S : [n] \rightarrow [m]$, i.e., uma alocação das tarefas às máquinas.

Objetivo Minimizar o *makespan* (tempo terminal)

$$\max_{j \in [m]} \sum_{i \in n | S(i)=j} l_i.$$

O problema é NP-completo. Seja $W = \sum_{i \in n} l_i$ o tempo total (workload). Sabemos que $S^* \geq W/m$ e também que $S^* \geq l_i$, $1 \leq i \leq n$.

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 com menor tempo final.

Proposição 8.3

Sequenciamento em lista com ordem arbitrária é uma $2 - 1/m$ -aproximação.

Prova. Seja $h \in [m]$ a máquina que define o *makespan* da solução gulosa, e $k \in [n]$ a última tarefa alocada à essa máquina. No momento em que essa tarefa foi alocada, todas as outras máquinas estavam com tempo ao menos $\sum_{i:S(i)=h} l_i - l_k$, portanto

$$\begin{aligned} m \left(\sum_{i:S(i)=h} l_i - l_k \right) + l_k &\leq W \\ \sum_{i:S(i)=h} l_i &\leq (W + (p-1)l_k)/p \\ &\leq \text{OPT} + p/(p-1)\text{OPT} = (2 - 1/p)\text{OPT} \end{aligned}$$

■

◇

O que podemos ganhar com algoritmos off-line? Uma abordagem é ordenar as tarefas por tempo execução não-crescente e aplicar o algoritmo guloso. Essa abordagem é chamada LPT (largest process first).

Proposição 8.4

LPT é uma $4/3 - 1/3p$ -aproximação.

Prova. Por análise de casos. TBD.

■

8.2. Aproximações com randomização

Randomização

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

Randomização: Exemplo

SATISFATIBILIDADE MÁXIMA, MAXIMUM SAT

Instância Fórmula $\varphi \in \mathcal{L}(V)$, $\varphi = C_1 \wedge C_2 \wedge \dots \wedge C_n$ em FNC.

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

Objetivo Maximiza o número de cláusulas satisfeitas

$$|\{C_i \mid \llbracket C_i \rrbracket_a = v\}|.$$

Nossa solução

```

1 SAT-R( $\varphi$ ) :=  

2   seja  $\varphi = \varphi(v_1, \dots, v_k)$   

3   for all  $i \in [1, k]$  do  

4     escolhe  $v_i = v$  com probabilidade  $1/2$   

5   end for

```

Aproximação?

- Surpresa: Algoritmo é 2-aproximação.

Prova. Seja X a variável aleatória que denota o número de cláusulas satisfeitas. Afirmação: $E[X] \geq (1 - 2^{-l})n$ com l o número mínimo de literais por cláusula. Portanto, com $l \geq 1$, temos $E[X] \geq n/2$.

8. Algoritmos de aproximação

Prova da afirmação: $P[\llbracket C_i \rrbracket = f] \leq 2^{-l}$ e logo $P[\llbracket C_i \rrbracket = v] \geq (1 - 2^{-l})$ e

$$E[X] = \sum_{1 \leq i \leq k} E[\llbracket C_i \rrbracket = v] = \sum_{1 \leq i \leq k} P[\llbracket C_i \rrbracket = v] = (1 - 2^{-l})n.$$

■

Outro exemplo

Cobertura por vértices guloso e randomizado.

```
1 VC-RG( $G$ ) :=  
2   seja  $\bar{w} := \sum_{v \in V} \deg(v)$   
3    $C := \emptyset$   
4   while  $E \neq \emptyset$  do  
5     escolhe  $v \in V$  com probabilidade  $\deg(v)/\bar{w}$   
6      $C := C \cup \{v\}$   
7      $G := G - v$   
8   end while  
9   return  $C \cup V$ 
```

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

8.3. Aproximações gulosas

Problema de mochila (Knapsack)

KNAPSACK

Instância Itens $X = [1, n]$ com valores $v_i \in \mathbb{N}$ e tamanhos $t_i \in N$, para $i \in X$, um limite M , tal que $t_i \leq M$ (todo item cabe na mochila).

Solução Uma seleção $S \subseteq X$ tal que $\sum_{i \in S} t_i \leq M$.

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

Observação: Knapsack é NP-completo.

Como aproximar?

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

Abordagem

```

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

```

Aproximação boa?

- Considere

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

- Então:

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

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

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

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

Tentativa 2: Modificação

```

1 K-G'( $v_i, t_i$ ) := 
2      $S_1 := \text{K-G}(v_i, t_i)$ 
3      $v_1 := \sum_{i \in S_1} v_i$ 
4      $S_2 := \{\text{argmax}_i v_i\}$ 
5      $v_2 := \sum_{i \in S_2} v_i$ 
6     if  $v_1 > v_2$  then

```

8. Algoritmos de aproximação

```

7     return  $S_1$ 
8 else
9     return  $S_2$ 
10 end if

```

Aproximação boa?

- O algoritmo melhorou?
- Surpresa

Proposição 8.5

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

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

$$\bar{v}_j = \sum_{1 \leq i < j} v_i \leq \varphi_{K\text{-}G}(x) \quad (8.1)$$

$$\bar{t}_j = \sum_{1 \leq i < j} t_i \leq M \quad (8.2)$$

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

1. Seja $v_j \leq \bar{v}_j$.

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

2. Seja $v_j > \bar{v}_j$

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

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

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



Problemas de particionamento

- Empacotamento unidimensional (ingl. bin packing).

BIN PACKING

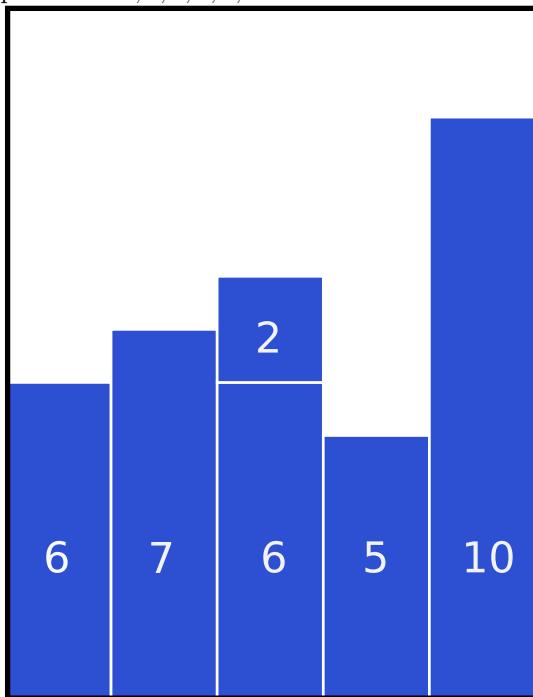
Instância Seqüência de itens v_1, \dots, v_n , $v_i \in]0, 1]$, $v_i \in \mathbb{Q}^+$

Solução Partição $\bigcup_{1 \leq i \leq k} P_i = [1, n]$ tal que $\sum_{j \in P_i} v_j \leq 1$ para todos $i \in [1, n]$.

Objetivo Minimiza o número de partições (“containeres”) k .

Abordagem?

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



8. Algoritmos de aproximação

Aproximação?

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

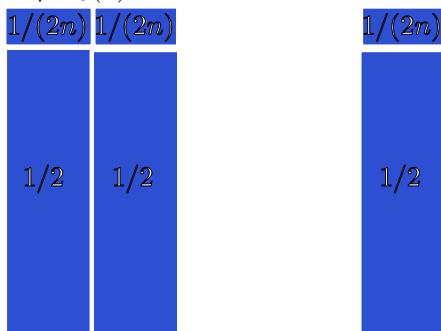
Prova. Seja B o número de containeres usadas, $V = \sum_{1 \leq i \leq n} v_i$. $B \leq 2 \lceil V \rceil$ porque duas containeres consecutivas contém uma soma > 1 . Mas precisamos ao menos $B \geq \lceil V \rceil$ containeres, logo $\text{OPT}(x) \geq \lceil V \rceil$. Portanto, $\varphi_{\text{PrC}}(x) \leq 2 \lceil V \rceil \leq 2\text{OPT}(x)$. ■

Aproximação melhor?

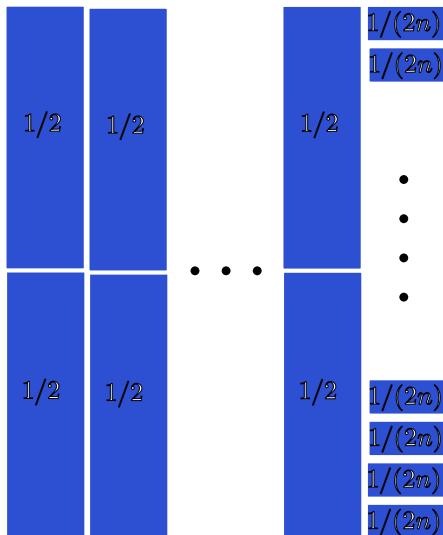
- Isso é o melhor possível!
- Considere os $4n$ itens

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

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



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



- 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?

$$\begin{aligned}\varphi_{\text{PiC}}(x) &\leq \lceil 1.7 \text{OPT}(x) \rceil \\ \varphi_{\text{PiCD}}(x) &\leq 1.5 \text{OPT}(x) + 1\end{aligned}$$

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

$$\begin{aligned}A &= \{v_i \mid v_i > 2/3\} \\ B &= \{v_i \mid 2/3 \geq v_i > 1/2\} \\ C &= \{v_i \mid 1/2 \geq v_i > 1/3\} \\ D &= \{v_i \mid 1/3 \geq v_i\}\end{aligned}$$

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

8. Algoritmos de aproximação

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

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

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

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

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

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

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

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

■

Aproximação melhor?

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

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

- Baker, 1985

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

8.4. 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 de mochila (veja página 166):
- Algoritmo MM-PD com programação dinâmica (pág. 106): tempo $O(n \sum_i v_i)$.
- Desvantagem: Pseudo-polynomial.

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

```

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

```

Teorema 8.1

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

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

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

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

8. Algoritmos de aproximação

ótima.

$$\begin{aligned}
 \varphi_{\text{MM-PTAS}}(I, S) &= \sum_{i \in S} v_i \geq \sum_{i \in S} 2^t \lfloor v_i/2^t \rfloor && \text{definição de } \lfloor \cdot \rfloor \\
 &\geq \sum_{i \in S^*} 2^t \lfloor v_i/2^t \rfloor && \text{otimalidade de MM-PD sobre } v'_i \\
 &\geq \sum_{i \in S^*} v_i - 2^t && (\text{A.10}) \\
 &= \left(\sum_{i \in S^*} v_i \right) - 2^t |S^*| \\
 &\geq \text{OPT}(I) - 2^t n
 \end{aligned}$$

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

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

■

Considerações finais 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 $O(n^{15000})$, $O(n^{37500})$ e $O(n^{523804})$, respectivamente!

8.5. Exercícios

Exercício 8.1

Um aluno propõe a seguinte heurística para Binpacking: 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

```

1 ordene itens em ordem crescente
2  $m := 1; M := n$ 
3 while ( $m < M$ ) do
4     abre novo container, coloca  $v_M$ ,  $M := M - 1$ 
5     while ( $v_m$  cabe e  $m < M$ ) do
6         coloca  $v_m$  no container atual
7          $m := m + 1$ 
8     end while
9 end while
```

Qual a qualidade desse algoritmo? É um algoritmo de aproximação? Caso sim, qual a taxa de aproximação dele? Caso não, por quê?

Exercício 8.2

Prof. Rapidez propõe o seguinte pré-processamento para o algoritmo SAT-R de aproximação para MAX-SAT (página 165): 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.

Este análise é correto ou não?

Parte III.

Algoritmos

9. Algoritmos em grafos

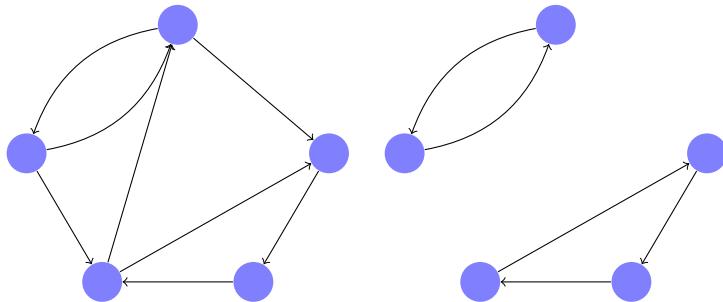


Figura 9.1.: Grafo (esquerda) com circulação (direita)

9.1. Fluxos em redes

Definição 9.1

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 saíentes de v e $\delta^-(v) = \{(u, v) \mid (u, v) \in E\}$ para os arcos entrantes em v .

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

$$f(v) := \sum_{e \in \delta^+(v)} f_e - \sum_{e \in \delta^-(v)} f_e = 0 \quad (9.1)$$

(ver Fig. 9.1).

Lema 9.1

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, 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 9.2

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

Prova. Pelo lema 9.1 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

$$\begin{array}{lll} \text{maximiza} & f(s) & (9.2) \\ \text{sujeito a} & f(v) = 0 & \forall v \in V \setminus \{s, t\} \\ & 0 \leq f_e \leq c_e & \forall e \in E. \end{array}$$

Observação 9.1

O programa (9.2) 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. ◇

9.1.1. Algoritmo de Ford-Fulkerson

Nosso objetivo: Achar um algoritmo *combinatorial* mais eficiente. Idéia básica: Começar com um fluxo viável $f_e = 0$ e aumentar ele gradualmente.

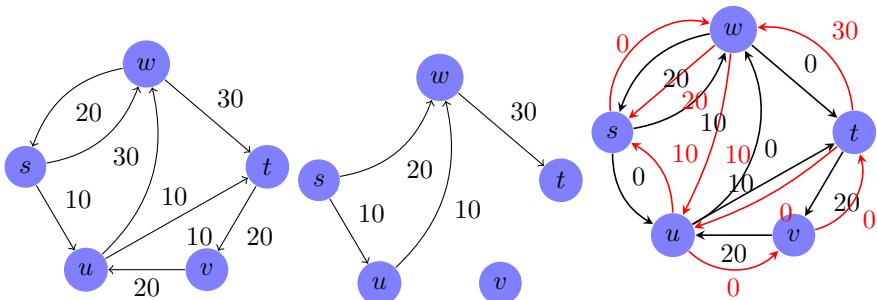


Figura 9.2.: Esquerda: Grafo com capacidades. Centro: Fluxo com valor 30. Direita: O grafo residual correspondente.

Observação: Se temos um s - t -caminho $P = (v_0 = s, v_1, \dots, v_{n-1}, v_n = t)$, podemos aumentar o fluxo atual f um valor que corresponde ao “gargalo”

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

Observação 9.2

Repetidamente procurar um caminho com gargalo positivo e aumentar nem sempre produz um fluxo máximo. Na Fig. 9.2 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. \diamond

Problema no caso acima: para aumentar o fluxo e manter a conservação do fluxo num vértice interno v temos quatro possibilidades: (i) aumentar o fluxo num arco entrante e saiente, (ii) aumentar o fluxo num arco entrante, e diminuir num outro arco entrante, (iii) diminuir o fluxo num arco entrante e diminuir num arco saiente e (iv) diminuir o fluxo num arco entrante e aumentar num arco entrante (ver Fig. 9.3).

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 $f_{(u, v)} > 0$.

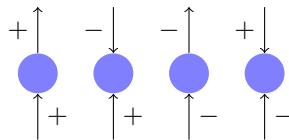


Figura 9.3.: Manter a conservação do fluxo.

Observe que na Fig. 9.2 o grafo residual possui um caminho $P = (s, w, u, t)$ que aumenta o fluxo por 10. O algoritmo de Ford-Fulkerson [24] consiste em, repetidamente, aumentar o fluxo num caminho $s-t$ no grafo residual.

Algoritmo 9.1 (Ford-Fulkerson)

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

Saída Um fluxo f .

```

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

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

Lema 9.3

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. ■

9. Algoritmos em grafos

Lema 9.4

Em cada iteração, o fluxo aumenta ao menos 1.

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

Lema 9.5

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 9.2

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 (9.1) 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 9.6

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 9.7

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 9.1 (Fluxo máximo – corte mínimo)

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

Lema 9.8

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

Prova. (Do teorema 9.1) 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. 9.4). Além disso, o algoritmo com complexidade $O((n + m)C)$ é somente pseudo-polynomial.
2. É possível que o algoritmo não termine para capacidades reais (veja Fig. 9.4). Usando uma busca por profundidade para achar caminhos $s-t$ ele termina, mas é ineficiente [16].

9.1.2. 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 (sem prova)

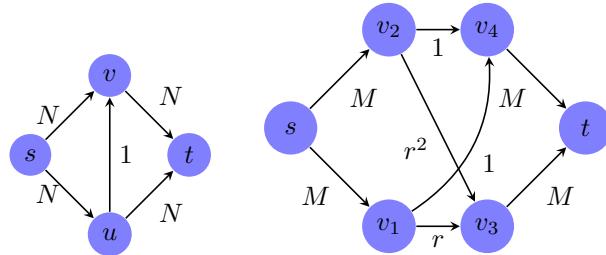


Figura 9.4.: Esquerda: Pior caso para o algoritmo de Ford-Fulkerson com pesos inteiros aumentando o fluxo por $2N$ vezes por 1 nos caminhos (s, u, v, t) e (s, v, u, t) . Direita: Menor grafo com pesos irracionais em que o algoritmo de Ford-Fulkerson falha [75]. $M \geq 3$, e $r = (1 + \sqrt{1 - 4\lambda})/2$ com $\lambda \approx 0.217$ a única raiz real de $1 - 5x + 2x^2 - x^3$. Aumentar (s, v_1, v_4, t) e depois repetidamente $(s, v_2, v_4, v_1, v_3, t)$, $(s, v_2, v_3, v_1, v_4, t)$, $(s, v_1, v_3, v_2, v_4, t)$, e $(s, v_1, v_4, v_2, v_3, t)$ converge para o fluxo máximo $2 + r + r^2$ sem terminar.

Teorema 9.2

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

Lema 9.9

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 .

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

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

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

Prova. (do teorema 9.2)

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

$$\begin{aligned}\delta_{f'}(u) &= \delta_{f'}(v) + 1 \\ &\geq \delta_f(v) + 1 \quad \text{pelo lema 9.9} \\ &= \delta_f(u) + 2,\end{aligned}$$

i.e., a distância do u entre dois instantes em que uv é crítico aumenta por pelo menos dois. Enquanto u é alcancável por s , a sua distância é no máximo $n - 2$, porque a 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. ■

Outras soluções (Goldberg 2008):

9.1.3. Variações do problema

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

$$\begin{array}{lll}\text{maximiza} & f(S) \\ \text{sujeito a} & f(v) = 0 & \forall v \in V \setminus (S \cup T) \\ & f_e \leq c_e & \forall e \in E.\end{array}\tag{9.3}$$

Ano	Referência	Complexidade	Obs
1951	Dantzig	$O(n^2 m C)$	Simplex
1955	Ford & Fulkerson	$O(n m C)$	Cam. aument.
1970	Dinitz	$O(n^2 m)$	Cam. min. aument.
1972	Edmonds & Karp	$O(m^2 \log C)$	Escalonamento
1973	Dinitz	$O(n m \log C)$	Escalonamento
1974	Karzanov	$O(n^3)$	Preflow-Push
1977	Cherkassky	$O(n^2 m^{1/2})$	Preflow-Push
1980	Galil & Naamad	$O(n m \log^2 n)$	
1983	Sleator & Tarjan	$O(n m \log n)$	
1986	Goldberg & Tarjan	$O(n m \log(n^2/m))$	Push-Relabel
1987	Ahuja & Orlin	$O(n m + n^2 \log C)$	
1987	Ahuja et al.	$O(n m \log(n \sqrt{\log C}/m))$	
1989	Cheriyan & Hagerup	$O(n m + n^2 \log^2 n)$	
1990	Cheriyan et al.	$O(n^3 / \log n)$	
1990	Alon	$O(n m + n^{8/3} \log n)$	
1992	King et al.	$O(n m + n^{2+\epsilon})$	
1993	Phillips & Westbrook	$O(n m (\log_{m/n} n + \log^{2+\epsilon} n))$	
1994	King et al.	$O(n m \log_{m/(n \log n)} n)$	
1997	Goldberg & Rao	$O(m^{3/2} \log(n^2/m) \log C)$ $O(n^{2/3} m \log(n^2/m) \log C)$	

Tabela 9.1.: Complexidade para diversos algoritmos de fluxo máximo [60].

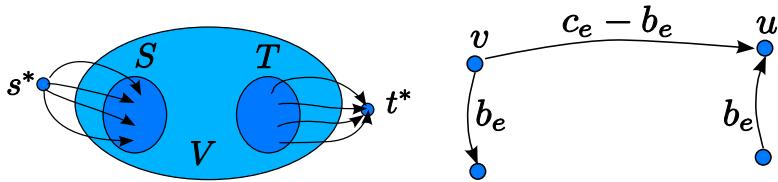


Figura 9.5.: Reduções entre variações do problema do fluxo máximo. Esquerda: Fontes e destinos múltiplos. Direita: Limite inferior e superior para a capacidade de arcos.

O problema (9.3) pode ser reduzido para um problema de fluxo máximo simples em $G' = (V', E', c')$ (veja Fig. 9.5(a)) com

$$\begin{aligned} V' &= V \cup \{s^*, t^*\} \\ E' &= E \cup \{(s^*, s) \mid s \in S\} \cup \{(t, t^*) \mid t \in T\} \\ c'_e &= \begin{cases} c_e & e \in E \\ c(\{s\}, \{\bar{s}\}) & e = (s^*, s) \\ c(\{\bar{t}\}, \{t\}) & e = (t, t^*) \end{cases} \end{aligned} \quad (9.4)$$

Lema 9.10

Se f' é solução máxima de (9.4), $f = f'|_E$ é uma solução máxima de (9.3). Conversamente, se f é uma solução máxima de (9.3),

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

é uma solução máxima de (9.4).

Prova. Supõe f é solução máxima de (9.3). Seja f' uma solução de (9.4) com valor $f'(s^*)$ maior. Então $f'|_E$ é um fluxo válido para (9.3) 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 . ■

9. Algoritmos em grafos

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

$$\begin{array}{ll} \text{maximiza} & f(s) \\ \text{sujeito a} & f(v) = 0 \quad \forall v \in V \setminus \{s, t\} \\ & b_e \leq f_e \leq c_e \quad e \in E. \end{array} \quad (9.5)$$

O problema (9.5) pode ser reduzido para um problema de fluxo máximo simples em $G' = (V', E', c')$ (veja Fig. 9.5(b)) com

$$\begin{aligned} V' &= V \\ E' &= E \cup \{(v, t) \mid (v, u) \in E\} \cup \{(s, u) \mid (v, u) \in E\} \\ c'_e &= \begin{cases} c_e - b_e & e \in E \\ b_{(v,u)} & e = (v, t) \\ b_{(v,u)} & e = (s, u) \end{cases} \end{aligned} \quad (9.6)$$

Lema 9.11

Problema (9.5) possui uma viável sse (9.6) possui uma solução máxima com todos arcos auxiliares $E' \setminus E$ saturados. Neste caso, se f é um fluxo máximo em (9.5),

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

é um fluxo máximo de (9.6) com arcos auxiliares saturados. Conversamente, se f' é um fluxo máximo para (9.6) com arcos auxiliares saturados, $f_e = f'_e + b_e$ é um fluxo máximo em (9.5).

Prova. (Exercício.) ■

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

$$\begin{array}{ll} \text{existe} & f \\ \text{s.a} & f(v) = -d_v \quad \forall v \in V \\ & f_e \leq c_e \quad e \in E. \end{array} \quad (9.7)$$

Evidentemente $\sum_{v \in V} d_v = 0$ é uma condição necessária (lema (9.1)). O problema (9.7) pode ser reduzido para um problema de fluxo máximo em $G' = (V', E')$ com

$$V' = V \cup \{s^*, t^*\}$$

$$E' = E \cup \{(s^*, v) \mid v \in V, d_v < 0\} \cup \{(v, t^*) \mid v \in V, d_v > 0\} \quad (9.8)$$

$$c_e = \begin{cases} c_e & e \in E \\ -d_v & e = (s^*, v) \\ d_v & e = (v, t^*) \end{cases}$$

Lema 9.12

Problema (9.7) possui uma solução sse problema (9.8) 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

$$\begin{array}{lll} \text{existe} & f \\ \text{s.a} & f(v) = d_v & \forall v \in V \\ & b_e \leq f_e \leq c_e & e \in E. \end{array} \quad (9.9)$$

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

$$\begin{aligned} V' &= V \\ E' &= E \end{aligned} \quad (9.10)$$

$$\begin{aligned} c_e &= c_e - b_e \\ d'_v &= d_v - \sum_{e \in \delta^-(v)} b_e + \sum_{e \in \delta^+(v)} b_e \end{aligned} \quad (9.11)$$

Lema 9.13

O problema (9.9) possui solução sse problema (9.10) possui solução.

Prova. (Exercício.) ■

9.1.4. 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) .

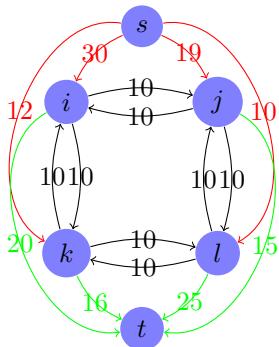
Segmentação de imagens O objetivo é segmentar um imagem em duas partes, por exemplo “foreground” e “background”. Supondo que temos uma “probabilidade” a_i de pertencer ao “foreground” e outra “probabilidade” de pertencer ao “background” b_i para cada pixel i , uma abordagem direta é definir que pixels com $a_i > b_i$ são “foreground” e os outros “background”. Um exemplo pode ser visto na Fig. 9.7 (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 $A \cup B$ tem um valor de

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

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

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

A solução mínima de $Q(A, B)$ pode ser visto como corte mínimo num grafo. O grafo possui um vértice para cada pixel e uma aresta com capacidade p_{ij} entre dois pixels adjacentes i e j . Ele possui ainda dois vértices adicionais s e t , arestas (s, i) com capacidade a_i para cada pixel i e arestas (i, t) com capacidade b_i para cada pixel i (ver Fig. 9.6).



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

Figura 9.6.: Exemplo da construção para uma imagem 2×2 . Direita: Tabela com valores pele/ não-pele. Esquerda: Grafo com penalidade fixa $p_{ij} = 10$.



Figura 9.7.: Segmentação de imagens com diferentes penalidades p . Acima:
(a) Imagem original (b) Segmentação somente com probabilidades ($p = 0$) (c) $p = 1000$ (d) $p = 10000$. Abaixo: (a) Walter Gramatté, Selbstbildnis mit rotem Mond, 1926 (b) Segmentação com $p = 5000$. A probabilidade de um pixel representar pele foi determinado conforme Jones e Rehg [40].

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

1. Porto Alegre (POA), 6.00 – Florianopolis (FLN), 7.00
2. Florianopolis (FLN), 8.00 – Rio de Janeiro (GIG), 9.00
3. Fortaleza (FOR), 7.00 – João Pessoa (JPA), 8.00
4. São Paulo (GRU), 11.00 – Manaus (MAO), 14.00
5. Manaus (MAO), 14.15 – Belém (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 acima. A idéia é representar aviões como fluxo: cada aresta origem-destino é obrigatório, e portanto recebe limites inferiores e superiores de 1, enquanto uma aresta destino-origem é facultativa e recebe limite inferior de 0 e superior de 1. Além disso, introduzimos dois vértices s e t , com arcos facultativos de s para qualquer origem e de qualquer destino para t , que representam os começos e finais da viagem completa de um avião. Para decidir se existe um solução com k aviões, finalmente colocamos um arco (t, s) com limite inferior de 0 e superior de k e decidir se existe uma circulação nesse grafo.

9.1.5. Outros problemas de fluxo

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

FLUXO DE MENOR CUSTO

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

Solução Um fluxo s - t f com valor v .

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

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

9.2. Emparelhamentos

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

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

EMPARELHAMENTO MÁXIMO (EM)

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

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

Objetivo Maximiza $|M|$.

EMPARELHAMENTO DE PESO MÁXIMO (EPM)

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

Solução Um emparelhamento $M \subseteq E$.

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

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

EMPARELHAMENTO PERFEITO DE PESO MÍNIMO (EPPM)

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

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

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

Observe que os pesos em todos problemas podem ser negativos. O problema de encontrar um emparelhamento de peso mínimo em $G = (V, E, c)$ é equivalente com EPM em $-G := (V, E, -c)$ (por quê?). Até EPPM pode ser reduzido para EPM.

Teorema 9.3

EPM e EPPM são problemas equivalentes.

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

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

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

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

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

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

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

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

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

$$\begin{array}{ll} \text{minimiza} & \sum_{e \in E} c_e x_e \\ \text{sujeito a} & \sum_{u \in N(v)} x_{uv} = 1, \quad \forall v \in V \\ & x_e \in \mathbb{B}. \end{array} \quad (9.12)$$

A formulação do problema do emparelhamento máximo é

$$\begin{array}{ll} \text{maximiza} & \sum_{e \in E} c_e x_e \\ \text{sujeito a} & \sum_{u \in N(v)} x_{uv} \leq 1, \quad \forall v \in V \\ & x_e \in \mathbb{B}. \end{array} \quad (9.13)$$

Observação 9.3

A matriz de coeficientes de (9.12) e (9.13) é totalmente unimodular no caso bipartido (pelo teorema de Hoffman-Kruskal). Portanto: a solução da relaxação linear é inteira. (No caso geral isso não é verdadeiro, K_3 é um contra-exemplo, com solução ótima $3/2$). Observe que isso resolve o caso ponderado sem custo adicional. \diamond

Observação 9.4

O dual da relaxação linear de (9.12) é

$$\begin{array}{ll} \text{maximiza} & \sum_{v \in V} y_v \\ \text{sujeito a} & y_u + y_v \leq c_{uv}, \quad \forall uv \in E \\ & y_v \in \mathbb{R}. \end{array} \quad (9.14)$$

e o dual da relaxação linear de (9.13)

$$\begin{array}{ll} \text{minimiza} & \sum_{v \in V} y_v \\ \text{sujeito a} & y_u + y_v \geq c_{uv}, \quad \forall uv \in E \\ & y_v \in \mathbb{R}_+. \end{array} \quad (9.15)$$

Com pesos unitários $c_{uv} = 1$ e restringindo $y_v \in \mathbb{B}$ o primeiro dual é a formulação do conjunto independente máximo e o segundo da cobertura por vértices mínima. Portanto, a observação 9.3 rende no caso não-ponderado:

Teorema 9.4 (Berge, 1951)

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

**9.2.1. 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 9.5**

[58, 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 intersecção. Um conjunto independente máximo é o complemento de uma cobertura por vértices mínima, o problema dual (9.15) de um emparelhamento máximo. Portanto, o tamanho de um emparelhamento máximo é igual $n - h$. Podemos obter o conjunto independente que procuramos usando “a metade” do emparelhamento (os vértices de uma parte só) e os vértices não emparelhados. Podemos achar o emparelhamento em tempo $O(n^{5/2})$ usando o algoritmo de Hopcroft-Karp, porque o grafo de intersecção é bi-partido (por quê?).

9.2.2. 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 9.6**

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

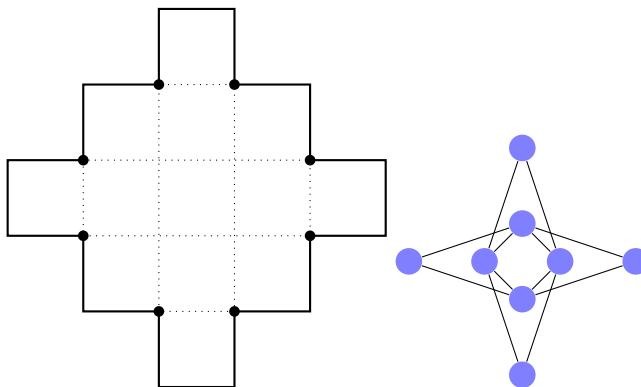


Figura 9.8.: Esquerda: Polígono ortogonal com vértices de reflexo (pontos) e cordas (pontilhadas). Direita: grafo de intersecção.

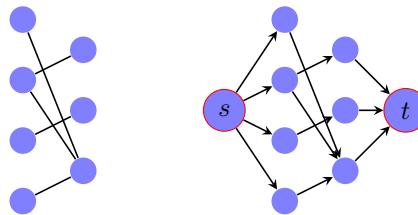


Figura 9.9.: Redução do problema de emparelhamento máximo para o problema do fluxo máximo

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

Teorema 9.7

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

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

Dado um fluxo máximo, existe um fluxo integral equivalente (veja lema (9.3)). 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-ponderado combinatorial Um caminho $P = v_1v_2v_3 \dots v_k$ é alternante em relação a M (ou M -alternante) se $v_i v_{i+1} \in M$ sse $v_{i+1} v_{i+2} \notin M$ para todos $1 \leq i \leq k-2$. Um vértice $v \in V$ é livre em relação a M se ele tem grau 0 em M , e emparelhado caso contrário. Um arco $e \in E$ é livre em relação a M , se $e \notin M$, e emparelhado caso contrário. Escrevemos $|P| \models k-1$ pelo comprimento do caminho P .

Observação 9.5

Caso temos um caminho $P = v_1v_2v_3 \dots v_{2k+1}$ que é M -alternante com v_1 é v_{2k+1} livre, podemos obter um emparelhamento $M \setminus (P \cap M) \cup (P \setminus M)$ de tamanho $|M| \models k + (k-1) = |M| + 1$. 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 9.8 (Hopcroft e Karp [39])

Seja M^* um emparelhamento máximo e M um emparelhamento arbitrário. O conjunto $M \oplus M^*$ contém ao menos $k = |M^*| \models |M|$ 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 $M := M \oplus M^*$. Cada vértice possui no máximo grau 2. Portanto, cada componente é ou um vértice livre, ou um caminhos simples ou um ciclo. Os caminhos e ciclos possuem alternadamente arcos de M e M^* . Portanto os ciclos tem comprimento par. Os caminhos de comprimento ímpar são ou M -aumentantes ou M^* -aumentantes, mas o segundo caso é impossível, porque M^* é máximo. Agora

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

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

Corolário 9.1 (Berge [10])

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

9. Algoritmos em grafos

Rascunho de um algoritmo:

Algoritmo 9.2 (Emparelhamento máximo)

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

Saída Um emparelhamento máximo M .

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

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

Observação 9.6

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

Teorema 9.9

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

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

Observação 9.7

O último teorema é o mesmo que teorema (9.6). \diamond

Observação 9.8

Pelo teorema (9.8) sabemos que em geral existem vários caminhos M -alternantes disjuntos (de vértices) e nos podemos aumentar M com todos eles em paralelo. Portanto, estruturamos o algoritmo em fases: cada fase procura um conjunto de caminhos aumentantes disjuntos e aplicá-los para obter um novo emparelhamento. Observe que pelo teorema (9.8) um aumento com o maior conjunto de caminhos M -alternantes disjuntos resolve o problema imediatamente, mas

não sabemos como achar esse conjunto de forma eficiente. Portanto, procuramos somente um conjunto máximo de caminhos M -alternantes disjuntos de menor comprimento.

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

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

Lema 9.14

Após cada fase, o comprimento de um caminho aumentante mínimo aumenta ao menos dois.

Lema 9.15

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

Teorema 9.10

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

Prova. Pelas lemas 9.14 e 9.15 e a observação que toda fase pode ser completada em $O(m)$. \blacksquare

Usaremos outro lema para provar os dois lemas acima.

Lema 9.16

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

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

Caso contrário: $P \oplus Q$ consiste em dois caminhos, e eventualmente um coleção de ciclos. Os dois caminhos são M -aumentantes, pelas seguintes observações:

1. O início e termo 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' .
3. Nenhum outro vértice de P ou Q é livre em relação a M : P só contém dois vértices livres e Q só contém dois vértices livres em Q mas não em P .
4. Temos dois caminhos M -aumentantes, começando com um vértice livre em Q e terminando com um vértice livre em P . O caminho em $Q \setminus P$ é M -alternante, porque as arestas livres em M' são exatamente as arestas livres em M . O caminho Q entra em P sempre após uma aresta livre em M , porque o primeiro vértice em P já é emparelhado em M e sai de P sempre antes de uma aresta livre em M , porque o último vértice em P já é emparelhado. Portanto os dois caminhos em $P \oplus Q$ são M -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

$$|Q| = |P \oplus Q| + 2|P \cap Q| \vdash |P| \geq |P| + 2|P \cap Q|. \quad \blacksquare$$

Prova. (do lema 9.14). 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 possua um vértice em comum com algum caminho em S , ele possui também um arco em comum (por quê?) e podemos aplicar lema 9.16. \blacksquare

Prova. (do lema 9.15). 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 9.14. Pelo teorema 9.8 existem ao menos $|M^*| \vdash |M|$ caminhos M -aumentantes disjuntos. Mas então $|M^*| \vdash |M| \leq \sqrt{n}$, porque no caso contrário eles possuem mais que n vértices em total. Como o emparelhamento cresce ao menos um em cada fase, o algoritmo executar no máximo mais \sqrt{n} fases. Portanto, o número total de fases é $O(\sqrt{n})$. \blacksquare

O algoritmo de Hopcroft-Karp é o melhor algoritmo conhecido para encontrar emparelhamentos máximos em grafos bipartidos não-ponderados¹. Para subclasses de grafos bipartidos existem algoritmos melhores. Por exemplo, existe um algoritmo randomizado para grafos bipartidos regulares com complexidade de tempo esperado $O(n \log n)$ [30].

¹Feder e Motwani [22, 23] melhoraram o algoritmo para $O(\sqrt{nm}(2 - \log_n m))$.

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

O núcleo de uma implementação do algoritmo de Hopcroft e Karp é descrito na observação 9.8: ele consiste em uma busca por largura até encontrar um ou mais caminhos M -alternantes mínimos e depois uma fase que extrai do DAG definido pela busca um conjunto máximo de caminhos disjuntos (de vértices). A busca por largura começa com todos vértices livres em V_1 . Usamos um vetor H para marcar os arcos que fazem parte do DAG definido pela busca por largura² e um vetor m para marcar os vértices visitados.

```

1 search_paths( $M$ ) :=
2   for all  $v \in V$  do  $m_v := \text{false}$ 
3   for all  $e \in E$  do  $H_e := \text{false}$ 
4
5    $U_1 := \{v \in V_1 \mid v \text{ livre}\}$ 
6
7   do
8     { determina vizinhos em  $U_2$  via arestas livres }
9      $U_2 := \emptyset$ 
10    for all  $u \in U_1$  do
11       $m_u := \text{true}$ 
12      for all  $uv \in E, uv \notin M$  do
13        if not  $m_v$  then
14           $H_{uv} := \text{true}$ 
15           $U_2 := U_2 \cup v$ 
16        end if
17      end for
18    end for
19
20    { determina vizinhos em  $U_1$  via arestas emparelhadas }
21    found := false           { ao menos um caminho encontrado? }
22     $U_1 := \emptyset$ 
23    for all  $u \in U_2$  do
24       $m_u := \text{true}$ 
25      if ( $u$  livre) then
26        found := true
27      else

```

² H , porque o DAG se chama *árvore húngara* na literatura.

```

28       $v := \text{mate}[u]$ 
29      if not  $m_v$  then
30           $H_{uv} := \text{true}$ 
31           $U_1 := U_1 \cup v$ 
32      end if
33  end for
34  end for
35  while (not found)
36 end

```

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

```

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

```

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

Observação 9.9

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). \diamond

Observação 9.10

A redução do teorema 9.6 para um problema de fluxo máximo não se aplica no caso ponderado. Mas, com a simplificação da observação 9.9, 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. \diamond

O dual do problema 9.15 é a motivação para

Definição 9.3

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

Pelo teorema forte de dualidade e o fato que a relaxação linear dos sistemas acima possui uma solução integral (ver observação 9.3) temos

Teorema 9.11 (Egervry [21])

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

O método húngaro A aplicação de um caminho M -aumentante $P = (v_1 v_2 \dots v_{2n+1})$ resulta num emparelhamento de peso $c(M) + \sum_i \text{impar } c_{v_i v_{i+1}} - \sum_i \text{par } c_{v_i v_{i+1}}$. Isso motiva a definição de uma árvore húngara ponderada. Para um emparelhamento M , seja H_M o grafo direcionado com as arestas $e \in M$ orientadas de T para S com peso $l_e := w_e$, e com as restantes arestas $e \in E \setminus M$ orientadas de S para T com peso $l_e := -w_e$. Com isso a aplicação do caminho 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 9.3 (Emparelhamento de peso máximo)

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

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

```

1   $M = \emptyset$ 
2  while (existe um caminho  $M$  aumentante  $P$ ) do
3      encontra o caminho  $M$  aumentante mínimo  $P$  em  $H_M$ 
4      caso  $l(P) \geq 0$  break;
```

```

5       $M := M \oplus P$ 
6  end while
7  return  $M$ 

```

Observação 9.11

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

Observação 9.12

Lembrando Bellman-Ford: Seja $d_k(t)$ a distância mínimo de qualquer caminho de s para t usando no máximo k arcos ou ∞ caso não existe. Temos

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

 \diamond

Para ver que o algoritmo é correto, chama um emparelhamento M *extremo* caso ele possua o maior peso entre todos emparelhamentos de tamanho $|M|$.

Teorema 9.12

Cada emparelhamento encontrado no algoritmo 9.3 é extremo.

Prova. Por indução. Para $M = \emptyset$ o teorema é correto. Seja M um emparelhamento extremo, P o caminho aumentante encontrado pelo algoritmo 9.3 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 9.8). Sabemos $l(Q) \geq l(P)$ pela minimalidade de P . $N \oplus Q$ é um emparelhamento de cardinalidade $|M|$, logo $c(N \oplus Q) \leq c(M)$. Com isso temos

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

 \blacksquare **Proposição 9.1**

Caso não existe caminho M -aumentante com comprimento negativo no algoritmo 9.3, M é máximo.

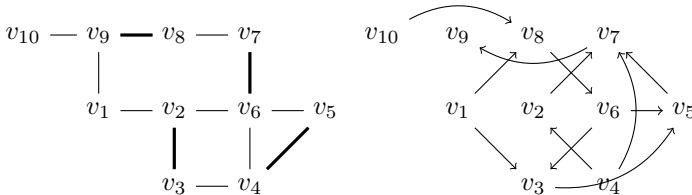


Figura 9.10.: Grafo com emparelhamento e grafo auxiliar.

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

Observação 9.13

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 obtemos um algoritmo em tempo $O(n(m + n \log n))$. ◇

9.2.3. 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 impares. 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 $v_1v_8v_6v_5v_7v_9$ 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 aumenta o emparelhamento. O problema é que o laço ímpar $v_6v_4v_5v_6$ não pode ser eliminado sem consequências.

	Cardinalidade	Ponderado
Bi-partido	$O(n\sqrt{\frac{mn}{\log n}})$ $O(m\sqrt{n}\frac{\log(n^2/m)}{\log n})$ [23]	[6] $O(nm + n^2 \log n)$ [46, 55]
Geral	$O(m\sqrt{n}\frac{\log(n^2/m)}{\log n})$ [31, 26]	$O(n^3)$ [20] $O(mn + n^2 \log n)$ [29]

Tabela 9.2.: Resumo emparelhamentos

9.2.4. Exercícios**Exercício 9.1**

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

10. Algoritmos de aproximação

(As notas seguem Vazirani [69].)

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

Definição 10.1

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

$$\text{OPT}_{\Pi_2}(x_2) \leq \text{OPT}_{\Pi_1}(x_1) \quad (10.1)$$

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

$$\varphi_{\Pi_1}(x_1, y_1) \leq \varphi_{\Pi_2}(x_2, y_2) \quad (10.2)$$

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

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

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

10.1. Aproximação para o problema da árvore de Steiner mínima

Seja $G = (V, A)$ um grafo completo, não-direcionado com custos $c_a \geq 0$ nos arcos. O problema da árvore Steiner mínima (ASM) consiste em achar o

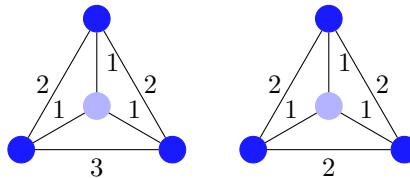


Figura 10.1.: Grafo com fecho métrico.

subgrafo conexo mínimo que inclui um dado conjunto de *vértices necessários* $R \subseteq V$ (terminais). Esse subgrafo sempre é uma árvore (ex. 10.1). 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 10.1

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

Definição 10.2

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

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

para qualquer tripla de vértices i, j, k .

Teorema 10.1

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

Prova. O “fecho métrico” de $G = (V, A)$ é um grafo G' completo sobre vértices e com custos $c'_{ij} := d_{ij}$, sendo d_{ij} o comprimento do menor caminho entre i e j em G . Evidentemente $c'_{ij} \leq c_{ij}$ é portanto (10.1) é satisfeita. Para ver que (10.2) é satisfeita, seja T' uma solução de ASM em G' . Define T como união de todos caminhos definidos pelos arcos em T' , menos um conjunto de arcos para remover eventuais ciclos. O custo de T é no máximo $c(T')$ porque o custo de todo caminho é no máximo o custo da aresta correspondente em T' . \blacksquare

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

Teorema 10.2

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

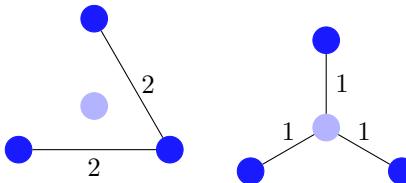


Figura 10.2.: AGM sobre R e melhor solução. ●: vértice em R , ○: vértice Steiner.

Prova. Considere a solução ótima S^* de ASM. Duplica todas arestas¹ tal que todo vértice possui grau par. Acha um caminho Euleriano nesse grafo. Remove vértices duplicados nesse caminho. O custo do caminho C obtido dessa forma não é mais que o dobro do custo original: o grafo com todas arestas custa $2c(S^*)$ e a remoção de vértices duplicados não aumenta esse custo, pela metricidade. Como esse caminho é uma árvore geradora, temos $c(A) \leq c(C) \leq 2c(S^*)$ para AGM A . ■

10.2. Aproximações para o PCV

Teorema 10.3

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

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

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

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

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

1. Determina uma AGM A de G .

¹Isso transforma G num multigrafo.

10. Algoritmos de aproximação

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

Teorema 10.4

O algoritmo acima define uma 2-aproximação.

Prova. A melhor solução do PCV menos uma aresta é uma árvore geradora de G . Portanto $c(A) \leq \text{OPT}$. 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 10.2. ■

O fator 2 dessa aproximação é resultado do passo 2 que duplica todas arestas para garantir a existência de um caminho Euleriano. Isso pode ser garantido mais barato: A AGM A possui um número par de vértices com grau ímpar (ver exercício 10.2), 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 10.5

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

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

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

e portanto

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

10.3. Algoritmos de aproximação para 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:

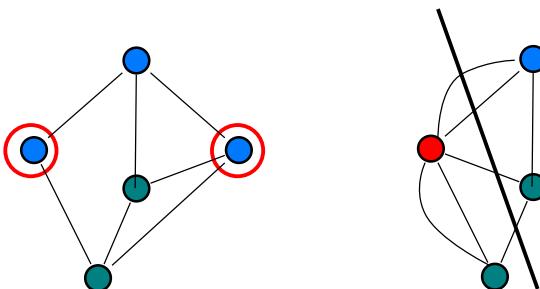


Figura 10.3.: Identificação de dois terminais e um corte no grafo reduzido. Vértices em verde, terminais em azul. O grafo reduzido possui múltiplas arestas entre vértices.

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

Fato 10.1

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

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

Isso leva ao algoritmo

Algoritmo 10.1 (CI)

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

Saída Um corte múltiplo que separa os s_i .

1. Para cada $i \in [1, k]$: Calcula o corte isolante C_i de s_i .

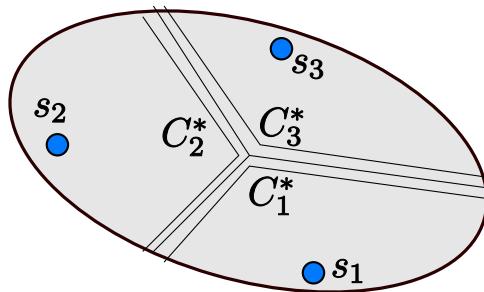


Figura 10.4.: Corte múltiplo e decomposição em cortes isolantes.

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

Teorema 10.6

Algoritmo 10.1 é uma $2 - 2/k$ -aproximação.

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

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

(Veja fig. 10.4.) Cada aresta de C^* faz parte das cortes das duas componentes adjacentes, e portanto

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

e ainda $w(C_i) \leq w(C_i^*)$ para os cortes C_i do algoritmo 10.1, porque nos usamos o corte isolante mínimo de cada componente. Logo para o corte C retornado pelo algoritmo temos

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

A análise do algoritmo é ótimo, como o seguinte exemplo da fig. 10.5 mostra. O menor corte que separa s_i tem peso $2 - \epsilon$, portanto o algoritmo retorne um corte de peso $(2 - \epsilon)k - (2 - \epsilon) = (k - 1)(2 - \epsilon)$, enquanto o menor corte que separa todos terminais é o ciclo interno de peso k .

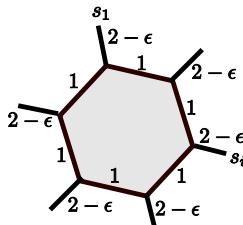


Figura 10.5.: Exemplo de um grafo em que o algoritmo 10.1 retorne uma $2 - 2/k$ -aproximação.

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

Fato 10.2

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 consequência, a AGH codifica o valor de todos cortes em G .

Ele pode ser calculado com $n - 1$ 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 e dois vértices do grafo original representados por ela.
 - 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 ao menos k componentes. Isso leva ao seguinte algoritmo.

Algoritmo 10.2 (KCM)

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

Saida Um k -corte.

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

Teorema 10.7

Algoritmo 10.2 é uma $2 - 2/k$ -aproximação.

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

Removendo C^* de G gera componentes V_1, \dots, V_k : Define um grafo sobre esses componentes identificando vértices de uma componente com arcos da AGH T entre os componentes, e eventualmente removendo arcos até obter uma nova árvore T' . Seja C_k^* o corte de maior peso, e define V_k como raiz da árvore. Desta forma, cada componente V_1, \dots, V_{k-1} possui uma aresta associada na direção da raiz. Para cada dessas arestas (u, v) temos

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

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

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

■

10.4. Exercícios**Exercício 10.1**

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

Exercício 10.2

Mostra que o número de vértices com grau ímpar num grafo sempre é par.

Parte IV.

Teoria de complexidade

11. Do algoritmo ao problema

11.1. Introdução

Motivação

- Análise e projeto: Foca em *algoritmos*.
- Teoria de complexidade: Foca em *problemas*.
- Qual a complexidade intrínseca 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 $\Sigma^* \rightarrow \Sigma^*$ Alternativa permitindo várias soluções: relação $R \subseteq \Sigma^* \times \Sigma^*$.
 - Problema de decisão: Função $\Sigma^* \rightarrow \{S, N\}$ 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 $\leq k$?”.
- Freqüentemente: Alfabeto $\Sigma = \{0, 1\}$ com codificação adequada.

Convenção 11.1

Sem perda de generalidade suporemos $\Sigma = \{0, 1\}$.

Definição 11.1

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.

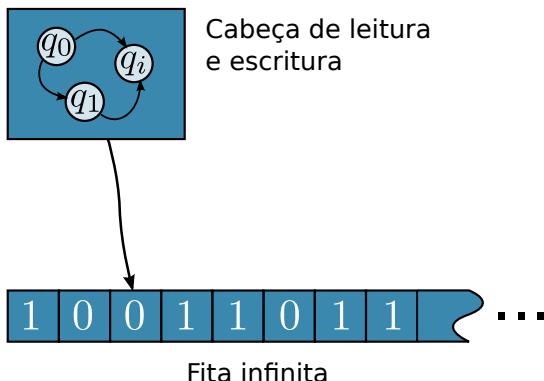
(Gödel em uma carta para Kreisel, em maio de 1968, falando sobre uma definição da computação.).



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 9999999999999999 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 9999999999999999 are the same. [68].

Máquina de Turing



Máquina de Turing (MT)

$$M = (Q, \Sigma, \Gamma, \delta)$$

- Alfabeto de entrada Σ (sem branco $\underline{\quad}$)
- 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 $\underline{\quad} \in \Gamma$)
- Regras $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, escritas da forma

$$q, a \rightarrow q' a' D$$

(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 $qa \rightarrow q'a'D$ (um L na primeira posição não tem efeito) até
 - não encontrar uma regra: a computação termina, ou

11. Do algoritmo ao problema

- 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.

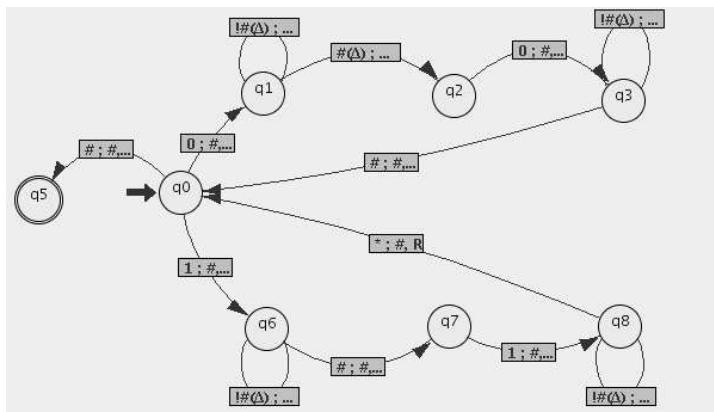
Exemplo 11.1 (Decidir ww^R)

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). \diamond

Máquinas não-determinísticas

- Observe: Num estado q lendo símbolo a temos exatamente uma regra da forma $qa \rightarrow q'a'D$.
- 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

$$\delta : Q \times \Gamma \rightarrow \mathcal{P}(Q \times \Gamma \times \{L, R\})$$

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 11.1

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 . \diamond

Linguagem de uma MTND

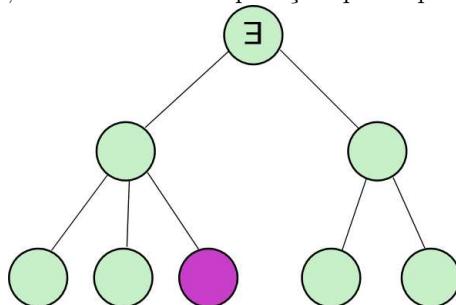
- Conjunto de linguagens Turing-reconhecíveis: linguagens *recursivamente enumeráveis* ou *computavelmente enumeráveis*.
- Conjunto de linguagens Turing-decidíveis: linguagens *recursivas* ou *computáveis*.
- Para uma máquina não-determinística temos que modificar a definição: ela precisa somente um 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:
 1. Uma MT com $k > 1$ fitas (cada uma com cabeça própria),
 2. uma MT com fita duplamente infinita,
 3. uma MT com alfabeto restrito $\Sigma = \{0, 1\}$,
 4. uma MTND e
 5. 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.)

1. 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 \grave{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.

2. Seja a_i o símbolo na posição i da fita, com $i \in \mathbb{Z}$. A máquina com uma xúnica 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.

3. 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.
4. A MTD simula uma MTN listando todas as execuções possíveis sistematicamente.
5. 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 11.2

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 é usado como *fita da saída* caso a MT calcula uma função. \diamond

Proposição 11.1

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 $\log |\Gamma|$ na posição atual, calcula o novo símbolo e estado usanda as regras de M , escreve os novos $\log |\Gamma|$ símbolos, e movimenta a cabeça $\log |\Gamma|$ 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). \blacksquare

Proposição 11.2

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' armazeno o conteúdo das k fitas de M de forma intercalada nas posições $1 + ki$, para $1 \leq i \leq k$. Para representar as posições das cabeças, usamos símbolos com uma marca \grave{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). \blacksquare

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 e 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 11.1 (Arora e Barak [7, 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 de 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 11.2. O número de passos de U então é limite por cT^2 com $c = 25c'$. ■

Observação 11.3

Uma simulação de MTND é possível com os mesmos limites. ◇

Exemplo 11.2 (Máquina universal)

Considere a máquina $M = (\{u, d\}, \{a, b\}, \{a, b, _\}, \delta)$ com

$$\delta = \{ua \rightarrow ubL, ub \rightarrow uaL, u_ \rightarrow dbR, da \rightarrow u_R, db \rightarrow daR, d_ \rightarrow uaL\}.$$

Essa máquina é universal? Ver <http://www.wolframscience.com/prizes/tm23>. Aparentemente o problema foi resolvido em outubro de 2007. ◇

Computabilidade e complexidade

Decidibilidade 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 11.2 (Leeuwen [48])

$$m - \text{tapes} \leq 1 - \text{tape}(\text{time } kn^2 \text{ & space Lin})$$

$$m - \text{tapes} \leq 2 - \text{tape}(\text{time } kn \log n \text{ & space Lin})$$

$$\text{SRAM} - \text{UTIME} \leq T(\text{time } n^2 \log n)$$

$$\text{RAM} - \text{UTIME} \leq T(\text{time } n^3)$$

$$\text{MRAM} - \text{UTIME} \leq T(\text{time Exp})$$

$$\text{SRAM} - \text{LTIME} \leq T(\text{time } n^2)$$

$$\text{RAM} - \text{LTIME} \leq T(\text{time } n^2)$$

$$\text{MRAM} - \text{LTIME} \leq T(\text{time Poly})$$

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 definida 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.

12. Classes de complexidade

12.1. Definições básicas

Complexidade pessimista

- Recursos básicos: *tempo* e *espaço*.
- A *complexidade de tempo (pessimista)* é uma função

$$t : \mathbb{N} \rightarrow \mathbb{N}$$

tal que $t(n)$ é o número máximo de passos para entradas de tamanho n .

- A *complexidade de espaço (pessimista)* é uma função

$$s : \mathbb{N} \rightarrow \mathbb{N}$$

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 12.1

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: $n^k, \log n, 2^n, n!, \dots$

Observação 12.1

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 $t, s : \mathbb{N} \rightarrow \mathbb{N}$ e um problema $L \subseteq \Sigma^*$
 - $L \in \text{DTIME}[t(n)]$ se existe uma máquina Turing determinística tal que aceita L com complexidade de tempo $t(n)$.
 - $L \in \text{NTIME}[t(n)]$ se existe uma máquina Turing não-determinística que aceita L com complexidade de tempo $t(n)$.
 - $L \in \text{DSPACE}[s(n)]$ se existe uma máquina Turing determinística que aceita L com complexidade de espaço $s(n)$.
 - $L \in \text{NSPACE}[s(n)]$ se existe uma máquina Turing não-determinística que aceita L com complexidade de espaço $s(n)$.

Hierarquia básica

- Observação

$$\text{DTIME}[F(n)] \subseteq \text{NTIME}[F(n)] \subseteq \text{DSPACE}[F(n)] \subseteq \text{NSPACE}[F(n)]$$

- Definições conhecidas:

$$P = \bigcup_{k \geq 0} \text{DTIME}[n^k]; \quad NP = \bigcup_{k \geq 0} \text{NTIME}[n^k]$$

- Definições similares para espaço:

$$PSPACE = \bigcup_{k \geq 0} \text{DSPACE}[n^k]; \quad NSPACE = \bigcup_{k \geq 0} \text{NSPACE}[n^k]$$

- Com a observação acima, temos

$$P \subseteq NP \subseteq \text{DSPACE} \subseteq \text{NSPACE}.$$

Prova. (Da observação.) Como uma máquina não-determinística é uma extensão da uma máquina determinística, temos obviamente $\text{DTIME}[F(n)] \subseteq \text{NTIME}[F(n)]$ e $\text{DSPACE}[F(n)] \subseteq \text{NSPACE}[F(n)]$. A inclusão $\text{NTIME}[F(n)] \subseteq \text{DSPACE}[F(n)]$ segue, porque todas computações que precisam menos que $F(n)$ passos, precisam menos que $F(n)$ espaço também. ■

Classes de complexidade

Zoológico de complexidade

12.2. Hierarquias básicas

Aceleração

Teorema 12.1

Podemos comprimir ou acelerar computações por um fator constante. Para todos $c > 0$ no caso de espaço temos

$$\begin{aligned} L \in \text{DSPACE}[s(n)] &\Rightarrow L \in \text{DSPACE}[cs(n)] \\ L \in \text{NSPACE}[s(n)] &\Rightarrow L \in \text{NSPACE}[cs(n)] \end{aligned}$$

e no caso do tempo, para máquinas de Turing com $k > 1$ fitas e $t(n) = \omega(n)$

$$\begin{aligned} L \in \text{DTIME}[s(n)] &\Rightarrow L \in \text{DTIME}[cs(n)] \\ L \in \text{NTIME}[s(n)] &\Rightarrow L \in \text{NTIME}[cs(n)] \end{aligned}$$

Prova. (Rascunho.) A idéia 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 em 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 12.2

Para $t(n)$ e $s(n)$ total e recursivo, existe um linguagem L tal que $L \notin \text{DTIME}[t(n)]$ ou $L \notin \text{DSPACE}[s(n)]$, respectivamente.

Prova. (Rascunho). Por diagonalização. As máquinas de Turing são enumeráveis: seja M_1, M_2, \dots uma enumeração deles e seja x_1, x_2, \dots 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 $\text{DTIME}[t(n)]$. 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 12.3 (Hartmanis,Stearns, 1965)

Para f, g com g tempo-construtível e $f \log f = o(g)$ temos

$$\text{DTIME}(f) \subsetneq \text{DTIME}(g).$$

Para funções f, g , com $g(n) \geq \log_2 n$ espaço-construtível e $f = o(g)$ temos

$$\text{DSPACE}(f) \subsetneq \text{DSPACE}(g).$$

Prova. (Rascunho.) Para provar o segundo parte (que é mais fácil) temos que mostrar que existe uma linguagem $L \subseteq \Sigma^*$ tal que $L \in \text{DSPACE}[g]$ mas $L \notin \text{DSPACE}[f]$. Vamos construir uma MT M sobre alfabeto de entrada $\Sigma = \{0, 1\}$ tal que $L(M)$ satisfaz essas características. A idéia 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 realizar essa ideia:

1. 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.
2. 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 $\langle M \rangle 10^n$).
3. 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 idéia 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 12.4 (Savitch)

Para cada função espaço-construtível $s(n) \geq \log_2 n$

$$\text{NSPACE}[s(n)] \subseteq \text{DSPACE}[s(n)^2]$$

- Corolário: $\text{DSPACE} = \text{NSPACE}$
- Não-determinismo ajuda pouco para espaço!

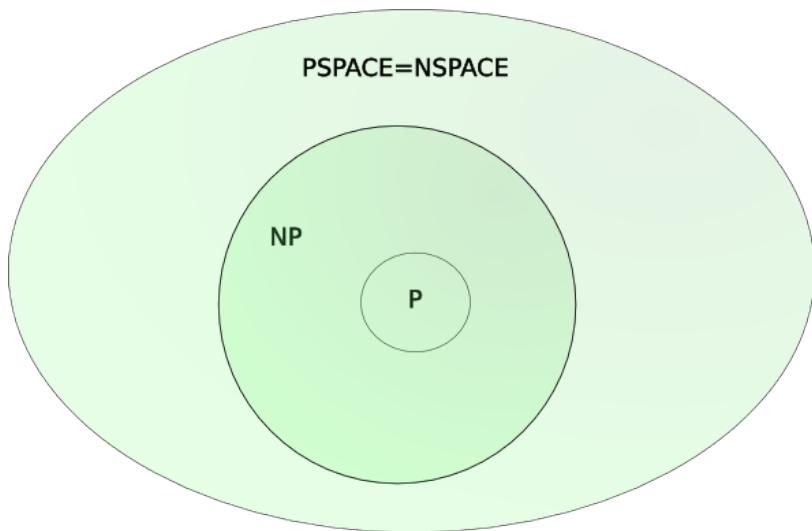


Walter
J. Savitch
(*1943)

Prova. (Rascunho.) Caso $L \in \text{NSPACE}[s(n)]$ 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 é 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 é $O(s(n))$, 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)



12.3. Exercícios

Exercício 12.1

Dado uma máquina de Turing com oráculo para o problema de parada, é possível calcular a função do “castor ocupado” (ingl. busy beaver)?

13. Teoria de NP-completude

13.1. 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 [12], [8, Th. 25.6] e [63, Th. 7.16]).

Teorema 13.1 (Complexidade das linguagens da hierarquia de Chomsky)

$$\text{REG} = \text{DSPACE}[O(1)] = \text{DSPACE}[o(\log \log n)]$$

$$\text{REG} \subseteq \text{DTIME}[n]$$

$$\text{CFL} \subseteq \text{DSPACE}[n^3]$$

$$\text{CSL} = \text{NSPACE}[n]$$

Normalmente, nosso interesse são soluções, não decisões: Ao seguir vamos definir P e NP em termos de soluções. As perguntas centrais como $P \neq NP$ acabam de ter respostas equivalentes.

P e NP em termos de busca

- A computação de uma solução pode ser vista como função $\Sigma^* \rightarrow \Sigma^*$
- 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.

13. Teoria de NP-completude

- Intuição: Uma relação é a especificação de um problema de busca: para entrada x queremos achar alguma solução y tal que $(x, y) \in R$.
- Nossa interesse são soluções que podem ser “escritas” em tempo polinomial:

Definição 13.1

Uma relação binária R é *polinomialmente limitada* se

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

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 13.2 ([67, Th. 2],[7, 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 se 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

```

1  if  $A(x, \epsilon) = \text{não}$  then
2    return “não existe solução”,
3  end if
4   $w := \epsilon$ 
5  while  $(x, w) \notin R$  do
6    if  $A(x, w \circ 0) = \text{sim}$  then
7       $w := w \circ 0$ 
8    else
9       $w := w \circ 1$ 
10   end if
11 end while
12 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

$$\text{CLIQUE} = \{\langle G, k \rangle \mid \text{Grafo não-direcionado } G \text{ com clique de tamanho } k\}$$

$$\text{SAT} = \{\langle \phi \rangle \mid \phi \text{ fórmula satisfatível da lógica proposicional em FNC}\}$$

$$\text{TSP} = \{\langle M, b \rangle \mid \text{Matriz simétrica de distâncias } M \text{ que tem TSP-círculo } \leq b\}$$

$$\text{COMPOSITE} = \{\langle n \rangle \mid n = m_1 m_2 \text{ com } m_1, m_2 > 1\}$$

13.2. Reduções

Reduções

Definição 13.2 (Redução em tempo polinomial)

Uma (many-one) *redução* entre duas linguagens L, L' com alfabetos Σ e Σ' é um função total $f : \Sigma^* \rightarrow \Sigma'^*$ 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 13.3 (Problemas difíceis e completos)

Dado uma classe de problemas C e um tipo de redução \leq , um problema L é

13. Teoria de NP-completude

C -≤-difícil, se $L' \leq L$ para todos $L' \in C$. Um problema L que é C -≤-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 13.1 (Fecho para baixo)

Se $A \leq_P B$ e $B \in P$ então $A \in P$.

Proposição 13.2 (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?

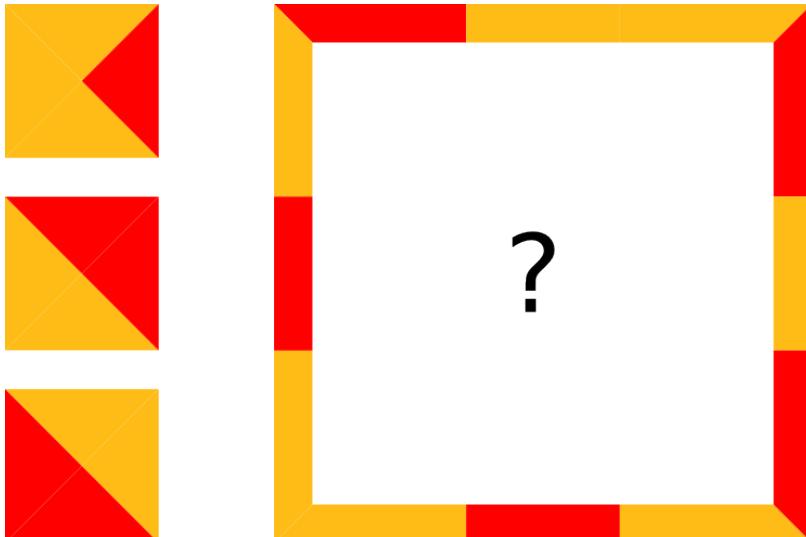
Teorema 13.3

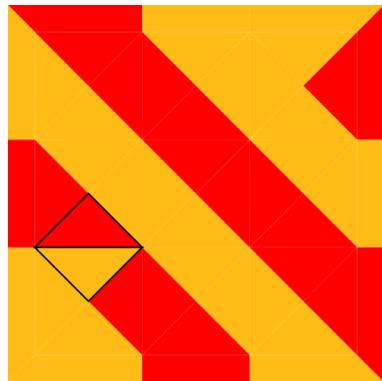
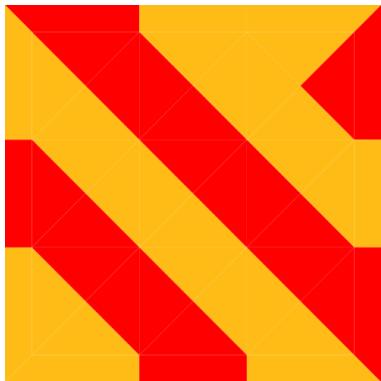
$$\text{BHALT} = \{\langle M, w, 1^n \rangle \mid \text{MT } M \text{ para com entrada } w \text{ em } n \text{ passos}\}$$

é NP-completo.

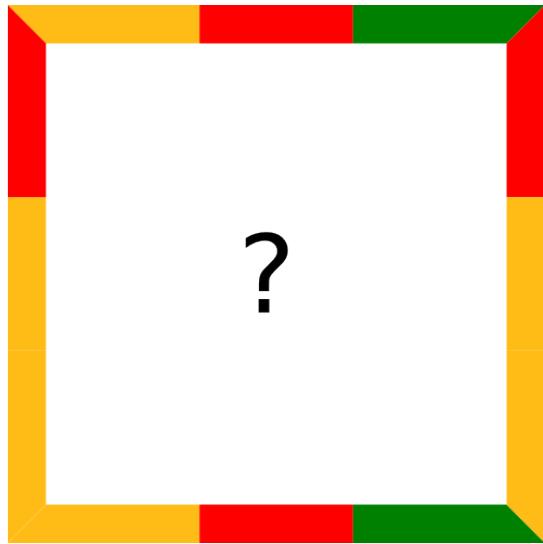
Prova. $\text{BHALT} \in \text{NP}$ porque podemos verificar uma execução em n passos em tempo polinomial. Observe que a codificação de uma execução em limitada polinomialmente em termos da entrada $\langle M, w, 1^n \rangle$ pela codificação de n em unário. Logo é suficiente de mostrar que qualquer problema em NP pode ser reduzido para BHALT.

Para alguma linguagem $L \in \text{NP}$, 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 $w \in L \Leftrightarrow w' \in \text{BHALT}$. Logo $L \leq_P \text{BHALT}$. ■

Ladrilhar: Exemplo**Ladrilhar: Solução**



Ladrilhar: Exemplo



Ladrilhar: O problema

- Para um conjunto finito de cores C , o *tipo* de um ladrilho é uma função

$$t : \{N, W, S, E\} \rightarrow C.$$

LADRILHAMENTO

Instância Tipos de ladrilhos t_1, \dots, t_k e um grade de tamanho $n \times n$ com cores nas bordas. (Cada ladrilho pode ser representado por quatro símbolos para as cores; a grade consiste de n^2 ladrilhos em branco e $4n$ cores; uma instância tem tamanho $O(k + n^2)$).

Questão Existe um ladrilhamento da grade tal que todas cores casam (sem girar os ladrilhos)?

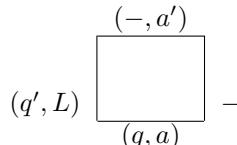
Teorema 13.4 (Levin)

Ladrilhamento é NP-completo.

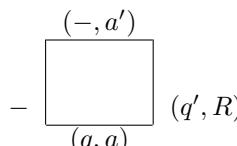
Prova. O problema é em NP, porque dado um conjunto de tipos de ladrilhos e um ladrilhamento, podemos verificar as restrições das cores em tempo polinomial.

Vamos reduzir qualquer problema em $L \in \text{NP}$ 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 $(|w|^k)^2$. Idéia: os cores dos cantos de sul e de norte vão codificar um símbolo da fita a informação se a cabeça está presente e o estado da máquina. Os cores dos cantos oeste e este vão codificar informação adicional para mover a cabeça. O canto sul da grade vão ser colorido conforme o estado inicial da máquina, o canto norte com o estado final e vamos projetar 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 sucessoras.

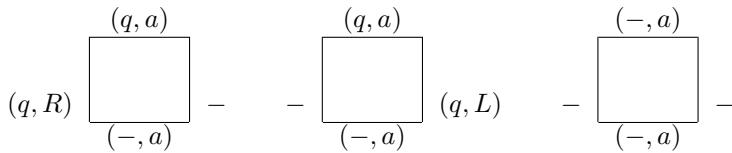
Nos vamos usar as cores $Q \cup \{-\} \times \Gamma$ na direção norte/sul e $Q \times \{L, R\} \cup \{-\}$ na direção oeste/este. Para uma regra $q, a \rightarrow q', a', L$ os ladrilhos tem a forma



e para $q, a \rightarrow q', a', R$



Além disso, tem ladrilhos



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 13.3

Se $A \subseteq B$ e A é fechado para baixo em relação à redução \leq e L é B - \leq -completo então

$$L \in A \iff A = B.$$

Exemplo: O problema SAT

SAT

Instância Fórmula proposicional em forma normal conjuntiva $\Phi(x_1, \dots, x_n)$.

Questão Tem uma atribuição $a_1, \dots, a_n \in \mathbb{B}$ que satisfaz Φ ?

Teorema 13.5 (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

$$\begin{aligned} L_{i,x,y} := & N_{x,y,t_i(N)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(N)}} \neg N_{x,y,c} \\ & \wedge W_{x,y,t_i(W)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(W)}} \neg W_{x,y,c} \\ & \wedge S_{x,y,t_i(S)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(S)}} \neg S_{x,y,c} \\ & \wedge E_{x,y,t_i(E)} \wedge \bigwedge_{\substack{c \in C \\ c \neq t_i(E)}} \neg E_{x,y,c} \end{aligned}$$

Prova (2)

13. Teoria de NP-completude

Sejam $c_{x,y}$ as cores na bordas. Seja ϕ a conjunção de

$$\begin{aligned}
 & \bigwedge_{x \in [1,n]} \bigwedge_{y \in [1,n]} \bigvee_{c \in [1,k]} L_{c,x,y} \quad \text{Toda posição tem um ladrilho} \\
 & \bigwedge_{x \in [1,n]} S_{x,1,c_{x,1}} \wedge N_{x,n,c_{x,n}} \quad \text{Cores corretas nas bordas N,S} \\
 & \bigwedge_{y \in [1,n]} W_{1,y,c_{1,y}} \wedge E_{n,y,c_{n,y}} \quad \text{Cores corretas nas bordas W,E} \\
 & \bigwedge_{x \in [1,n[} \bigwedge_{y \in [1,n]} E_{x,y,c} \Rightarrow W_{x+1,y,c} \quad \text{Correspondência E-W} \\
 & \bigwedge_{x \in]1,n]} \bigwedge_{y \in [1,n]} W_{x,y,c} \Rightarrow E_{x-1,y,c} \quad \text{Correspondência W-E} \\
 & \bigwedge_{x \in [1,n[} \bigwedge_{y \in [1,n]} N_{x,y,c} \Rightarrow S_{x,y+1,c} \quad \text{Correspondência N-S} \\
 & \bigwedge_{x \in [1,n[} \bigwedge_{y \in]1,n]} S_{x,y,c} \Rightarrow N_{x,y-1,c} \quad \text{Correspondência S-N}
 \end{aligned}$$

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. [64]

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.

A 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 da 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! [1]

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)?

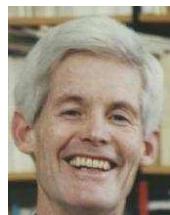
1	?	?	?	1
?	3	?	3	1
?	4	?	3	
?	2	?	2	
?	?	2	2	
1	?	?	1	
?	?	2	2	1
1	?	1	?	1

O mundo agora

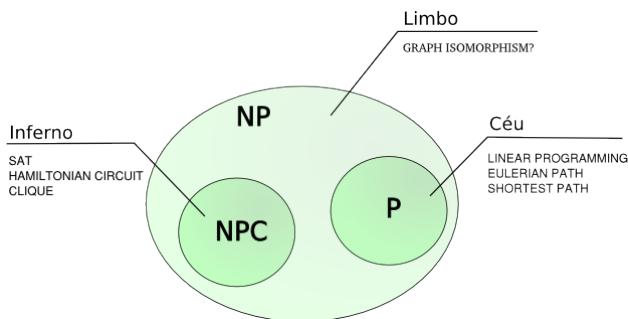
- O milagre da NP-completude
 - Qualquer problema em NP tem uma redução polinomial para SAT!
 - Por que não usar só SAT? (soluções em 1.3^n)?

Teorema 13.6 (Ladner [47])

- Se $P \neq NP$, existe uma linguagem $L \in NP$ que nem é NP-completo nem em P.



Stephen Arthur Cook (*1939)



Leonid Levin (*1948)

Muitos se interessavam

Woeginger's página sobre P vs. NP

13.3. Exercícios

Exercício 13.1

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.

14. Fora do NP

Classes fora do P-NP

$$\text{L} = \text{DSPACE}[\log n]$$

$$\text{NL} = \text{NSPACE}[\log n]$$

$$\text{EXPTIME} = \bigcup_{k>0} \text{DTIME}[2^{n^k}]$$

$$\text{NEXPTIME} = \bigcup_{k>0} \text{NTIME}[2^{n^k}]$$

$$\text{EXPSPACE} = \bigcup_{k>0} \text{DSPACE}[2^{n^k}]$$

$$\text{NEXPSPACE} = \bigcup_{k>0} \text{NSPACE}[2^{n^k}]$$

Co-classes

Definição 14.1 (Co-classes)

Para uma linguagem L , a *linguagem complementar* é $\overline{L} = \Sigma^* \setminus L$. Para uma classe de complexidade C , a *co-classe* $co - C = \{\overline{L} \mid L \in C\}$ é a classe das linguagens complementares.

Proposição 14.1

$$\text{P} = \text{co-P}.$$

- Qual problema pertence à NP?

$\overline{\text{CLIQUE}}, \overline{\text{SAT}}, \overline{\text{TSP}}, \overline{\text{COMPOSITE}}$.

Prova. Seja $L \in \text{P}$. 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 $\overline{\text{CLIQUE}}, \overline{\text{SAT}}, \overline{\text{TSP}}$ pertencem à NP. Em 2002 foi provado, que $\overline{\text{COMPOSITE}} \in \text{P}$ [3]. Observe que se aplica só para o *teste* se um número é 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 $\text{TAUT} \notin \text{NP}$.
- Em outras palavras, NP parece de não ser fechado sobre a complementação:

$$\text{co-NP} \neq \text{NP} ?$$

Proposição 14.2

Se $L \in \text{NPC}$ então $L \in \text{co-NP} \iff \text{NP} = \text{co-NP}$.

Proposição 14.3

TAUT é co-NP-completo.

Prova. (Proposição 14.2.) Seja $L \in \text{NPC}$. (\rightarrow): Seja $L \in \text{co-NP}$. Se $L' \in \text{NP}$, temos $L' \leq_P L \in \text{co-NP}$, logo $\text{NP} \subseteq \text{co-NP}$. Se $L' \in \text{co-NP}$, então $\overline{L'} \in \text{NP}$ e $\overline{L'} \leq_P L \in \text{co-NP}$, logo $\overline{L'} \in \text{co-NP}$ e $L' \in \text{NP}$. (\leftarrow): Como $L \in \text{NPC} \subseteq \text{NP}$, e $\text{NP} = \text{co-NP}$, também $L \in \text{co-NP}$. ■

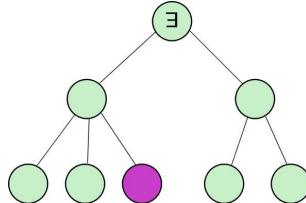
Prova. (Proposição 14.3, rascunho.) TAUT $\in \text{co-NP}$, porque uma MT com um estado universal pode testar todas atribuições das variáveis proposicionais e aceita se todas são verdadeiras.

Para provar a completude, temos que provar, que toda linguagem $\overline{L} \in \text{co-NP} \leq_P \text{TAUT}$. 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

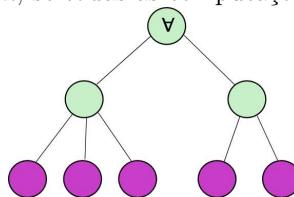
$$\begin{array}{ll} w \in \overline{L} \iff \varphi_w \text{ satisfável} & \text{pela def. de } \varphi_w \\ w \in L \iff \varphi_w \text{ insatisfável} & \text{negação da afirmação} \\ \iff \neg\varphi_w \text{ é tautologia} & \end{array}$$

A classe co-NP

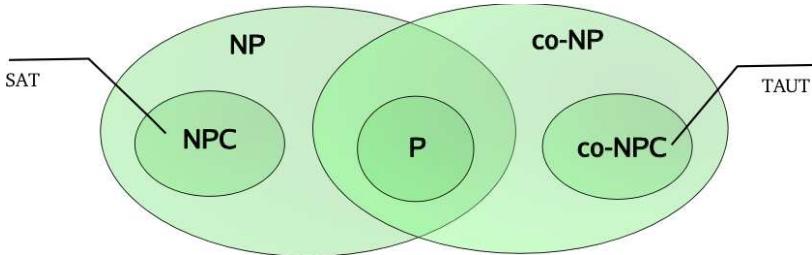
- NP: Resposta *sim*, se existe uma computação que responde *sim*.

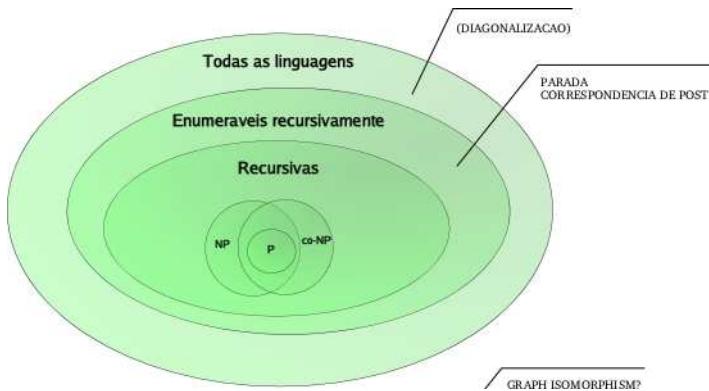


- Ou: Dado um certificado, *verificável* em tempo polinomial.
- co-NP: Resposta *sim*, se todas as computações respondem *sim*



- Ou: Dado um “falsificado”, *falsificável* em tempo polinomial.

O mundo da complexidade ao redor do P**14.1. De P até PSPACE****O mundo inteiro (2)**



Problemas PSPACE-completos

- Não sabemos, se $\text{NP} = \text{PSPACE}$ ou até $\text{P} = \text{PSPACE}$
- Como resolver isso? Da mesma forma que a questão $\text{P} = \text{NP}$: busque problemas PSPACE-completos (relativo a \leq_P).
- Considere

FORMULAS BOOLEANAS QUANTIFICADAS (INGL. QUANTIFIED BOOLEAN FORMULAS, QBF)

Instância Uma sentença booleana

$$\Phi := (Q_1 x_1)(Q_2 x_2) \cdots (Q_n x_n)[\varphi(x_1, x_2, \dots, x_n)]$$

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 14.1

QBF é PSPACE-completo.

Prova. (Rascunho.) É fácil de provar que $\text{QBF} \in \text{PSPACE}$: Podemos verificar recursivamente que a sentença é verdadeira: Para uma fórmula $Qx_1\varphi(x_1, \dots)$ 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 $L \in \text{PSPACE}$ 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 idéia 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

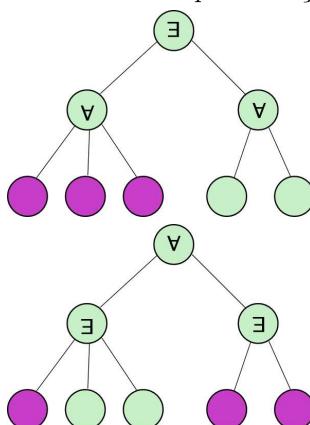
PALAVRA EM LINGUAGEM SENSÍVEL AO CONTEXTO

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

$$\forall(x, y_1, \dots, y_i) \in R \exists p \in \text{poly} \forall i |y_i| \leq p(|x|)$$

- **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

$$x \in L \iff \exists y_1 \forall y_2 \dots Q_i : (x, y_1, \dots, y_i) \in R$$

- **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

$$x \in L \iff \forall y_1 \exists y_2 \dots Q_i : (x, y_1, \dots, y_i) \in R$$

- 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 14.2 (Chandra, Kozen, Stockmeyer)

Para $t(n) \geq n$

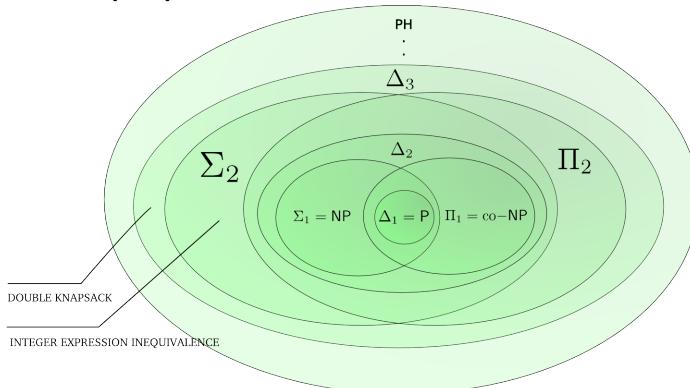
$$\text{ATIME}[t(n)] \subseteq \text{DSPACE}[t(n)] \subseteq \bigcup_{c>0} \text{ATIME}[ct(n)^2].$$

Corolário 14.1

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:

$$\exists p_1 \forall p_2 \exists p_3 \forall p_4 \dots \exists p_{2k+1} :$$

$p_1, p_2, p_3, \dots, p_{2k+1}$ é uma sequência de passos para ganhar.

- Portanto, vários jogos são PSPACE-completos: Generalized Hex, generalized Geography, ...

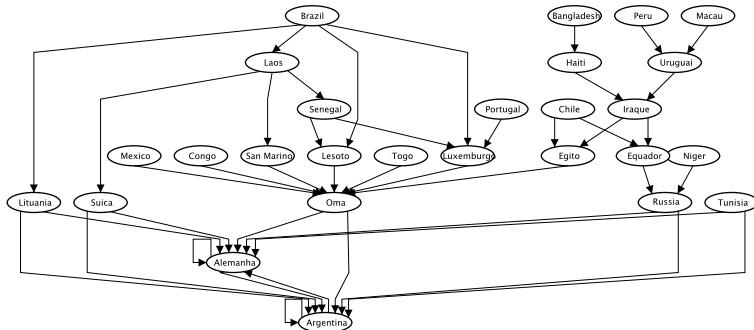
Mais exemplos da classe PSPACE (2)

Jogo de geografia para dois jogadores.

Em alternação 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)

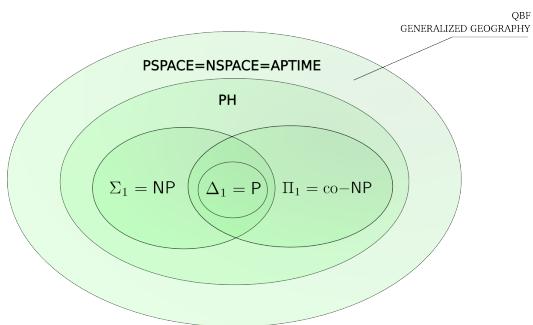
Instância Um grafo $G = (V, E)$ e um nó $v_0 \in V$

Decisão Jogando "geografia" com este grafo, o primeiro jogador pode ganhar com certeza?

Teorema 14.3

Geografia generalizada é PSPACE-completo.

O mundo até PSPACE



14.2. De PSPACE até ELEMENTAR

Problemas intratáveis demonstráveis

- Agora, consideramos os seguintes classes

$$\text{EXP} = \text{DTIME}[2^{n^{O(1)}}] = \bigcup_{k \geq 0} \text{DTIME}[2^{n^k}]$$

$$\text{NEXP} = \text{NTIME}[2^{n^{O(1)}}] = \bigcup_{k \geq 0} \text{NTIME}[2^{n^k}]$$

$$\text{EXPSPACE} = \text{DSPACE}[2^{n^{O(1)}}] = \bigcup_{k \geq 0} \text{DSPACE}[2^{n^k}]$$

- 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

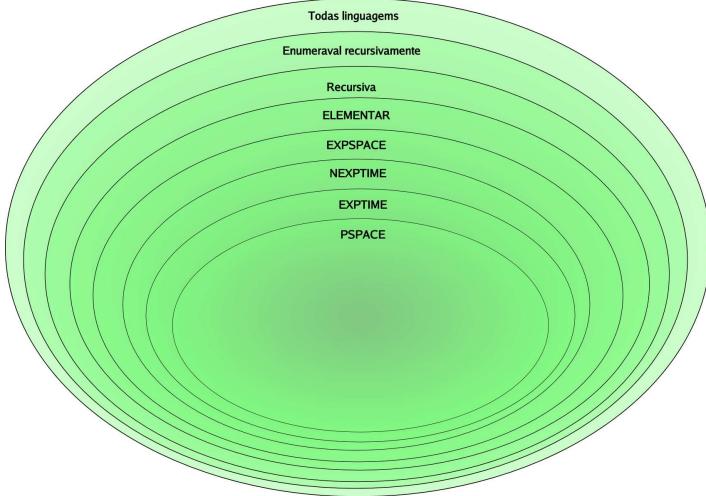
- As classes $k - \text{EXP}$, $k - \text{NEXP}$ e $k - \text{EXPSPACE}$ tem k níveis de exponenciação!
- Por exemplo, considere a torre de dois de altura três: $2^{2^2^k}$
- Problemas desse tipo são *bem* intratáveis

$$\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 \\ \hline 4 & 16 & 65536 & \approx 1.16 \times 10^{77} & \approx 2 \times 10^{19728} \end{array}$$

$$\text{ELEMENTAR} = \bigcup_{k \geq 0} k - \text{EXP}$$

- 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 $\lg n$.

14.3. Exercícios

Exercício 14.1

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 and Stockmeyer [2]). Booth [1] 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 [3] for example, shows that graph isomorphism is PSPACE-complete. [17]

Sabendo que GI pertence a NP isso implicaria $\text{PSPACE} = \text{NP}$. Acha o erro na prova.

15. Complexidade de circuitos

(As notas seguem Arora/Barak.)

Um modelo alternativo de computação são *circuitos booleanos*. Circuitos tem a vantagem de ser matematicamente 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 15.1 (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 *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

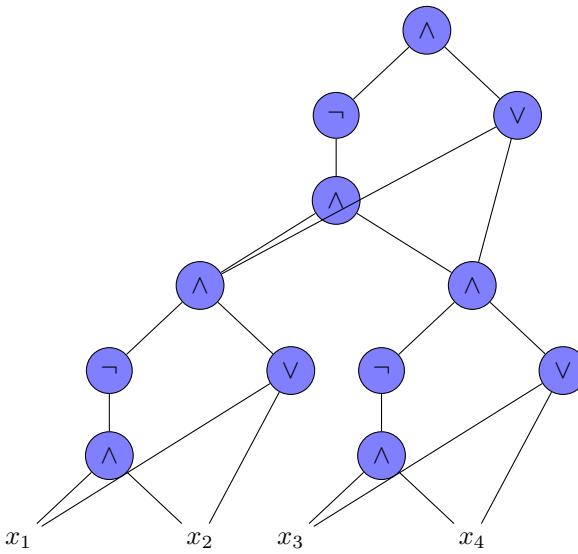
$$v(d) = \begin{cases} v(e_1) \wedge v(e_2) & \text{caso } d \text{ é rotulado com } \wedge \text{ e } (e_1, d), (e_2, d) \in E \\ v(e_1) \vee v(e_2) & \text{caso } d \text{ é rotulado com } \vee \text{ e } (e_1, d), (e_2, d) \in E \\ \neg v(e) & \text{caso } d \text{ é rotulado com } \neg \text{ e } (e, d) \in E \\ x_i & \text{case } d \text{ é a } i\text{-gésima entrada} \end{cases}$$

Observação 15.1

A definição permite um fan-out arbitrariamente grande. Um fan-in $k > 2$ pode ser implementado por uma cadeia de $k - 1$ portas com fan-in 2. \diamond

Exemplo 15.1

A função booleana simétrica $S_{1,3}(x_1, \dots, x_4)$ 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 15.2

Para uma função $t : \mathbb{N} \rightarrow \mathbb{N}$, 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 $\text{SIZE}(t(n))$ 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 15.1

Cada função $f : \{0, 1\}^n \rightarrow \{0, 1\}$ 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 $\text{SIZE}(2n2^n)$ 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 15.3 (Circuitos com um número polinomial de portas)

A classe de complexidade P/poly contém todas linguagens decidíveis por famílias de circuitos de tamanho polinomial, i.e.,

$$\text{P/poly} = \bigcup_{k>0} \text{SIZE}(n^k)$$

O seguinte lema estabelece que tudo que pode ser decidido por uma MT, pode ser decidido por uma família de circuitos booleanos de tamanho não mais que o quadrado do tempo de execução da MT.

Lema 15.1

Para uma função $t : \mathbb{N} \rightarrow \mathbb{N}$ tal que $L \in \text{DTIME}(t(n))$ temos $L \in \text{SIZE}(O(t(n)^2))$.

Prova. (Rascunho.) Seja $L \in \text{P}$ 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 celula 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 celula t_{ij} depende somente do conteúdo das celulas $t_{i-1,j-1}$, $t_{i-1,j}$ e $t_{i-1,j+1}$. Seja $s_{ijk} \in \{0,1\}$ um valor booleano que é verdadeira caso a celula $t_{i,j}$ 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 celula 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 celula 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 celula, e portanto $O(t(n)^2)$. ■

Corolário 15.1 $P \subseteq P/\text{poly}$

Prova. Caso $L \in P$, $L \in \text{DTIME}(n^k)$ para algum $k \in \mathbb{N}$. Pelo lema 15.1 temos $L \in \text{SIZE}(O(n^{2k}))$ e portanto $L \in P/\text{poly}$. ■

Existe uma separação entre P e P/poly ? Sim.

Proposição 15.2

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 idéia é só permitir as famílias de circuitos construtíveis:

Definição 15.4 (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 15.1

Uma linguagem L é decidível por uma família P -uniforme de circuitos sse $L \in P$.

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 15.1 observando que a construção é possível em tempo polinomial em n . ■

Máquinas de Turing com conselho Uma caracterização alternativa de P/poly é como MT *com conselho*.

Definição 15.5 (Máquina de Turing com conselho)

Para funções $T, a : \mathbb{N} \rightarrow \mathbb{N}$ a classe de complexidade $\text{DTIME}(t(n))/a(n)$ 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 15.2 (Caracterização alternativa de P/poly)

$$\text{P/poly} = \bigcup_{k, l > 0} \text{DTIME}(n^k)/n^l$$

Prova. (Rascunho.)

Caso $L \in \text{P/poly}$ 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 15.1 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 15.6 (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 polinomial, com fan-in ilimitado e profundidade $O(\log^d n)$. A classe AC é $\bigcup_{k > 0} \text{AC}^d$.

Lema 15.2 (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 atribuiçao 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 15.3

Seja $\oplus(x_1, \dots, x_n) = \sum_{1 \leq i \leq n} x_i \pmod{2}$ a função de paridade. $\oplus \notin \text{AC}^0$.

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 idéia 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 15.6 $d - 2$ 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 -ésima 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 -ésima restrição e o penúltimo nível contém disjunções. Os últimos dois nível representam fórmulas em k_i -DNF. Pelo lema 15.6 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 é $\leq 1/10n^b$, 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 cláusula, por exemplo), 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 repetir algumas definições básicas da matemática.

Definição A.1

Para um conjunto C o *fecho de Kleene* C^* denota o conjunto de todos os 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 $C_- := \{x \in C | x < 0\}$. Por exemplo

$$\mathbb{R}_+ = \{x \in \mathbb{R} | x > 0\}.$$
¹

Para um conjunto finito S , $\mathcal{P}(S)$ denota o conjunto de todos subconjuntos de S .

Definição A.2 (Valor absoluto)

O valor absoluto $|\cdot|$ 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)

$$\vdash x = |x| \tag{A.1}$$

$$x \leq |x| \tag{A.2}$$

$$|x + y| \leq |x| + |y| \quad \text{Desigualdade triangular} \tag{A.3}$$

$$|xy| = |x||y| \tag{A.4}$$

Prova. (i) Se $-x > 0$ temos $x < 0$, logo $\vdash x = -x$ e $|x| = -x$. Os casos restantes podem ser analisadas analogamente. (ii) Analise os casos. (iii) Para $x + y < 0$: $|x + y| = -(x + y) = (-x) + (-y) \leq \vdash x + \vdash y = |x| + |y|$. Para

¹Alguns autores usam \mathbb{R}^+ .

A. Conceitos matemáticos

$x + y \geq 0$: $|x + y| = x + y \leq |x| + |y|$. (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

$$\left| \sum_{1 \leq i \leq n} x_i \right| \leq \sum_{1 \leq i \leq n} |x_i| \quad (\text{A.5})$$

$$\left| \prod_{1 \leq i \leq n} x_i \right| = \prod_{1 \leq i \leq n} |x_i| \quad (\text{A.6})$$

(A.7)

Prova. Prova com indução sobre n . ■

Proposição A.2 (Regras para o máximo)

Para $a_i, b_i \in \mathbb{R}$

$$\max_i a_i + b_i \leq \max_i a_i + \max_i b_i \quad (\text{A.8})$$

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

$$\lfloor x \rfloor = \max\{y \in \mathbb{Z} | y \leq x\}$$

$$\lceil x \rceil = \min\{y \in \mathbb{Z} | y \geq x\}$$

O *parte fracionária* de x é $\{x\} = x - \lfloor x \rfloor$.

Observe que a parte fracionária sempre é positiva, por exemplo $\{-0.3\} = 0.7$.

Proposição A.3 (Regras para pisos e tetos)

Pisos e tetos satisfazem

$$x \leq \lceil x \rceil < x + 1 \quad (\text{A.9})$$

$$x - 1 < \lfloor x \rfloor \leq x \quad (\text{A.10})$$

Definição A.4

O *fatorial* é a função

$$n! : \mathbb{N} \rightarrow \mathbb{N} : n \mapsto \prod_{1 \leq i \leq n} i.$$

Temos a seguinte aproximação do fatorial (fórmula de Stirling)

$$n! = \sqrt{2\pi n} \left(\frac{n}{e}\right)^n (1 + O(1/n)) \quad (\text{A.11})$$

Uma estimativa menos preciso, pode ser obtido pelas observações

$$e^n = \sum_{i \geq 0} \frac{n^i}{i!} > \frac{n^n}{n!}$$

que combinado ficam

$$(n/e)^n \leq n! \leq n^n.$$

Revisão: Logaritmos

$$\log_a(1) = 0 \quad (\text{A.12})$$

$$a^{\log_a(n)} = n \quad \text{por definição} \quad (\text{A.13})$$

$$\log_a(n \cdot m) = \log_a(n) + \log_a(m) \quad \text{propriedade do produto} \quad (\text{A.14})$$

$$\log_a\left(\frac{n}{m}\right) = \log_a(n) - \log_a(m) \quad \text{propriedade da divisão} \quad (\text{A.15})$$

$$\log_a(n^m) = m \cdot \log_a(n) \quad \text{propriedade da potência} \quad (\text{A.16})$$

$$\log_a(n) = \log_b(n) \cdot \log_a(b) \quad \text{troca de base} \quad (\text{A.17})$$

$$\log_a(n) = \frac{\log_c(n)}{\log_c(a)} \quad \text{mudança de base} \quad (\text{A.18})$$

$$\log_b(a) = \frac{1}{\log_a(b)} \quad (\text{A.19})$$

$$a^{\log_c(b)} = b^{\log_c(a)} \quad \text{expoentes} \quad (\text{A.20})$$

Os números harmônicos

$$H_n = \sum_{1 \leq i \leq n} \frac{1}{i}$$

ocorrem freqüentemente na análise de algoritmos.

A. Conceitos matemáticos

Proposição A.4

$$\ln n < H_n < \ln n + 1.$$

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 $\int 1/x = \ln x$:

$$\begin{aligned}\ln(n) &\leq \ln(n+1) = \int_1^{n+1} \frac{1}{x} dx \\ &\quad \int_2^{n+1} \frac{1}{x-1} = \ln(n)\end{aligned}$$

■

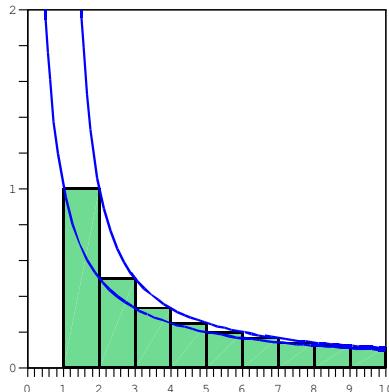


Figura A.1.: Cota inferior e superior dos números harmônicos.

Definição A.5

O *logaritmo iterado* é

$$\log^* n = \begin{cases} 0 & \text{se } n \leq 1 \\ 1 + \log^*(\log n) & \text{caso contrário} \end{cases}$$

O logaritmo iterado é uma função que cresce extremamente lento; para valores práticos de n , $\log^* n$ não ultrapassa 5.

A.2. Somatório

Revisão: Notação Somatório

Para k uma constante arbitrária temos

$$\sum_{i=1}^n k a_i = k \sum_{i=1}^n a_i \quad \text{Distributividade} \quad (\text{A.21})$$

$$\sum_{i=1}^n k = nk \quad (\text{A.22})$$

$$\sum_{i=1}^n \sum_{j=1}^m a_i b_j = \left(\sum_{i=1}^n a_i \right) \left(\sum_{j=1}^m b_j \right) \quad \text{Distributividade generalizada} \quad (\text{A.23})$$

$$\sum_{i=1}^n (a_i + b_i) = \sum_{i=1}^n a_i + \sum_{i=1}^n b_i \quad \text{Associatividade} \quad (\text{A.24})$$

$$\sum_{i=1}^p a_i + \sum_{i=p+1}^n a_i = \sum_{i=1}^n a_i \quad (\text{A.25})$$

$$\sum_{i=0}^n a_{p-i} = \sum_{i=p-n}^p a_i \quad (\text{A.26})$$

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

$$\sum_{0 \leq i \leq n} a_{p-i} = \sum_{0 \leq p-i \leq n} a_{p-(i-p)} = \sum_{p-n \leq i \leq p} a_i.$$

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 $matA$ e $matB$, faça um algoritmo que copie a matriz triangular inferior de $matB$ para $matA$.

Algoritmo A.1 (CopiaMTI)

Entrada Matrizes quadráticas A e B e dimensão n .

A. Conceitos matemáticos

Saída Matriz A com a matriz triangular inferior copiada de B .

```
1  for i := 1 to n do
2      for j := 1 to i do
3          Aij = Bij
4      end for
5  end for
```

Uma análise simples deste algoritmo seria:

$$\sum_{i=1}^n \sum_{j=1}^i 1 = \sum_{i=1}^n i = n(n+1)/2 = O(n^2)$$

Séries

$$\sum_{i=1}^n i = \frac{n(n+1)}{2} \quad \text{série aritmética} \quad (\text{A.27})$$

$$\sum_{i=0}^n x^i = \frac{x^{n+1} - 1}{x - 1} \quad \text{série geométrica, para } x \neq 1 \quad (\text{A.28})$$

se $|x| < 1$ então

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x} \quad \text{série geométrica infinitamente decrescente} \quad (\text{A.29})$$

Série geométrica com limites arbitrários:

$$\sum_{a \leq i \leq b} x^i = \frac{x^{b+1} - x^a}{x - 1} \quad \text{para } x \neq 1$$

Séries

$$\sum_{i=1}^n 2^i = 2^{n+1} - 2 \quad (\text{A.30})$$

$$\sum_{i=0}^n i^2 = \frac{n(n+1)(2n+1)}{6} \quad (\text{A.31})$$

$$\sum_{i=0}^n i2^i = 2 + (n-1)2^{n+1} \quad (\text{A.32})$$

Mais geral para alguma sequência f_i temos

$$\begin{aligned} \sum_{1 \leq i \leq n} if_i &= \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq i} f_i = \sum_{1 \leq j \leq i \leq n} f_i = \sum_{1 \leq j \leq n} \sum_{j \leq i \leq n} f_i \\ &= \sum_{1 \leq j \leq n} \left(\sum_{1 \leq i \leq n} f_i - \sum_{1 \leq i < j} f_i \right). \end{aligned}$$

Uma aplicação:

$$\begin{aligned} \sum_{1 \leq i \leq n} ix^i &= \sum_{1 \leq j \leq n} \left(\sum_{1 \leq i \leq n} x^i - \sum_{1 \leq i < j} x^i \right) = \sum_{1 \leq j \leq n} \left(\frac{x^{n+1} - x^1}{x - 1} - \frac{x^j - x^1}{x - 1} \right) \\ &= \frac{1}{x - 1} \sum_{1 \leq j \leq n} (x^{n+1} - x^j) \\ &= \frac{1}{x - 1} \left(nx^{n+1} - \frac{x^{n+1} - x^1}{x - 1} \right) = \frac{x}{(x - 1)^2} (x^n(nx - n - 1) + 1) \end{aligned}$$

e com $x = 1/2$ temos

$$\sum_{1 \leq i \leq n} i2^{-i} = 2(2(2^{-1} - 2^{-n-1}) - n2^{-n-1}) = 2((1 - 2^{-n}) - n2^{-n-1}) = 2 - 2^{-n}(n+2) \quad (\text{A.33})$$

A.3. Indução

Revisão: Indução matemática

- Importante para provar resultados envolvendo inteiros.

A. Conceitos matemáticos

- 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

$$\sum_{i=1}^n 2i - 1$$

e prove seu resultado via indução matemática.

- Use indução matemática para provar que

$$\sum_{i=0}^{n-1} q^i = \frac{q^n - 1}{q - 1}.$$

- Resolva os exercícios do capítulo 1.

A.4. Limites

Definição A.6 (Limites)

Para $f : \mathbb{N} \rightarrow \mathbb{R}$ o limite de n para ∞ é definido por

$$\lim_{n \rightarrow \infty} f(n) = c \iff \exists c \forall \epsilon > 0 \exists n_0 \forall n > n_0 |f(n) - c| < \epsilon. \quad (\text{A.34})$$

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,

$$\lim_{n \rightarrow \infty} f(n) = \infty \iff \forall c \exists n_0 \forall n > n_0 f(n) > c \quad (\text{A.35})$$

Também temos

$$\begin{aligned}\liminf_{n \rightarrow \infty} f(n) &= \lim_{n \rightarrow \infty} \left(\inf_{m \geq n} f(m) \right) \\ \limsup_{n \rightarrow \infty} f(n) &= \lim_{n \rightarrow \infty} \left(\sup_{m \geq n} f(m) \right)\end{aligned}$$

Lema A.1 (Definição alternativa do limite)

É possível substituir $<$ com \leq na definição do limite.

$$\lim_{n \rightarrow \infty} f(n) = c \iff \forall \epsilon > 0 \exists n_0 \forall n > n_0 |f(n) - c| \leq \epsilon$$

Prova. \Rightarrow é obvio. Para \Leftarrow , escolhe $\epsilon' = \epsilon/2 < \epsilon$. ■

A.5. Probabilidade discreta

Probabilidade: Noções básicas

- *Espaço amostral* finito Ω de *eventos elementares* $e \in \Omega$.
- Distribuição de probabilidade $\Pr[e]$ tal que

$$\Pr[e] \geq 0; \quad \sum_{e \in \Omega} \Pr[e] = 1$$

- *Eventos* (compostos) $E \subseteq \Omega$ com probabilidade

$$\Pr[E] = \sum_{e \in E} \Pr[e]$$

A. Conceitos matemáticos

Exemplo A.1

Para um dado sem bias temos $\Omega = \{1, 2, 3, 4, 5, 6\}$ e $\Pr[i] = 1/6$. O evento Par = {2, 4, 6} tem probabilidade $\Pr[\text{Par}] = \sum_{e \in \text{Par}} \Pr[e] = 1/2$. \diamond

Probabilidade: Noções básicas

- Variável aleatória

$$X : \Omega \rightarrow \mathbb{N}$$

- Escrevemos $\Pr[X = i]$ para $\Pr[X^{-1}(i)]$.
- Variáveis aleatórias *independentes*

$$P[X = x \text{ e } Y = y] = P[X = x]P[Y = y]$$

- Valor esperado

$$E[X] = \sum_{e \in \Omega} \Pr[e]X(e) = \sum_{i \geq 0} i \Pr[X = i]$$

- Linearidade do valor esperado: Para variáveis aleatórias X, Y

$$E[X + Y] = E[X] + E[Y]$$

Prova. (Das formulas equivalentes para o valor esperado.)

$$\begin{aligned} \sum_{0 \leq i} \Pr[X = i]i &= \sum_{0 \leq i} \Pr[X^{-1}(i)]i \\ &= \sum_{0 \leq i} \sum_{e \in X^{-1}(i)} \Pr[e]X(e) = \sum_{e \in \Omega} \Pr[e]X(e) \end{aligned}$$

Prova. (Da linearidade.)

$$\begin{aligned} E[X + Y] &= \sum_{e \in \Omega} \Pr[e](X(e) + Y(e)) \\ &= \sum_{e \in \Omega} \Pr[e]X(e) \sum_{e \in \Omega} \Pr[e]Y(e)) = E[X] + E[Y] \end{aligned}$$

Exemplo A.2

(Continuando exemplo A.1.)

Seja X a variável aleatório que denota o número sorteado, e Y a variável aleatória tal que $Y = [a \quad \text{face em cima do dado tem um ponto no meio}]$.

$$E[X] = \sum_{i \geq 0} \Pr[X = i]i = 1/6 \sum_{1 \leq i \leq 6} i = 21/6 = 7/2$$

$$E[Y] = \sum_{i \geq 0} \Pr[Y = i]i = \Pr[Y = 1] = 1/2E[X + Y] = E[X] + E[Y] = 4$$

◊

A.6. 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 $C = (\{v_0, \dots, v_k\}, \{\{v_i, v_{i+1}\} \mid 0 \leq i < k\})$ 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:

“ \Rightarrow ”: 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 $\forall n > n_0 \sup_{m \geq n} f(m)/g(m) \leq c$.

“ \Leftarrow ”: Seja $\limsup_{n \rightarrow \infty} f(n)/g(n) < \infty$. 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:

“ \Rightarrow ”: 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 $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ (veja lema A.1).

“ \Leftarrow ”: Seja $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$, 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:

“ \Rightarrow ”: 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. $i(n_0 + 1) > 0$.

“ \Leftarrow ”: Suponha $\liminf_{n \rightarrow \infty} f(n)/g(n) = l > 0$. 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 $f \in \omega(g) \subseteq \Omega(g)$.

B. Soluções dos exercícios

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

$$g'(n) = \begin{cases} h(n)/f(n) & \text{se } f(n) \neq 0 \\ cg(n) & \text{caso contrário} \end{cases}$$

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.

1. Temos as equivalências

$$\begin{aligned}
 f \preceq g &\iff f \in O(g) \\
 &\iff \exists c \exists n_0 \forall n > n_0 f \leq cg \\
 &\iff \exists c' \exists n_0 \forall n > n_0 g \geq c'f \quad \text{com } c' = 1/c \\
 &\iff g \in \Omega(f)
 \end{aligned}$$

2. 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 .

3. Temos as equivalências

$$\begin{aligned}
 f \prec g &\iff f \in o(g) \\
 &\iff \forall c \exists n_0 \forall n > n_0 f \leq cg \\
 &\iff \forall c' \exists n_0 \forall n > n_0 g \geq c'f \quad \text{com } c' = 1/c \\
 &\iff g \in \omega(f)
 \end{aligned}$$

4. O argumento é essencialmente o mesmo que no item (a).

5. 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 $g \in \Omega(f)$.

B. Soluções dos exercícios

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 $h = f'g' \geq c_f c_g f g$.

Prova de B.6: $h \geq c f g$. Escrevendo $h = f g'$ temos que mostrar $g' \in \Omega(g)$. Mas $g' = h/f \geq c f g/f = c g$. ■

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.

1. Para $n \geq 2$ temos $\log 1 + n \leq \log 2n = \log 2 + \log n \leq 2 \log n$.
2. 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 $n > \max(c, n_0)$.
3. 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 [66, p. 2.3].

1. $3n + 7 \leq 5n + 2 \iff 5 \leq 2n \iff 2.5 \leq n$ (equação linear)
2. $5n + 7 \leq 3n^2 + 1 \iff 0 \leq 3n^2 - 5n - 6 \iff 5/6 + \sqrt{97}/6 \leq n$ (equação quadrática)
3. $5 \log_2 n + 7 \leq 5n + 1 \iff 7^5 + 2^7 - 2 \leq 2^{5n} \iff 16933 \leq 2^{5n} \iff 2.809\dots \leq n$
4. Veja item (b)
5. $52^n + 3 \geq 3n^2 + 5n \iff n \geq 2^n \geq (3n^2 + 5n - 3)/5 \iff 2^n \geq n^2$.

B. Soluções dos exercícios

$$6. n^2 3^n \geq n^3 2^n + 1 \Leftrightarrow n^2 3^n \geq n^3 2^{n+1} \Leftrightarrow 2 \log_2 n + n \log_2 3 \geq 3 \log_2 n + (n+1) \log_2 2 \Leftrightarrow n \log_2 3 \geq \log_2 n + (n+1) \Leftrightarrow n(\log_2 3 - 1)/2 \geq \log_2 n$$

Solução do exercício [66, p. 2.9].

Com $f \in \Theta(n^r)$ e $g \in \Theta(n^s)$ temos

$$c_1 n^r \leq f \leq c_2 n^r; \quad d_1 n^s \leq g \leq d_2 n^s \quad \text{a partir de um } n_0$$

(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.

Temos que mostrar que $\log n \leq cn^\epsilon$ para constantes c, n_0 a partir de um $n \geq n_0$. Temos

$$\log n \leq cn^\epsilon \Leftrightarrow \log n \log n \leq \log c + \epsilon \log n \Leftrightarrow \log m \leq \epsilon m + c'$$

com a substituição $m = \log n$ e $c' = \log c$. Nos vamos mostrar que $\log m \leq m\epsilon$ para qualquer ϵ a partir de $m \geq m_0$ que é equivalente com $\lim_{m \rightarrow \infty} \log m/m = 0$. Pela regra de L'Hospital

$$\lim_{m \rightarrow \infty} \frac{\log m}{m} = \lim_{m \rightarrow \infty} \frac{1}{m} = 0$$

Solução do exercício 2.1.

$$C_p[\text{Alg1}] = \sum_{i=1}^n \sum_{j=1}^{2^{i-1}} c = \frac{c}{2} \cdot \sum_{i=1}^n 2^i = c \cdot 2^n - c = O(2^n)$$

Os detalhes da resolução do algoritmo abaixo foram suprimidos. Resolva com detalhes e confira se a complexidade final corresponde à encontrada na análise

abaixo.

$$\begin{aligned}
 C_p[\text{Alg2}] &= \sum_{1 \leq i \leq n} \sum_{\substack{1 \leq j \leq 2^i \\ j \text{ ímpar}}} j^2 \leq \sum_{1 \leq i \leq n} \sum_{1 \leq j \leq 2^i} j^2 \\
 &= O\left(\sum_{1 \leq i \leq n} (2^i)^3\right) \\
 &= O\left(\sum_{1 \leq i \leq n} 8^i\right) = \frac{8^{n+1} - 8}{7} \leq 8^{n+1} = O(8^n)
 \end{aligned}$$

$$\begin{aligned}
 C_p[\text{Alg3}] &= \sum_{i=1}^n \sum_{j=i}^n 2^i = \sum_{i=1}^n 2^i \cdot (n - i + 1) \\
 &= \sum_{i=1}^n (n2^i - i2^i + 2^i) = \sum_{i=1}^n n \cdot 2^i - \sum_{i=1}^n i \cdot 2^i + \sum_{i=1}^n 2^i \\
 &= n \cdot (2^{n+1} - 2) - (2 + (n-1) \cdot 2^{n+1}) + (2^{n+1} - 2) \\
 &= n2^{n+1} - 2n - 2 - n2^{n+1} + 2^{n+1} + 2^{n+1} - 2 \\
 &= 2^{n+2} - 2n - 4 = O(2^n)
 \end{aligned}$$

$$\begin{aligned}
 C_p[\text{Alg4}] &= \sum_{i=1}^n \sum_{j=1}^i 2^j = \sum_{i=1}^n (2^{i+1} - 2) \\
 &= 2 \sum_{i=1}^n 2^i - \sum_{i=1}^n 2 = 2 \cdot (2^{n+1} - 2) - 2n \\
 &= 4 \cdot 2^n - 4 - 2n = O(2^n)
 \end{aligned}$$

$$\begin{aligned}
 C_p[\text{Alg5}] &= \sum_{i=1}^n \sum_{j=i}^n 2^j = \sum_{i=1}^n \left(\sum_{j=1}^n 2^j - \sum_{j=1}^{i-1} 2^j \right) \\
 &= \sum_{i=1}^n (2^{n+1} - 2 - (2^{i-1+1} - 2)) = \sum_{i=1}^n (2 \cdot 2^n - 2 - 2^i + 2) \\
 &= 2 \sum_{i=1}^n 2^n - \sum_{i=1}^n 2^i = 2 \cdot n2^n - (2^{n+1} - 2) \\
 &= 2 \cdot n2^n - 2 \cdot 2^n + 2 = O(n2^n)
 \end{aligned}$$

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

$$\begin{aligned} T_n &= n - 1 + 2/n \sum_{0 \leq i < n} O(i) \\ &\leq n - 1 + 2c/n \sum_{0 \leq i < n} i \\ &= n - 1 + c(n - 1) = cn + (n - 1 - c) \\ &\not\leq cn \quad \text{Não funciona para } n > c + 1 \end{aligned}$$

Solução do exercício 2.3.

Uma solução simples é manter um máximo M e o segundo maior elemento m no mesmo tempo:

```

1  M := ∞
2  m := ∞
3  for i = 1, ..., n do
4      if ai > M then
5          m := M
6          M := ai
7      else if ai > m do
8          m := ai
9      end if
10 end for
11 return m

```

O número de comparações é ao máximo dois por iteração, e esse limite ocorre numa sequência crescente 1, 2, ..., 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

```

1  m := ∑1 ≤ i ≤ n ai
2  for C ⊆ [1, n] do
3      m' := | ∑i ∈ C ai - ∑i ∉ C ai |

```

```

4   if  $m' < m$  then
5        $m := m'$ 
6   end if
7 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, \dots, 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

$$E[A] = E\left[\sum_{1 \leq i \leq n} A_i\right] = n/6.$$

Com o mesmo argumento a segunda distribuição leva a $E[A_i] = 1/10$ e $E[A] = n/10$ finalmente.

Solução do exercício 2.7.

Cada chave em nível $i \in [1, k]$ precisa i comparações e a árvore tem $\sum_{1 \leq i \leq k} 2^{i-1} = 2^k - 1$ nós e folhas em total. Para o número de comparações C temos

$$E[C] = \sum_{1 \leq i \leq k} P[C = i]i = \sum_{1 \leq i \leq k} \frac{2^{i-1}}{2^{k-1}}i = 2^{-k} \sum_{1 \leq i \leq k} 2^i i = 2(k-1) + 2^{1-k}.$$

Solução do exercício 4.1.

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)

```

1   if  $m > n$  then
2       return false
3   end if
4    $i := 1$ 

```

```

5      for  $j := 1, \dots, n$  do
6          if  $s'_i = s_j$  then
7               $i := i + 1$ 
8              if  $i > m$  then
9                  return true
10             end if
11         end if
12     end for
13     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 4.2.

O seguinte algoritmo resolve o problema:

Algoritmo B.2 (Bases)

Entrada Uma sequência de posições x_i de n cidades, $1 \leq i \leq n$.

Saída Uma sequência mínima de posições b_i de bases.

```

1      Sejam  $S = x'_1 \dots x'_n$  as posições em ordem crescente
2       $B = \epsilon$ 
3      while  $S \neq \emptyset$  do
4          Seja  $S = x' S'$ 
5           $B := B, (x' + 4)$  { aumenta a sequência B }
6          Remove todos os elementos  $x \leq x' + 8$  de  $S$ 
7      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 6.1.

- $T(n) = 9T(n/3) + n$

$$\begin{aligned} T(n) &= \sum_{0 \leq i < \log_3 n} 9^i (n/3^i) + \Theta(9^{\log_3 n}) \\ &= n \sum_{0 \leq i < \log_3 n} 3^i + n^2 \\ &= n \frac{3^{\log_3 n} - 1}{2} + n^2 = \Theta(n^2) \end{aligned}$$

- $T(n) = 2T(n/2) + n \log n$

$$\begin{aligned} T(n) &= \sum_{0 \leq i < \log n} 2^i (n/2^i) \log_2 n/2^i + \Theta(2^{\log_2 n}) \\ &= n \sum_{0 \leq i < \log n} \log_2 n - i + \Theta(n) \\ &= n \log_2^2 n - \frac{n \log_2 n (\log_2 n - 1)}{2} + \Theta(n) \\ &= O(n \log_2^2 n) \end{aligned}$$

Solução do exercício 6.3.

- Produto de dois números binários (exemplo 5.3.8 em [66]).

Algoritmo B.3 (mult-bin)

Entrada Dois números binários p, q com n bits.

Saída O produto $r = pq$ (que tem $\leq 2n$ bits).

```

1  if n = 1 then
2    return pq      { multiplica dois bits em O(1) }
3  else

```

B. Soluções dos exercícios

```

4       $x := p_1 + p_2$ 
5       $y := q_1 + q_2$ 
6       $z := \text{MULT-BIN}(x_2, y_2)$ 
7       $t := (x_1 y_1) 2^n + (x_1 y_2 + x_2 y_1) 2^{n/2} + z$ 
8       $u := \text{MULT-BIN}(p_1, q_1)$ 
9       $v := \text{MULT-BIN}(p_2, q_2)$ 
10      $r := u 2^n + (t - u - v) 2^{n/2} + v$ 
11     return  $r$ 
12 end if

```

É importante de observar que x é a soma de dois números com $n/2$ bits e logo tem no máximo $n/2 + 1$ bits. A divisão de x em x_1 e x_2 é tal que x_1 represente o bit $n/2 + 1$ e x_2 o resto.

$$\begin{aligned}
p &= \underbrace{| \quad p_1 \quad |}_{n/2\text{bits}} \underbrace{| \quad p_2 \quad |}_{n/2\text{bits}} \\
x &= \underbrace{| \quad p_1 \quad |}_{n/2\text{bits}} \\
&\quad + \underbrace{| \quad p_2 \quad |}_{n/2\text{bits}} \\
&= \underbrace{| x_1 |}_{1\text{bit}} \underbrace{| x_2 |}_{n/2\text{bits}}
\end{aligned}$$

(y tem a mesma subdivisão.)

Corretude do algoritmo Temos a representação $p = p_1 2^{n/2} + p_2$ e $q = q_1 2^{n/2} + q_2$ e logo obtemos o produto

$$pq = (p_1 2^{n/2} + p_2)(q_1 2^{n/2} + q_2) = p_1 q_1 2^n + (p_1 q_2 + p_2 q_1) 2^{n/2} + p_2 q_2 \quad (\text{B.7})$$

Usando $t = (p_1 + p_2)(q_1 + q_2) = p_1 q_1 + p_1 q_2 + p_2 q_1 + p_2 q_2$ obtemos

$$p_1 q_2 + p_2 q_1 = t - p_1 q_1 - p_2 q_2. \quad (\text{B.8})$$

A linha 7 do algoritmo calcula t (usando uma chamada recursiva com $n/2$ bits para obter $z = x_2y_2$). Com os produtos $u = p_1q_1$, $v = p_2q_2$ (que foram obtidos com duas chamadas recursivas com $n/2$ bits) temos

$$\begin{aligned} p_1q_2 + p_2q_1 &= t - u - v \\ pq &= u2^n + (t - u - v)2^{n/2} + v \end{aligned}$$

substituindo u e v nas equações B.7 e B.8.

Complexidade do algoritmo Inicialmente provaremos pelo método da substituição que a recorrência deste algoritmo é $O(n^{\log_2 3})$. Se usarmos a hipótese de que $T(n) \leq cn^{\log_2 3}$, 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: $T(n) \leq cn^{\log_2 3} - dn$

$$\begin{aligned} T(n) &\leq 3(c(n/2)^{\log_2 3} - d(n/2)) + bn \\ &\leq 3cn^{\log_2 3}/(2^{\log_2 3}) - 3d(n/2) + bn \\ &\leq cn^{\log_2 3} - 3d(n/2) + bn \\ &\leq cn^{\log_2 3} - dn \end{aligned}$$

A desigualdade acima é verdadeira para $-3d(n/2) + bn \leq -dn$, ou seja, para $d \leq -2b$: $T(n) \in O(n^{\log_2 3} - dn) \in O(n^{\log_2 3})$.

Com a hipótese que uma *multiplicação com 2^k precisa tempo constante* (shift left), a linha 7 do algoritmo também precisa tempo constante, porque uma multiplicação com x_1 ou y_1 precisa tempo constante (de fato é um if). O custo das adições é $O(n)$ e temos a recorrência

$$T_n = \begin{cases} 1 & \text{se } n = 1 \\ 3T(n/2) + cn & \text{se } n > 1 \end{cases}$$

cuja solução é $\Theta(n^{1.58})$ (com $1.58 \approx \log_2 3$, aplica o teorema Master).

Exemplo B.1

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 $z = x_2y_2 = 0110.0011$, $t = y_22^4 + z = 21.0001.0011$, $u = p_1q_1 = 0111.0101$, $v = p_2q_2 = 0001.1000$ e finalmente $r = 1111.1010.111.1000 = 32120$. \diamond

B. Soluções dos exercícios

O algoritmo acima não é limitado para números binários, ele pode ser aplicado para números com base arbitrário. Ele é conhecido como algoritmo de Karatsuba [42]. Um algoritmo mais eficiente é do Schönhage e Strassen [59] que multiplica em $O(n \log n \log \log n)$. Fürer [27] apresenta um algoritmo que multiplica em $n \log n 2^{O(\log^* n)}$, um pouco acima do limite inferior $\Omega(n \log n)$.

2. Algoritmo de Strassen para multiplicação de matrizes.

O algoritmo está descrito na seção 6.3. 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

$$\begin{aligned} T(n) &\leq \sum_{0 \leq i \leq h} cn^2(7/4)^i + 7^{h+1} \\ &= (4/3)cn^2((7/4)^{h+1} - 1) + 7^{h+1} \quad \text{com } 4^{h+1} \geq 4n^2 \\ &\leq (7c/3 + 1)7^h - (4/3)cn^2 \quad \text{com } 7^h \leq 77^{\log_2 n} \\ &\leq (49c/3 + 1)n^{\log_2 7} = O(n^{\log_2 7}). \end{aligned}$$

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 método Master, 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 $\Theta(n^{\log_2 7})$.

3. Algoritmo de seleção do k -ésimo elemento.

Esse 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 subproblemas tem tamanho diferente (portanto o método Master 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 $h = \lceil \log_{c_2} 20/n \rceil$. Portanto, obtemos uma cota superior para o custo da árvore

$$\begin{aligned} T(n) &\leq n \sum_{0 \leq i \leq h} c^i + F(n) \\ &\leq n \sum_{0 \leq i < \infty} c^i + F(n) \quad \text{porque } c < 1 = 10n/(1 - \epsilon) + F(n) \end{aligned}$$

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 método Master não se aplica nesta recorrência.

Índice

- DSPACE, 194
DTIME, 194
NP, 202
NSPACE, 194, 195
NTIME, 194, 195
 Ω (Notação), 22
PSPACE, 195
 Π_n , 219
 P , 195, 202
 Σ_n , 219
 Θ (Notação), 22
 \asymp (relação de crescimento), 26
FP, 202
PF, 202
 ω (Notação), 22
 \prec (relação de crescimento), 26
 \preceq (relação de crescimento), 26
 \succ (relação de crescimento), 26
 \succeq (relação de crescimento), 26
árvore
 binária, 117
 de busca, 117
 espalhada mínima, 79
árvore binário, 92
APX, 163
NPO, 161
PO, 161
BHALT, 204

ABB-ÓTIMA (algoritmo), 121
absorção (de uma função), 26
adjacência
 de vértices, 237

AEM-Kruskal (algoritmo), 82
AEM-Prim (algoritmo), 82, 83
alfabeto, 185
algoritmo
 de aproximação, 157
 de Karatsuba, 249
 guloso, 75
 PrefixTree, 92
 randomizado, 179
algoritmo ϵ -aproximativo, 163
algoritmo r -aproximativo, 163
algoritmo de Grover, 225
algoritmo de Hirschberg, 104
algoritmo de Shor, 225
all pairs shortest paths, 84
aproximação
 absoluta, 162
 relativa, 163
aresta, 237
atribuição, 38, 41
aval (função de complexidade), 34

backtracking, 143
Bayes, 237
bottom-up, 99
Bubblesort (algoritmo), 43, 55
Busca binária (algoritmo), 48
Busca em Largura (algoritmo), 49
busca linear, 225
Busca seqüencial (algoritmo), 46, 53
Busca1 (algoritmo), 36, 53
código

- livre de prefixos, 91
cache, 99
caixeiro viajante, 116
caminho, 237
Caminho Hamiltoniano, 51
caminho mais curto
 entre todas pares, 84
 entre um nó e todos outros, 84
certificado, 202
ciclo, 237
 euleriano, 18
 hamiltoniano, 18
classe de complexidade, 194
cobertura por vértices, 157
coloração mínima, 90
complexidade
 média, 35, 50
 otimista, 37
 pessimista, 35
componente
 conjuntiva, 38, 39
 disjuntiva, 38, 42
composicionalidade, 38
condicional, 38, 42
conjunto compatível de intervalos,
 85
conjunto independente, 87
 máximo (problema), 87
Cook, Stephen Arthur, 211
CopiaMTI (algoritmo), 232
corte, 80
cota assintótica superior, 20
Counting-Sort (algoritmo), 47
custo (função de custos), 34

desemp (função de desempenho), 34
Dijkstra, Edsger, 85
distância de Levenshtein, 105
distribuição, 236
divisão e conquista, 58, 123

Eliminação de Gauss (algoritmo),
 11
espaço amostral, 236
espaço-construtível, 194
Euler, Leonhard, 18
evento, 236
 elementar, 236
exec (função de execução), 34

fórmula de Stirling, 229
fatoração, 225
fatorial, 229
Flajolet, Philippe, 52
Floyd, Robert W, 114
Floyd-Warshall (algoritmo), 114
função
 de complexidade (aval), 34
 de custos (custo), 34
 de desempenho (desemp), 34
 de execução (exec), 34
função de otimização, 161
função objetivo, 161

grafo, 49, 237
 k-partido, 150
 bipartido, 150
 conexo, 18, 79
 de intervalo, 88
 direcionado, 238
 não-direcionado, 18
 perfeito, 150
Grover, Lev, 225

Hamilton, Sir William Rowan, 18
hierarquia polinomial, 219
Hirschberg, algoritmo de, 104
Hoare, Charles Anthony Richard,
 58
Huffman (algoritmo), 94

independent set, 87

- indução natural, 234
 informação
 quantidade de, 93
 inversão, 56
 tabela de, 57
 iteração
 definida, 38, 41
 indefinida, 38, 41
- Karatsuba, Anatolii Alekseevitch, 249
- Kruskal, Joseph Bernard, 81
- Levensthein, Vladimir Iosifovich, 105
- Levin, Leonid, 211
- linearidade do valor esperado, 237
- linguagem, 185
- logaritmo, 229
- Loteria Esportiva (algoritmo), 47
- máquina de RAM, 33
- máquina de Turing, 187
 determinística, 189
 não-determinística, 189
- Máximo (algoritmo), 45, 57
- método
 da substituição, 125, 126
 de árvore de recursão, 125, 131
 mestre, 125, 133
- maximum independent set (problema), 87
- maximum Knapsack, 109
- memoização, 99
- Mergesort, 17
 recorrência, 125
- mochila máxima, 109
- Multiplicação de matrizes, 17, 50, 112
 algoritmo de Coppersmith-Winograd, 50
 algoritmo de Strassen, 50, 139
- multiplicação de números (algoritmo), 249
- número cromático, 150
- número de clique, 150
- número Fibonacci, 97
- números harmônicos, 230
- notação assintótica
 Ω , 22
 Θ , 22
 ω , 22
 O, 19
 o, 22
- O (notação), 19
- o (Notação), 22
- ordenação
 Bubblesort, 43
 por inserção direta (algoritmo), 44, 54
 Quicksort, 58
- palavra, 185
- Parada não-determinístico em k passos, 51
- particionamento
 de intervalos, 88–90
 de um vetor, 58
- Partition (algoritmo), 58
- PD-matriz, 110
- potenciação, 134
- PrefixTree (algoritmo), 92
- Prim, Robert C., 81
- probabilidade, 236
- probabilidade condicional, 237
- problema
 completo, 203
 de avaliação, 161
 de construção, 161
 de decisão, 161
 difícil, 203

Índice

- problema de otimização, 161
programação dinâmica, 97, 100
- quantidade de informação, 93
Quicksort (algoritmo), 58, 60
- recorrência
 simplificar, 125
- redução, 203
- relação
 polinomialmente limitada, 161,
 202
- relação de crescimento, 25
 \asymp , 26
 \prec , 26
 \preceq , 26
 \succ , 26
 \succeq , 26
- retrocedimento, 143
- série aritmética, 232
série geométrica, 232
- Savitch, Walter J., 198
- seqüênciaria, 38, 39
seqüenciamento
 de intervalos (algoritmo), 86
 de intervalos (problema), 85
- Shannon, Claude, 93
- Shor, Peter, 225
- single-source shortest paths, 84
- somatório, 231
- straight insertion sort (algoritmo),
 44, 54
- Strassen, Volker, 139
- subestrutura ótima, 77
- subgrafo, 238
 induzido, 238
- subseqüênciaria, 101
- subseqüênciaria comum mais longa, 101
- tabela de inversões, 57
- tam (tamanho de entradas), 34
tempo-construtível, 194
teorema de Bayes, 237
teorema de Savitch, 198
tese de Cobham-Edmonds, 8
top-down, 99
transposição, 56
traveling salesman, 116
troca mínima (algoritmo), 76
Turing, Alan Mathison, 186
- vértice, 237
valor esperado, 237
variável aleatória, 237
vertex cover, 157
- Vinogradov, I. M., 26
 notação de, 26
- Vitter, Jeffrey Scott, 52
- vizinhança, 237
- Warshall, Stephen, 114

Bibliografia

- [1] Scott Aaronson. “NP-complete problems and physical reality”. Em: *ACM SIGACT News* (mar. de 2005).
- [2] G. Adelson-Velskii e E. M. Landis. “An algorithm for the organization of information (in Russian)”. Em: *Proceedings of the USSR Academy of Sciences*. Vol. 146. 1962, pp. 263–266.
- [3] Manindra Agrawal, Neeraj Kayal e Nitin Saxena. “PRIMES is in P”. Em: *Annals of Mathematics* 160 (2004), pp. 781–793.
- [4] Mohamad Akra e Louay Bazzi. “On the Solution of Linear Recurrence Equations”. Em: *Computational Optimization and Applications* 10 (1998), pp. 195–210.
- [5] Noga Alon et al. “Witnesses for Boolean Matrix Multiplication and for Shortest Paths”. Em: *FOCS*. 1992.
- [6] H. Alt et al. “Computing a maximum cardinality matching in a bipartite graph in time $O(n^{1.5}\sqrt{m \log n})$ ”. Em: *Information Processing Letters* 37 (1991), pp. 237–240.
- [7] Sanjeev Arora e Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009.
- [8] Mikhail J. Atallah, ed. *Algorithms and theory of computation handbook*. CRC Press, 1999.
- [9] Richard Bellman. “Dynamic Programming Treatment of the Travelling Salesman Problem”. Em: *J. ACM* 9.1 (1962), pp. 61–63.
- [10] Claude Berge. “Two theorems in graph theory”. Em: *Proc. National Acad. Science* 43 (1957), pp. 842–844.
- [11] Timothy M. Chan. “More Algorithms for All-Pairs Shortest Paths in Weighted Graphs”. Em: *STOC'07*. 2007.
- [12] *Complexity zoo*. Online.
- [13] Don Coppersmith e Shmuel Winograd. “Matrix multiplication via arithmetic progressions”. Em: *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*. 1987, pp. 1–6.
- [14] Thomas H. Cormen et al. *Introduction to Algorithms*. 2nd. The MIT Press, 2001.

- [15] L.J. Cowen, Robert Cowen e Arthur Steinberg. “Totally Greedy Coin Sets and Greedy Obstructions”. Em: *The Electronic Journal of Combinatorics* 15 (2008).
- [16] Brian C. Dean, Michel X. Goemans e Nicole Immorlica. “Finite termination of ”augmenting path”algorithms in the presence of irrational problem data”. Em: *ESA '06: Proceedings of the 14th conference on Annual European Symposium*. Zurich, Switzerland: Springer-Verlag, 2006, pp. 268–279. DOI: http://dx.doi.org/10.1007/11841036_26.
- [17] Matthew Delacorte. *Graph Isomorphism is PSPACE-complete*. arXiv:0708.4075. 2007.
- [18] Reinhard Diestel. *Graph theory*. 3rd. Springer, 2005.
- [19] Ding-Zhu Du e Ker-I Ko, eds. *Advances in Algorithms, Languages, and Complexity - In Honor of Ronald V. Book*. Kluwer, 1997.
- [20] J. Edmonds. “Paths, Trees, and Flowers”. Em: *Canad. J. Math* 17 (1965), pp. 449–467.
- [21] Jenő Egerváry. “Matrixok kombinatorius tulajdonságairól (On combinatorial properties of matrices)”. Em: *Matematikai és Fizikai Lapok* 38 (1931), pp. 16–28.
- [22] T. Feder e R. Motwani. “Clique partitions, graph compression and speeding-up algorithms”. Em: *Proceedings of the Twenty Third Annual ACM Symposium on Theory of Computing (23rd STOC)*. 1991, pp. 123–133.
- [23] T. Feder e R. Motwani. “Clique partitions, graph compression and speeding-up algorithms”. Em: *Journal of Computer and System Sciences* 51 (1995), pp. 261–272.
- [24] L. R. Ford e D. R. Fulkerson. “Maximal flow through a network”. Em: *Canadian Journal of Mathematics* 8 (1956), pp. 399–404.
- [25] András Frank. *On Kuhn's Hungarian Method – A tribute from Hungary*. Rel. téc. Egerváry Research Group on Combinatorial Optimization, 2004.
- [26] C. Fremuth-Paeger e D. Jungnickel. “Balanced network flows VIII: a revised theory of phase-ordered algorithms and the $O(\sqrt{nm} \log(n^2/m)/\log n)$ bound for the nonbipartite cardinality matching problem”. Em: *Networks* 41 (2003), pp. 137–142.
- [27] Martin Fürer. “Faster Integer Multiplication”. Em: *STOC '07: Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*. San Diego, California, USA: ACM, 2007, pp. 57–66. ISBN: 978-1-59593-631-8. DOI: <http://doi.acm.org/10.1145/1250790.1250800>.

- [28] Martin Fürer e Balaji Raghavachari. “Approximating the minimum-degree steiner tree to within one of optimal”. Em: *Journal of Algorithms* (1994).
- [29] H. N. Gabow. “Data structures for weighted matching and nearest common ancestors with linking”. Em: *Proc. of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms* (1990), pp. 434–443.
- [30] Ashish Goel, Michael Kapralov e Sanjeev Khanna. “Perfect Matchings in $O(n \log n)$ Time in Regular Bipartite Graphs”. Em: *STOC 2010*. 2010.
- [31] A. V. Goldberg e A. V. Karzanov. “Maximum skew-symmetric flows and matchings”. Em: *Mathematical Programming A* 100 (2004), pp. 537–568.
- [32] Ronald Lewis Graham, Donald Ervin Knuth e Oren Patashnik. *Concrete Mathematics: a foundation for computer science*. Addison-Wesley, 1988.
- [33] Yuri Gurevich e Saharon Shelah. “Expected computation time for Hamiltonian Path Problem”. Em: *SIAM J. on Computing* 16.3 (1987), pp. 486–502.
- [34] Ken Habgood e Itamar Arel. “Revisiting Cramer’s rule for solving dense linear systems”. Em: *Proceedings of the 2010 Spring Simulation Multi-conference*. 2010. DOI: [10.1145/1878537.1878623](https://doi.org/10.1145/1878537.1878623).
- [35] Juris Hartmanis e Richard E. Stearns. “On the Computational Complexity of Algorithms”. Em: *Transactions of the American Mathematical Society* 117 (1965), pp. 285–306.
- [36] Thomas N. Hibbard. “Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting”. Em: *JACM* 9.1 (1962), pp. 16–17.
- [37] Dan S. Hirschberg. “A linear space algorithm for computing maximal common subsequences”. Em: *Comm. of the ACM* 18.6 (1975), pp. 341–343.
- [38] C. A. R. Hoare. “Quicksort”. Em: *Computer Journal* 5 (1962), pp. 10–15.
- [39] J. E. Hopcroft e R. Karp. “An $n^{5/2}$ algorithm for maximum matching in bipartite graphs”. Em: *SIAM J. Comput.* 2 (1973), pp. 225–231.
- [40] Michael J. Jones e James M. Rehg. *Statistical Color Models with Application to Skin Detection*. Rel. téc. CRL 98/11. Cambridge Research Laboratory, 1998.
- [41] Erich Kaltofen e Gilles Villard. “On the complexity of computing determinants”. Em: *Computational complexity* 13 (2004), pp. 91–130.

- [42] Anatolii Alekseevich Karatsuba e Yu Ofman. “Multiplication of Many-Digital Numbers by Automatic Computers”. Em: *Doklady Akad. Nauk SSSR* 145.2 (1962). Translation in Soviet Physics-Doklady 7 (1963), pp. 595–596, pp. 293–294.
- [43] Jon Kleinberg e Éva Tardos. *Algorithm design*. Addison-Wesley, 2005.
- [44] Donald E. Knuth. *The art of computer programming*. 2nd. Vol. III, Sorting and searching. Addison-Wesley, 1998.
- [45] Donald Ervin Knuth. “Big omicron and big omega and big theta”. Em: *SIGACT News* (1976).
- [46] H. W. Kuhn. “The Hungarian Method for the assignment problem”. Em: *Naval Research Logistic Quarterly* 2 (1955), pp. 83–97.
- [47] Richard Ladner. “On the structure of polynomial time reducibility”. Em: *Journal of the ACM* (1975). URL: <http://weblog.fortnow.com/2005/09/favorite-theorems-np-incomplete-sets.html>.
- [48] Jan van Leeuwen, ed. *Handbook of theoretical computer science*. Vol. A: Algorithms and complexity. MIT Press, 1990. URL: <http://www.amazon.com/Handbook-Theoretical-Computer-Science-Vol/dp/0262720140>.
- [49] Tom Leighton. Manuscript, MIT. 1996.
- [50] Vladimir Iosifovich Levenshtein. “Binary codes capable of correcting deletions, insertions, and reversals”. Em: *Soviet Physics Doklady* (1966), pp. 707–710.
- [51] Leonid Levin. “Universal’nye perebornyye zadachi (Universal Search Problems: in Russian)”. Em: *Problemy Peredachi Informatsii* 9.3 (1973), pp. 265–266.
- [52] Seth Lloyd. “Computational capacity of the universe”. Em: *Physical Review Letters* 88.23 (2002). <http://focus.aps.org/story/v9/st27>.
- [53] M.J. Magazine, G.L.Nemhauser e L.E.Trotter. “When the greedy solution solves a class of Knapsack problems”. Em: *Operations research* 23.2 (1975), pp. 207–217.
- [54] A. R. Meyer e L. J. Stockmeyer. “The equivalence problem for regular expression with squaring requires exponential time”. Em: *Proc. 12th IEEE Symposium on Switching and Automata Theory*. 1972, pp. 125–129.
- [55] J. Munkres. “Algorithms for the assignment and transporation problems”. Em: *J. Soc. Indust. Appl. Math* 5.1 (1957), pp. 32–38.

- [56] D. Pearson. “A polynomial time algorithm for the change-making problem”. Em: *Operations Research Letters* 33.3 (2005), pp. 231–234.
- [57] Salvador Roura. “Improved Master Theorems for Divide-and-Conquer Recurrences”. Em: *Journal of the ACM* 48.2 (2001), pp. 170–205.
- [58] J.R. Sack e J. Urrutia, eds. *Handbook of computational geometry*. Elsevier, 2000.
- [59] A. Schönhage e V. Strassen. “Schnelle Multiplikation grosser Zahlen”. Em: *Computing* 7 (1971), pp. 281–292.
- [60] Alexander Schrijver. *Combinatorial optimization. Polyhedra and efficiency*. Vol. A. Springer, 2003.
- [61] Robert Sedgewick. *Algorithmen in C++*. Addison-Wesley, 1992.
- [62] Robert Sedgewick. *Algorithms for the masses*. URL: <http://www.cs.princeton.edu/~rs/talks/AlgsMasses.pdf> (acedido em 14/03/2011).
- [63] Michael Sipser. *Introduction to the theory of computation*. Thomson, 2006.
- [64] Michael Sipser. “The history and status of the P versus NP question”. Em: *Proceedings of the 24th Annual ACM Symposium on the Theory of Computing*. 1992, pp. 603–619.
- [65] Volker Strassen. “Guassian Elimination is not Optimal”. Em: *Numer. Math* 13 (1969), pp. 354–356.
- [66] Laira Vieira Toscani e Paula A. S. Veloso. *Complexidade de Algoritmos*. 2a. Editora Sagra Luzzatto, 2005. URL: <http://www.inf.ufrgs.br/~laira/>.
- [67] Luca Trevisan. P and NP. <http://lucatrevisan.wordpress.com/2010/04/24/cs254-lecture-2-p-and-np>. 2010.
- [68] Alan Mathison Turing. “On computable numbers with an application to the Entscheidungsproblem”. Em: *Proc. London MathSoc.* 2.42 (1936), pp. 230–265.
- [69] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2001.
- [70] Paul M. B. Vitányi e Lambert Meertens. “Big Omega Versus the Wild Functions”. Em: *SIGACT News* 16.4 (1985).
- [71] J.S. Vitter e Philipe Flajolet. “Handbook of theoretical computer science”. Em: ed. por Jan van Leeuwen. Vol. A: Algorithms and complexity. MIT Press, 1990. Cap. Average-case analysis of algorithms and data structures. URL: <http://www.amazon.com/Handbook-Theoretical-Computer-Science-Vol/dp/0262720140>.

Bibliografia

- [72] Jie Wang. “Advances in Algorithms, Languages, and Complexity - In Honor of Ronald V. Book”. Em: Kluwer, 1997. Cap. Average-Case Intractable NP Problems.
- [73] *Wilkinson Microwave Anisotropy Probe*. Online. 2010. URL: <http://map.gsfc.nasa.gov> (acedido em 13/03/2011).
- [74] V. Vassilevska Williams e Ryan Williams. *Triangle detection versus matrix multiplication: a study of truly subcubic reducibility*. 2010.
- [75] Uri Zwick. “The smallest networks on which the Ford-Fulkerson maximum flow procedure may fail to terminate”. Em: *Theoretical Computer Science* 148.1 (1995), pp. 165 –170. DOI: [DOI:10.1016/0304-3975\(95\)00022-0](https://doi.org/10.1016/0304-3975(95)00022-0).