4 | Computação probabilística

Algoritmos, como estudados em disciplinas introdutórias e apresentados nos capítulos anteriores, são descritos no que chamamos de *pseudocódigo*. Trata-se de uma mistura de palavras-chave em português com sentenças estruturadas de forma semelhante a linguagens de programação como a linguagem C. Neste capítulo, introduzimos brevemente os modelos formais de computação, justificamos o uso de pseudocódigo e, em seguida, exploramos as classes de complexidade. Elas classificam problemas computacionais de acordo com a "dificuldade" de serem resolvidos por máquinas abstratas desses modelos.

4.1 MODELOS DE COMPUTAÇÃO

Nesta seção, discutimos modelos de computação de maneira *resumida* e *informal*. Para quem deseja se aprofundar no tema, recomendamos o texto de Savage (1998).

A Máquina de Turing, proposta por Alan Turing, e o λ-cálculo, introduzido por

Alonzo Church, foram os primeiros modelos formais de computação a capturar o conceito intuitivo de *algoritmo*. Esses modelos pioneiros, juntamente com outros, como o modelo formal RAM e o modelo genérico de linguagens de programação (exemplificado pela linguagem C), são equivalentes. Isso significa que tudo o que pode ser computado em um desses modelos também pode ser nos outros. Simulações entre modelos formais e um modelo genérico são possíveis, e, sob condições razoáveis, há pouca perda de desempenho, o que torna o estudo das classes de complexidade independente do modelo adotado.

Os modelos formais de computação são úteis para compreender as dificuldades inerentes à resolução de problemas computacionais. Embora sejam mais simples e bem definidos que linguagens de programação, têm a desvantagem de serem menos práticos para descrever algoritmos específicos. A especificação de um modelo de computação inclui uma descrição detalhada do dispositivo (máquina ou programa) do modelo, como a *entrada* é apresentada, e o que o dispositivo pode calcular para produzir a *saída* a partir da entrada. Além disso, o modelo deve definir uma noção de *passo elementar* de computação, a partir da qual derivamos medidas de *tempo de execução*. O tempo de execução de um dispositivo com uma dada entrada é o número de passos elementares necessários para concluir a computação. Geralmente, esse tempo é analisado como uma função do *tamanho* da entrada, utilizando critérios como o *pior caso*.

Problema computacional Vimos que um **problema computacional** é caracterizado por uma tripla $(\mathcal{I}, \mathcal{O}, \mathcal{R})$ em que $\mathcal{I}$ denota o conjunto das instâncias (entradas) do problema, $\mathcal{O}$ o conjunto das respostas (saídas) e $\mathcal{R} \subset \mathcal{I} \times \mathcal{O}$ é uma relação que associa a cada instância uma ou mais respostas. Uma solução para um problema computacional computacional é um dispositivo de computação que computa uma função $\mathsf{A} \subseteq \mathcal{R}$ e nesse caso dizemos que A é uma **função computável**. Usaremos a notação funcional para denotar o resultado de um processo computacional de modo que o dispositivo C com instância $x \in \mathcal{I}$ responde $\mathsf{C}(x) \in \mathcal{O}$.

Os problemas computacionais podem ser classificados em problemas de *decisão* e problemas de *busca*. Num problema de busca cada instância x de um problema está associada a um conjunto $\mathsf{R}_x = \{y \in \mathcal{O} \colon (x, y) \in \mathcal{R}\}$ das soluções possíveis e que pode ser vazio. Uma *solução* do problema é uma função computável A tal que $\mathsf{A}(x) \in \mathsf{R}_x$ sempre que $\mathsf{R}_x \neq \varnothing$, caso contrário $\mathsf{A}(x) := \perp$ (ou algum símbolo fora de $\mathcal{O}$ convencionado previamente, a função é parcial).

Em um problema de decisão $\mathcal{O} = \{\text{sim}, \text{não}\}$ e, usualmente, especificamos um subconjunto $\mathsf{L} \subseteq \mathcal{I}$ tal que $\mathsf{A}(x) = \text{sim}$ se e somente se $x \in \mathsf{L}$; desse modo o problema é decidir se uma instância qualquer x pertence ao conjunto L. Por exemplo,

o problema de decidir se um número é primo tem $\mathcal{I}$ como o conjunto dos inteiros positivos e tomamos L como o subconjunto dos inteiros positivos e primos. O teste em si é uma função computável A que responde sim se x pertence a L, caso contrário responde não, isto é, decide se x é ou não é um número primo.

A Complexidade Computacional é uma disciplina que, dentre outros objetivos, classifica problemas em classes de complexidade com relação a um modelo. Em geral, essas classes consideram problemas computacionais de decisão. Isso se justifica pela simplicidade e porque, embora pareça muito restritivo, para um grande classe de problemas de interesse os problemas de busca podem ser reduzidos a problemas de decisão (Goldreich, 2008, teoremas 2.6 e 2.10). O seguinte exemplo ilustra esse fenômeno.

Exemplo 4.1 (SAT). O problema denominado por SAT é o seguinte problema de decisão: dado uma fórmula booleana Φ, existe uma valoração das suas variáveis que a torna verdadeira? Nesse caso o conjunto $\mathcal{I}$ das instâncias é a família de todas as fórmulas booleanas sobre as variáveis $x_1, x_2, x_3, \ldots$. Como é um problema de decisão as respostas são $\mathcal{O} = \{\text{sim}, \text{não}\}$ e a relação $\mathcal{R}$ é dada pelos pares (Φ, sim) e $(\Phi', \text{não})$ para cada $\Phi \in \mathcal{I}$ satisfatível e cada $\Phi' \in \mathcal{I}$ não satisfatível ou, ainda, podemos colocar esse problema como o problema de decidir pertinência em

$$\text{SAT} := \{\Phi : (\Phi, \text{sim}) \in \mathcal{R}\}.$$

A versão de busca do problema SAT pede para determinar uma valoração das variáveis de Φ que torne a fórmula verdadeira ou determinar que tal valoração não existe. Entretanto, se $\Phi = \Phi(x_1, x_2, \ldots, x_n)$ e existe um algoritmo A para SAT então, caso Φ seja satisfatível, podemos escrever uma algoritmo A' para determinar uma valoração da seguinte maneira: A' usa A para decidir se $\Phi(1, x_2, \ldots, x_n)$ é satisfatível, se não for satisfatível então $\Phi(0, x_2, \ldots, x_n)$ é satisfatível; determinado o valor de x_1, o algoritmo A' repete o processo para determinar o valor de x_2, e assim por diante. Em n repetições desse processo descobrimos, valoração, caso exista, uma valoração ou descobrimos que não existe uma valoração que torne Φ verdadeira. Ademais, se A é um algoritmo de tempo polinomial então A' também será um algoritmo de tempo polinomial. ◊

Há, ainda, uma classe importante de problemas computacionais formada pelos problemas de *otimização*, como o MAX-SAT e o MAX-3SAT apresentados na seção 2.3.1. Cada par $(x, y) \in \mathcal{R}$ tem um custo $f(x, y) \in \mathbb{R}$ e buscamos pelas respostas que excedem um limiar ou atingem um valor máximo. No primeiro caso, dados x e c procuramos por $y \in \mathsf{R}_x$ tal que $f(x, y) \geqslant c$. No segundo caso, dado x, buscamos por y tal que $f(x, y)$ é máximo dentre todos $y \in \mathsf{R}_x$. Um problema computacional de otimização pode ser

reduzido a um problema computacional de busca (Goldreich, 2008, seção 2.2.2) que, por sua vez, pode ser reduzido a um problema computacional de decisão.

4.1.1 MÁQUINAS DE TURING

Uma **máquina de Turing** é uma sêxtupla $(Q, \Gamma, \Sigma, \delta, q_0, q_{\text{para}})$, em que Q é um conjunto finito de *estados* da máquina, Γ é o *alfabeto da fita* que contém um símbolo especial $\flat$ que representa *espaço em branco* e contém o conjunto Σ que é um *alfabeto* finito de modo que $\flat \notin \Sigma$, $q_0 \in Q$ é o *estado inicial*, $q_{\text{para}} \notin Q$ é o *estado final*, e $\delta\colon Q \times \Gamma \to (Q \cup \{q_{\text{para}}\}) \times \Gamma \times \{\leftarrow, —, \rightarrow\}$ é a *função de transição*. No que segue Σ^* denota o conjunto de todas as *palavras* formadas por símbolos de Σ.

Uma **computação** de uma máquina de Turing M começa com uma entrada w, digamos que $w = w_1 w_2 w_3 \dots w_n \in \Sigma^*$, escrita nas n primeiras posições de uma *fita* que é infinita para a direita; as outras posições contêm o símbolo $\flat$, um símbolo por posição. A máquina M começa no estado q_0 lê w_1 na posição mais a esquerda da fita; essa é a **configuração inicial**. Se a máquina está num estado $q \in Q$ e lê $\gamma \in \Gamma$ na fita então o próximo passo é dado por $\delta(q, \gamma) = (q', \gamma', \ell)$ em que $q' \in Q \cup \{q_{\text{para}}\}$ é o novo estado, $\gamma' \in \Gamma$ é o símbolo que a máquina deixa na posição que estava o símbolo γ e o passo $\ell \in \{\leftarrow, —, \rightarrow\}$ diz qual posição da fita é a próxima a ser lida, respectivamente, a que está a esquerda da atual, a atual ou a posição a direita da atual. A computação continua com $\delta(q', \gamma')$ se q' não é o estado final. Se M é uma máquina de Turing e $w \in \Sigma^*$ uma entrada para M, então uma computação de M com entrada w pode

- *terminar*, o que significa que M atingiu o estado final q_{para}. Nesse caso, a resposta da computação é a sequência z de símbolos escrita na fita desde a posição mais a esquerda até a última posição antes do primeiro $\flat$, e escrevemos $M(w) = z$ ou, senão, $M(w) = \flat$;

- *não terminar*, o que significa que a computação nunca chega ao estado final. Máquinas de Turing que não terminam são úteis para classificar problemas computacionais mas não são dispositivos úteis de computação.

Exemplo 4.2. Considere o problema de decidir a paridade de uma sequência binária $x_1 x_2 x_3 x_4 \dots x_n$ para qualquer $n \geqslant 1$, ou seja, determinar se a quantidade de 1's é ímpar, nesse caso responder 1, ou se é par, nesse caso responder 0. Uma máquina de Turing para esse problema tem estados $\{P, I, q_P, q_I\}$ além de q_0 e q_{para}. A função de transição é representada pelo diagrama de estados da Figura 4.2, explicado com o exemplo da Figura 4.1.

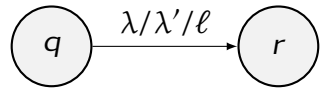

Figura 4.1: legenda para um diagrama de estados: se a máquina está no estado $q \in Q$ e lê $\lambda \in \Gamma$ na fita, então escreve $\lambda' \in \Gamma$ nessa posição e movimenta-se, de acordo com $\ell \in \{\leftarrow, —, \rightarrow\}$ ou uma posição para a esquerda, ou uma para a direita ou permanece na posição que está e, em seguida, entra no estado $r \in Q \cup \{q_{\text{para}}\}$.

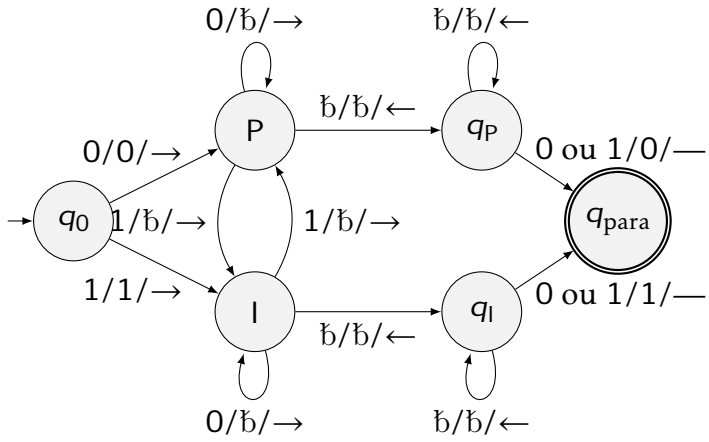

Figura 4.2: diagrama de estados da máquina de Turing para o problema da paridade.

Uma computação dessa máquina com uma entrada particular fica completamente descrita por: a palavra escrita na fita, o estado da máquina e a posição de leitura/escrita a cada passo. A execução dessa máquina de Turing com entrada 011 está mostrada na Figura 4.3, que deve ser lida de cima para baixo, da esquerda para a direita. Na última configuração lemos 0 na fita, o que significa que a a quantidade de 1's nessa entrada particular é par. ◊

Uma função $f\colon \Sigma^* \to \Sigma^* \cup \perp$ é uma **função computável** se existe uma máquina de Turing M tal que para todo $w \in \Sigma^*$, quando M começa a computação na configuração inicial com entrada w, termina com somente $f(w)$ escrito na fita (todo outro símbolo na fita é ƀ), ou seja, $M(w) = f(w)$.

Seja M uma máquina de Turing que termina a computação com qualquer entrada e $T\colon \mathbb{N} \to \mathbb{N}$ uma função. Dizemos que M é uma **máquina de Turing com tempo de execução** T, ou simplesmente máquina de Turing de tempo T, se para todo $w = w_1 w_2 w_3 \ldots w_n \in \Sigma^*$ a máquina computa $M(w)$ após no máximo $T(n)$ transições.

Um conjunto de palavras $L \subseteq \Sigma^*$ é uma **linguagem**. Por abuso de notação denotamos também por L a função característica[1] $L\colon \Sigma^* \to \{0,1\}$ do conjunto L. Se L é uma linguagem e M uma máquina de Turing, então dizemos que M **decide** L se para

[1] A *função característica de um conjunto* $A \subseteq U$ é a função $f\colon U \to \{0,1\}$ tal que $f(x) = 1$ se e somente se $x \in A$.

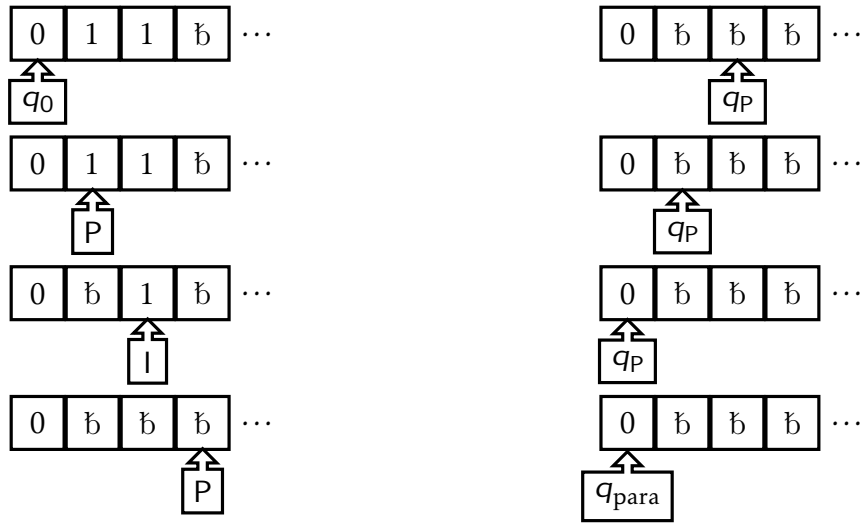

Figura 4.3: a fita na execução da máquina de Turing para a paridade com entrada 011.

todo $w \in \Sigma^*$

$$\mathsf{M}(w) = \mathsf{L}(w)$$

ou seja, decide pertinência na linguagem L. Ainda, se M decide L então dizemos que M **aceita** as palavras $w \in \mathsf{L}$ e **rejeita** as palavras $w \notin \mathsf{L}$.

Codificação Vamos assumir que as instâncias e respostas dos problemas computacionais estão codificadas usando símbolos de um alfabeto Σ finito e com pelo menos dois símbolos. De fato, assumimos, sem perda de generalidade (veja o Exercício 4.22, página 241) que

$$\Sigma = \{0, 1\}$$

de modo que Σ^* é o conjunto de todas as sequências binárias. O **tamanho** ou **comprimento** de uma palavra $w \in \Sigma^*$ é a quantidade de símbolos de Σ que constituem w, denotado por $|w|$.

Usamos o símbolo especial λ para a *palavra vazia* de modo que $\lambda \in \Sigma^*$ e $|\lambda| = 0$. Também, convencionamos que Σ^n denota o conjuntos das palavras de comprimento n. Assim, podemos escrever

$$\Sigma^* = \bigcup_{n \geqslant 0} \Sigma^n.$$

As computações são realizadas sobre sequências binárias, logo os elementos do conjunto das instâncias $\mathcal{I}$ de um problema computacional devem ser representados usando somente 0's e 1's por uma **codificação**

$$c \colon \mathcal{I} \cup \mathcal{O} \to \{0, 1\}^*$$

e no caso $\mathcal{O} = \{\text{sim}, \text{não}\}$ convencionamos

$$c(\text{sim}) = 1 \text{ e } c(\text{não}) = 0.$$

Por exemplo, números, grafos, fórmulas booleanas devem ser codificados em binário. Não nos preocuparemos com detalhes dessas codificações e *assumiremos que sempre é possível fixarmos uma codificação canônica*. No caso de números consideramos a sua representação em base 2 e um grafo é dado pela matriz de adjacências.

Sempre que for oportuno deixar claro que estamos usando uma representação binária (canônica) usamos a notação $\langle x \rangle$ para $c(x)$. Por exemplo, uma representação binária da fórmula booleana Φ é denotada por $\langle \Phi \rangle$ e a linguagem $\textsc{sat} \subset \Sigma^*$ é o conjunto $\textsc{sat} = \{\langle \Phi \rangle : \Phi \text{ é fórmula booleana satisfatível}\}$.

Desse modo, quando falamos em computar uma função $f : \mathcal{I} \to \mathcal{O}$, de fato nos referimos a uma função $f_c : c(\mathcal{I}) \to \{0,1\}^*$ de maneira que para uma representação binária de $x \in \mathcal{I}$, a função f_c determina uma representação binária de $f(x)$.

Universalidade e Indecidibilidade Uma propriedade do modelo, de grande importância, é chamada de universalidade, descrita abaixo em (4.1). Primeiro o leitor deve se convencer de que uma máquina de Turing M tem uma descrição finita e em seguida perceber que essa descrição pode ser codificada por $\langle M \rangle$, ou seja, *é possível estabelecermos uma codificação binária canônica para as máquinas de Turing* (Hopcroft, Motwani e Ullman, 2000, seção 9.1.2); denotamos por M_α a máquina cuja representação binária é $\alpha \in \{0,1\}^*$. O fato importante é que

$$\textit{existe uma máquina de Turing } \mathcal{U} \textit{ dita} \text{ universal } \textit{tal que } \mathcal{U}(w, \alpha) = M_\alpha(w) \tag{4.1}$$

e essa simulação é bastante eficiente, se M_α é uma máquina de Turing de tempo T então $\mathcal{U}(\cdot, \alpha)$ é uma máquina de tempo $O(T \log(T))$ (Arora e Barak, 2009, seção 1.4.1, teorema 1.9).

O conjunto de todas as máquinas de Turing é enumerável. A cardinalidade do conjunto formado pelas funções $f : \mathbb{N} \to \{0,1\}$ é a cardinalidade do conjunto $2^{\mathbb{N}}$, das partes de $\mathbb{N}$, e pelo Teorema de Cantor não há função sobrejetiva de $\mathbb{N}$ em $2^{\mathbb{N}}$, portanto existem funções de $\mathbb{N}$ em $\{0,1\}$ que não são computáveis por máquina de Turing ou, pondo de outro modo

existem linguagens que não são decidíveis

e dizemos que uma linguagem dessas é **indecidível**.

Um problema indecidível bastante conhecido é o famoso *Problema da Parada*: decidir se a computação de uma máquina de Turing M com uma entrada x termina. Consideremos uma enumeração $(M_i : i \in \mathbb{N})$ das máquinas de Turing e suponhamos

que exista uma máquina de Turing H que com entrada $(\langle i \rangle, x)$ decide se a computação $M_i(x)$ termina. Tomemos P a máquina de Turing que com entrada x computa da seguinte maneira:

```
1 se H(x, x) = 0 então responda 1;
2 senão se 𝔘(x, x) = 1 então responda 0;
3 senão responda 1.
```

Algoritmo 38: $P(x)$

A resposta é $P(x) = 0$ se, e só se, $\mathfrak{U}(x, x) = 1$. Agora, se tomarmos como entrada para P a palavra $x := \langle n \rangle$ de tal sorte que $M_n = P$, então $\mathfrak{U}(x, x) = M_n(\langle n \rangle) = P(x)$ e

$$P(x) = 0 \text{ se, e só se, } P(x) = 1$$

uma contradição. Logo, não deve existir a máquina H.

4.1.2 MODELO RAM E ALGORITMOS EFICIENTES

Um modelo mais próximo, na sua concepção, do computador (eletrônico) que conhecemos é o modelo RAM que tem um número infinito e enumerável de registradores (memória), $M_0, M_1, \ldots$, um conjunto finito de instruções para transferência de valores entre registradores, endereçamento (direto e indireto), aritmética (soma e subtração), desvio condicional e um contador de programa que indica qual instrução de um programa vai ser executada. Um *programa* RAM é uma sequência finita de instruções $N_0, N_1, \ldots, N_m$ dentre as instruções válidas do modelo.

Um conjunto típico de instruções é dado na Tabela 4.1.

`INC Mi`	Incrementa o conteúdo do registrador i de 1
`DEC Mi`	Decrementa o conteúdo do registrador i de 1
`CLR Mi`	Substitui o conteúdo de M_i por 0
`Mi ← Mj`	Substitui o conteúdo de M_i pelo de M_j
`Mi ← MMj`	Substitui o conteúdo de M_i pelo de M_{M_j}
`MMi ← Mj`	Substitui o conteúdo de M_{M_i} pelo de M_j
`JMP Ni`	Atribui N_i ao contador de programa
`Mj JIfZ Ni`	Se $M_j = 0$, atribui N_i ao contador de programa
`CONTINUE`	Continue na próxima instrução, caso exista, ou pare

Tabela 4.1: Instruções básicas do modelo RAM.

Cada posição de memória pode armazenar um inteiro e as instruções operam sobre inteiros. As instruções são executadas sequencialmente, a cada instrução executada o contador de programa é incrementado, exceto nas instruções de desvio.

A computação começa com a entrada nas primeiras posições de memória, digamos $M_0, M_1, \ldots, M_n$, as outras posições são nulas; o contador de programa indica N_0; a resposta é dada em M_0. Em qualquer instante, apenas uma quantidade finita dos números armazenados na memória são diferente de 0.

Uma diferença importante com relação às máquinas de Turing é que numa máquina RAM temos acesso direto as posições de memória[2], numa só instrução da máquina, como em um computador. Por outro lado, uma diferença importante com relação aos computadores é que assumimos que uma máquina RAM tem memória ilimitada e isso faz com que em cada posição de memória deva ser permitido armazenar um inteiro arbitrariamente grande pois precisamos guardar na memória os endereços de memória para ser possível o endereçamento direto. Exceto pelas computações que abusam do fato de ser permitido armazenar e operar um inteiro arbitrariamente grande, um computador e uma máquina RAM têm os mesmos programas.

O **tempo de execução** de um programa RAM que sempre termina é definida pelo número de instruções da RAM que o programa executa em função do tamanho da entrada. Usualmente consideramos dois modelos RAM quando levamos em conta o tempo de execução: o modelo de **custo logarítmico** que leva em conta o custo das operações em função do tamanho dos operandos, e o modelo de **custo unitário** com instruções em tempo constante, que é o mais comum quando analisamos algoritmos. A soma de dois números, por exemplo, tem custo proporcional à quantidade de dígitos no primeiro caso e no segundo caso tem custo constante. Se um programa RAM usa números grandes o suficiente para que se torne irreal assumir que a adição e outras operações podem ser executadas com custo unitário, o modelo de custo logarítmico de RAM é mais adequado. A escolha de qual o modelo é mais adequado depende da aplicação. Por exemplo, para algoritmos que lidam com números, como o que encontramos na seção 1.7.2 em que o produto vC de um vetor de dimensão n por uma matriz quadrada de ordem n foi atribuído custo (unitário) $O(n^2)$, a análise com custo unitário só é representativa quando os operandos têm tamanho significativamente menor que a instância do problema, caso as entradas da matriz e do vetor tivessem n dígitos, por exemplo, esse custo estaria muito subestimado.

Simulações Uma máquina de Turing que executa T transições pode ser simulada por um programa RAM que executa $O(T)$ instruções (Savage, 1998, teorema 3.8.1). Um programa RAM com inteiros de tamanho limitado que executa T instruções pode ser simulado por uma máquina de Turing que executa $O(T^3)$ transições (Savage, 1998, teorema 8.4.1) (veja também Papadimitriou, 1994, teoremas 2.4 e 2.5).

[2] Esse é o motivo do nome *random access*, não tem relação com "aleatório".

Um programa RAM é muito semelhante a um programa em linguagem de montagem (*assembly*), que é especifica de um processador e um sistema operacional. Há compiladores que transformam programas em linguagens como C em programas em linguagem de montagem para, em seguida, criar um código que seja executável num computador eletrônico.

Algoritmos eficientes Tradicionalmente *computação eficiente* é sinônimo de computação feita por um dispositivo com *tempo de execução polinomial* no tamanho da entrada. Dizemos que uma máquina de Turing, ou programa RAM, tem **tempo polinomial** se existe constante inteira e positiva k tal que nas entradas de tamanho n o tempo de execução do dispositivo é $O(n^k)$. Comentamos acima que máquina de Turing e o modelo RAM são **polinomialmente equivalentes**, ou seja, tudo o que pode ser computado em tempo polinomial num modelo também pode ser computado em tempo polinomial no outro. Portanto, a classe dos problemas computacionais que podem ser resolvidos em tempo polinomial para o modelo máquina de Turing coincide com a classe dos problemas computacionais que podem ser resolvidos em tempo polinomial para o modelo RAM. De fato, essa classe é invariante para todos os modelos clássicos que são razoáveis.

Para simplificar nós dizemos, de modo genérico, **algoritmo de tempo polinomial** para nos referirmos a um dispositivo de um modelo formal tradicional que tenha tempo de execução polinomial. Ademais, temos a facilidade de descrever o algoritmo usando um pseudocódigo[3]. Neste texto assumimos o custo unitário quando analisamos o tempo de execução de um algoritmo e a distribuição de probabilidade da resposta, a não ser que seja dito explicitamente outra coisa. Nesse sentido, para nos mantermos independentes de um modelo formal tomamos o cuidado de o tamanho dos operandos usados nas instruções serem limitados polinomialmente no tamanho da entrada e dos sorteios serem feitos num conjunto de tamanho polinomial no tamanho da entrada de modo a garantir a equivalência polinomial (assintótica) entre os modelos de custo unitário e logarítmico.

4.1.3 CIRCUITO BOOLEANO

Um **circuito booleano** é representado por um grafo orientado acíclico em que cada vértice está associado a: (*i*) ou a uma porta lógica dentre *e*, *ou*, *não*; (*ii*) ou a uma

[3]Em Teoria da Computação, tradicionalmente, "algoritmo" é uma palavra usada para se referir à máquinas de Turing que terminam a computação. Aqui adotamos a convenção de que se refere a qualquer dispositivo formal que termina a computação. Em particular, quando o término se dá em tempo polinomial o modelo formal de fato não importa.

variável x_i para $i \in \{1, 2, \ldots, n\}$ para algum $n \in \mathbb{N}$; (*iii*) ou uma saída s_i, $i \in \{1, 2, \ldots, m\}$ para algum $m \in \mathbb{N}$. O grau de entrada dos vértices pode assumir um de três valores: 0 (variáveis), 1 (porta *não* e saídas) e 2 (portas *e* e *ou*). Um circuito computa uma **função binária** (o caso $m = 1$ é chamado de **função booleana**)

$$\begin{aligned} \mathsf{C}\colon \{0,1\}^n &\to \{0,1\}^m \\ (x_1, x_2, \ldots, x_n) &\mapsto (s_1, s_2, \ldots, s_m), \end{aligned}$$

que também denotaremos por C, do seguinte modo, uma entrada $(x_1, x_2, \ldots, x_n) \in \{0,1\}^n$ valora as variáveis do circuito; sempre que as entradas de uma porta lógica estão valoradas, a porta opera sobre os valores e devolve o resultado, esse resultado é propagado no circuito até que encontra uma saída.

Exemplo 4.3. O circuito na Figura 4.4 abaixo computa a paridade da sequência bi-

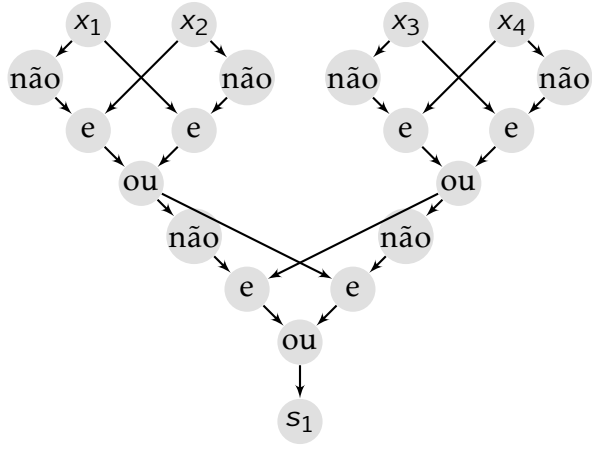

Figura 4.4: circuito booleano que computa $s_1(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3 + x_4 \bmod 2$.

nária $x_1x_2x_3x_4$, ou seja, $s_1 = 1$ se e somente se o número de ocorrências de 1's na sequência é ímpar. ◊

Se L é uma linguagem, então uma família de circuitos $\mathcal{C} = (\mathsf{C}_n\colon n \in \mathbb{N})$ **decide** L se para cada $x \in \{0,1\}^*$ o circuito $\mathsf{C}_{|x|}$ com entrada x responde 1 se e somente se $x \in \mathsf{L}$, em outras palavras, se denotamos também por L a função característica $\mathsf{L}\colon \{0,1\}^* \to \{0,1\}$ do conjunto L, temos que

$$\mathsf{C}_n(x) = \mathsf{L}(x)$$

para todo $x \in \{0,1\}^n$, para todo n.

O **tamanho do circuito** C, denotado por $|\mathsf{C}|$, é o número de vértices do grafo subjacente a definição de circuito e, em geral, estamos interessado no tamanho dos circuitos de uma família $\mathcal{C}$ em função do tamanho da entrada. Outro parâmetro

usado para medir a complexidade de um circuito é a sua **profundidade** que é o número de arestas no caminho mais longo através do circuito.

Seja $\mathcal{C} = (C_n : n \in \mathbb{N})$ uma família de circuitos e $s : \mathbb{N} \to \mathbb{N}$ uma função. Dizemos que $\mathcal{C}$ é uma família de **tamanho** $s(n)$ se $|C_n| \leqslant s(n)$ para todo n. A **complexidade de uma linguagem** L **com respeito a circuitos** é a função $s_L : \mathbb{N} \to \mathbb{N}$ se $s_L(n)$ é o menor tamanho de uma família de circuitos que decide L.

Um fato conhecido sobre circuitos e que será usado várias vezes mais adiante é o seguinte resultado cuja prova é um exercício (Sipser, 1996, teorema 9.30).

Exercício 4.4 (teorema 9.30 em Sipser (1996)). Prove que se L é uma linguagem que pode ser decidida por uma máquina de Turing de tempo $T(n)$ então L pode ser decidida por uma família de circuitos $(C_n : n \in \mathbb{N})$ de tamanho $O(T(n)^2)$ (dica: suponha tempo $T(n)$, exatamente, considere apenas as $T(n)$ primeiras posições da fita e as $T(n)$ configurações da fita dadas por uma computação, uma para cada passo (veja a Figura 4.3, página 198). Cada posição de uma fita, depende somente de três posições da fita da configuração imediatamente anterior).

A recíproca dessa proposição não vale pois há linguagens indecidíveis por máquina de Turing que admitem circuito de tamanho polinomial (veja o Exercício 4.23, página 241). Circuito booleano e máquina de Turing não são computacionalmente equivalentes.

Ao contrário da máquina de Turing, com circuito booleano temos um dispositivo de computação para cada tamanho de entrada, nesse caso dizemos que circuito booleano é um **modelo não-uniforme** de computação enquanto que máquina de Turing é um **modelo uniforme** de computação. A não uniformidade faz com que seja possível *desaleatorizar* eficientemente circuitos probabilísticos, ou seja é possível evitar o uso de bits aleatórios sem aumentar substancialmente o tamanho dos circuitos (veja o Exercício 4.25, página 241).

Circuito aritmético Um circuito aritmético é como um circuito booleano no qual as entradas são variáveis ou constantes, as portas *ou* são substituídas pela *soma* e as portas *e* são substituídas pela *multiplicação*. De modo mais geral, em um **circuito aritmético sobre um corpo** $\mathbb{F}$ as constantes e as operações aritméticas são as do corpo. Por exemplo, o circuito na Figura 4.5 abaixo computa $x_1 x_2 + 1$. Circuito aritmético é o modelo padrão para computação sobre polinômios, a saída de um circuito aritmético é um polinômio (ou um conjunto de polinômios) nas de variáveis entrada. As medidas de complexidade associadas com tais circuitos são *tamanho*, que é a quantidade de vértices no grafo subjacente, e *profundidade*, que é a maior

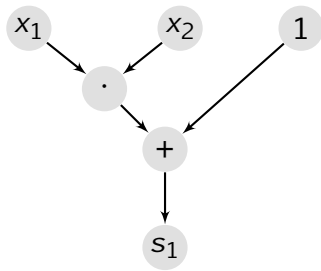

Figura 4.5: Circuito aritmético que computa $s_1(x_1, x_2) = x_1 \cdot x_2 + 1$.

distância entre uma entrada e uma saída.

Observamos que toda função booleana $f\colon \{0,1\}^n \to \{0,1\}$ pode ser expressa por uma fórmula booleana que, por sua vez, equivale a um polinômio sobre $\mathbb{F}_2$ (O'Donnell, 2021).

4.2 CLASSES DE COMPLEXIDADE DE TEMPO POLINOMIAL

Uma classe de complexidade computacional com respeito a um modelo de computação é o conjunto dos problemas de computação que são resolvidos por um dispositivo computacional naquele modelo com alguma restrição de recurso como, por exemplo, o tempo de execução. Como nos restringimos aos problemas de decisão, as classes de complexidade são apresentadas como classes de linguagens. As definições que apresentaremos abaixo são elaboradas tendo em mente máquinas de Turing como modelo formal de algoritmo, mas com alguns cuidados que já descrevemos isso significa que *algoritmo* pode ser entendido no sentido corriqueiro, escrito em pseudocódigo.

A classe P — *Polynomial time* — é a classe das linguagens decididas por algoritmos de tempo polinomial. Por exemplo, o problema de decidir se um inteiro positivo é primo está em P (Agrawal, Kayal e Saxena, 2004). Não sabemos se SAT pertence a P pois não conhecemos nenhum algoritmo de tempo polinomial para SAT.

A classe descrita a seguir é dada por uma definição alternativa, porém equivalente, à definição clássica (e.g. Papadimitriou, 1994; Sipser, 1996, ou veja o Exercício 4.5).

A classe NP — *Nondeterministic Polynomial time* — é a classe das linguagens L para as quais existe um polinômio p e um algoritmo *verificador* M de tempo polinomial tal que todo $w \in \{0,1\}^*$

1. se $w \in L$ então $M(w, B) = 1$ para algum $B \in \{0,1\}^{p(|w|)}$,
2. se $w \notin L$ então $M(w, B) = 0$ qualquer que seja $B \in \{0,1\}^{p(|w|)}$,

ou seja, se $w \in \mathsf{L}$ então existe um a *testemunha* que faz M responder *sim*, se $w \notin \mathsf{L}$, M responde *não* qualquer que seja a *testemunha*.

Por exemplo, $\textsc{sat} \in \mathtt{NP}$ pois podemos escrever um algoritmo M que com entrada (Φ, v) verifica em tempo polinomial se v é uma valoração das variáveis da fórmula booleana Φ que torna a fórmula verdadeira. Como $\Phi \in \textsc{sat}$ se, e somente se, existe uma valoração das variáveis de Φ que torna a fórmula verdadeira, existe uma testemunha v que convence M nos casos de fórmulas satisfazíveis. Caso contrário, qualquer que seja a sequência v, ela não satisfaz uma fórmula $\Phi \notin \textsc{sat}$, logo o algoritmo responde não. Ademais, o tamanho de $\langle v \rangle$ é polinomial no tamanho de $\langle \Phi \rangle$.

Uma sequência B no algoritmo de verificação é chamada de **certificado**. Dizemos que para $w \in \mathsf{L}$ e $\mathsf{L} \in \mathtt{NP}$ há um certificado curto, de tamanho polinomial em $|w|$, que prova tal fato, senão, caso $w \notin \mathsf{L}$, não há um certificado curto para $w \in \mathsf{L}$. Por exemplo, um inteiro $d > 1$ que divide outro inteiro positivo n é um certificado curto de que n é composto. O problema de decidir se um inteiro positivo é composto está em $\mathtt{NP}$ pois $|\langle d \rangle| \leqslant |\langle n \rangle|$ e há um algoritmo de tempo polinomial em $|\langle n \rangle|$ que verifica se d divide n.

Não é difícil mostrar que $\mathtt{P} \subseteq \mathtt{NP}$: se $\mathsf{L} \in \mathtt{P}$ e D é um algoritmo polinomial para L então um algoritmo verificador de tempo polinomial M para L simula D e ignora a entrada auxiliar de modo que $\mathsf{M}(w, \mathsf{B}) = \mathsf{D}(w)$ para todo w e todo B.

Por outro lado, não é sabido se a inclusão $\mathtt{NP} \subseteq \mathtt{P}$ é verdadeira. Esse é um dos principais, senão o principal, problema não resolvido da Teoria da Computação e foi formulado independentemente por Stephen Cook e Leonid Levin em 1971

Problema 5. $\mathtt{P} \neq \mathtt{NP}$?

Muito do que está exposto neste capítulo nasceu das tentativas de entender melhor esse problema.

Exercício 4.5 (definição clássica de $\mathtt{NP}$*).* A definição original da classe $\mathtt{NP}$ envolve máquinas de Turing não-determinísticas: uma sêxtupla como a que define a máquina de Turing exceto pelo função de transição que tem múltiplos valores $\delta(q, \gamma) = \{(q_1, \gamma_1, \ell_1), \ldots, (q_k, \gamma_k, \ell_k)\}$. Uma palavra é *aceita* se algum "ramo" da computação termina com aceite (um ramo da computação é uma sequência de transições dentro das possibilidades em cada passo). Cabe ressaltar que o sentido de "não-determinístico" que aparece na definição não é o mesmo que de "probabilístico", a definição (sintática) de máquina é semelhante mas a definição de computação (semântica) é diferente. Essa máquina não é um modelo realista de computação (veja mais em Sipser, 1996). Prove que a classe $\mathtt{NP}$ é a mesma para as definições descritas acima.

Problemas completos A linguagem L é chamada de NP-**difícil** se existe um algoritmo eficiente R tal que para toda linguagem $\mathsf{I} \in \mathsf{NP}$ temos $x \in \mathsf{I}$ se e só se $\mathsf{R}(x) \in \mathsf{L}$. O algoritmo R é chamado de **redução** de tempo polinomial. Se L está em NP e é NP-difícil, então é chamada de NP-**completa** se está em NP.

Decorre dessa definição que um algoritmo eficiente para uma linguagem NP-completa L implica num algoritmo eficiente para toda linguagem em NP, e isso resolveria o Problema 5. São conhecidos muitos problemas cuja linguagem é NP-completa, a primeira foi estabelecida pelo famoso Teorema de Cook–Levin que prova que SAT é NP-completa. Reduzindo SAT a uma linguagem $\mathsf{L} \in \mathsf{NP}$ prova que L é NP-completa.

A definição análoga para a classe P é: a linguagem $\mathsf{L} \in \mathsf{P}$ é P-completa se para toda linguagem em P há um redução em tempo polinomial para L. Entretanto, tal redução não é interessante para a teoria porque não traz informação nova sobre a complexidade relativa dos problemas (todo problema em P é completo para redução de tempo polinomial). Nesse caso, há outras reduções de interesse como, por exemplo, as de espaço logarítmico.

Complemento de uma classe Se $\mathcal{P}$ é um problema computacional de decisão, então suas instâncias são particionadas em $\mathcal{I}_{\text{sim}}$, o conjunto das instâncias sim, e $\mathcal{I}_{\text{não}}$, o conjunto das instâncias não. Como vimos acima, a codificação dos elementos em $\mathcal{I}_{\text{sim}}$ é a linguagem $\mathsf{L}(\mathcal{P})$ associada ao problema. O **complemento do problema de decisão** $\mathcal{P}$ é obtido trocando-se os papéis das intâncias *sim* e *não* e denotado $\neg\mathcal{P}$. Com respeito à linguagem associada, $\mathsf{L}(\neg\mathcal{P})$ é a linguagem definida pela codificação de $\mathcal{I}_{\text{não}}$.

O **complemento de uma linguagem** $\mathsf{L} \subseteq \Sigma^*$ é a linguagem $\overline{\mathsf{L}} = \Sigma^* \setminus \mathsf{L}$. O complemento $\overline{\mathsf{L}}(\mathcal{P})$ é formado pelas cadeias de símbolos de $\mathsf{L}(\mathcal{P})$ mais as cadeias de símbolos que não são instâncias de $\mathcal{P}$.

O problema SAT é o de decidir se uma fórmula booleana é satisfazível, a linguagem SAT é formada por uma codificação (canônica) das fórmulas booleanas satisfazíveis. O complemento do problema SAT é o problema apelidado de UNSAT de decidir se uma fórmula booleana não é satisfazível e linguagem UNSAT é formada pela codificação (canônica) das fórmulas booleanas não satisfazíveis. Notemos que, a rigor, a linguagem $\overline{\text{SAT}}$ é diferente de UNSAT.

Se C é uma classe de complexidade de problemas de decisão o **complemento da classe de complexidade**, denotada por $\mathbf{co}\mathsf{C}$, é a classe dos problemas de decisão formada pelos complementos dos problemas em C. Sob a hipótese de que é fácil identificar as entradas que não codificam instâncias do problema assumimos que $\mathbf{co}\mathsf{C}$ é formada pelas linguagens L tais que $\overline{\mathsf{L}} \in \mathsf{C}$.

Pode-se deduzir facilmente da definição que $\mathsf{P} = \mathsf{coP}$: se $\mathsf{L} \in \mathsf{P}$ então existe um algoritmo polinomial M que decide L. O algoritmo $\overline{\mathsf{M}}(w) := 1 - \mathsf{M}(w)$ de tempo polinomial decide $\overline{\mathsf{L}}$, portanto $\mathsf{L} \in \mathsf{coP}$. A recíproca desse argumento vale, ou seja, $\mathsf{coP} \subseteq \mathsf{P}$, portanto esses conjuntos são iguais.

Na definição de NP, Exercício 4.5, há uma assimetria entre aceitar e rejeitar uma palavra o que invalida o mesmo argumento para tentar provar que $\mathsf{NP} = \mathsf{coNP}$. De fato, atualmente, não é sabido se vale tal igualdade

Problema 6. $\mathsf{NP} \neq \mathsf{coNP}$?

em que a classe coNP é a classe das linguagens L para as quais existe um polinômio p e um algoritmo *verificador* M de tempo polinomial tal que todo $w \in \{0,1\}^*$

$w \in \mathsf{L}$ se e somente se $\mathsf{M}(w, \mathsf{B}) = 1$ *para todo* $\mathsf{B} \in \{0,1\}^{p(|w|)}$.

Por exemplo, a linguagem TAUT formada pelas fórmulas booleanas verdadeiras (tautologias) está em coNP. O complemento da linguagem é dado pelas fórmulas booleanas que não são tautológicas e para uma dada fórmula dessa linguagem um certificado é uma valoração que faça fórmula falsa, portanto em NP. É fácil verificar que se $\mathsf{NP} \neq \mathsf{coNP}$ então $\mathsf{P} \neq \mathsf{NP}$.

Complexidade de circuitos A classe $\mathsf{P/poly}$ é a classe de todas as funções booleanas $f\colon \{0,1\}^* \to \{0,1\}$ que são computáveis por uma família de circuitos de tamanho polinomial.

O principal motivo para a definição da classe $\mathsf{P/poly}$ foi a esperança de poder separar P de NP. Do Exercício 4.4 acima nós deduzimos que uma cota inferior para o tamanho dos circuitos que computam um determinado problema implica numa cota inferior para o tempo de um algoritmo que computa o mesmo problema. Esse fato fornece uma estratégia de prova de que $\mathsf{P} \neq \mathsf{NP}$. Por exemplo, se soubéssemos provar que SAT não pode ser decidido por uma família de circuitos de tamanho polinomial[4] então SAT não poderia ser decidido por um algoritmo de tempo polinomial e esse fato estabeleceria que $\mathsf{P} \neq \mathsf{NP}$.

Ademais, funções booleanas que precisam de circuitos grandes abundam. O número de funções booleanas em n variáveis é 2^{2^n} e a fração delas que são computadas com um circuito com $t = 2^n/n$ portas lógicas tende a 0 quando n cresce. A quantidade de circuitos com t portas lógicas pode ser estimada, de fato super estimada, com a seguinte descrição: cada uma das t portas pode ser de 3 tipos (e, ou, não)

[4] Não há nada especial em SAT a não ser pelo fato de ser NP-completo; qualquer outro problema NP-completo bastaria.

e recebe bits de, no máximo, outras duas portas dentre as $n+t-1$. Se $C(n,t)$ é a quantidade de circuitos com n entradas e t portas, então

$$C(n,t) < \frac{(3(n+t)^2)^t}{t!} < \frac{(9(n+t)^2)^t}{t^t} < 9^t t^t \left(1+\frac{n}{t}\right)^{2t} \leqslant 9^t t^t e^{2n}$$

descontando as $t!$ permutações de cada descrição, usando o limitante $t! > (t/3)^t$ (segue da estimativa de Stirling, (d.3)). Algumas descrições não correspondem a circuitos, mas todo circuito tem uma tal descrição. Ademais, toda função computada por um circuito de tamanho no máximo t também é computável por um circuito de tamanho exatamente t e cada uma das permutações das descrições portas leva a uma descrição diferente de um circuito computando a mesma função.

Assim, uma estimativa para a quantidade de circuitos com no máximo $t = 2^n/n$ portas é

$$C(n, 2^n/n) < \left(9\frac{2^n}{n}\right)^{\frac{2^n}{n}} e^{2n} = \left(\frac{9e^{\frac{n^2}{2^{n-1}}}}{n}\right)^{\frac{2^n}{n}} 2^{2^n} \ll 2^{2^n}$$

ou seja, a quantidade de circuitos com $2^n/n$ portas lógicas é assintoticamente muito menor que a quantidade de funções booleanas em n variáveis, assim concluímos que, assintoticamente, quase todas as funções (uma fração $(1-o(1))$ delas) necessitam de circuitos maiores que $2^n/n$ para serem computadas.

Portanto, existem muitas (na verdade, a maioria) funções booleanas exponencialmente difíceis mas não conseguimos explicitar uma família natural dessas funções booleanas. Uma justificativa para a dificuldade de explicitarmos um problema difícil para o qual conseguimos provar uma cota inferior superpolinomial, como $2^n/n$, é dada na seção 7.2.

Também, decorre do Exercício 4.4 acima que $\mathsf{P} \subseteq \mathsf{P/poly}$. Atualmente, não é sabido se $\mathsf{NP} \nsubseteq \mathsf{P/poly}$.

Problema 7. $\mathsf{NP} \nsubseteq \mathsf{P/poly}$?

Se esse for o caso, ou seja a resposta para o problema acima é sim, como explicamos no parágrafo acima, concluímos que $\mathsf{P} \neq \mathsf{NP}$, resolvendo o Problema 5 (página 206).

4.2.1 CLASSES PROBABILÍSTICAS

O estudo sistemático da computabilidade e complexidade de modelos probabilísticos, bem como das classes de complexidade, começou com Leeuw *et al.*, 1970 e Gill, 1974. Este último prova algumas propriedades básicas de máquinas de Turing probabilísticas, como a enumerabilidade efetiva e a universalidade entre outros resultados. Podemos associar a cada modelo de computação um modelo aumentado

que permite elementos aleatórios, por exemplo, podemos modificar a classe de programas RAM introduzindo uma instrução especial chamada RANDOM(N) que atribui para a variável N um bit com igual probabilidade e de forma independente de chamadas anteriores. Como já dissemos antes, dentro das classes de tempo polinomial, os resultados relatados neste texto não dependem de máquina.

Uma **máquina de Turing probabilística** M é definida como uma máquina em que cada transição tem dois movimentos legais δ_0 e δ_1, entretanto, a cada transição durante uma computação só um deles é escolhido como o próximo passo; dado um estado q e um símbolo da fita σ a máquina executa a transição $\delta_i(q,\sigma)$ para uma escolha aleatória $i \in \{0,1\}$. A resposta da máquina para uma entrada w é uma variável aleatória que depende de w; nesse caso, consideramos a distribuição da variável aleatória $M(w)$ da máquina probabilística M com entrada fixa $w \in \Sigma^*$ em que a probabilidade é tomada sobre as escolhas aleatórias feitas por M. O espaço de probabilidades é formado por todas as sequências binárias que representam as escolhas internas da máquina numa computação desde a configuração inicial no estado q_0 até alcançar q_{para}. Para uma tal sequência de comprimento m associamos a probabilidade 2^{-m}. Na computação de M com entrada w a probabilidade

$$\mathbb{P}[\, M(w) = z \,]$$

é a soma das probabilidades de todas as sequências de escolhas aleatórias internas que terminam com a palavra $z \in \Sigma^*$ escrita na fita.

Notemos que para uma uma entrada w fixa o número de bits aleatórios usados numa computação com w pode variar, entretanto, para toda sequência binária r que representa as escolhas aleatórias numa computação com w, não existe uma sequência r' que também representa as escolhas aleatórias numa computação e tal que r seja prefixo de r' ou vice-versa.

Uma máquina probabilística M **decide** L **com probabilidade de erro** ϵ **em tempo** T, para $T\colon \mathbb{N} \to \mathbb{N}$, se para toda palavra $w \in \Sigma^*$

$$\mathbb{P}[\, M(w) \neq L(w) \,] \leqslant \varepsilon$$

e $M(w)$ termina a computação em no máximo $T(|w|)$ transições para *qualquer que seja a sequência de bits aleatórios* usados na computação.

Uma alternativa à definição acima é dada a seguir pelo modelo chamado em algumas referências bibliográficas de máquina de Turing probabilística *off-line*: é uma máquina de Turing determinística que tem uma entrada auxiliar escrita numa fita auxiliar só de leitura, além da entrada principal escrita na fita principal. Essa fita auxiliar contém uma sequência B de bits aleatórios, cada posição tem um bit com distribuição uniforme e as ocorrências na sequência são independentes. Cada

posição da fita auxiliar é usada (lida) no máximo uma única vez. Notemos que o comprimento da entrada auxiliar pode ser definido como igual ao tempo T de execução da máquina. Nesse caso, consideramos a distribuição da variável aleatória $M(w, B)$ da máquina probabilística M com entrada fixa $w \in \Sigma^*$ e um vetor aleatório $B \in_R \{0,1\}^{T(|w|)}$. Uma máquina probabilística *off-line* M com tempo de execução T **decide** L **com probabilidade de erro** ϵ se para toda palavra $w \in \Sigma^*$

$$\mathbb{P}_{B \in_R \{0,1\}^{T(|w|)}}[M(w, B) \neq L(w)] \leqslant \varepsilon.$$

Os dois modelos são computacionalmente equivalentes (Leeuw *et al.*, 1970). Uma computação da máquina probabilística *off-line* N como definida acima pode ser simulada por uma máquina probabilística M que com entrada w seleciona internamente uma sequência de bits aleatórios b e computa $N(w, b)$. Por outro lado, a computação de uma máquina probabilística M é simulada pela máquina determinística N que realiza transições de $M(w)$ usando uma sequência de bits aleatórios b dados com entrada auxiliar como as escolhas aleatórias internas a M. De fato, mais do que o que foi dito vale, as definições são equivalentes no contexto do tempo de execução, assim, na sequência, vamos usar livremente o modelo probabilístico que for mais conveniente.

Com a definição de máquina *off-line*, a classe NP pode ser definida (verifique) como a classe das linguagens L para as quais existe um polinômio p e um algoritmo probabilístico M de tempo polinomial tal que todo $w \in \{0,1\}^*$

1. se $w \in L$ então

   $$\mathbb{P}_{B \in_R \{0,1\}^{p(|w|)}}[M(w, B) = L(w)] > 0,$$

   ou seja, se $w \in L$ então existe uma *entrada auxiliar* $B \in \{0,1\}^{p(|w|)}$ que faz M responder *sim*;

2. se $w \notin L$ então

   $$\mathbb{P}_{B \in_R \{0,1\}^{p(|w|)}}[M(w, B) = L(w)] = 1$$

   nesse caso, se $w \notin L$, M responde *não* qualquer que seja a *entrada auxiliar*.

Classes de complexidade probabilísticas As seguir, vamos definir as classes de complexidade de tempo polinomial que estão associadas as algoritmos probabilísticos.

A classe RP — *Randomized Polynomial time* — é formada pelas linguagens L decididas por um algoritmo probabilístico M de tempo polinomial tal que para algum

polinômio p e para todo $w \in \{0,1\}^*$, a probabilidade de erro é

$$\mathop{\mathbb{P}}_{B \in_R \{0,1\}^{p(|w|)}}[M(w,B) \neq L(w)] \leqslant \begin{cases} 1/3, & \text{se } w \in L \\ 0, & \text{se } w \notin L \end{cases}$$

ou seja, se $w \in L$ então a M erra com probabilidade limitada, e se $w \notin L$ então M não erra. Assim, uma resposta *sim* do algoritmo M está sempre certa, isto é $w \in L$, enquanto que uma resposta *não* pode ser um falso negativo, o que ocorre com probabilidade no máximo 1/3.

O problema de determinar se um gafo G tem um corte pequeno, de tamanho menor que um dado k, está em RP pois o Algoritmo 9, página 75, pode ser executado duas vezes o que limita a probabilidade de erro a 1/4, além disso se o algoritmo acha um corte pequeno responde sim (corretamente), se não acha (o que não significa que não tenha) então responde não (com chance de erro).

A constante 1/3 nos algoritmos que definem RP é arbitrária. Se realizamos k execuções independentes desse algoritmo (com a mesma entrada), então uma resposta 1 é definitiva enquanto que k respostas 0 estão todas erradas com probabilidade menor que $(1/3)^k$ e se $k = O(n^c)$ para alguma constante positiva c então as execuções terminam em tempo polinomial. O mesmo vale se trocarmos 1/3 por qualquer constante $\varepsilon \in (0,1)$ de modo que se a probabilidade de erro for ε então podemos executar o algoritmo várias vezes (com a mesma entrada) até que a probabilidade de erro fique menor que 1/3, o que certifica a pertinência em RP.

Observemos que se M decide uma linguagem L que pertence a RP então as escolhas de bits aleatórios por $M(w)$ que terminam em $M(w) = 1$ é um certificado c para $w \in L$, portanto, podemos concluir que L pertence a NP.

Proposição 4.6 $RP \subseteq NP$. □

A classe coRP é a classe das linguagens L tais que $\overline{L} \in RP$, isto é, a classe das linguagens L para as quais existe um algoritmo probabilístico M de tempo polinomial tal que uma resposta *não* está sempre certa enquanto que uma resposta *sim* pode ser um falso positivo. Para algum polinômio p e para todo $w \in \{0,1\}^*$,

$$\mathop{\mathbb{P}}_{B \in_R \{0,1\}^{p(|w|)}}[M(w,B) \neq L(w)] \leqslant \begin{cases} 1/3, & \text{se } w \notin L \\ 0, & \text{se } w \in L \end{cases}$$

Nos algoritmos 11 e 13 (pág. 77 e seguintes) para, respectivamente, os testes de igualdade do produto de matrizes e de identidade polinomial, uma resposta *não* está sempre certa enquanto que uma resposta *sim* pode estar errada, ademais duas rodadas independentes de tais algoritmos reduzem a probabilidade de erro para o limiar da definição da classe coRP, logo as linguagens definidas por esses exemplos

são linguagens em coRP. O problema de testar se um número é primo também está em coRP como atesta o algoritmo de Miller–Rabin (página 168), que também atesta que o problema de testar se um número é composto está em RP.

Um fato interessante ocorre quando consideramos a possibilidade de uma linguagem $L \in \mathsf{RP} \cap \mathsf{coRP}$, pois ela pode se beneficiar dos resultados exatos do algoritmo M_{RP}, o que prova que $L \in \mathsf{RP}$, e do algoritmo M_{coRP}, o que prova que $L \in \mathsf{coRP}$, para especificar um algoritmo probabilístico M' que nunca erra, porém uma escolha ruim de bits aleatórios pode levar o algoritmo a executar por muito tempo: dada uma entrada $w \in \{0,1\}^*$

```
1 enquanto verdadeiro faça
2 |   se M_RP(w) = 1 então responda 1;
3 |   se M_coRP(w) = 0 então responda 0.
```

Algoritmo 39: $\mathsf{RP} \cap \mathsf{coRP}$

Se M_{RP} responder 1 então por definição garantimos que $w \in L$. Analogamente, se M_{coRP} responder 0 então por definição garantimos que $w \notin L$. Assim nunca obtemos uma resposta errada. Porém não temos como saber exatamente quantas rodadas vamos esperar até que um certificado apropriado seja encontrado.

Um algoritmo probabilístico tem **tempo esperado** $T(n)$ se a variável aleatória t_w, que denota o tempo de execução do algoritmo com entrada w, tem valor esperado $\mathbb{E}\, t_w \leqslant T(|w|)$ para todo w. A esperança é computada sobre os bits aleatórios usados na computação pelo algoritmo. Nos casos em que T é um polinômio dizemos que a máquina é de **tempo esperado polinomial**. Por exemplo, o algoritmo aproximativo para o problema MAX-E3SAT, Algoritmo 19 na página 134, tem tempo esperado polinomial.

No Algoritmo 39 acima, o número esperado de execuções até o passo 2 ter sucesso é no máximo 2 e o mesmo vale para o passo 3, portanto o tempo esperado de M' é no máximo um número constante de simulações de $M_{\mathsf{RP}}(w)$ e de $M_{\mathsf{coRP}}(w)$, ou seja, o tempo esperado de execução é polinomial em $|w|$.

A classe ZPP — *Zero-error Probabilistic Polynomial time* — é a classe de complexidade de todas as linguagens para as quais existe um algoritmo probabilístico M de tempo *esperado* polinomial e tal que $\mathbb{P}[\,M(w,B) = L(w)\,] = 1$, para todo $w \in \{0,1\}^*$. Equivalentemente, ZPP é a classe das linguagens para as quais existe um algoritmo probabilístico de tempo polinomial M tal que, para todo w, $M(w) \in \{0,1,\perp\}$ em que 0 significa $w \notin L$ e 1 significa $w \in L$, como é usual, e $\perp$ significa "não sei" e $\mathbb{P}[\,M(w) = \perp\,] \leqslant 1/2$. Intuitivamente, entendemos essa definição como que em tempo polinomial ou o algoritmo decide ou interrompe a execução.

O Algoritmo 39 acima mostra que

$$\mathsf{RP} \cap \mathsf{coRP} \subseteq \mathsf{ZPP} \tag{4.2}$$

mas nesse caso vale a igualdade dessas classes como demonstra o seguinte resultado.

Lema 4.7 $\mathsf{ZPP} = \mathsf{RP} \cap \mathsf{coRP}$.

Demonstração. De (4.2), só precisamos provar que $\mathsf{ZPP} \subseteq \mathsf{RP} \cap \mathsf{coRP}$.

Seja $\mathsf{L} \in \mathsf{ZPP}$ uma linguagem e M um algoritmo probabilístico de tempo esperado $p(n)$ que decide pertinência em L sem errar. Para provar pertinência em RP, definimos um algoritmo N que computa da seguinte maneira: dada uma entrada $w \in \{0,1\}^*$

> simula M com entrada w até no máximo $3p(|w|)$ passos;
> **se** $\mathsf{M}(w) = 1$ **então responda** 1;
> **senão responda** 0.

Algoritmo 40: $\mathsf{N}(w)$

Se $w \notin \mathsf{L}$ então N termina sem aceitar w, portanto $\mathbb{P}[\,\text{erro}\,] = \mathbb{P}[\,\mathsf{N}(w) = 1\,] = 0$. Por outro lado, se $w \in \mathsf{L}$ então o algoritmo N aceita w (corretamente) ou termina pelo limite dos $3p(|w|)$ passos. No segundo caso $\mathsf{N}(w) = 0$ e a resposta está errada. A probabilidade da resposta errada é $\mathbb{P}[\,t_w > 3p(|w|)\,]$ em que, como acima, t_w é a variável aleatória para o tempo de execução do algoritmo M com entrada w. Pela desigualdade de Markov (Exercício 2.43 na página 131),

$$\mathbb{P}[\,t_w \geqslant 3p(|w|)+1\,] \leqslant \frac{p(|w|)}{3p(|w|)+1} < \frac{1}{3}$$

portanto, se $w \in \mathsf{L}$ então $\mathbb{P}[\,\text{erro}\,] = \mathbb{P}(\,\mathsf{N}(w) = 0\,) < 1/3$ e com isso temos que $\mathsf{L} \in \mathsf{RP}$.

Do maneira análoga, podemos provar que $\mathsf{ZPP} \subseteq \mathsf{coRP}$; definimos um algoritmo N' que com entrada w

> simula M com entrada w até no máximo $3p(|w|)$ passos;
> **se** $\mathsf{M}(w) = 0$ **então responda** 0;
> **senão responda** 1.

Algoritmo 41: $\mathsf{N}'(w)$

Assim, se $w \in \mathsf{L}$ então $\mathbb{P}[\,\text{erro}\,] = \mathbb{P}[\,\mathsf{N}(w) = 0\,] = 0$. Se $w \notin \mathsf{L}$ então o algoritmo pode aceitar erroneamente e $\mathbb{P}[\,\text{erro}\,] = \mathbb{P}[\,t_w > 3p(|w|)\,] < 1/3$. Portanto, $\mathsf{ZPP} \subseteq \mathsf{RP} \cap \mathsf{coRP}$. □

NP é tão fácil quanto detectar soluções únicas Valiant e Vazirani, 1986, em "NP is as easy as detecting unique solutions", mostraram que, se existe um algoritmo de tempo polinomial para USAT, a linguagem das fórmulas booleanas com exatamente uma valoração que a satisfaz, então $\mathsf{NP} = \mathsf{RP}$.

A demonstração desse resultado envolve o projeto de um algoritmo, uma *redução probabilística de tempo polinomial*, que recebe uma fórmula booleana φ e devolve uma fórmula booleana ϕ tal que, se φ é satisfazível, então ϕ é, com probabilidade positiva, satisfazível; caso contrário, se φ não é satisfazível, então ϕ não é satisfazível. Ou seja, existe um algoritmo A, probabilístico e de tempo polinomial, tal que para toda fórmula booleana φ

$$\begin{aligned} &\text{se } \varphi \in \textsc{sat}, \text{ então } \mathbb{P}[\,A(\varphi) \in \textsc{usat}\,] \geqslant \frac{1}{8n} \\ &\text{se } \varphi \notin \textsc{sat}, \text{ então } \mathbb{P}[\,A(\varphi) \in \textsc{sat}\,] = 0. \end{aligned} \tag{4.3}$$

A ideia para construir esse algoritmo é a seguinte: se φ tem n variáveis e $\approx 2^k$ valorações delas satisfazem φ e, além disso $h\colon \{0,1\}^n \to \{0,1\}^{k+2}$ é uma função de uma família *hash* 2-universal, então podemos construir uma fórmula $\phi = \varphi \wedge (h = 0)$ tal que, com probabilidade positiva, o número de valorações v tais que $\phi(v) = 1$ e $h(v) = 0$ é 1. Finalmente, instanciando o algoritmo para $\textsc{usat}$ com $\phi = A(\varphi)$, ele responde 1se ϕ for satisfazível por uma única valoração e, caso contrário, responde 0 com probabilidade 1. Assim, temos um algoritmo probabilístico que põe em RP a linguagem $\textsc{sat}$. A existência de um algoritmo RP para $\textsc{sat}$ implica $\mathsf{NP} \subseteq \mathsf{RP}$ e assim $\mathsf{RP} = \mathsf{NP}$.

Proposição 4.8 *Sejam $S \subseteq \{0,1\}^n$ tal que $2^k \leqslant |S| < 2^{k+1}$ e $\mathcal{H}$ uma família de funções de* hash *2-universal definidas em $\{0,1\}^n$ e com valores em $\{0,1\}^{k+2}$. Então,*

$$\mathop{\mathbb{P}}_{h \in_R \mathcal{H}}[\,|x \in S\colon h(x) = 0| = 1\,] \geqslant \frac{1}{8}$$

em que $0 = (0,0,\ldots,0)$.

Demonstração. Sejam S e $\mathcal{H}$ como no enunciado da proposição. A definição de *hashing* 2-universal (seção 5.2.2) diz que

$$\mathop{\mathbb{P}}_{h \in \mathcal{H}}([h(x) = i] \cap [h(y) = j]) = \frac{1}{2^{2(k+2)}},$$

para quaisquer x, y distintos no domínio da função e para quaisquer i, j no contradomínio. Decorre disso que $\mathbb{P}_{h\in\mathcal{H}}[h(x) = 0] = 2^{-(k+2)}$ para todo $x \in \{0,1\}^n$ e $0 = (0,0,\ldots,0) \in \{0,1\}^{k+2}$. Agora, para $y \in S$ e $y \neq x$

$$\mathop{\mathbb{P}}_{h \in_R \mathcal{H}}[\,h(y) \neq 0 \mid h(x) = 0\,] = \mathop{\mathbb{P}}_{h \in_R \mathcal{H}}[h(y) \neq 0] = \frac{1}{2^{k+2}},$$

portanto

$$\mathop{\mathbb{P}}_{h \in_R \mathcal{H}}[\,\exists y,\, h(y) \neq 0 \mid h(x) = 0\,] \leqslant \sum_{y \in S \setminus \{x\}} \frac{1}{2^{k+2}} = \frac{|S| - 1}{2^{k+2}} \leqslant \frac{1}{2}$$

e, então

$$\mathop{\mathbb{P}}_{h\in_R\mathcal{H}}[\,\forall y \in S \setminus \{x\},\, h(y) \neq 0 \mid h(x) = 0\,] \geqslant \frac{1}{2}.$$

Disso, podemos deduzir da lei do produto que

$$\mathop{\mathbb{P}}_{h\in_R\mathcal{H}}([\forall y \in S \setminus \{x\},\, h(y) \neq 0] \cap [h(x) = 0]) \geqslant \frac{1}{2}\frac{1}{2^{k+2}} = \frac{1}{2^{k+3}}.$$

Para concluir, observamos que a probabilidade de haver um único elemento de S mapeado para 0 é dada pela soma sobre $x \in S$ das probabilidades de que x seja o único elemento mapeado para 0. Esses eventos são disjuntos, então a probabilidade de um elemento único ser mapeado para 0 é pelo menos $|S|/2^{k+3} \geqslant 1/8$. □

Vamos aplicar esse lema a uma família específica de funções de *hash* 2-universal. Embora os resultados apresentados a seguir não dependam de nenhuma especificidade dessa família, ela facilita a codificação da função como uma expressão booleana. A família

$$\mathcal{H}_{n,k} := \left\{ h_{A,b} = Ax + b \colon A \in \{0,1\}^{k\times n} \text{ e } b \in \{0,1\}^k \right\}$$

com as operações módulo 2 é 2-universal (Exercício 5.21).

Lema 4.9 *Existe um algoritmo probabilístico de tempo polinomial que, dados uma fórmula booleana φ e um inteiro positivo k, produz uma fórmula ϕ tal que: (1) se φ não for satisfazível, então ϕ também não é satisfazível; (2) se φ tiver pelo menos 2^k e menos que 2^{k+1} valorações que a satisfazem, então, com probabilidade pelo menos $1/8$, a fórmula ϕ tem exatamente uma valoração que a satisfaz.*

Demonstração. Seja φ uma fórmula sobre n variáveis. O algoritmo

1. sorteia, aleatória e uniformemente, $a_1, \ldots, a_{k+2} \in \{0,1\}^n$ e bits $b_1, \ldots, b_{k+2}$;
2. escreve a fórmula ϕ equivalente à expressão:

$$\varphi(x) \wedge (\ell_1(x) = 0) \wedge (\ell_2(x) = 0) \wedge \cdots \wedge (\ell_{k+2}(x) = 0)$$

   em que, para todo i, $\ell_i(x) = \sum_l a_{ij} \cdot x_j + b_i$ (com as operações módulo 2) são as linhas que resultam da aplicação da função $h_{A,b}$ em que A é a matriz cujas linhas são os a_i's e b é a matriz coluna cujas linhas são os b_i's.

Se φ não for satisfazível, então ϕ também não será satisfazível, quaisquer que sejam as escolhas de a_i e b_i, para todo i. Senão, a quantidade valorações que satisfazem ϕ é igual a quantidade de $x \in \{0,1\}^n$ que satisfazem φ e tais que $\ell_1(x) = \ell_2(x) = \cdots \ell_{k+2}(x) = 0$.

Se φ tiver entre 2^k e 2^{k+1} valorações que a tornam verdadeira, então a proposição acima implica que, com probabilidade de pelo menos 1/8, haverá uma e somente uma atribuição que satisfaz ϕ. □

Para finalizar, seja A o algoritmo probabilístico que recebe uma fórmula booleana φ sorteia $k \in \{0,1,\ldots,n-1\}$, sorteia $h_{A,k} \in \mathcal{H}_{n,k}$ e escreve ϕ de acordo com (1) e (2) do lema acima. Esse algoritmo acerta o valor de k com probabilidade $1/n$. Condicionado ao fato do valor correto de k, ou seja, temos a hipótese do item (2) do lema, temos uma única valoração que satisfaz ϕ com probabilidade 1/8. Assim, verificamos (4.3).

Agora, se B é o algoritmo para o USAT, então B(A(φ)) é um algoritmo que, dado uma fórmula booleana satisfazível φ, responde 1 com probabilidade $1/8n$, e, dado uma fórmula booleana não satisfazível φ, responde 0 com probabilidade 1.

TEOREMA 4.10 (VALIANT E VAZIRANI, 1986) *Se existe um algoritmo de tempo polinomial que decide* USAT*, então,* NP = RP. □

BPP— *Bounded-error Probabilistic Polynomial time* A classe BPP é a classe das linguagens decididas por algoritmos probabilísticos de tempo polinomial com probabilidade de erro 1/3, ou seja, $L \in BPP$ se existe um polinômio p e um algoritmo probabilístico M de tempo polinomial tal que para todo $w \in \{0,1\}^*$

$$\mathbb{P}_{B \in_R \{0,1\}^{p(|w|)}}[M(w,B) \neq L(w)] \leqslant 1/3.$$

A probabilidade de erro 1/3 na definição não tem nada de especial, qualquer constante $\varepsilon \in (0,1/2)$ serviria para definir a mesma classe de linguagens, como veremos no Lema 4.11 abaixo a probabilidade de erro pode ser feita menor que qualquer constante. Se M é uma máquina probabilística de tempo polinomial que aceita a linguagem L com probabilidade de erro ε, então podemos escrever uma máquina N probabilística e de tempo polinomial que aceita a mesma linguagem L com probabilidade de erro 1/3.

LEMA 4.11 *Para toda constante $0 < \varepsilon < 1/2$, todo polinômio p, toda linguagem $L \subseteq \{0,1\}^*$ e todo algoritmo probabilístico M de tempo polinomial que decide L com probabilidade de erro ε, existe um algoritmo probabilístico N de tempo polinomial e que decide L com probabilidade de erro $2^{-p(n)}$ nas entradas de tamanho n, para todo n.*

DEMONSTRAÇÃO. Sejam ε, p, L e M como no enunciado. Definimos

$$\delta := 4\varepsilon(1-\varepsilon) \text{ e } k(n) := \left\lceil \frac{p(n)}{\log_2(1/\delta)} \right\rceil$$

e notemos que $\delta < 1$ pois $\varepsilon < 1/2$.

Consideremos o algoritmo N que com entrada $w \in \{0,1\}^*$ simula $M(w)$ um número ímpar de vezes e decide pela resposta dada na maioria das simulações:

```
1 para i de 1 até 2k+1 faça
2 |   m_i ← M(w);
3 se ∑_i m_i > k então responda 1;
4 senão responda 0.
```

Certamente, N é uma algoritmo probabilístico de tempo polinomial. Seja $S = S(w) \in \{0,1\}^{2k+1}$ uma sequência de respostas dadas na linha 2 em uma execução de N com entrada w. Sejam $c = c(S)$ e $e = e(S)$ o número de respostas certas e respostas erradas, respectivamente, em S, onde resposta certa quer dizer que M decidiu corretamente a pertinência de w em L.

O algoritmo N responde errado se para a sequência S correspondente a uma execução de N ocorre $c < e$. Queremos limitar a probabilidade desse evento. Como M tem probabilidade de erro limitada por $\varepsilon < 1/2$, a probabilidade de uma sequência S que faz N responder errado é no máximo $\varepsilon^e(1-\varepsilon)^c \leqslant \varepsilon^{k+1}(1-\varepsilon)^k$ pois $e \geqslant k+1$ e $\varepsilon < 1-\varepsilon$. Assim, a probabilidade de erro é

$$\mathbb{P}[\,N(w) \neq L(w)\,] \leqslant \sum_S \varepsilon^{e(S)}(1-\varepsilon)^{c(S)} \leqslant 2^{2k+1}\varepsilon^{k+1}(1-\varepsilon)^k = 2\varepsilon(2^2\varepsilon(1-\varepsilon))^k < (4\varepsilon(1-\varepsilon))^k = \delta^k$$

em que a soma é sobre toda sequência S de respostas de M que faz N responder errado. Pela escolha de k e como $\log_2(1/\delta) = 1/\log_\delta(1/2)$ temos $\delta^k \leqslant 2^{-p(|w|)}$. □

Da definição das classes deduzimos que

$$\mathsf{RP}, \mathsf{coRP}, \mathsf{ZPP} \subseteq \mathsf{BPP}.$$

Exemplo 4.12 (PIT∈BPP). Seja $p \in \mathbb{F}[x_1, \ldots, x_n]$ um polinômio dado por um circuito aritmético, como definido na página 204. O PIT é o problema da identidade polinomial: decidir se o polinômio p definido pelo circuito $\langle p \rangle$ é identicamente nulo.

Comentamos na seção 1.7.3 que é um problema importante na teoria da computação sabermos se é possível resolvê-lo em tempo polinomial no tamanho do circuito. Também comentamos que, a princípio, temos dois problemas computacionais: dado um polinômio p, EZE – *Evaluates to Zero Everywhere* – é o problema de decidir se, como função, p vale zero em todo elemento do corpo e PIT que é decidir se p na forma canônica tem todos os coeficientes nulos. EZE está em coNP, pois está em coRP como mostra o algoritmo 13. De fato é coNP-difícil, pois é possível escrever uma fórmula 3-CNF como um polinômio sobre $\mathbb{F}_2$.

Um circuito aritmético de tamanho m tem profundidade no máximo m, portanto realiza no máximo m multiplicações, logo define um polinômio de grau no máximo 2^m. Nessa situação, o Algoritmo 13, página 82, sorteia n valores em $\{1, \ldots, 2^{m+2}\}$,

avalia o valor de p com esses valores simulando o circuito aritmético em tempo polinomial em m e resolve PIT com probabilidade de erro 1/4. Porém, a sequência sorteada $(x_1,\ldots,x_n)$ tem tamanho $O(nm)$ bits enquanto que x^{2^m} calculado em 2^{m+2} resulta num número com $O(m2^m)$ bits, logo só para escrevê-lo o tempo consumido seria exponencialmente grande, o que não resulta num algoritmo de tempo polinomial para o problema.

Um modo de contornarmos esse problema é calcularmos as operações aritméticas nas portas do circuito módulo um inteiro positivo k apropriado, resultando, no final do cômputo, $p(x_1,\ldots,x_n) \bmod k$. Com isso o tempo de simulação do circuito continua polinomial e os números que ocorrem nas operações têm tamanho controlado, mas aumenta a probabilidade de erro pois podemos ter $p(x_1,\ldots,x_n) \neq 0$ e $p(x_1,\ldots,x_n) \bmod k = 0$, caso k divida $p(x_1,\ldots,x_n)$. Agora, devemos estimar essa probabilidade de erro e fazê-la pequena usando rodadas independentes de escolhas para k.

Tomemos $k \in \{1,2,\ldots,2^{2m}\}$. A quantidade de números primos nesse conjunto é, pelo Teorema dos Números Primos, maior que

$$\frac{2^{2m}}{2m+2}$$

se $m > 2$ (Rosser, 1941).

Assumamos que $p := p(x_1,\ldots,x_n) \neq 0$. A quantidade de fatores primos distintos em p é $(m+2)2^m$, pois se $p = p_1^{\alpha_1} p_2^{\alpha_2} \cdots p_t^{\alpha_t}$ então $p \geqslant 2^t$, portanto $t \leqslant \log_2 p \leqslant \log_2 (2^{m+2})^{2^m} = (m+2)2^m$. Assim, a quantidade de primos em $\{1,2,\ldots,2^{2m}\}$ que não é um dos t fatores primos de p é maior que

$$\frac{2^{2m}}{2m+2} - (m+2)2^m > \frac{2^{2m}}{8m}$$

para todo $m > 7$, de modo que a probabilidade com que uma escolha aleatória em $\{1,2,\ldots,2^{2m}\}$ resulte num número *bom*, isto é um primo que não divide m, é maior que $1/8m$. Em r sorteios para o valor de k, basta um deles resultar num número *bom* para podermos responder que $p(x_1,\ldots,x_n) \neq 0$. A probabilidade com que nenhum dentre $r := 16m$ sorteios resulte num número *bom* é no máximo

$$\left(1 - \frac{1}{8m}\right)^r = \left(\left(1 - \frac{1}{8m}\right)^{-8m}\right)^{-2} < 0{,}15$$

para todo inteiro $m > 0$, pois $(1 - 1/8m)^{-8m} \geqslant$ e (veja (s.8)). Portanto, se $\langle p \rangle$ é nulo o algoritmo sempre descobre, caso contrário o polinômio não nulo é declarado nulo com probabilidade 0,15, logo PIT $\in$ BPP. $\Diamond$

Notemos que P $\subseteq$ BPP pois para todo algoritmo de tempo polinomial M podemos escrever um algoritmo probabilístico de tempo polinomial M′ que simula M e ignora os bits aleatórios. Não sabemos se a recíproca vale ou não

Problema 8. $\mathsf{BPP} \subseteq \mathsf{P}$?

Também não sabemos situar BPP com relação a NP

Problema 9. $\mathsf{BPP} \subseteq \mathsf{NP}$?

não sabemos se $\mathsf{BPP} \subseteq \mathsf{NP}$, se $\mathsf{NP} \subseteq \mathsf{BPP}$ ou se nenhuma dessas duas relações valem. Também, não conhecemos nenhuma linguagem que é completa para BPP. Agora, é um exercício fácil, que deixamos para o leitor, verificar que $\mathsf{BPP} = \mathsf{coBPP}$.

Problema 10. Alguma(s) das inclusões $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subseteq \mathsf{BPP}$ são próprias?

Os algoritmos que reconhecem linguagens em BPP são chamados, em algumas referências bibliográficas, de **Atlantic city**, os de RP e coRP são conhecidos como **Monte-Carlo** e os de ZPP são os **Las Vegas**.

Finalizamos essa seção enunciado o seguinte resultado sem prova.

Teorema. (Impagliazzo e Wigderson, 1999) *Se existe uma linguagem* L *decidida em tempo (determinístico)* $2^{O(n)}$ *que nas palavras de tamanho* n *requer circuito de tamanho* $2^{\Omega(n)}$, *então* $\mathsf{P} = \mathsf{BPP}$.

Exercício 4.13. Prove que se $\mathsf{L} \in \mathsf{BPP}$, então existe um algoritmo probabilístico que com entrada w decide se $w \in \mathsf{L}$ com probabilidade de erro menor que $1/(3m)$ e em tempo polinomial em $|w|$, em que $m = m(|w|)$ é o número (polinomial) de bits aleatórios usados na computação (dica: ajuste a quantidade de rodadas independentes de simulações no algoritmo acima).

$\mathsf{BPP} \subseteq \mathsf{P/poly}$ O próximo resultado mostra que toda linguagem decidida por algoritmo probabilístico de tempo polinomial também é decidida por circuitos de tamanho polinomial. Como circuito é um modelo não-uniforme de computação e como há muitas sequências binárias aleatórias que testemunham a favor da decisão correta, podemos escolher uma tal sequência para cada n e todo $w \in \{0,1\}^n$ e projetar os circuitos com as sequências escolhidas. Essa ideia está formalizada no teorema a seguir. A inclusão é própria porque há problemas não decidíveis por algoritmos que são decididos por família de circuitos de tamanho polinomial, como já dissemos acima (Exercício 4.23, página 241).

Teorema 4.14 (Adleman, 1978) $\mathsf{BPP} \subsetneq \mathsf{P/poly}$.

Demonstração. Sejam $\mathsf{L} \in \mathsf{BPP}$ e M um algoritmo probabilístico de tempo polinomial que decide L com probabilidade de erro exponencialmente pequena

$$\mathop{\mathbb{P}}_{\mathsf{B} \in_R \{0,1\}^{p(n)}} [\mathsf{M}(w, \mathsf{B}) \neq \mathsf{L}(w)] < 2^{-n}$$

nas entradas de tamanho n, para algum polinômio p. A existência desse algoritmo é garantido pelo Lema 4.11 acima.

Fixado um inteiro positivo n, podemos afirmar que existe uma sequência binária b que é um certificado de pertinência em L para toda entrada $w \in \mathsf{L}$ de tamanho n. De fato,

$$\mathop{\mathbb{P}}_{\mathsf{B}\in_{\mathrm{R}}\{0,1\}^{p(n)}}\left[\bigcup_{w\in\{0,1\}^n}[\mathsf{M}(w,\mathsf{B})\neq\mathsf{L}(w)]\right] < \sum_{w\in\{0,1\}^n} 2^{-n} = 1$$

logo, com probabilidade maior que 0 para pelo menos um $\mathsf{B} \in \{0,1\}^{p(n)}$ fixo o algoritmo M responde corretamente para todo $w \in \{0,1\}^n$, ou seja, existe (ao menos) uma sequência $b_n \in \{0,1\}^{p(n)}$, com a qual M não erra nas entradas de tamanho n. O algoritmo (determinístico) M' dado por

$$\mathsf{M}'(w) := \mathsf{M}(w, b_n)$$

não erra em entradas de tamanho n.

A partir de M' construímos, para cada n, um circuito booleano C_n de tamanho polinomial, conforme construção do Exercício 4.4 (página 204), tal que $\mathsf{C}_n(w) = \mathsf{M}'(w)$, para todo w de tamanho n. Dessa forma, a família de circuitos $(\mathsf{C}_n : n > 0)$ decide L, portanto $\mathsf{L} \in \mathsf{P/poly}$. □

4.2.2 BPP ESTÁ NA HIERARQUIA POLINOMIAL

A hierarquia polinomial é uma hierarquia formada por classes de problemas com complexidade polinomial, as *classes do nível* i são denotadas por Σ_i e Π_i, para todo $i \in \mathbb{N}$. Essas classes generalizam P e NP de um certo modo natural. Uma motivação para essa hierarquia pode ser lida no capítulo 17 de Papadimitriou (1994), assim como outros resultados que estão fora do escopo deste texto pois precisam de vários pré-requisitos da Complexidade Computacional. Nesta seção, falaremos brevemente sobre a hierarquia polinomial, sua relação com as classes probabilísticas e daremos alguns outros resultados sem prova para que o leitor possa ter alguma referência sobre a importância de PH e sua relação com outras classes de complexidade.

Por enquanto vamos nos concentrar nos primeiros níveis da hierarquia e em seguida mostrar uma relação com a classe BPP. Começamos declarando que

$$\Sigma_0 = \Pi_0 := \mathsf{P} \quad \text{e} \quad \Sigma_1 := \mathsf{NP} \quad \text{e} \quad \Pi_1 := \mathsf{coNP}$$

ou seja, por definição Σ_0 é a classe das linguagens L para as quais existe um algoritmo M de tempo polinomial tal que para todo $w \in \{0,1\}^*$

$$w \in \mathsf{L} \Leftrightarrow \mathsf{M}(w) = 1,$$

Σ_1 é a classe das linguagens L para as quais existe um algoritmo M de tempo polinomial e um polinômio p tais que para todo $w \in \{0,1\}^*$

$$w \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|w|)},\ M(w,u) = 1,$$

definimos Σ_2 como a classe das linguagens L para as quais existe um algoritmo M de tempo polinomial e polinômios p e q tais que para todo $w \in \{0,1\}^*$

$$w \in L \Leftrightarrow \exists u \in \{0,1\}^{p(|w|)}\ \forall v \in \{0,1\}^{q(|w|)},\ M(w,u,v) = 1. \tag{4.4}$$

Exemplo 4.15 (*conjunto independente máximo*). Em um grafo G, um conjunto $U \subseteq V(G)$ é dito *independente* (ou *estável*) se $|e \cap U| \leqslant 1$ para toda aresta $e \in E(G)$, isto é, U não contêm os dois vértices de qualquer aresta do grafo. Um conjunto independente é *máximo* se tem cardinalidade

$$\alpha(G) := \max\{|U| : U \subseteq V(G) \text{ é independente}\}.$$

Decidir se um grafo G tem conjunto independente de cardinalidade (pelo menos) k está em NP: dados G e U é possível verificarmos em tempo polinomial se U é um subconjunto com pelo menos k vértices de G e independente e, se esse é o caso, então uma codificação de U tem tamanho polinomial no tamanho de G. Agora, um tal conjunto independente é máximo em G se o grafo não tem conjunto independente de cardinalidade $k+1$. Decidir se um grafo G não tem conjunto independente de cardinalidade $k+1$ está em coNP (por quê?).

Decidir "$\alpha(G) = k$?" tem um certificado curto (tamanho polinomial) para uma parte do problema e não tem um certificado curto para a outra parte do problema. A linguagem

$$\textsc{max-ind} := \{\langle G,k\rangle : G \text{ é um grafo com } \alpha(G) = k\} \tag{4.5}$$

está em Σ_2. De fato, escrevendo na forma (4.4), um par $\langle G,k\rangle$ está na linguagem se e somente se existe um conjunto independente U de cardinalidade k e todo subconjunto V de cardinalidade (pelo menos) $k+1$ não é independente. Deve ficar claro ao leitor que a codificação dos subconjuntos tem tamanho polinomial no tamanho do grafo. Ainda não sabemos se a linguagem descrita em (4.5) acima está ou não está em $\mathsf{NP} = \Sigma_1$. ◊

O próximo teorema é o principal resultado dessa seção. A demonstração do teorema é de Lautemann (1983). Usaremos a notação $a \oplus u$ para denotar a operação *ou exclusivo* (ou soma módulo 2) coordenada-a-coordenada das sequências binárias $a,u \in \{0,1\}^m$. Também, usaremos o Exercício 1.44, página 45, que afirma que se $X \in_R \{0,1\}^m$ e $Y \in_D \{0,1\}^m$ é uma variável aleatória com distribuição arbitrária sobre $\{0,1\}^m$ então $X \oplus Y \in_R \{0,1\}^m$.

Teorema 4.16 (Sipser–Gács–Lautemann, 1983) $\mathsf{BPP} \subseteq \Sigma_2$.

A ideia da prova é que se $w \in \mathsf{L}$ e $\mathsf{L} \in \mathsf{BPP}$ então há um algoritmo M para o qual quase toda sequência de um conjunto $\{0,1\}^m$ (m polinomial em $|w|$) é um certificado disso de modo que, dado $r \in \{0,1\}^m$, é certo que alguma translação de r tomada de um número polinomial delas, digamos $r \oplus u_1, \ldots, r \oplus u_m$, é um certificado para w, ou seja, $\mathsf{M}(w, r \oplus u_i) = 1$. Por outro lado, se $w \notin \mathsf{L}$, então uma fração ínfima dos elementos de $\{0,1\}^m$ certificaria erroneamente a pertinência de x em L de modo que é possível haver $r \in \{0,1\}^m$ tal que nenhuma das translações $r \oplus u_1, \ldots, r \oplus u_m$ é um certificado, ou seja, para todo i temos $\mathsf{M}(w, r \oplus u_i) = 0$. Essa afirmação é capturada na sentença (4.6) abaixo que caracteriza L como uma linguagem de Σ_2

$$w \in \mathsf{L} \Leftrightarrow \exists u_1, u_2, \ldots, u_m \in \{0,1\}^m \forall r \in \{0,1\}^m, \bigvee_{i=1}^{m} \mathsf{M}(w, r \oplus u_i) = 1 \tag{4.6}$$

em que $\bigvee$ denota o operador *ou* lógico dos m termos "$\mathsf{M}(w, r \oplus u_i) = 1$". Além disso, o predicado $\bigvee_{i=1}^{m} \mathsf{M}(w, r \oplus u_i) = 1$ é uma computação de tempo polinomial, pois M é executada em tempo polinomial e um número polinomial de vezes. Feito isso estabeleceremos que $\mathsf{L} \in \Sigma_2$.

Demonstração do Teorema 4.16. Sejam $\mathsf{L} \in \mathsf{BPP}$ e M um algoritmo probabilístico de tempo $q(n)$, que nas entradas de tamanho n usa $m = m(n)$ bits aleatórios e erra com probabilidade menor que $1/(3m)$, tal algoritmo existe pelo Exercício 4.13, página 220. Vamos provar que vale a equação (4.6). Suponhamos que $w \in \mathsf{L}$. Então

$$\mathop{\mathbb{P}}_{U_1,\ldots,U_m \in_R \{0,1\}^m} [\exists r \in \{0,1\}^m \forall i \in \{1,\ldots,m\},\ \mathsf{M}(w, U_i \oplus r) = 0] \leqslant$$
$$\sum_{r \in \{0,1\}^m} \mathop{\mathbb{P}}_{U_1,\ldots,U_m \in_R \{0,1\}^m} [\forall i \in \{1,\ldots,m\},\ \mathsf{M}(w, U_i \oplus r) = 0] \leqslant 2^m \left(\frac{1}{3m}\right)^m < 1$$

portanto, qualquer que seja $r \in \{0,1\}^m$, existe uma sequência $u_1, \ldots, u_m \in \{0,1\}^m$ tal que vale $\bigvee_{i=1}^{m} \mathsf{M}(w, r \oplus u_i) = 1$.

Agora, suponhamos que $w \notin \mathsf{L}$. Fixada uma sequência $u_1, \ldots, u_m \in \{0,1\}^m$

$$\mathop{\mathbb{P}}_{R \in_R \{0,1\}^m} [\exists i \in \{1,\ldots m\},\ \mathsf{M}(w, R \oplus u_i) = 1] \leqslant$$
$$\sum_{i=1}^{m} \mathop{\mathbb{P}}_{R \in_R \{0,1\}^m} [\mathsf{M}(w, r \oplus u_i) = 1] \leqslant m \frac{1}{3m} = \frac{1}{3}$$

portanto, existe $r \in \{0,1\}^m$ tal que $\mathsf{M}(w, r \oplus u_1) = \mathsf{M}(w, r \oplus u_2) = \cdots = \mathsf{M}(w, r \oplus u_m) = 0$, o que prova a equação (4.6). □

A definição de BPP é simétrica, isto é, $\mathsf{BPP} = \mathsf{coBPP}$ de modo que do teorema acima temos $\mathsf{BPP} \subseteq \mathsf{co}\Sigma_2$. Definimos $\Pi_2 := \mathsf{co}\Sigma_2$ e, portanto, o Teorema 4.16 garante

que

$$\mathsf{BPP} \subseteq \Sigma_2 \cap \Pi_2.$$

Uma visão geral de PH A **hierarquia polinomial** é definida pela sequências de classes de complexidade: $\Sigma_0 = \Pi_0 = \mathsf{P}$ e

- para todo $i \geqslant 1$, Σ_i é a classe das linguagens L para as quais existe um algoritmo M de tempo polinomial e polinômios $p_1, p_2, \ldots, p_i$ tais que para todo $w \in \Sigma^*$

  $$w \in \mathsf{L} \Leftrightarrow \exists u_1,\ \forall u_2,\ \exists u_3,\ \ldots,\ \mathrm{Q}u_i,\ \ \mathsf{M}(w, u_1, u_2, \ldots, u_i) = 1$$

  onde $u_j \in \{0,1\}^{p_j(|w|)}$, para todo j, e Q é um quantificador $\forall$ ou $\exists$ dependendo da paridade do índice i;

- para todo $i \geqslant 1$, Π_i é a classe das linguagens L para as quais existe um algoritmo M de tempo polinomial e polinômios $p_1, p_2, \ldots, p_i$ tais que para todo $w \in \Sigma^*$

  $$w \in \mathsf{L} \Leftrightarrow \forall u_1,\ \exists u_2,\ \forall u_3,\ \ldots,\ \mathrm{Q}u_i,\ \ \mathsf{M}(w, u_1, u_2, \ldots, u_i) = 1$$

  onde $u_j \in \{0,1\}^{p_j(|w|)}$, para todo j, e Q é um quantificador $\forall$ ou $\exists$ dependendo da paridade do índice i.

Não é difícil notar que, para cada $i \geqslant 1$, temos que as classes definidas acima são complementares, isto é, $\Pi_i = \mathrm{co}\Sigma_i$. Também vale, e não é difícil provar, que $\Pi_i \cup \Sigma_i \subseteq \Sigma_{i+1} \cap \Pi_{i+1}$, em particular $\mathsf{NP} \cup \mathrm{co}\mathsf{NP} \subseteq \Sigma_2 \cap \Pi_2$. Notemos que P, NP e BPP são todos subconjuntos de Σ_2.

Uma generalização do Problema 5 ($\mathsf{P} \neq \mathsf{NP}$?) é a conjetura $\Sigma_i \neq \Sigma_{i+1}$? E uma generalização do Problema 6 ($\mathsf{NP} \neq \mathrm{co}\mathsf{NP}$?) é a conjectura que $\Pi_i \neq \Sigma_i$ para todo $i \geqslant 1$? É possível mostrar que, para $i \geqslant 1$,

$$\textit{se } \Sigma_i = \Pi_i \textit{ então } \Sigma_i = \Sigma_{i+1}$$

donde deduzimos

$$\textit{se } \Sigma_i = \Pi_i \textit{ então } \Sigma_j = \Sigma_i \textit{ para todo } j \geqslant i$$

e podemos por o problema abaixo.

Problema 11. $\Sigma_i \subsetneq \Sigma_{i+1}$?

Se, para algum i, $\Sigma_i = \Sigma_{i+1}$ então $\Sigma_j = \Pi_j = \Sigma_i$ para todo $j \geqslant i$. Nesse caso dizemos que a hierarquia **colapsa** no i-ésimo nível.

Definimos a classe PH — *Polynomial Hierarchy* — por

$$\mathsf{PH} := \bigcup_{i \geqslant 0} \Sigma_i = \bigcup_{i \geqslant 0} \Pi_i.$$

O seguinte resultado pode ser estudado no livro de Papadimitriou, 1994.

Teorema. *Para todo* $i \geqslant 1$, *se* $\Sigma_i = \Pi_i$ *então* $\Pi_{i+1} = \Sigma_{i+1} = \Sigma_i$ *que implica em* $\mathsf{PH} = \Sigma_i$. *Em particular, se* $\mathsf{P} = \mathsf{NP}$ *então* $\mathsf{PH} = \mathsf{P}$.

Vimos que $\mathsf{P} \subseteq \mathsf{P/poly}$, portanto, se $\mathsf{P} = \mathsf{NP}$ então $\mathsf{NP} \subseteq \mathsf{P/poly}$ (o que não é sabido, Problema 7, página 209). Uma questão interessante é saber se $\mathsf{NP} \subseteq \mathsf{P/poly}$ poderia ocorrer somente se $\mathsf{P} = \mathsf{NP}$. Um resultado nessa direção, num nível acima da hierarquia, é descrito a seguir, mostrando que $\mathsf{NP} \subseteq \mathsf{P/poly}$ pode ocorrer somente se PH colapsa para o segundo nível.

Teorema. (Karp e Lipton, 1980) *Se* $\mathsf{NP} \subseteq \mathsf{P/poly}$ *então* $\Sigma_2 = \Pi_2$.

Como comentamos acima, se $\mathsf{P} = \mathsf{NP}$ então $\mathsf{P} = \mathsf{PH}$ e como $\mathsf{BPP} \subseteq \mathsf{PH}$, obtemos $\mathsf{BPP} \subseteq \mathsf{P}$. Já sabíamos que $\mathsf{P} \subseteq \mathsf{BPP}$, portanto, se $\mathsf{P} = \mathsf{NP}$ então $\mathsf{P} = \mathsf{BPP}$.

Corolário 4.17 *Se* $\mathsf{P} = \mathsf{NP}$ *então* $\mathsf{BPP} = \mathsf{P}$.

Logo se não há problemas difíceis no sentido de que todo problema NP tem algoritmo eficiente, então as soluções probabilísticas eficientes podem ser *desaleatorizadas*.

4.3 SISTEMAS PROBABILÍSTICOS DE PROVA

Em um sistema de prova dois algoritmos interagem comunicando-se através de leitura e escrita num espaço comum. Chamaremos esses algoritmos de *Verificador*, denotado V, e *Provador*, denotado P. Ambos têm acesso a uma memória somente de leitura contendo a entrada w que é o fato a ser provado. O sistema ainda tem mais duas outras memórias compartilhadas por P e por V, numa delas, denotada $M_{P\to V}$, o Provador P pode escrever e V somente fazer leituras, na outra, denotada $M_{V\to P}$, os papéis são trocados, ou seja, V pode escrever e P somente fazer leituras. As memórias compartilhadas estão vazias (há somente $\flat$) no começo da computação. O Verificador *aceita* (responde 1) ou *rejeita* (responde 0) a prova dada pelo Provador e a resposta do Verificador é designada por $(V,P)(w)$. Ademais V é limitado, tem complexidade de tempo polinomial, enquanto que P não tem limitação. O número de rodadas de uma computação do sistema de provas é o número de mensagens $m \in \{0,1\}^*$ trocadas nas duas direções entre os algoritmos.

Uma linguagem L admite um sistema de prova se existe um verificador V de tempo polinomial e um protocolo com k rodadas, com k polinomial no tamanho da entrada, tal que são satisfeitos:

1. *completude*: se $w \in L$ então em k rodadas $(V,P)(w) = 1$ para algum provador P;
2. *consistência*: se $w \notin L$ então em k rodadas $(V,P)(w) = 0$ qualquer que seja o

provador P.

Sistema P de prova A classe P pode ser definida em termos de sistemas de provas da seguinte maneira: uma linguagem L está em P se admite um sistema de provas tal que para todo $w \in \{0,1\}^*$, se $w \in L$ então quando o algoritmo V termina ele responde 1 sem nunca ter consultado $M_{P\to V}$; se $w \notin L$ o algoritmo V, no término de sua computação, responde 0 sem nunca ter consultado $M_{P\to V}$.

Sistema NP de prova Podemos definir um sistema de provas para qualquer linguagem $L \in NP$ pois, se $w \in L$ então o Provador escreve um certificado c de tamanho polinomial em $M_{P\to V}$ e o Verificador checa em tempo polinomial e com a ajuda da prova c, se $w \in L$. Tal certificado existe se, e somente se, $w \in L$.

Exemplo 4.18. Denotamos por 3-COL a linguagem definida pelos grafos que admitem um 3-coloração própria dos seus vértices, ou seja, G está nessa linguagem se e somente se todos os vértices de G podem ser coloridos com três cores distintas de modo que vértices adjacentes têm cores diferentes. A linguagem 3-COL está em NP.

Um sistema de provas para 3-COL consiste de uma instância G conhecida por V e P; o Verificador pergunta a cor de um determinado vértice de G ao Provador que responde com uma de três cores. A cada resposta, o Verificador verifica se a cor dada não conflita com as cores que já foram atribuídas. Se o Provador responde uma cor que viola a propriedade então o Verificador rejeita. Se, ao final, todos os vértices estão coloridos propriamente então o Verificador aceita.

Notemos que o Provador poderia, com seu poder computacional ilimitado, simplesmente computar e responder de uma só vez uma coloração. ◊

Agora, suponhamos que uma linguagem L admite um sistema de prova (V, P) de k (polinomial) rodadas. Para $w \in L$ o conteúdo de $M_{P\to V}$ ao final da computação é uma sequência $(m_1, m_2, \ldots, m_k)$ de tamanho polinomial de mensagens enviada pelo Provador. A sequência de mensagens é um certificado para V (de tempo polinomial) aceitar a entrada w, portanto *a classe das linguagens que admitem tal sistema de provas coincide com* NP.

O Provador só pode antecipar as respostas na estratégia acima se o comportamento do Verificador é determinístico, caso o Verificador use bits aleatórios, o Provador teria que gerar um certificado exponencialmente grande para cobrir todas as possibilidades. Portanto, esse cenário muda quando permitimos um Verificador aleatorizado. A aleatorização é essencial para que a classe das linguagens que admitem um sistema de prova vá além de NP, como ilustra o seguinte exemplo pitoresco e clássico na literatura sobre provas interativas: Alice tem duas bolas de bilhar, uma

é vermelha e a outra verde; ambas parecem idênticas ao seu amigo Bob, que é daltônico e cético com respeito à diferença das bolas que lhe parecem iguais. Alice quer convencer Bob que, de fato, as bolas são de cores diferentes e para isso entrega as bolas ao Bob deixando uma em cada mão, sem dizer qual é a cor de cada uma (fato que ela se recordará sempre que for necessário). Em seguida, Bob esconde as mãos para trás e, sem que Alice veja, lança uma moeda e se o resultado for cara ele troca as bolas de mãos, se der coroa ele deixa as bolas como foi entregue por Alice. Feito isso, Bob questiona Alice sobre o seu ato, isto é, se ele trocou ou não as bolas. Se as bolas são diferentes então Alice acerta, sempre. Se as bolas são idênticas Alice acerta com probabilidade 1/2, e essa probabilidade reduz pela metade cada repetição do teste; Bob repete até estar seguro de que Alice está certa ou não, com 10 repetições e bolas idênticas a Alice acerta todas as respostas com probabilidade menor que 0,001.

4.3.1 A CLASSE IP

Um **sistema interativo de prova** é um sistema de prova como acima com a exceção de que o Verificador é um algoritmo *probabilístico* de tempo polinomial e P é um algoritmo probabilístico sem restrições de tempo. Esse conceito foi introduzido por Goldwasser, Micali e Rackoff (1985) e de modo independente por Babai (1985). Uma interação em k rodadas define uma sequência de mensagens $m_1, m_2, \ldots, m_k$ tal que

$$
\begin{aligned}
m_1 &:= \mathsf{V}(w), \\
m_2 &:= \mathsf{P}(w, m_1), \\
&\vdots \\
m_i &:= \begin{cases} \mathsf{V}(w, m_1, \ldots, m_{i-1}) \text{ se } i \text{ é ímpar menor ou igual a } k \\ \mathsf{P}(w, m_1, \ldots, m_{i-1}) \text{ se } i \text{ é par e menor ou igual a } k \end{cases}
\end{aligned}
$$

e $(\mathsf{V}, \mathsf{P})(w) = \mathsf{V}(w, m_1, \ldots, m_k) \in \{0, 1\}$.

Uma linguagem L admite um sistema interativo de provas se existe um Verificador V probabilístico de tempo polinomial tal que com entrada w

1. *completude*: se $w \in \mathsf{L}$ então *existe* P tal que

$$\mathbb{P}[\,(\mathsf{V}, \mathsf{P})(w) = 1\,] \geqslant \frac{2}{3};$$

2. *consistência*: se $w \notin \mathsf{L}$ então *qualquer que seja* o Provador P

$$\mathbb{P}[\,(\mathsf{V}, \mathsf{P})(w) = 1\,] \leqslant \frac{1}{3};$$

além disso, os algoritmos trocam no máximo um número polinomial em $|w|$ de mensagens e a *aleatoriedade é privada*, ou seja, P não tem acesso aos bits aleatórios de V e vice-versa.

IP — *Interactive Proof* — é a classe de linguagens que admitem um sistema interativo de prova.

IP(k) é a subclasse de linguagens em IP que admitem um sistema interativo de prova em k rodadas.

A classe IP é invariante com respeito às seguintes modificações na definição: a probabilidade de erro é arbitrária e poderia ser qualquer constante positiva, na completude, assumir erro com probabilidade zero. Não provaremos esses fatos aqui, o leitor pode consultar Arora e Barak (2009, seção 8.3). Ademais, o Provador probabilístico não acrescenta poder ao modelo com respeito ao reconhecimento de linguagens, poderíamos assumir P determinístico e de complexidade de espaço[5] polinomial.

Convencionamos que um protocolo interativo é escrito como

Entrada: aqui descrevemos a entrada comum as duas partes.

V: aqui estão as computações de V;

V→P: aqui estão as mensagens enviadas de V para P;

P: aqui, são as computações de P;

P→V: aqui, são as mensagens enviadas de P para V;

O exemplo abaixo mostra um problema computacional que está na classe IP mas não se sabe se está na classe NP, o que é um indício de que com a aleatoriedade o modelo interativo vai além de NP.

Exemplo 4.19 (NONISO ∈ IP). Os grafos a seguir são sobre o mesmo conjunto V de vértices, fixamos $V = \{1, 2, \ldots, n\}$ e um isomorfismo é uma permutação σ do conjunto $\mathbb{S}_n$ de todas as permutações de $\{1, 2, \ldots, n\}$ de modo que se G é um grafo então $\sigma(G)$ é o grafo com arestas $\{\{\sigma(u), \sigma(v)\} : \{u, v\} \in E(G)\}$. Definimos a linguagem formada por pares de grafos não isomorfos

$$\text{NONISO} := \{\langle G, H\rangle : G \text{ e } H \text{ não são grafos isomorfos}\}.$$

Não é sabido se NONISO está em NP, de modo que um provador não conhece um certificado efetivo que possa ser enviado a um verificador.

[5] Um máquina de Turing tem complexidade de espaço $S(n)$ se $S(n)$ é um limitante superior para a posição mais a direita na fita que é lida ou escrita em qualquer computação com instância de tamanho n.

Apesar disso, em 2 rodadas um Provador consegue convencer um Verificador probabilístico desse fato. O Verificador escolhe ao acaso um dos dois grafos e envia ao Provador um cópia isomorfa do grafo escolhido, o Provador tem que descobrir qual foi o grafo escolhido, o que só é possível se os grafos da entrada não forem isomorfos. O protocolo é: com entrada G_0 e G_1; V sorteia $i \in \{0,1\}$ e sorteia uma permutação π, envia $\pi(G_i)$; o poderoso P testa se $\pi(G_i)$ é isomorfo a G_0 ou a G_1 e envia o que descobriu $j \in \{0,1\}$; se $i = j$ então V aceita, senão rejeita. Se G_0 não é isomorfo a G_1 (a entrada pertence a linguagem), então existe um Provador que descobre qual isomorfismo o Verificador construiu, por exemplo testando todas as permutações sobre os n vértices, e envia o bit correto para o Verificado, que aceita. Agora se G_0 é isomorfo a G_1 (a entrada não pertence a linguagem) o Provador não consegue descobrir qual gerou $\pi(G_i)$ de modo que o melhor que ele pode fazer é chutar um valor. Nesse caso, o Verificador aceita ou não com probabilidade 1/2. Esse protocolo executado duas vezes ("em paralelo") diminui a probabilidade de erro:

Entrada: os grafos G_0 e G_1.

V: $i \leftarrow_R \{0,1\}$; $j \leftarrow_R \{0,1\}$;
$\sigma \xleftarrow{R} \mathbb{S}_n$;$\pi \xleftarrow{R} \mathbb{S}_n$;
$H \leftarrow \sigma(G_i)$; $J \leftarrow \pi(G_j)$;

V→P: H, J;

P: se G_0 não é isomorfo a H, então $d \leftarrow 1$, senão
se G_1 não é isomorfo a H, então $d \leftarrow 0$, senão
$d \xleftarrow{R} \{0,1\}$;

se G_0 não é isomorfo a J, então $e \leftarrow 1$, senão
se G_1 não é isomorfo a J, então $e \leftarrow 0$, senão
$e \xleftarrow{R} \{0,1\}$;

P→V: d, e;

V: se $(i,j) = (d,e)$ então responda 1, senão responde 0.

Se os grafos G_0 e G_1 não são isomorfos, então o Provador pode sempre distinguir o caso em que H é isomorfo a G_0 do caso em que H é isomorfo a G_1, o mesmo vale para J, e sempre acertar os valores de d e e. Nesse caso o verificador responde 1 (completude *sem erro*).

Se os grafos são isomorfos então mesmo com um grande poder computacional o Provador não sabe distinguir H e J de G_0 ou G_1 de modo que d e e são sorteados

pelo Provador. O Verificador responderá errado se $d = i$ e $e = j$, o que ocorre com probabilidade no máximo 1/4.

Notemos que a tarefa do Provador durante a execução é testar isomorfismo entre grafos, o que não sabemos fazer de forma eficiente, entretanto, P não tem restrição de tempo, ele pode testar todas as $n!$ permutações possíveis para descobrir o isomorfismo. ◊

O exemplo acima nos mostra que NONISO $\in$ IP mas, como já dissemos, ainda não sabemos responder se NONISO $\in$ NP, mais que isso, não sabemos se a inclusão NP $\subseteq$ IP é própria.

Problema 13. NP $\subsetneq$ IP?

A linguagem

$$\text{ISO} := \{\langle G, H\rangle : G \text{ e } H \text{ são grafos isomorfos}\}$$

por sua vez, está em NP pois um isomorfismo é um certificado curto que atesta pertinência na linguagem. Não é sabido se ISO é uma linguagem NP-completa. Se ISO for NP-completa, então NONISO será coNP-completa e chegaríamos a uma resposta afirmativa para o problema

Problema 14. coNP $\subseteq$ IP(2)?

É sabido (Boppana, Håstad e Zachos, 1987) que se o Problema 14 for respondido com sim, coNP $\subseteq$ IP(2), então a Hierarquia Polinomial colapsa no segundo nível.

A classe IP contém a hierarquia polinomial (Lund *et al.*, 1992) e o que foi surpresa para os pesquisadores é o fato de que IP é igual a classe PSPACE — *Polynomial Space* — a classe das linguagens que podem ser decididas por algoritmos com complexidade de espaço polinomial (Shamir, 1992; Shen, 1992). Assim o que sabemos é que

$$\text{P} \subseteq \text{NP} \subseteq \text{PH} \subseteq \text{IP} = \text{PSPACE}.$$

Exemplo 4.20 (*não-resíduo quadrático está em* IP). Seja p um primo. Lembremos que a é um resíduo quadrático modulo p se para algum inteiro x temos $x^2 \equiv a \pmod{p}$. A linguagem definida pelos pares (a, p) tais que p é primo e a é um resíduo quadrático modulo p está em NP pois uma raiz quadrada de a é um certificado que pode ser verificado de modo eficiente, assim como a primalidade de p. Por outro lado, a linguagem definida pelos pares (a, p) tais que p é primo e a *não* é um resíduo quadrático modulo p não se sabe se está em NP, mas tem uma prova interativa como mostra o seguinte protocolo:

Entrada: um par de inteiros (a, p), p primo.

V: $r \leftarrow_R \{1, \ldots, p-1\}$;
$b_1 \leftarrow_R \{0,1\}$; $b_2 \leftarrow_R \{0,1\}$;
Para cada $i \in \{1,2\}$,
se $b_i = 0$, então $w_i \leftarrow r^2 \bmod p$,
senão $w_i \leftarrow ar^2 \bmod p$;

V→P: w_1, w_2;

P: Para cada $i \in \{1,2\}$,
se w_i é resíduo quadrático, então $c_i \leftarrow 0$,
senão $c_i \leftarrow 1$;

P→V: c_1, c_2;

V: se $(c_1, c_2) = (b_1, b_2)$, então responde 1, senão responde 0.

Se a é resíduo quadrático então ar^2 também é um resíduo quadrático então w_1 e w_2 serão resíduos quadráticos, portanto $c_1 = c_2 = 0$. Como o Provador não conhece os bits b_1 e b_2 a probabilidade de que $(c_1, c_2) = (b_1, b_2)$ é 1/4. Agora, se a não é resíduo quadrático então ar^2 também não é, enquanto que r^2 é resíduo quadrático, portanto o Provador consegue distinguir corretamente o bit sorteado pelo Verificador e a resposta, nesse caso, é sempre 1. ◊

4.3.2 SISTEMAS DE PROVA COM BITS ALEATÓRIOS PÚBLICOS

Um fato crucial para o sucesso dos protocolos de comunicação nos dois sistemas de prova dos exemplos acima é que o Provador não conhece os bits sorteados pelo Verificador. Se permitimos que o Verificador envie ao Provador os bits aleatórios, então temos um *sistemas de provas de bits públicos* de k rodadas que dá origem à hierarquia de classes de complexidade $\mathtt{AM}(k)$ — *Arthur–Merlin proofs* — e $\mathtt{MA}(k)$ — *Merlin–Arthur proofs*.

O protocolo AM foi apresentado em Babai (1985) com o objetivo de construir uma versão aleatorizada de NP, e que estivesse "logo acima" de NP, para acomodar certas linguagens. Arthur, um ser humano com suas limitações, refere-se ao Verificador e Merlin, um mago poderoso, ao Provador. Ambas as classes são subclasses de IP obtidas quando restringimos as mensagens que o Verificador envia: todos, e somente os, bits aleatórios que ele usa. Qualquer outra informação que o Verificador precise enviar pode ser computada pelo poderoso Provador.

A diferença entre essas classes AM e MA reside em quem manda a primeira mensa-

gem, o Verificador (Arthur) na classe $\mathtt{AM}(k)$, ou o Provador (Merlin) na classe $\mathtt{MA}(k)$. Em particular, em $\mathtt{MA}(1)$ Merlin envia a mensagem inicial (um certificado) e como não há mais mensagens Arthur toma sua decisão sem sorteio de bits, logo $\mathtt{MA}(1) = \mathtt{NP}$; em $\mathtt{AM}(1)$ Arthur sorteia bits, os envia ao Merlin que não comunica nada, e toma a sua decisão usando esses bits sorteados, logo $\mathtt{AM}(1) = \mathtt{BPP}$; em $\mathtt{MA}(2)$ o Merlin manda uma mensagem inicial, Arthur sorteia seus bits, os manda para Merlin que não tomará outra providência e usa esses bits para tomar sua decisão de aceite ou não.

Seguindo a tradição bibliográfica

$$\mathtt{AM} := \mathtt{AM}(2) \quad \text{e} \quad \mathtt{MA} := \mathtt{MA}(2)$$

logo $\mathtt{AM}$ é a classe das linguagens com prova interativa onde o Verificador manda uma mensagem com bits aleatórios e o Provador responde. Para $p(n)$ uma função limitada polinomialmente, com $p(n) \geqslant 2$, Babai (1985) mostrou que $\mathtt{AM}(p(n)) = \mathtt{AM}(p(n)+1) = \mathtt{MA}(p(n)+1)$em particular, $\mathtt{AM} = \mathtt{AM}(k) = \mathtt{MA}(k+1)$ para qualquer $k \geqslant 2$ fixo. Ainda, Goldwasser e Sipser (1986) mostram a segunda inclusão de $\mathtt{AM}(p(n)) \subseteq \mathtt{IP}(p(n)) \subseteq \mathtt{AM}(p(n)+2)$logo, reunindo várias informações que temos até aqui, podemos escrever

$$\mathtt{NP} \cup \mathtt{BPP} \subseteq \mathtt{MA} \subseteq \mathtt{AM} \subseteq \mathtt{AM}(\mathtt{poly}) = \mathtt{IP} = \mathtt{PSPACE}$$

em que

$$\mathtt{AM}(\mathtt{poly}) = \mathtt{MA}(\mathtt{poly}) := \bigcup_{k>0} \mathtt{AM}(n^k).$$

Protocolo com bits aleatórios públicos para NONISO Fixamos o conjunto de vértices dos grafos em $V = \{1, 2, \ldots, n\}$. Sejam G_0 e G_1 dois grafos. A quantidade de grafos isomorfos à G_i é

$$|I(G_i)| := \frac{n!}{|\mathrm{Aut}(G_i)|} \tag{4.7}$$

em que $\mathrm{Aut}(G_i)$ é o conjunto de todas as permutações $\pi\colon V \to V$ que definem um isomorfismo de G_i em G_i. Esse conjunto munido da composições de funções define um grupo chamado de grupo dos automorfismos de G_i (o qual é subgrupo do grupo de todas as permutações, logo (4.7) é inteiro, pelo Teorema de Lagrange).

Por exemplo, no caso do grafo $G := (\{1,2,3\},\{12\})$ com três vértices e uma aresta, se $\pi(1) = 2$, $\pi(2) = 1$, $\pi(3) = 3$, então π é um isomorfismo de G em G. Agora se se $\pi'(1) = 3$, $\pi'(3) = 1$, $\pi(2) = 2$, então π' é um isomorfismo de G em $\pi'(G) := (\{1,2,3\},\{13\})$ que é diferente de G. No caso do triângulo, denotado C_3, toda bijeção define um isomorfismo de C_3 em C_3, portanto $|\mathrm{Aut}(C_3)| = 6$. No caso do circuito com 4 vértices, o $C_4 := (\{1,2,3,4\},\{12,23,34,14\})$, das 24 permutações apenas 8 definem automorfismos de C_4, portanto há 3 grafos distintos sobre $\{1,2,3,4\}$ isomorfos ao C_4. Agora, cada grafo em $I(G_i) := \{\pi(G_i)\colon \pi \text{ permutação}\}$ ocorre $|\mathrm{Aut}(G_i)|$ vezes dentre todas $n!$ permutações, logo $|I(G_i)|$ é dado pela equação (4.7).

Seja S o conjunto dos grafos sobre V que são isomorfos a algum G_i, para $i = 0, 1$,

$$S = \{(H, \pi) \colon H \equiv G_0 \text{ ou } H \equiv G_1 \text{ e } \pi \in \mathrm{Aut}(H)\}.$$

A ideia do protocolo é separar (com probabilidade razoavelmente grande) o caso em que G_0 e G_1 não são isomorfos do caso que são isomorfos usando o tamanho de S pois

$$\text{se } G_0 \not\equiv G_1 \text{ então } |S| = 2n! \tag{4.8}$$

$$\text{se } G_0 \equiv G_1 \text{ então } |S| = n! \tag{4.9}$$

a afirmação (4.9) segue imediatamente de (4.7) pois $|\mathrm{Aut}(H)| = |\mathrm{Aut}(G_i)|$. Para a afirmação (4.8), cada $H \equiv G_i$ contribui com $|\mathrm{Aut}(G_i)|$ pares e são $n!/|\mathrm{Aut}(G_i)|$ tais H's, no outro caso, como os grafos G_0 e G_1 não são isomorfos, são $2n!$ pares.

Se o Verificador gerar um grafo H então o Provador pode verificar se esse grafo é isomorfo a G_0 ou a G_1 e quando for o caso enviar o isomorfismo para o Verificador conferir. Repetindo esse procedimento nós observaríamos que no caso (4.8) o Verificador atesta isomorfismo duas vezes mais que no caso (4.9), permitindo uma conclusão a respeito do grafos de entrada. O problema desse protocolo é que para termos uma probabilidade de erro limitada por uma constante o número de repetições tem que ser exponencial, pois o número total de grafos sobre $\{1, 2, \ldots, n\}$ é $2^{\binom{n}{2}}$ e $|S|/2^{\binom{n}{2}}$ é exponencialmente pequeno.

Uma solução é usar uma ferramenta introduzida no Exemplo 5.7, que estudaremos com mais profundidade na seção 5.2.2. No Exemplo 5.7, página 250, ao sortear uma função de hash de uma família de funções de hash $\mathcal{H} \subseteq M^U$ 2-a-2 independentes, isto é, tal que

$$\mathop{\mathbb{P}}_{h \in \mathcal{H}}([h(x) = i] \cap [h(y) = j]) = \frac{1}{|M|^2}, \quad \forall x \neq y \in U, \quad \forall i, j \in M,$$

a probabilidade p com que exista $x \in S \subseteq U$ tal que $h(x) = 0$ é

$$\frac{|S|}{|S| + m} \leqslant p \leqslant \frac{|S|}{m}.$$

Fixado um inteiro K, tome m a maior potência de 2 menor ou igual a $K/2$, e seja $h \colon U \to \{0, \ldots, m-1\}$ escolhida uniformemente de uma família de funções hash 2-a-2 independente. Então a probabilidade p de existir $x \in S$ tal que $h(x) = 0$ satisfaz $p \geqslant 2/3$, se $|S| \geqslant K$, e $p \leqslant 1/4$ se $|S| \leqslant K/16$.

Com isso, temos o conjunto S cujo tamanho difere por um fator $1/2$ dependendo de se os grafos são isomorfos ou não, equações (4.9) e (4.8) respectivamente, e para usar o resultado estabelecido no parágrafo anterior temos que amplificar este fator

para 1/16. Nós podemos fazer isso usando uma quádrupla $S' = S \times S \times S \times S$ de modo que

$$|S'| = \begin{cases} 16(n!)^4 & \text{se } G_0 \not\equiv G_1 \\ (n!)^4 & \text{se } G_0 \equiv G_1. \end{cases}$$

Com isso, o Provador afirma que $|S| \geqslant 16(n!)^4$; o Verificador escolhe uma função hash aleatória h de uma família independente aos pares e pede ao Provador um $x \in S$ tal que $h(x) = 0$. O Provador convence com probabilidade pelo menos 2/3 se $G_0 \not\equiv G_1$, e no máximo 1/4 se $G_0 \equiv G_1$.

Vejamos, agora, um protocolo de troca de mensagens com bits aleatórios públicos, devido a Goldwasser e Sipser, em que o objetivo do Provador é convencer o Verificador de que $|S| \geqslant K$ e caso $|S| \leqslant K/16$ o Verificador deve rejeitar com probabilidade de erro positiva. Notemos que se existe $x \in S$ tal que $h(x) = y$ então o Provador consegue determinar x e convencer o Verificador de que sua imagem é y. A única restrição em S é que a pertinência tem que ser determinada de forma eficiente. Essa estratégia do Provador convencer o Verificador de um limitante inferior no tamanho de um conjunto pode ser usada substituir moedas privadas por públicas em qualquer prova interativa. No Exercício 5.54, página 290, é apresentada uma família de funções de hash que podem ser usadas nesse protocolo.

Entrada: $S \subseteq U$ e um natural K.

V: $h \overset{R}{\leftarrow} \{f : U \to \{0,1\}^{\lceil \log(K) \rceil} : f \text{ satisfaz } (5.6)\}$;
$y \leftarrow_R \{0,1\}^{\lceil \log(K) \rceil}$;

V→P: h, y;

P: determina, se possível, $x \in S$ tal que $h(x) = y$;

P→V: x e um certificado de que $x \in S$;

V: se $h(x) = y$ e o certificado valida a pertinência $x \in S$
então responde 1,
senão responde 0.

Como ISO, o problema de isomorfismo de grafos, está em NP, concluímos que $\text{ISO} \in \text{NP} \cap \text{coAM}$. Acredita-se que AM seja uma classe que inclua propriamente NP e também que NP-completos não estejam em coAM. Pode-se provar que se $\text{NP} \subseteq \text{coAM}$ então a hierarquia polinomial colapsa. Por esta razão, acreditas-se que ISO não seja NP-completo.

4.3.3 PROVAS COM CONHECIMENTO ZERO

Conhecimento zero em um sistema de prova é uma característica de certos Provadores de que somente a validade da prova é transmitida ao Verificador no processo de interação, ou seja, o Verificador não ganha informação no processo e mesmo assim convence-se da validade da prova no final do processo (Goldwasser, Micali e Rackoff, 1985). Por exemplo, no caso das bolas de bilhar, página 226, Bob fica convencido de que as bolas têm cor diferente mas não sabe a cor de cada bola individualmente. Essa propriedade é bastante interessante para projetos de protocolos em Criptografia pois ela permite que a prova de um fato sem revelar segredos.

O fato do Verificador não ganhar informação significa que num sistema de prova (V, P) com entrada w tudo aquilo que o Verificador V puder computar ao interagir com P a partir da entrada w é igual a tudo aquilo que um algoritmo da classe do Verificador (com o mesmo poder de computação) consegue computar com a entrada w por ele mesmo, isto é, não interagindo com um Provador P. Por exemplo, no sistema para 3-COL, exemplo 4.18 na página 226, o Verificador recebe do Provador uma 3-coloração, informação que um algoritmo de tempo polinomial não consegue computar. Nesse exemplo o Verificador ganha informação a partir da interação.

Um sistema de prova tem conhecimento zero se para todo Verificador V existe um algoritmo eficiente M, chamado **simulador**, tal que a distribuição da resposta de V quando interage com P, que denotamos por $(V, P)(w)$, e a distribuição $M(w)$ são idênticas. Formalmente, relaxamos um pouco a exigência sobre o tempo do simulador, permitindo que seja polinomial em média. Dizemos que um Provador de um sistema de prova interativo para a linguagem L tem **conhecimento-zero perfeito** se para todo Verificador V existe um algoritmo probabilístico M de tempo *esperado* polinomial, tal que para todo $w \in L$, as variáveis aleatórias $(V, P)(w)$ e $M(w)$ são identicamente distribuídas.

Para propósitos práticos podemos exigir menos da relação entre as distribuições $(V, P)(x)$ e $M(x)$, por exemplo, que sejam indistinguíveis estatisticamente ou por algoritmos eficientes. De acordo com alguma definição de distância entre essas distribuições, temos a seguinte hierarquia de conhecimento zero

perfeito – as distribuições são idênticas, como foi discutido acima;

estatístico – as distribuições são estatisticamente próximas[6];

computacional – as distribuições são computacionalmente indistinguíveis por algoritmos eficientes,

[6] segundo a métrica $\max_{S \subset \Omega} \left| \mathbb{P}[X \in S] - \mathbb{P}[Y \in S] \right|$.

claramente, os requisitos de conhecimento zero perfeito são mais exigentes que o estatístico que, por sua vez, são mais exigentes que o computacional, que é o mais prático. Essas definições dão origem as classes de linguagens com conhecimento-zero computacional CSK — *Computational Zero-knowledge* – e conhecimento-zero estatístico SZK – *Statistical Zero-knowledge*. É sabido que

$$\mathsf{BPP} \subset \mathsf{PZK} \subset \mathsf{SZK} \subset \mathsf{CZK} \subset \mathsf{IP}.$$

Com a hipótese de existirem funções unidirecionais temos que CZK = IP (Ben-Or *et al.*, 1990) e que NP ⊂ CZK (Goldreich, Micali e Wigderson, 1991). Por outro lado, é improvável que NP ⊂ SZK (Fortnow, 1987); apesar disso vários problemas difíceis estão em SZK, como resíduo e não-resíduo quadrático, isomorfismo e não-isomorfismo de grafos, logaritmo discreto. Fortnow (1987) provou que a existência de um problema NP-completo em SZK implica no colapso de PH no nível 2. Sob a hipótese de existir função unidirecional e de não haver colapso em PH temos CZK ≠ SZK.

Conjectura 4.21 $\mathsf{BPP} \subsetneqq \mathsf{PZK}$ *e* $\mathsf{SZK} \subsetneqq \mathsf{CZK}$ *e* $\mathsf{CZK} = \mathsf{IP}$.

logaritmo discreto ∈ PZK Um provador deseja convencer qualquer verificador de que ele conhece k, o logaritmo discreto de β com relação à base (raiz primitiva) α e um primo p grande. Os parâmetros α, β e p são públicos.

Entrada: p primo, α raiz primitiva módulo p e β um resíduo módulo p.

P: $r \overset{R}{\leftarrow} \{0,\ldots,p-1\}$;
$h \leftarrow \alpha^r \bmod p$;

P→V: h;

V: $b \leftarrow_R \{0,1\}$;

V→P: b;

P: $a \leftarrow r + bk \bmod p$

P→V: a;

V: se $\alpha^a \equiv h\beta^b \pmod p$ então responde 1,
senão responde 0.

Notemos que V executa em tempo polinomial. Ademais, dado que P conhece k, o logaritmo discreto de β, o verificador fica convencido (sem erro) desse fato pois $\alpha^a = \alpha^r \alpha^{kb} = h\beta^b$, logo, o sistema interativo é completo.

Para provar a consistência suponhamos que não haja provador que conheça k. Fixemos um deles. No protocolo P deve enviar um h na primeira mensagem, suponhamos que P sorteia r e envia h. Se V sorteia 0, então o provador pode enviar $a = r$ e V aceita, senão V sorteia 1 e o provador precisa descobrir o logaritmo discreto de $h\beta$, que é o logaritmo discreto de h mais o de β, ou seja, $r + k$. O melhor que ele pode fazer é chutar e acertar com probabilidade $1/(p-1)$. A probabilidade de um aceite nesse caso é

$$\frac{1}{2} + \frac{1}{2}\frac{1}{p-1}.$$

Agora, vamos considerar que P envia h sem conhecer r. Se V sorteia 0, o provador precisa descobrir o logaritmo discreto de h, então ele chuta. Se V sorteia 1, então um provador esperto poderia ter enviado $h = \alpha^r\beta^{-1}$ pois, com isso, agora basta enviar $a = r$ que V aceita pois $\alpha^a = \alpha^r = (\alpha^r\beta^{-1})\beta = h\beta$. Agora, a probabilidade de aceite é no máximo

$$\frac{1}{2}\frac{1}{p-1} + \frac{1}{2}.$$

Se p é primo ímpar então $\frac{1}{2}\frac{1}{p-1} + \frac{1}{2} < 1$ e a repetição do protocolo faz a probabilidade de erro pequena suficiente para valer a consistência.

Para verificar o conhecimento-zero do protocolo nós precisamos provar que nenhum verificador recebe informação adicional àquelas transmitidas nas mensagens do protocolo. Nesse protocolo, podemos ver a computação do verificador em duas etapas: (1) $\mathsf{V}_1(h)$ que recebe a informação h do Provador e responde com um bit b e (2) $\mathsf{V}_2(h, a)$ que é a computação com as informações h e a recebidas do Provador. Essa duas etapas são de tempo polinomial. Fixado um verificador V que interage com P dado acima, escrevemos o seguinte simulador que denominamos M.

Entrada: p primo, α raiz primitiva módulo p e β um resíduo módulo p.

1. sorteia $b' \in_R \{0, 1\}$ e $r' \in_R \{0, \ldots, p-1\}$;
2. $[\mathsf{V}_1(h)]$ simula V com $h := \alpha^{r'}\beta^{b'}$ na memória $\mathsf{F}_{\mathsf{P}\to\mathsf{V}}$ para obter $b \in \{0, 1\}$ após um número polinomial de passos;
3. se $b = b'$ então $[\mathsf{V}_2(h, a)]$ simula V com o envio de $a := r'$ por P e devolve a resposta da simulação; caso contrário, recomeça no passo 1.

Para provar que a resposta do simulador M tem a mesma distribuição da resposta de V na interação basta provar que nas respostas parciais a distribuição é a mesma.

Primeiro, *a distribuição de h computado por* M *é idêntica a de h escolhida pelo provador*: em ambos os casos h é uma potência de α com expoente aleatório com distribuição uniforme. Quando o simulador atribui $h := \alpha^{r'}\beta^{b'}$ temos $h = \alpha^{r'-b'k}$ e o expoente tem distribuição uniforme.

Segundo, $b = b'$ *com probabilidade pelo menos* $1/2$: segue do fato no parágrafo anterior que as distribuições dos h's são idênticas, portanto, o comportamento do verificador não muda, de modo que $b = b'$ com probabilidade $1/2$. Portanto, o número esperado de execuções do simulador é no máximo 2, o que resulta num tempo esperado de execução que é polinomial no tamanho da entrada.

Finalmente, para concluir que as respostas têm a mesma distribuição, *condicionado a* $b' = b$ *e ao valor de* h, *o valor* a *computado por* M *é o mesmo que* P *envia quando a primeira mensagem foi* h *e a resposta de* V *foi* b: o valor enviado de a por P é

$$a = \begin{cases} \text{log discreto de } h & \text{se } b = 0 \\ \text{log discreto de } h\beta & \text{se } b = 1 \end{cases}$$

e no caso de M, como $\alpha^{r'} = h\beta^{b'}$ e $a = r'$

$$a = \begin{cases} \text{log discreto de } h & \text{se } b' = 0 \\ \text{log discreto de } h\beta & \text{se } b' = 1 \end{cases}.$$

Nessa condições a resposta de V_2 é a mesma resposta que V daria. Assim toda escrita de V na fita ocorre com as mesma distribuições de V_1 e V_2 donde concluímos que M e o sistema (V, P) respondem com a mesma distribuição.

ISO $\in$ PZK Vejamos um sistema de prova interativa com conhecimento zero para a linguagem ISO a qual não sabemos se pertence a BPP.

Sejam G_0 e G_1 grafos isomorfos sobre o mesmo conjunto de vértices, sem perda de generalidade $V := \{1, 2, \ldots, n\}$. Consideremos uma permutação dos vértices $\pi \in_R \mathbb{S}_n$ e $\pi(G_X)$ o grafo isomorfo a G_X pela permutação π. Então as variáveis aleatórias $X \in_R \{0, 1\}$ e $\pi(G_X)$ são independentes se X e π são independentes.

De fato, sejam G_0 e G_1 grafos isomorfos e $X \in_R \{0, 1\}$ e $\pi \in_R \mathbb{S}_n$ variáveis aleatórias independentes. Os conjuntos de permutações $\Pi_0 = \{\sigma : \sigma(G_0) = H\}$ e $\Pi_1 = \{\sigma : \sigma(G_1) = H\}$ têm a mesma cardinalidade, portanto, $\mathbb{P}[\,\pi \in \Pi_1\,] = \mathbb{P}[\,\pi \in \Pi_0\,]$. Como $\mathbb{P}[\,\pi(G_X) = H \mid X = 1\,] = \mathbb{P}[\,\pi(G_1) = H\,] = \mathbb{P}[\,Y \in \Pi_1\,]$ e, analogamente, $\mathbb{P}[\,\pi \in \Pi_0\,] = \mathbb{P}[\,\pi(G_X) = H \mid X = 0\,]$ temos, pelo Teorema de Bayes, que

$$\begin{aligned} \mathbb{P}[\,X = 1 \mid \pi(G_X) = H\,] &= \frac{\mathbb{P}[\,\pi(G_X) = H \mid X = 1\,]\mathbb{P}[\,X = 1\,]}{\mathbb{P}[\,\pi(G_X) = H \mid X = 1\,]\mathbb{P}[\,X = 1\,] + \mathbb{P}[\,\pi(G_X) = H \mid X = 0\,]\mathbb{P}[\,X = 0\,]} \\ &= \frac{\mathbb{P}[\,\pi(G_X) = H \mid X = 1\,]\mathbb{P}[\,X = 1\,]}{\mathbb{P}[\,\pi(G_X) = H \mid X = 1\,]\big(\mathbb{P}[\,X = 1\,] + \mathbb{P}[\,X = 0\,]\big)} \\ &= \frac{\mathbb{P}[\,\pi(G_X) = H \mid X = 1\,]\mathbb{P}[\,X = 1\,]}{\mathbb{P}[\,\pi(G_X) = H \mid X = 1\,]} = \frac{1}{2} \end{aligned}$$

e $\mathbb{P}[\,X = 0 \mid \pi(G_X) = H\,] = 1/2$ pela mesma razão, assim, para todo grafo H isomorfo a G_0 e G_1

$$\mathbb{P}[\,X = 1 \mid \pi(G_X) = H\,] = \mathbb{P}[\,X = 0 \mid \pi(G_X) = H\,] = \frac{1}{2} = \mathbb{P}[\,X = 1\,] = \mathbb{P}[\,X = 0\,]$$

ou seja, conhecer a cópia isomorfa não dá nenhuma pista sobre o bit sorteado.

Suponha agora que um Provador queira provar que dois grafos dados, G_0 e G_1, são isomorfos, e que ele *de fato conhece um isomorfismo*. Vamos apresentar um protocolo para este problema em que tanto o provador quanto o verificador são eficientes.

Entrada: G_0 e G_1 grafos.

P: $\pi \xleftarrow{R} \mathbb{S}_n$;
$H \leftarrow \pi(G_1)$;

P→V: H;

V: $i \leftarrow_R \{0,1\}$;

V→P: i;

P: escolhe uma permutação φ, se possível com $\varphi(G_0) = G_1$;
se $i = 1$ então $\sigma \leftarrow \pi$;
se $i = 0$, então $\sigma \leftarrow \pi \circ \varphi$;

P→V: σ;

V: se $H = \sigma(G_i)$ então responde 1,
senão responde 0.

Se os dois grafos de entrada são isomorfos, P conhece um isomorfismo e o usa em φ e então o Verificador sempre irá responder 1. Suponha que os grafos da entrada não são isomorfos. Não importa como é construído o grafo H que o Provador envia ao Verificador, sempre haverá $i \in \{0,1\}$ tal que H e G_i não são isomorfos, portanto, se o Verificador cumpre seu papel então aceita a entrada somente no caso de sortear o i correto com probabilidade 1/2. Repetindo o protocolo obtemos que a probabilidade de erro no máximo 1/4.

Resta verificarmos que vale o *conhecimento zero* no protocolo. Precisamos mostrar que o Provador não transmite conhecimento qualquer que seja o verificador, mesmo que esse trapaceie com o objetivo de extrair informação do Provador. Seja V um programa fixo e arbitrário de um algoritmo probabilístico de tempo polinomial interagindo com P de acordo com o protocolo. Vamos projetar um algoritmo M de tempo polinomial que gera uma distribuição de probabilidade que é idêntica à distribuição de probabilidade induzida nas memórias de bits aleatório e de comunicação de V durante sua interação. A estratégia de V pode ser vista como duas funções: (1) a computação de V com H (a primeira mensagem passada por P) que responde com um bit i sorteado por V e (2) a computação de V com H e σ (a duas mensagens passadas por P) é qualquer coisa que V computa depois da resposta de P

ao bit i.

Entrada: G_0 e G_1 grafos.

1. escolhe $i' \in_R \{0,1\}$ e escolhe $\pi \in_R \mathbb{S}_n$;
2. simula V com entrada G_0 e G_1 e com $H := \pi(G_{i'})$ na memória $F_{P \to V}$ para obter $i \in \{0,1\}$ após um número polinomial de passos;
3. se $i = i'$ então simula V com o envio de π por P e devolve a mesma resposta; caso contrário, recomeça a simulação.

Reparemos, e isso é um ponto importante, que se G_0 e G_1 são isomorfos então o grafo H gerado não revela a escolha de i', portanto $i = i'$ com probabilidade 1/2; o seguinte resultado, de que $\pi(G_{i'})$ não dá informação sobre i' será provado adiante nessa seção.

O simulador escolhe i' uniformemente na esperança de que, a frente, o Verificador escolha $i = i'$. Agora, para qualquer algoritmo aleatorizado A que com entrada $H = \pi(G_{i'})$ tenta computar i' vale

$$\begin{aligned}
\mathbb{P}[\, A(\pi(G_{i'})) = i' \,] &= \sum_G \mathbb{P}[\, A(G) = i' \mid \pi(G_{i'}) = G \,]\mathbb{P}[\, \pi(G_{i'}) = G \,] \\
&= \sum_G \sum_b \mathbb{P}[\, [A(G) = b] \cap [i' = b] \mid \pi(G_{i'}) = G \,]\mathbb{P}[\, \pi(G_{i'}) = G \,] \\
&= \sum_G \sum_b \mathbb{P}[\, A(G) = b \,]\mathbb{P}[\, i' = b \mid \pi(G_{i'}) = G \,]\mathbb{P}[\, \pi(G_{i'}) = G \,] \\
&= \sum_G \sum_b \mathbb{P}[\, A(G) = b \,]\frac{1}{2}\mathbb{P}[\, \pi(G_{i'}) = H \,] = \frac{1}{2}.
\end{aligned}$$

Sendo assim, $\mathbb{P}[\, i = i' \,] = 1/2$ e dado que $i = i'$ o Verificador responde com a distribuição correta e, além disso, M reinicia no passo 3 com probabilidade 1/2, portanto o número esperado de rodadas antes de terminar é 2 e como cada rodada é de tempo polinomial, concluímos que o simulador é de tempo esperado polinomial.

Para finalizar, notemos que o Verificador do protocolo dado acima, no inicio desse exemplo, executa em tempo polinomial (probabilístico) e o mesmo vale para o Provador quando um isomorfismo φ é dado como entrada auxiliar. Também, a rigor, precisamos mostrar que vale a propriedade de *conhecimento zero* para qualquer verificador que interage com P, isto é, a análise acima aplica-se no caso em que V segue o protocolo mas esse não é sempre o caso, só o protocolo de P é fixo. Uma prova detalhada dessa análise pode ser vista em Goldreich (2001) ou em Goldreich, Micali e Wigderson (1991).

4.4 EXERCÍCIOS

Exercício 4.22 (Arora e Barak, 2009). Prove que toda função $f\colon \{0,1\}^* \to \{0,1\}$ computável por uma Máquina de Turing com alfabeto Γ em tempo $t(n)$ também é computável por uma Máquina de Turing com alfabeto $\{0,1,\text{b}\}$ em tempo $O(\log(|\Gamma|)t(n))$.

Exercício 4.23. Seja $I \subseteq \{0,1\}^*$ uma linguagem indecidível por máquina de Turing (por exemplo, a linguagem definida pelo Problema da Parada, definido na página 199). Defina

$$L = \{1^n : n \text{ em base 2 pertence a } I\}.$$

Prove que L é indecidível. Exiba uma família de circuitos $(C_n : n \in \mathbb{N})$ com número de vértices polinomial em n e que decide pertinência em L.

Exercício 4.24 (Problema de busca NP). Um problema de busca definido pela relação R é um problema de busca NP se dados x e y, decidir $(x,y) \in R$ pode ser feito em tempo polinomial e existe um polinômio p tal que se $(x,y) \in R$ então $|y| \leqslant p(|x|)$. Prove que para todo problema de busca NP, existe um problema de decisão NP tal que se o problema de decisão tem solução em tempo $T(n)$, então o problema de busca tem solução em tempo $O(n^{c_1} \cdot T(n^{c_2}))$, para constantes positivas c_1 e c_2. Conclua que P = NP se, e só se, todo problema de busca NP tem solução em tempo polinomial.

Exercício 4.25. (Adleman, 1978) Defina *circuito booleano aleatorizado* como um circuito booleano que além das n portas da entrada contém portas que recebem, cada uma, um bit aleatório de modo uniforme e independente; esse circuito computa uma função $f\colon \{0,1\}^n \to \{0,1\}$ se quando $f(x_1,\ldots,x_n) = 0$ o circuito com entrada $(x_1,\ldots,x_n)$ responde 0 e quando $f(x_1,\ldots,x_n) = 1$ o circuito com entrada $(x_1,\ldots,x_n)$ responde 1 com probabilidade pelo menos 1/2. Prove que se $f\colon \{0,1\}^* \to \{0,1\}$ é computada por uma família de circuitos booleanos aleatorizados de tamanho polinomial então f é computada por uma família de circuitos booleanos de tamanho polinomial (que não usa bits aleatórios).

Exercício 4.26. Prove que

1. BPP = coBPP.
2. ZPP = coZPP.
3. Se NP ≠ coNP então P ≠ NP.
4. Se NP $\not\subseteq$ P/poly então P ≠ NP.
5. Se NP ⊆ BPP então NP = RP.

6. Se $\mathsf{RP} \subseteq \mathsf{coPP}$ então $\mathsf{ZPP} = \mathsf{RP}$ e $\mathsf{RP} \subseteq \mathsf{NP} \cap \mathsf{coNP}$. (Atualmente, não se sabe $\mathsf{RP} = \mathsf{coRP}$, se $\mathsf{RP} \subseteq \mathsf{NP} \cap \mathsf{coNP}$ e se $\mathsf{NP} = \mathsf{coNP}$.)

Exercício 4.27. Sejam M uma máquina de Turing probabilística de tempo polinomial e L uma linguagem tais que para constantes $0 < \varepsilon_1 < \varepsilon_2 < 1$ e todo $w \in \{0,1\}^*$

1. se $w \in \mathsf{L}$, então $\mathbb{P}[\,\mathsf{M}(w) = 1\,] \geqslant \varepsilon_2$, e
2. se $w \notin \mathsf{L}$, então $\mathbb{P}[\,\mathsf{M}(w) = 1\,] < \varepsilon_1$;

prove que L está em BPP.

Exercício 4.28. A classe PP — *Probabilistic Polynomial time* — é a classe das linguagens L para as quais existe um algoritmo probabilístico M de tempo polinomial tal que

$$\mathbb{P}[\,\mathsf{M}(w) \neq \mathsf{L}(w)\,] \leqslant 1/2, \text{ para todo } w \in \{0,1\}^*.$$

Prove as seguintes inclusões

1. $\mathsf{P} \subseteq \mathsf{ZPP} \subseteq \mathsf{RP} \subset \mathsf{NP} \subseteq \mathsf{PP}$.
2. $\mathsf{RP} \subseteq \mathsf{BPP} \subseteq \mathsf{PP}$.
3. $\mathsf{PP} = \mathsf{coPP}$.

Exercício 4.29. Suponha que exista um algoritmo determinístico eficiente A que, dado inteiros a e $\mathsf{N} = pq > 1$, para p e q primos distintos, determina de modo eficiente uma solução de $x^2 \equiv a \pmod{\mathsf{N}}$. Sorteie $b \in \mathbb{Z}_{\mathsf{N}}^*$ e compute $b^2 \bmod \mathsf{N}$, em seguida use o algoritmo A para determinar uma solução x de $\mathsf{X}^2 \equiv b^2 \pmod{\mathsf{N}}$. Verifique que com probabilidade $1/2$ temos $x \not\equiv \pm b \pmod{\mathsf{N}}$. Prove que nesse caso $\mathrm{mdc}(b - x, \mathsf{N}) \in \{p, q\}$. Conclua que se é possível determinar raiz quadrada módulo N de maneira eficiente então é possível fatorar N de maneira eficiente.

Exercício 4.30. Dê uma prova direta (isto é, sem usar o teorema 4.14) de que $\mathsf{RP} \subseteq \mathsf{P/poly}$.

Exercício 4.31. Prove que a linguagem do Exemplo 4.15 está em Π_2.

Exercício 4.32. Mostre que se o i-ésimo bit de $p \in (0,1)$ pode ser determinado em tempo polinomial em i então a distribuição sobre $\{0,1\}$ com $\mathbb{P}(1) = p$ pode ser simulada por um algoritmo probabilístico de tempo esperado $O(1)$.

Exercício 4.33. Mostre que uma moeda com $\mathbb{P}[\,\text{cara}\,] = 1/2$ pode ser simulada por um algoritmo probabilístico que lança moeda com $\mathbb{P}[\,\text{cara}\,] = p$ com tempo esperado $O(1/(p - p^2))$.

Exercício 4.34. Mostre que se trocarmos $1/2$ por $1/p(|w|)$ ou por $1-(1/2)^{p(|w|)}$, para qualquer polinômio positivo p nas definições de RP e de coRP, obteremos as mesmas classes de linguagens.

Exercício 4.35. Prove que se trocarmos a constante na definição de sistema interativo por $(1/2)^{n^c}$, para qualquer constante positiva c então a classe de linguagens reconhecidas não muda, ou seja, é IP.

Exercício 4.36. Como consequência de $\mathsf{AM}(p(n)) = \mathsf{AM}(p(n)+1) = \mathsf{MA}(p(n)+1)$ temos: se $\mathsf{coNP} \subseteq \mathsf{AM}$ então a hierarquia polinomial colapsa, $\Sigma_2 = \Pi_2 = \mathsf{AM}$ (veja o Problema 14, página 230, e o parágrafo seguinte). Para provar esse corolário

1. Prove que $\mathsf{AM} \subseteq \Pi_2$ e que $\mathsf{MA} \subseteq \Sigma_2 \cap \Pi_2$ (generalize a prova de que $\mathsf{BPP} \subseteq \mathsf{PH}$, seção 4.2.2).

2. Prove que se $\mathsf{L} \in \Sigma_2$ então existe $\mathsf{L}_1 \in \mathsf{coNP}$ tal que

$$w \in \mathsf{L} \Leftrightarrow \exists u \in \{0,1\}^{p(|w|)},\ (w,u) \in \mathsf{L}_1$$

   para algum polinômio p (veja (4.4), página 222). Conclua que $\mathsf{L} \in \mathsf{AM}$.

3. Prove que $\Sigma_2 = \Pi_2 = \mathsf{AM}$.

Exercício 4.37. (Arora e Barak, 2009, Lema 9.2) Considere um sistema de codificação (seção 1.2.1) com funções de codificação E_k e de decodificação D_k computáveis em tempo polinomial com chave menor que o texto, em que k é uma chave aleatória compartilhada entre as partes, e $\mathsf{D}_k(\mathsf{E}_k(w)) = w$ para todo texto w.

Prove que se $\mathsf{P} = \mathsf{NP}$ então existe um algoritmo A de tempo polinomial tal que para todo w, existem $x_0, x_1 \in \{0,1\}^{|w|}$ com

$$\mathop{\mathbb{P}}_{\substack{b \in_R \{0,1\} \\ k \in_R \{0,1\}^n}} [\mathsf{A}(\mathsf{E}_k(x_b)) = b] \geqslant \frac{3}{4}$$

em que $n < |w|$. (*Dica*: tome S o conjunto das sequências binárias y tais que $y = \mathsf{E}_k(0^m)$ para algum k; $\mathsf{A}(y) = 0$ se $y \in \mathsf{S}$ e $\mathsf{A}(y) = 1$ caso contrário.)

Exercício 4.38. (Goldwasser, Micali e Rackoff, 1989) Sejam a e b inteiros, $0 < b < a$, tais que $\mathrm{mdc}(a,b) = 1$. Prove que a linguagem

$$\textsc{qr} := \{\langle a,b\rangle : b \text{ é um resíduo quadrático módulo } a\}$$

admite o seguinte sistema de prova com conhecimento zero: com entrada a, b (**P** conhece supostamente conhece s tal que $b = s^2 \pmod a$). **P** sorteia s' e envia $b' = bs'^2 \pmod a$. **V** sorteia um bit i e envia. Se $b = 0$ então **P** revela ss' (uma raiz para x'); se $b = 1$ then **P** revela s' (uma raiz para $x'x^{-1}$). **V** verifica se o valor revelado é uma raiz de $x'x^{-b}$, se for aceita.

Bibliografia

Achlioptas, Dimitris (2021). "Random Satisfiabiliy". Em: *Handbook of Satisfiability - Second Edition*. Ed. por Armin Biere *et al.* Vol. 336. Frontiers in Artificial Intelligence and Applications. IOS Press, pp. 437–462 (ver p. 247).

Achlioptas, Dimitris e Yuval Peres (2004). "The threshold for random k-SAT is $2^k \log 2 - O(k)$". Em: *J. Amer. Math. Soc.* 17.4, pp. 947–973 (ver p. 248).

Adleman, Leonard (1978). "Two theorems on random polynomial time". Em: *19th Annual Symposium on Foundations of Computer Science (Ann Arbor, Mich., 1978)*. Long Beach, Calif.: IEEE, pp. 75–83 (ver pp. 220, 241).

Agrawal, Manindra e Somenath Biswas (2003). "Primality and identity testing via Chinese remaindering". Em: *J. ACM* 50.4, 429–443 (electronic) (ver pp. 171, 172).

Agrawal, Manindra, Neeraj Kayal e Nitin Saxena (2004). "PRIMES is in P". Em: *Ann. of Math. (2)* 160.2, pp. 781–793 (ver pp. 158, 205).

Alexander, K. S., K. Baclawski e G. C. Rota (1993). "A stochastic interpretation of the Riemann zeta function". Em: *Proceedings of the National Academy of Sciences of the United States of America* 2.90, 697–699 (ver p. 125).

Arora, Sanjeev e Boaz Barak (2009). *Computational complexity*. A modern approach. Cambridge: Cambridge University Press, pp. xxiv+579 (ver pp. 66, 199, 228, 241, 243, 301).

Arvind, V. e Partha Mukhopadhyay (2008). "Derandomizing the Isolation Lemma and Lower Bounds for Circuit Size". Em: *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques*. Ed. por Ashish Goel *et al.* Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 276–289 (ver p. 94).

Babai, László (1985). "Trading Group Theory for Randomness". Em: *STOC*. Ed. por Robert Sedgewick. ACM, pp. 421–429 (ver pp. 227, 231, 232).

Bach, Eric e Jeffrey Shallit (1996). *Algorithmic Number Theory*. Cambridge, MA, USA: MIT Press (ver pp. 61, 180, 181).

Bansal, Nikhil (2010). "Constructive Algorithms for Discrepancy Minimization". Em: *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pp. 3–10 (ver p. 272).

Bartle, Robert G. (1976). *The elements of real analysis*. Second. John Wiley & Sons, New York-London-Sydney, pp. xv+480 (ver p. 316).

Ben-Or, Michael (1981). "Probabilistic Algorithms in Finite Fields". Em: *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981*. IEEE Computer Society, pp. 394–398 (ver p. 188).

Ben-Or, Michael *et al.* (1990). "Everything provable is provable in zero-knowledge". Em: *Advances in cryptology—CRYPTO '88 (Santa Barbara, CA, 1988)*. Vol. 403. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 37–56 (ver p. 236).

Bernstein, Daniel J. (1998). "Detecting perfect powers in essentially linear time". Em: *Math. Comput.* 67.223, pp. 1253–1283 (ver p. 173).

Bernstein, Sergei (1924). "On a modification of Chebyshev's inequality and of the error formula of Laplace". Em: *Ann. Sci. Inst. Sav. Ukraine, Sect. Math* 1.4, pp. 38–49 (ver p. 270).

Billingsley, P. (1979). *Probability and measure*. Wiley series in probability and mathematical statistics. Probability and mathematical statistics. Wiley (ver p. 87).

Blum, Manuel (1986). "How to prove a theorem so no one else can claim it". Em: *Proceedings of the International Congress of Mathematicians*. Vol. II. disponível em http://www.mathunion.org/ICM/ICM1986.2/. International Mathematical Union (IMU). Berkeley, pp. 1444–1451 (ver p. 310).

Boppana, Ravi B., Johan Håstad e Stathis Zachos (1987). "Does co-NP have short interactive proofs?" Em: *Inform. Process. Lett.* 25.2, pp. 127–132 (ver p. 230).

Bourgain, J. (1985). "On lipschitz embedding of finite metric spaces in Hilbert space". Em: *Israel Journal of Mathematics* 52.1, 46–52 (ver p. 287).

Carter, J. Lawrence e Mark N. Wegman (1979). "Universal classes of hashing functions". Em: *Journal of Computer and System Sciences* 18, pp. 143–154 (ver pp. 113, 265, 291).

Chazelle, B. (2000). *The Discrepancy Method: Randomness and Complexity*. Randomness and Complexity. Cambridge University Press (ver p. 272).

Chernoff, Herman (1952). "A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations". Em: *The Annals of Mathematical Statistics* 23.4, pp. 493 –507 (ver p. 270).

— (2014). "A career in statistics". Em: *Past, Present, and Future of Statistical Science*. Ed. por X. Lin *et al.* CRC Press, pp. 29–40 (ver p. 270).

Cormen, Thomas H., Charles E. Leiserson e Ronald L. Rivest (1990). *Introduction to algorithms*. The MIT Electrical Engineering and Computer Science Series. Cambridge, MA: MIT Press, pp. xx+1028 (ver pp. 127, 130).

Coron, Jean-Sébastien e Alexander May (2007). "Deterministic polynomial-time equivalence of computing the RSA secret key and factoring". Em: *J. Cryptology* 20.1, pp. 39–50 (ver p. 309).

DeMillo, Richard A. e Richard J. Lipton (1978). "A Probabilistic Remark on Algebraic Program Testing". Em: *Inf. Process. Lett.* 7.4, pp. 193–195 (ver p. 82).

Diffie, Whitfield e Martin E. Hellman (1976). "New directions in cryptography". Em: *IEEE Trans. Information Theory* IT-22.6, pp. 644–654 (ver pp. 177, 306).

Ding, Jian, Allan Sly e Nike Sun (2022). "Proof of the satisfiability conjecture for large k". Em: *Ann. of Math. (2)* 196.1, pp. 1–388 (ver p. 248).

ElGamal, Taher (1985). "A public key cryptosystem and a signature scheme based on discrete logarithms". Em: *Advances in cryptology (Santa Barbara, Calif., 1984)*. Vol. 196. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 10–18 (ver p. 306).

Erdős, P. (1956). "On pseudoprimes and Carmichael numbers". Em: *Publ. Math. Debrecen* 4, pp. 201–206 (ver p. 161).

Feller, William (1968). *An introduction to probability theory and its applications. Vol. I*. Third edition. New York: John Wiley & Sons Inc., pp. xviii+509 (ver p. 100).

Fortnow, L. (1987). "The complexity of perfect zero-knowledge". Em: *Proceedings of the nineteenth annual ACM symposium on Theory of computing*. STOC '87. New York, New York, United States: ACM, pp. 204–209 (ver p. 236).

Freivalds, Rusins (1977). "Probabilistic Machines Can Use Less Running Time". Em: *IFIP Congress*, pp. 839–842 (ver p. 76).

Gao, Shuhong e Daniel Panario (1997). "Tests and Constructions of Irreducible Polynomials over Finite Fields". Em: *Foundations of Computational Mathematics*. Ed. por Felipe Cucker e Michael Shub. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 346–361 (ver p. 188).

Garfinkel, Simson (1994). *PGP: Pretty Good Privacy*. O'Reilly (ver p. 161).

Gathen, Joachim von zur e Jürgen Gerhard (2013). *Modern Computer Algebra*. 3ª ed. Cambridge University Press (ver p. 183).

Gelbaum, B.R. e J.M.H. Olmsted (1964). *Counterexamples in Analysis*. Dover books on mathematics. Holden-Day (ver p. 15).

Gill, John T. (1974). "Computational complexity of probabilistic Turing machines". Em: *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*. STOC '74. Seattle, Washington, USA: Association for Computing Machinery, 91–95 (ver p. 209).

Goldreich, O. e L. A. Levin (1989). "A hard-core predicate for all one-way functions". Em: *STOC '89: Proceedings of the twenty-first annual ACM symposium on Theory of computing*. Seattle, Washington, United States: ACM, pp. 25–32 (ver p. 311).

Goldreich, Oded (2001). *Foundations of cryptography*. Basic tools. Cambridge: Cambridge University Press, pp. xx+372 (ver p. 240).

Goldreich, Oded (2008). *Computational complexity*. A conceptual perspective. Cambridge: Cambridge University Press, pp. xxiv+606 (ver pp. 195, 196).

Goldreich, Oded, Silvio Micali e Avi Wigderson (1991). "Proofs that yield nothing but their validity, or All languages in NP have zero-knowledge proof systems". Em: *J. Assoc. Comput. Mach.* 38.3, pp. 691–729 (ver pp. 236, 240, 310, 313).

Goldwasser, S, S Micali e C Rackoff (1985). "The knowledge complexity of interactive proof-systems". Em: *STOC '85: Proceedings of the seventeenth annual ACM symposium on Theory of computing*. Providence, Rhode Island, United States: ACM Press, pp. 291–304 (ver pp. 227, 235).

Goldwasser, S e M Sipser (1986). "Private coins versus public coins in interactive proof systems". Em: *Proceedings of the eighteenth annual ACM symposium on Theory of computing*. STOC '86. Berkeley, California, United States: ACM, pp. 59–68 (ver p. 232).

Goldwasser, Shafi e Silvio Micali (1984). "Probabilistic encryption". Em: *Journal of Computer and System Sciences* 28.2, pp. 270 –299 (ver p. 297).

Goldwasser, Shafi, Silvio Micali e Charles Rackoff (1989). "The knowledge complexity of interactive proof systems". Em: *SIAM J. Comput.* 18.1, pp. 186–208 (ver p. 243).

Graham, Paul (2002). *A Plan for Spam*. Acesso em 06/04/2009. URL: http://www.paulgraham.com/spam.html (ver p. 34).

Graham, Ronald L., Donald E. Knuth e Oren Patashnik (1994). *Concrete mathematics*. Second. A foundation for computer science. Reading, MA: Addison-Wesley Publishing Company, pp. xiv+657 (ver pp. 175, 286).

Harman, Glyn (2005). "On the Number of Carmichael Numbers up to x". Em: *Bulletin of the London Mathematical Society* 37.5, pp. 641–650. eprint: https://londmathsoc.onlinelibrary.wiley.com/doi/pdf/10.1112/S0024609305004686 (ver p. 161).

Harvey, David e Joris Van Der Hoeven (mar. de 2019). "Integer multiplication in time $O(n \log n)$". working paper or preprint (ver p. 65).

Håstad, Johan (jul. de 2001). "Some Optimal Inapproximability Results". Em: *J. ACM* 48.4, pp. 798–859 (ver pp. 136, 138).

Hoeffding, Wassily (1994). "Probability inequalities for sums of bounded random variables". Em: *The collected works of Wassily Hoeffding*, pp. 409–426 (ver pp. 270, 280).

Hopcroft, John E., Rajeev Motwani e Jeffrey D. Ullman (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley (ver p. 199).

Håstad, Johan *et al.* (mar. de 1999). "A Pseudorandom Generator from Any One-way Function". Em: *SIAM J. Comput.* 28.4, pp. 1364–1396 (ver p. 302).

Impagliazzo, Russell e Avi Wigderson (1999). "P = BPP if E requires exponential circuits: derandomizing the XOR lemma". Em: *STOC '97 (El Paso, TX)*. New York: ACM, 220–229 (electronic) (ver p. 220).

Ireland, Kenneth e Michael Rosen (1990). *A classical introduction to modern number theory*. Second. Vol. 84. Graduate Texts in Mathematics. New York: Springer-Verlag, pp. xiv+389 (ver p. 300).

Johnson, David S. (1974). "Approximation algorithms for combinatorial problems". Em: *Journal of Computer and System Sciences* 9.3, pp. 256–278 (ver p. 136).

Johnson, William B. e Joram Lindenstrauss (1984). "EXTENSIONS OF LIPSCHITZ MAPPINGS INTO A HILBERT SPACE". Em: *Contemporary Mathematics* 26, pp. 189–206 (ver p. 288).

Karger, David R. (1993). "Global min-cuts in RNC, and other ramifications of a simple min-cut algorithm". Em: *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (Austin, TX, 1993)*. New York: ACM, pp. 21–30 (ver p. 74).

Karp, Richard M. e Richard J. Lipton (1980). "Some connections between nonuniform and uniform complexity classes". Em: *STOC '80: Proceedings of the twelfth annual ACM symposium on Theory of computing*. Los Angeles, California, United States: ACM, pp. 302–309 (ver p. 225).

Karp, Richard M. e Michael Luby (1983). "Monte-Carlo algorithms for enumeration and reliability problems". Em: *Foundations of Computer Science, IEEE Annual Symposium on* 0, pp. 56–64 (ver p. 293).

Kimbrel, Tracy e Rakesh Kumar Sinha (1993). "A probabilistic algorithm for verifying matrix products using $O(n^2)$ time and $\log_2 n + O(1)$ random bits". Em: *Inform. Process. Lett.* 45.2, pp. 107–110 (ver p. 78).

Klivans, Adam R. e Daniel A. Spielman (2001). "Randomness efficient identity testing of multivariate polynomials". Em: *Proceedings on 33rd Annual ACM Symposium on Theory of Computing, July 6-8, 2001, Heraklion, Crete, Greece*, pp. 216–223 (ver p. 94).

Knuth, Donald E. (1981). *The art of computer programming. Vol. 2*. Second. Seminumerical algorithms, Addison-Wesley Series in Computer Science and Information Processing. Addison-Wesley Publishing Co., Reading, Mass., pp. xiii+688 (ver pp. 76, 298).

Lautemann, Clemens (1983). "BPP and the Polynomial Hierarchy". Em: *Inf. Process. Lett.* 17.4, pp. 215–217 (ver p. 222).

Leeuw, K. de *et al.* (1970). "Computability by Probabilistic Machines". Em: *Journal of Symbolic Logic* 35.3, pp. 481–482 (ver pp. 209, 211).

Lehmann, Daniel e Michael O. Rabin (1981). "On the advantages of free choice: a symmetric and fully distributed solution to the dining philosophers problem". Em: *POPL '81: Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Williamsburg, Virginia: ACM, pp. 133–138 (ver p. 85).

Lenstra, H. W. (2002). "Primality Testing with Gaussian Periods". Em: *FST TCS 2002: Foundations of Software Technology and Theoretical Computer Science*. Ed. por Manindra Agrawal e Anil Seth. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 1–1 (ver p. 158).

Lidl, Rudolf e Harald Niederreiter (1997). *Finite fields*. Second. Vol. 20. Encyclopedia of Mathematics and its Applications. With a foreword by P. M. Cohn. Cambridge: Cambridge University Press, pp. xiv+755 (ver pp. 183, 185, 186).

Linial, Nathan, Eran London e Yuri Rabinovich (1994). "The geometry of graphs and some of its algorithmic applications". Em: *Combinatorica* 15, pp. 215–245 (ver p. 287).

Lund, Carsten *et al.* (1992). "Algebraic methods for interactive proof systems". Em: *J. Assoc. Comput. Mach.* 39.4, pp. 859–868 (ver p. 230).

Menezes, Alfred J., Paul C. van Oorschot e Scott A. Vanstone (1997). *Handbook of applied cryptography*. CRC Press Series on Discrete Mathematics and its Applications. With a foreword by Ronald L. Rivest. Boca Raton, FL: CRC Press, pp. xxviii+780 (ver pp. 177, 305).

Miller, Gary L. (1975). "Riemann's hypothesis and tests for primality". Em: *Seventh Annual ACM Symposium on Theory of Computing (Albuquerque, N.M., 1975)*. Assoc. Comput. Mach., New York, pp. 234–239 (ver p. 164).

Mitzenmacher, Michael e Eli Upfal (2005). *Probability and computing*. Randomized algorithms and probabilistic analysis. Cambridge: Cambridge University Press, pp. xvi+352 (ver p. 77).

Monier, Louis (1980). "Evaluation and comparison of two efficient probabilistic primality testing algorithms". Em: *Theoretical Computer Science* 12.1, pp. 97 –108 (ver p. 158).

Mulmuley, Ketan, Umesh V. Vazirani e Vijay V. Vazirani (1987). "Matching is as easy as matrix inversion". Em: *Combinatorica* 7.1, pp. 105–113 (ver p. 49).

Nisan, Noam (1992). "Pseudorandom generators for space-bounded computation". Em: *Combinatorica* 12.4, pp. 449–461 (ver p. 270).

O'Donnell, Ryan (2021). *Analysis of Boolean Functions*. arXiv: 2105.10386 [cs.DM] (ver p. 205).

Papadimitriou, Christos H. (1994). *Computational complexity*. Reading, MA: Addison-Wesley Publishing Company, pp. xvi+523 (ver pp. 201, 205, 221, 224).

Pugh, William (1989). "Skip lists: a probabilistic alternative to balanced trees". Em: *Algorithms and data structures (Ottawa, ON, 1989)*. Vol. 382. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 437–449 (ver p. 145).

Raab, Martin e Angelika Steger (1998). ""Balls into Bins- A Simple and Tight Analysis". Em: *Proceedings of the Second International Workshop on Randomization and Approximation Techniques in Computer Science*. RANDOM '98. Berlin, Heidelberg: Springer-Verlag, pp. 159–170 (ver pp. 155, 294).

Rabin, Michael O. (1980a). "Probabilistic algorithm for testing primality". Em: *J. Number Theory* 12.1, pp. 128–138 (ver p. 164).

— (1980b). "Probabilistic algorithms in finite fields". Em: *SIAM J. Comput.* 9.2, pp. 273–280 (ver p. 187).

Ribenboim, Paulo (1996). *The new book of prime number records*. New York: Springer-Verlag, pp. xxiv+541 (ver p. 176).

Rivest, R. L., A. Shamir e L. Adleman (1978). "A method for obtaining digital signatures and public-key cryptosystems". Em: *Commun. ACM* 21.2, pp. 120–126 (ver p. 308).

Rosenthal, Jeffrey S. (2006). *A first look at rigorous probability theory*. Second. World Scientific Publishing Co. Pte. Ltd., Hackensack, NJ, pp. xvi+219 (ver p. 15).

Ross, Sheldon M. (2010). *A first course in Probability*. 8th. New Jersey: Prentice Hall (ver p. 30).

Rosser, Barkley (1941). "Explicit Bounds for Some Functions of Prime Numbers". Em: *American Journal of Mathematics* 63.1, pp. 211–232 (ver p. 219).

Savage, John E. (1998). *Models of computation - exploring the power of computing*. Addison-Wesley, pp. I–XXIII, 1–672 (ver pp. 193, 201).

Schwartz, Jacob T. (1979). "Probabilistic algorithms for verification of polynomial identities". Em: *Symbolic and algebraic computation (EUROSAM '79, Internat. Sympos., Marseille, 1979)*. Vol. 72. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 200–215 (ver p. 82).

Shamir, Adi (1992). "IP = PSPACE". Em: *J. Assoc. Comput. Mach.* 39.4, pp. 869–877 (ver p. 230).

Shen, A. (out. de 1992). "IP = SPACE: simplified proof". Em: *J. ACM* 39.4, pp. 878–880 (ver p. 230).

Sipser, Michael (1996). *Introduction to the Theory of Computation*. Course Technology (ver pp. 204–206).

Spencer, Joel (jun. de 1985). "Six standard deviations suffice". Em: *Transactions of the American Mathematical Society* 289.2, pp. 679–706 (ver p. 272).

Vadhan, Salil Pravin e Shafi Goldwasser (1999). "A Study of Statistical Zero-Knowledge Proofs". AAI0801528. Tese de dout. USA: Massachusetts Institute of Technology (ver p. 315).

Valiant, L. G. e V. V. Vazirani (nov. de 1986). "NP is as easy as detecting unique solutions". Em: *Theor. Comput. Sci.* 47.1, 85–93 (ver pp. 214, 217).

Vuillemin, Jean (1980). "A unifying look at data structures". Em: *Commun. ACM* 23.4, pp. 229–239 (ver p. 283).

Yao, Andrew C. (1982). "Theory and applications of trapdoor functions". Em: *23rd annual symposium on foundations of computer science (Chicago, Ill., 1982)*. New York: IEEE, pp. 80–91 (ver p. 304).

Zippel, Richard (1979). "Probabilistic algorithms for sparse polynomials". Em: *Symbolic and algebraic computation (EUROSAM '79, Internat. Sympos., Marseille, 1979)*. Vol. 72. Lecture Notes in Comput. Sci. Berlin: Springer, pp. 216–226 (ver p. 82).