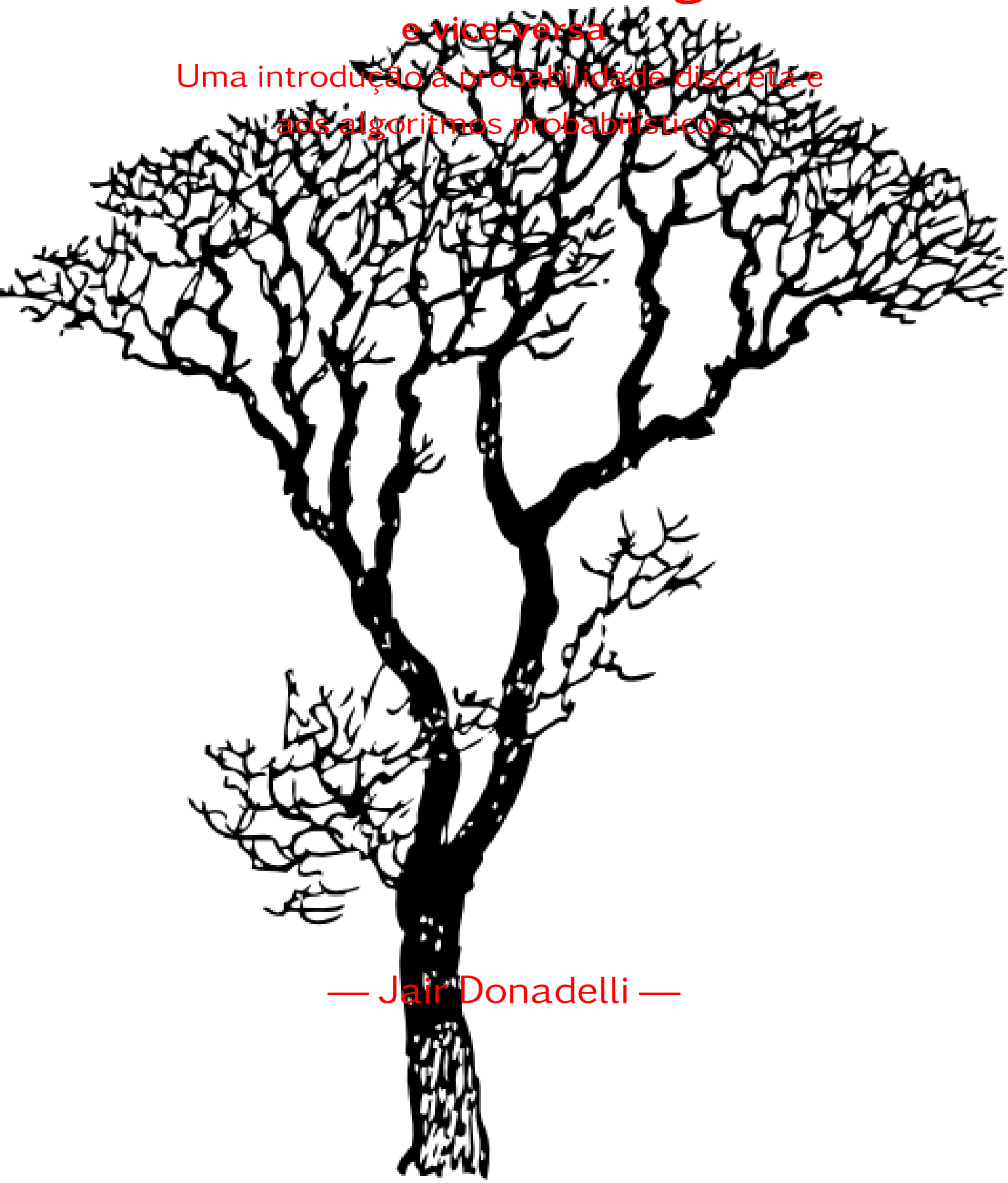
Probabilidade com Algoritmos

e vice-versa

Uma introdução à probabilidade discreta e aos algoritmos probabilísticos



— Jair Donadelli —

4 | Computação probabilística

Conteúdo

Algoritmos, como são estudados atualmente em disciplinas introdutórias e como foram apresentados nos capítulos anteriores, são descritos no que costumamos chamar de *pseudocódigo*, uma mistura de algumas palavras-chave em português com sentenças construídas como em uma linguagem de programação estruturada como a linguagem C. Apesar de ser um conceito antigo, o algoritmo de Euclides tem mais de dois mil anos, uma formalização satisfatória do que é um algoritmo só se deu por volta de 1936 com os trabalhos de alguns matemáticos, notadamente, Alan Turing. A aleatorização de algoritmos parece ser bem mais recente na história dos algoritmos. No século 18, Buffon usou um método aleatorizado para determinar uma aproximação para π. Em um artigo de 1917, Henry Pocklington apresentou um algoritmo para encontrar $\sqrt{a} \bmod p$ que dependia de escolher números ao acaso. Em 1953 foi publicado um método para se obter uma sequência de amostras aleatórias de acordo com uma distribuição de probabilidade para a qual a amostragem direta é difícil, o conhecido algoritmo Metropolis–Hastings.

O estudo moderno e sistematizado de modelos probabilísticos de computação começou por volta de 1976 com Michael Rabin, Robert Solovay e Volker Strassen que escreveram algoritmos aleatorizados muito eficientes para testar a primalidade de um número. Atualmente, para esse problema, conhecemos um algoritmo determinístico eficiente, porém não tanto quanto o probabilístico. Também por volta de 1976, John Gill estendeu a definição de Máquina de Turing para incluir bits aleatórios e definiu classes de complexidade computacional para essas máquinas.

algoritmos probabilísticos– Computacao probabilistica

4.1 Modelos de Computação

Esta seção é dedicada a apresentação de modelos de computação de maneira *resumida* e *informal*. O leitor interessado em aprender mais sobre esses modelos que apresentaremos sugerimos o texto de Savage (1998).

A Máquina de Turing, proposta por Alan Turing, e o λ-cálculo, proposto por Alonzo Church, foram os primeiros modelos formais de computação que capturam o conceito do que intuitivamente os matemáticos chamam de algoritmo. Esses modelos pioneiros, o modelo genérico dado por uma linguagem de programação (pela linguagem C por exemplo) e outros modelos que vieram depois como o modelo formal RAM, são equivalentes no sentido de que tudo que pode ser computado em um também pode ser no outro. Simulações entre modelos formais e um modelo genérico (como o fornecido pela linguagem C) são possíveis e não há muita perda de desempenho sob condições razoáveis o que torna o estudo de classes de complexidade independente do modelo adotado. Os modelos formais de computação ajudam na compreensão das dificuldades inerentes de se resolver problemas computacionais, têm a vantagem de serem mais simples e bem definidos que as linguagens de programação e a desvantagem de serem mais difíceis de descrever um algoritmo específico.

A especificação de um modelo de computação inclui uma descrição do que é um dispositivo (máquina ou programa) do modelo, como uma *entrada* é apresentada, o que é permitido para o dispositivo do modelo calcular para que a *saída* seja obtida a partir da entrada. Além disso, o modelo deve fornecer uma noção de *passo elementar* de computação da qual derivamos medidas de *tempo de execução* do dispositivo. O tempo de execução de um dispositivo do modelo com uma dada entrada é o número de passos elementares necessários para concluir uma computação. Em geral, o tempo de execução é dado como função do *tamanho* da entrada segundo algum critério como, por exemplo, o de *pior caso*.

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 $\mathsf{A} \subset \mathcal{R}$ e nesse caso dizemos que A é uma **função computável**. Usaremos uma notação funcional para denotar o resultado de um processo computacional no sentido de 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 a cada instância x de um problema está associada a um conjunto $\mathsf{R}_x = \{y \in \mathcal{O} : (x, y) \in \mathcal{R}\}$ das soluções possíveis que, ocasionalmente, 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 $\mathsf{A}(x) := \varnothing$ (ou algum símbolo fora de $\mathcal{O}$). Num problema de decisão $\mathcal{O} = \{\text{sim}, \text{não}\}$ e, usualmente, especificamos um subconjunto $\mathsf{L} \subset \mathcal{I}$ 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, decidir se um número é primo tem $\mathcal{I}$ como o conjunto dos inteiros positivos e L 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.

Nosso foco a seguir será nos problemas de decisão. Faremos isso seguindo uma tradição na área e isso se justifica porque a apresentação e a formalização são consideravelmente mais simples 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 150 (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 $\Phi \in \mathcal{I}$ satisfazível e cada $\Phi' \in \mathcal{I}$ não satisfazível ou, ainda,

algoritmos probabilísticos Computação probabilística

podemos colocar esse problema como o problema de decidir pertinência em

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

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 satisfazível, 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 satisfazível, se não for satisfazível então $\Phi(0, x_2, \ldots, x_n)$ é satisfazível; 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 eficiente (isso será definido precisamente adiante) então A′ também será um algoritmo eficiente (um pouco mais lento, mas eficiente). ◊

Há, ainda, uma classe importante de problemas computacionais formada pelos problemas de *otimização*, como o MAX-SAT e o MAX-3-SAT apresentados na seção 2.1.5. 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 $(\mathsf{Q}, \Gamma, \Sigma, \delta, q_0, q_{\text{para}})$, em que Q é um conjunto finito de *estados* da máquina, Γ é o *alfabeto da fita* que contém um símbolo especial ƀ que representa *espaço em branco* e contém o conjunto Σ que é um *alfabeto* finito de modo que $\text{ƀ} \notin \Sigma$, $q_0 \in \mathsf{Q}$ é o *estado inicial*, $q_{\text{para}} \notin \mathsf{Q}$ é o *estado final*, e $\delta\colon \mathsf{Q} \times \Gamma \to (\mathsf{Q} \cup \{q_{\text{para}}\}) \times \Gamma \times \{\leftarrow, —, \rightarrow\}$ 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 $w = w_1 w_2 w_3 \ldots w_n \in \Sigma^*$, escrita nas n primeiras posições de uma *fita* que é infinita para a direita; as outras posições contêm o símbolo ƀ, 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 $q \in \mathsf{Q}$ e lê $\gamma \in \Gamma$ na fita então o próximo passo é dado por $\delta(q, \gamma) = (q', \gamma', \ell)$ em que $q' \in \mathsf{Q} \cup \{q_{\text{para}}\}$ 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 ƀ, e escrevemos $\mathsf{M}(w) = z$ ou, senão, $\mathsf{M}(w) = \text{ƀ}$;
- *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 151. Considere o problema de computar a paridade de uma sequência binária $x_1 x_2 x_3 x_4 \ldots x_n$ 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 $\{\mathsf{P}, \mathsf{I}, q_{\mathsf{P}}, q_{\mathsf{I}}\}$ além de q_0 e q_{para}. A função de transição é representada pelo diagrama de estados da figura 4.2, explicado com a subestrutura da figura 4.1.

algoritmos probabilísticos — Computação Probabilística

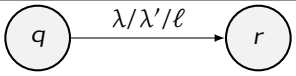


Figura 4.1: legenda para um diagrama de estados: se a máquina está no estado $q \in Q$ e lê $\lambda \in \Gamma$ na fita, então escreve $\lambda' \in \Gamma$ nessa posição e movimenta-se, de acordo com $\ell \in \{\leftarrow, —, \rightarrow\}$ ou uma posição para a esquerda, ou uma para a direita ou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

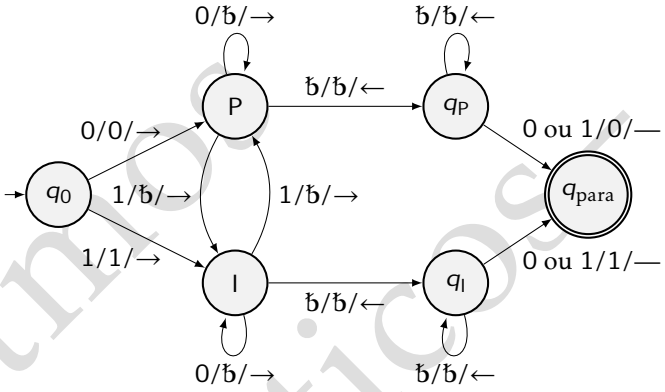


Figura 4.2: diagrama de estados da máquina de Turing para o problema da paridade.

Uma execução 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

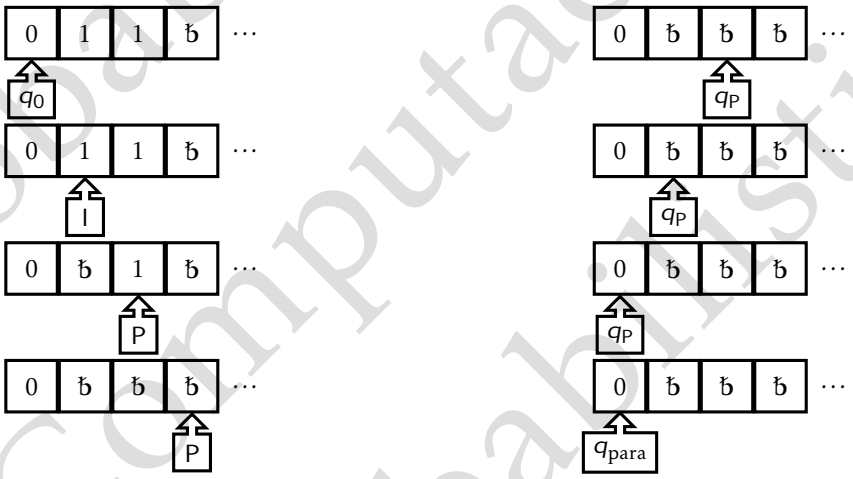


Figura 4.3: execução da máquina de Turing para a paridade com entrada 011.

lemos 0 na fita, o que significa que a a quantidade de 1's nessa entrada particular é par. ◊

Uma função $f\colon \Sigma^* \to \Sigma^*$ 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

algoritmos probabilísticos Computação probabilística

todo $w = w_1 w_2 w_3 \dots w_n \in \Sigma^*$ a máquina computa $\mathsf{M}(w)$ após no máximo $\mathsf{T}(n)$ transições.

Linguagem: um conjunto de palavras $\mathsf{L} \subset \Sigma^*$ é uma **linguagem**. Por abuso de notação denotamos também por L a função característica[1] $\mathsf{L}\colon \Sigma^* \to \{0,1\}$ do conjunto L. Se L é uma linguagem e M uma máquina de Turing, então dizemos que M **decide** L se para 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 das instâncias e respostas: 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 2, página 155) 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 $\textsc{sat} = \{\langle \Phi \rangle : \Phi \text{ é fórmula booleana satisfazível}\}$.

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

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

algoritmos probabilísticos- Computacao probabilistica

Universalidade e indecidibilidade: um fato de grande importância a respeito de máquina de Turing, chamado de universalidade, está descrito a seguir em (4.1). Primeiro o leitor deve se convencer de que uma máquina de Turing tem uma descrição finita e em seguida perceber que essa descrição pode ser codificada, ou seja, *é possível estabelecermos uma codificação binária canônica para as máquinas de Turing* (Hopcroft et al., 2000, seção 9.1.2); denotamos por M_α a máquina cuja representação binária é $\alpha \in \{0,1\}^*$. O fato importante é que

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

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

Um efeito colateral dessa codificação de máquinas de Turing é que o conjunto de todas as máquinas de Turing é enumerável, ou seja, o conjunto de todas as descrições de máquinas de Turing a mesma cardinalidade que $\mathbb{N}$. A cardinalidade do conjunto formado pelas funções $f\colon \mathbb{N} \to \{0,1\}$ é a cardinalidade do conjunto $2^{\mathbb{N}}$ das partes de $\mathbb{N}$ e pelo Teorema de Cantor não há função sobrejetiva $\mathbb{N} \to 2^{\mathbb{N}}$, 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 $(\mathsf{M}_i : i \in \mathbb{N})$ das máquinas de Turing e suponhamos que exista uma máquina de Turing H que com entrada $(\langle i\rangle, x)$ decide se a computação $\mathsf{M}_i(x)$ 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 se 𝒰(x, x) = 1 então responda 0;
3 responda 1.
```

Algoritmo 13: $\mathsf{P}(x)$

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

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

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

Modelo probabilístico: uma **máquina de Turing probabilística** M é definida como uma máquina que sempre termina e 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 $\mathsf{M}(w)$ 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}. A uma tal sequência de comprimento m associamos a probabilidade 2^{-m}. Na computação de M com entrada w a probabilidade

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

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

algoritmos probabilísticos Computação probabilística

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 $T: \mathbb{N} \to \mathbb{N}$, 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 equivalentes: 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 uma leve preferência ao *off-line*).

4.1.2 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

`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` $\leftarrow$ `Mj`	Substitui o conteúdo de M_i pelo de M_j
`Mi` $\leftarrow$ `MMj`	Substitui o conteúdo de M_i pelo de M_{M_j}
`MMi` $\leftarrow$ `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

algoritmos probabilísticos Computação Probabilística

Cada posição de memória pode armazenar um inteiro e as instruções operam sobre inteiros. As instruções são executadas sequencialmente, a cada instruções 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. Para obtermos um modelo RAM probabilístico acrescentamos a instrução `RAND(k)` que retorna um inteiro aleatório de k bits.

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 1.5.3 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.

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).

Um programa RAM é muito semelhante a um programa em linguagem de montagem (*assembly*), que é especifica a 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 $O(n^k)$.

Vimos 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.

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

algoritmos probabilísticos- Computação probabilística

A classe das funções polinomiais é fechada para soma, multiplicação e composição de funções, o que é bastante apropriado do ponto de vista teórico pois a noção de eficiência é preservada por práticas comuns de programação. Além disso, os modelos tradicionais de computação são polinomialmente equivalentes o que torna a escolha do modelo irrelevante para essa definição de eficiência. Além disso, com algum cuidado, as várias representações computacionais de objetos abstratos, como um grafo por exemplo, têm tamanhos polinomialmente relacionados, o que faz a codificação ser irrelevante.

Dito isso, 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.

Neste texto assumimos implicitamente 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 um valor lógico dentre 0 e 1; (*iii*) ou a uma variável x_i para $i \in \{1,2,\ldots,n\}$ para algum $n \in \mathbb{N}$; (*iv*) 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 e valores lógicos), 1 (porta *não* e saídas) e 2 (portas *e* e *ou*). Um circuito computa uma função booleana, *que também denotaremos por* C,

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

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 152. O circuito na figura 4.4 abaixo computa a paridade da sequência binária $x_1x_2x_3x_4$, ou seja, $s_1 = 1$ se e

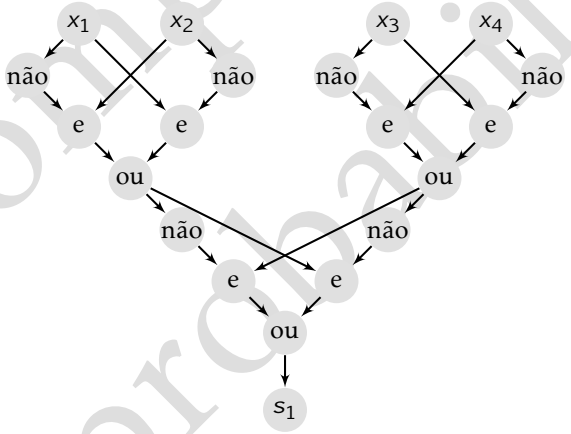


Figura 4.4: Circuito booleano que computa $s_1(x_1,x_2,x_3,x_4) = x_1 + x_2 + x_3 + x_4 \bmod 2$.

somente se o número de ocorrências de 1's na sequência é ímpar. ◊

algoritmos probabilísticos- Computação probabilística

Se L é uma linguagem, então uma família de circuitos $\mathcal{C} = (\mathsf{C}_n)_{n\in\mathbb{N}}$ **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.

Seja $\mathcal{C} = (\mathsf{C}_n)_{n\in\mathbb{N}}$ uma família de circuitos e $s\colon \mathbb{N} \to \mathbb{N}$ uma função. Dizemos que $\mathcal{C}$ é uma família de **tamanho** $s(n)$ se $|\mathsf{C}_n| \leqslant s(n)$ para todo n. A **complexidade de uma linguagem** L **com respeito a circuitos** é a função $s_{\mathsf{L}}\colon \mathbb{N} \to \mathbb{N}$ se $s_{\mathsf{L}}(n)$ 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 153. Prove que se L é uma linguagem que pode ser decidida por uma máquina de Turing de tempo $\mathsf{T}(n)$ então L pode ser decidida por uma família de circuitos $(\mathsf{C}_n)_{n\in\mathbb{N}}$ de tamanho $\mathsf{O}(\mathsf{T}(n)^2)$ (dica: suponha tempo $\mathsf{T}(n)$, exatamente, considere apenas as $\mathsf{T}(n)$ primeiras posições da fita e as $\mathsf{T}(n)$ configurações da fita dadas por uma computação, uma para cada passo (veja a figura 4.3, página 116). Cada posição da fita, depende somente de três posições da configuração imediatamente anterior).

A recíproca dessa proposição não vale pois há linguagens indecidíveis por máquina de Turing que admitem circuito de tamanho polinomial (exercício 3, página 155). 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 (exercício 5, página 156).

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 $x_1x_2 + 1$. Circuito aritmético é o modelo padrão para computação sobre polinômios, a saída de um circuito

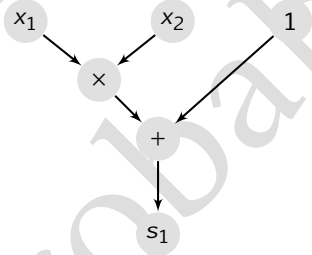


Figura 4.5: Circuito aritmético que computa $s_1(x_1, x_2) = x_1x_2 + 1$.

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 distância entre uma entrada e uma saída.

algoritmos probabilísticos — Computação probabilística

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 o tempo de execução, por exemplo. Como nos restringimos aos problemas de decisão, as classes de complexidade são apresentadas como classe 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 que equivale ao modelo RAM de custo unitário.

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 et al., 2004). Não sabemos se SAT $\in$ P pois não conhecemos nenhum algoritmo de tempo polinomial para SAT e não sabemos sequer se tal algoritmo pode existir.

A classe descrita a seguir é dada por uma definição alternativa, porém equivalente, àquela que normalmente aparece na literatura (e.g. Sipser, 1996, Papadimitriou, 1994). A definição clássica é dada no o exercício 154 a seguir.

A classe NP — *Nondeterministic Polynomial time* — é 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 $\mathbb{P}_{B \in_R \{0,1\}^{p(|w|)}}[M(w,B) = 1] > 0$,
2. se $w \notin L$ então $\mathbb{P}_{B \in_R \{0,1\}^{p(|w|)}}[M(w,B) = 0] = 1$.

Por exemplo, SAT $\in$ NP pois podemos escrever um algoritmo M que com entrada (Φ, c) verifica em tempo polinomial se c é uma valoração das variáveis da fórmula booleana Φ que torna a fórmula verdadeira ou não. Como $\Phi \in$ SAT se, e somente se, existe uma valoração c das variáveis de Φ que torna a fórmula verdadeira, temos que uma escolha aleatória para c satisfaz a fórmula $\Phi \in$ SAT com probabilidade maior que zero e qualquer escolha aleatória para c satisfaz a fórmula $\Phi \notin$ SAT com probabilidade 0. Ademais, o tamanho de $\langle c \rangle$ é polinomial no tamanho de $\langle \Phi \rangle$.

Uma sequência c como no exemplo acima é chamada de **certificado**. Se $w \in L$ e $L \in$ NP então há um certificado curto, de tamanho polinomial em $|w|$, que certifica tal fato, senão, caso $w \notin L$, não há um certificado curto de que $w \in L$. 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 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 P $\subset$ NP: se L $\in$ P e D é um algoritmo polinomial para L então um algoritmo probabilístico polinomial M para L simula D e ignora os bits aleatórios de modo que $M(w,B) = D(w)$ para todo w e todo B. Por outro lado, não é sabido se a inclusão NP $\subset$ 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 1. P $\neq$ NP?

Muito do que está descrito a seguir neste capítulo nasceu das tentativas de solucionar ou entender melhor esse problema.

Exercício 154 (*definição clássica de* NP). A definição original da classe 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. Cabe ressaltar que o sentido de "não-determinístico" que aparece na definição não é o mesmo que de "probabilístico", a definição de máquina é semelhante mas a definição de computação é muito diferente. Não nos estenderemos nessa definição (veja mais em Sipser, 1996), essa máquina não é um modelo realista de computação e uma alternativa mais atual para essa definição é a seguinte. Um **verificador** de tempo polinomial para uma linguagem L é um algoritmo de tempo polinomial M tal que para todo $w \in \{0,1\}^*$

algoritmos probabilísticos Computação probabilística

- se $w \in \mathsf{L}$ então existe $c \in \{0,1\}^{p(|w|)}$ tal que $\mathsf{M}(w,c) = 1$, para algum polinômio p e
- se $w \notin \mathsf{L}$ então $\mathsf{M}(w,c) = 0$ para todo $c \in \{0,1\}^*$.

Prove que a classe NP, como definida acima no texto, é a classe das linguagens que admitem um verificador de tempo polinomial.

Problemas completos: A linguagem L é chamada de NP-**completa** se está em NP e 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. 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 1.

O famoso Teorema de Cook–Levin estabelece que SAT é NP-completa. A partir de SAT várias outras linguagens foram provadas ser completas para NP mostrando-se uma redução para SAT.

A definição análoga para a classe P, a linguagem $\mathsf{L} \in \mathsf{P}$ é P-completa se para toda linguagem em P há um redução em tempo polinomial para L, não é interessante para a teoria pois todo problema em P é completo (por quê?). Nesse caso, há outras reduções de interesse como, por exemplo, as de espaço logarítmico.

Complemento de uma classe: se C é uma classe de complexidade e L uma linguagem então $\mathsf{L} \in \mathsf{coC}$ se, e somente se, o complemento de L, denotado $\overline{\mathsf{L}}$, está em 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, $\mathsf{coP} \subset \mathsf{P}$, portanto esses conjuntos são iguais. No entanto, na definição de NP 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 2. $\mathsf{NP} \neq \mathsf{coNP}$?

Por exemplo, a linguagem TAUT formada pelas fórmulas booleanas verdadeiras está em coNP. É fácil verificar que se $\mathsf{P} = \mathsf{NP}$ então $\mathsf{NP} = \mathsf{coNP}$.

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

O principal motivo para a definição da classe P/poly foi a esperança de poder separar P de NP. Do exercício 153 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[3] então SAT não poderia ser decidido por um algoritmo de tempo polinomial e esse fato estabeleceria que $\mathsf{P} \neq \mathsf{NP}$. Funções com cotas não triviais são abundantes, como enuncia o exercício 155 a seguir. Uma justificativa para a dificuldade de explicitarmos um problema difícil para o qual conseguimos provar uma cota inferior superpolinomial é dada na seção 4.5.

Exercício 155. Prove que existe uma função booleana $\{0,1\}^n \to \{0,1\}$ não computada por circuito de tamanho menor que $\frac{2^n}{n}$, para todo natural $n \geqslant 2$. Mais que isso, prove que a maioria dessas funções não admite circuito com tamanho menor que $2^{n/3}$ (dica: estime a quantidade de grafos que modelam circuitos e compare com a quantidade de funções).

Também decorre do exercício 153 acima que $\mathsf{P} \subset \mathsf{P/poly}$. Atualmente, não é sabido se $\mathsf{NP} \not\subset \mathsf{P/poly}$ e se esse for o caso, como explicamos no parágrafo acima, concluímos que $\mathsf{P} \neq \mathsf{NP}$, resolvendo o problema 1.

Problema 3. $\mathsf{NP} \not\subset \mathsf{P/poly}$?

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

algoritmos probabilísticos- Computação probabilística

4.2.1 Classes probabilísticas

Lembremos que quando $L \subset \{0,1\}^*$ nós também usamos L para denotar a função característica da linguagem. Além disso, as medidas de probabilidades nas definições das classes probabilísticas dizem respeito somente aos bits aleatórios usados nas computações.

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

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

Exemplo 156 (PIT). Seja $p \in \mathbb{Q}[x_1,\ldots,x_n]$ um polinômio dado por um circuito aritmético, como definido na página 122. O problema da identidade polinomial, denominado PIT (*polynomial identity testing*), é o problema de decidir se o polinômio p definido pelo circuito $\langle p \rangle$ é identicamente nulo.

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 6, página 39, sorteia n valores em $S := \{1,\ldots,2^{m+2}\}$, em tempo polinomial em m avalia o valor de $p(x_1,\ldots,x_n)$ simulando o circuito aritmético e resolve PIT com probabilidade de erro 1/4. Porém, $(x_1,\ldots,x_n)$ 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 $t \leqslant \lg p \leqslant \lg(2^{m+2})^{2^m} = (m+2)2^m$. 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 $\mathrm{PIT} \in \mathsf{BPP}$. ◊

algoritmos probabilísticos Computação probabilística

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 158 abaixo. 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 ou, de fato, muito menor que 1/3.

Exemplo 157. Sejam A, B e C matrizes quadrada de ordem n sobre $\mathbb{Z}$ e $L = \{\langle A, B, C\rangle : AB = C\}$. Podemos reescrever o algoritmo 4, página 36, usando que cada coordenada de $v \in \{0,1,2,3\}^n$ é sorteada com probabilidade 1/4 e independentemente uma das outras de modo que o algoritmo erra com probabilidade no máximo 1/4, ou seja, de acordo com o parágrafo acima, L está em BPP. ◊

Lema 158 *Para toda constante $0 < \varepsilon < 1/2$, todo polinômio p, toda linguagem $L \subset \{0,1\}^*$ 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

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

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 k ← ⌈p(|w|)/log2(1/δ))⌉
2 para i de 1 até 2k + 1 faça
3 |   m_i ← M(w);
4 se Σ_i m_i > k então responda 1;
5 senão responda 0.
```

Certamente, N é uma algoritmo probabilístico de tempo polinomial. Seja $S \in \{0,1\}^{2k+1}$ uma sequência de respostas dadas na linha 3 em uma execução de N. 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 $\varepsilon^e(1-\varepsilon)^c \leqslant \varepsilon^{k+1}(1-\varepsilon)^k$. Assim, a probabilidade de erro é

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

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

Exercício 159. Prove que se $L \in$ BPP, então existe um algoritmo probabilístico que com entrada w decide se $w \in L$ 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).

Exercício 160. Prove que $L \in$ BPP se, e só se, L é decidida por algoritmo probabilístico de tempo polinomial com probabilidade de erro $(1/2) - (1/p(n))$ nas entradas de tamanho n, para algum polinômio p (dica: como acima, simule um algoritmo que garante pertinência em BPP uma quantidade suficientemente grande de vezes e use a desigualdade de Chernoff para estimar a probabilidade de erro).

algoritmos probabilísticos — Computação probabilística

Notemos que $\mathsf{P} \subset \mathsf{BPP}$ 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 4. $\mathsf{BPP} \subset \mathsf{P}$?

Também não sabemos situar BPP com relação a NP

Problema 5. $\mathsf{BPP} \subset \mathsf{NP}$?

não sabemos se $\mathsf{BPP} \subset \mathsf{NP}$, se $\mathsf{NP} \subset \mathsf{BPP}$ 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 $\mathsf{BPP} = \mathsf{coBPP}$.

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 $w \in \{0,1\}^*$,

1. se $w \in \mathsf{L}$, então $\mathbb{P}_{\mathsf{B} \in_R \{0,1\}^{p(|w|)}}[\mathsf{M}(w, \mathsf{B}) = 0] \leqslant 1/3$, e
2. se $w \notin \mathsf{L}$, então $\mathbb{P}_{\mathsf{B} \in_R \{0,1\}^{p(|w|)}}[\mathsf{M}(w, \mathsf{B}) = 1] = 0$

ou seja, se $w \in \mathsf{L}$ então a probabilidade de erro é limitada, mas se $w \notin \mathsf{L}$ então a probabilidade de erro é nula. Assim, uma resposta 1 (sim) do algoritmo M está sempre certa, isto é $w \in \mathsf{L}$, enquanto que uma resposta 0 (não) pode ser um falso negativo, o que ocorre com probabilidade no máximo 1/3.

A constante 1/3 nos algoritmos que definem RP é arbitrária. Se realizamos k execuções independentes desse algoritmo 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.

Sejam G um grafo, $k \in \mathbb{N}$ e L a linguagem formada pelos pares $\langle \mathsf{G}, k \rangle$ tais que $\mathrm{mincut}(\mathsf{G}) \leqslant k$. O algoritmo 3, página 34, com uma instância do problema responde *sim* caso $\mathrm{mincut}(\mathsf{G}) \leqslant k$, senão responde *não* com probabilidade de erro $< 1/2$. Assim, duas repetições independentes desse algoritmo estabelece que $\mathsf{L} \in \mathsf{RP}$.

Da definição de RP acima deduzimos que

$$\mathsf{RP} \subset \mathsf{BPP}.$$

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

$$\mathsf{RP} \subset \mathsf{NP}.$$

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

1. se $w \in \mathsf{L}$, então $\mathbb{P}_{\mathsf{B} \in_R \{0,1\}^{p(|w|)}}[\mathsf{M}(w, \mathsf{B}) = 0] = 0$, e
2. se $w \notin \mathsf{L}$, então $\mathbb{P}_{\mathsf{B} \in_R \{0,1\}^{p(|w|)}}[\mathsf{M}(w, \mathsf{B}) = 1] \leqslant 1/3$.

Equivalentemente, coRP é a classe das linguagens L tais que $\overline{\mathsf{L}} \in \mathsf{RP}$. Nos exemplos 156 e 157 uma resposta 0 está sempre certa enquanto que uma resposta 1 pode estar errada, logo as linguagens definidas nesses exemplos são linguagens em coRP.

Um fato interessante ocorre quando consideramos a possibilidade de uma linguagem $\mathsf{L} \in \mathsf{RP} \cap \mathsf{coRP}$, pois ela pode se beneficiar dos resultados exatos do algoritmo M_{RP} que prova que $\mathsf{L} \in \mathsf{RP}$ e do algoritmo $\mathsf{M}_{\mathsf{coRP}}$ que prova que $\mathsf{L} \in \mathsf{coRP}$.

algoritmos probabilísticos — Computação probabilística

Essa linguagem admite 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 14: RP ∩ coRP

Se M_{RP} responder 1 então por definição garantimos que $w \in \mathsf{L}$. Analogamente, se $\mathsf{M}_{\mathsf{coRP}}$ responder 0 então por definição garantimos que $w \notin \mathsf{L}$. 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** $\mathsf{T}(n)$ se a variável aleatória t_w, que denota o tempo de execução do algoritmo com entrada w, tem valor esperado $\mathbb{E}\, t_w \leqslant \mathsf{T}(|w|)$ 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-3-SAT, algoritmo 9 na página 61, tem tempo esperado polinomial.

No algoritmo 14 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 $\mathsf{M}_{\mathsf{RP}}(w)$ e de $\mathsf{M}_{\mathsf{coRP}}(w)$, 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 $\mathbb{P}[\,\mathsf{M}(w,\mathsf{B}) = \mathsf{L}(w)\,] = 1$, 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, $\mathsf{M}(w) \in \{0,1,?\}$ em que 0 significa $x \notin \mathsf{L}$ e 1 significa $x \in \mathsf{L}$, como é usual, e ? significa "não sei" e $\mathbb{P}[\,\mathsf{M}(w) = ?\,] \leqslant 1/2$. Intuitivamente, entendemos essa definição como que em tempo polinomial ou o algoritmo decide ou interrompe a execução.

O algoritmo 14 acima mostra que

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

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

Lema 161 ZPP = RP ∩ coRP.

Demonstração. De (4.2), só precisamos provar que ZPP ⊂ RP ∩ 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 $w \in \{0,1\}^*$

```
1 simula M com entrada w até no máximo 3p(|w|) passos;
2 se M(w) = 1 então responda 1;
3 senão responda 0.
```

Algoritmo 15: 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, equação (3.6) na página 98,

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

algoritmos probabilísticos- Computacao probabilistica

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 $\mathsf{ZPP} \subset \mathsf{coRP}$; definimos um algoritmo N' que com entrada w

```
1 simula M com entrada w até no máximo 3p(|w|) passos;
2 se M(w) = 0 então responda 0;
3 senão responda 1.
```

Algoritmo 16: $\mathsf{N}'(w)$

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, $\mathsf{ZPP} \subset \mathsf{RP} \cap \mathsf{coRP}$. □

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 com o seguinte problema ainda sem solução

Problema 6. Alguma(s) das inclusões $\mathsf{P} \subset \mathsf{ZPP} \subset \mathsf{RP} \subset \mathsf{BPP}$ são próprias?

4.2.2 $\mathsf{BPP} \subset \mathsf{P/poly}$

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 3, página 155).

Teorema 162 (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 158 acima.

Fixado n, podemos afirmar que existe uma sequência binária b que é um certificado de pertinência em L para toda entrada w de tamanho n, pois para $\mathsf{B} \in_R \{0,1\}^{p(n)}$ a probabilidade de existir $w \in \{0,1\}^n$ para o qual $\mathsf{M}(w, \mathsf{B}) \neq \mathsf{L}(w)$ é

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

portanto, a probabilidade de $\mathsf{M}(w, \mathsf{B}) = \mathsf{L}(w)$ para todo w é positiva, ou seja, existe 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 153 (página 122), 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 $\mathsf{L} \in \mathsf{P/poly}$. □

algoritmos probabilísticos
Computação probabilística

4.2.3 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. Aqui, 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

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

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 $w \in \{0,1\}^*$

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

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 $w \in \{0,1\}^*$

$$w \in \mathsf{L} \Leftrightarrow \exists u \in \{0,1\}^{p(|w|)}\ \forall v \in \{0,1\}^{q(|w|)},\ \mathsf{M}(w,u,v) = 1. \tag{4.3}$$

Exemplo 163 (*conjunto independente máximo*). Em um grafo G, um conjunto $\mathsf{U} \subset \mathsf{V}(\mathsf{G})$ é dito *independente* (ou *estável*) se $|e \cap \mathsf{U}| \leqslant 1$ para toda aresta $e \in \mathsf{E}(\mathsf{G})$, isto é, U não contêm os dois vértices de qualquer aresta do grafo. Um conjunto independente é *máximo* se tem cardinalidade

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

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 se G 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?). Então, decidir $\alpha(\mathsf{G}) = k$ 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

$$\textsc{max-ind} := \{\langle \mathsf{G}, k\rangle : \mathsf{G} \text{ é um grafo com } \alpha(\mathsf{G}) = k\} \tag{4.4}$$

está em Σ_2. De fato, escrevendo na forma (4.3), um par $\langle \mathsf{G}, k\rangle$ 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 (4.4) 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

algoritmos probabilísticos — Computacao probabilistica

binárias $a, u \in \{0,1\}^m$. Também, usaremos o exercício 5, página 44, 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 164 (Sipser–Gács–Lautemann, 1983) $\mathsf{BPP} \subset \Sigma_2$.

A ideia da prova é que se $w \in L$ e $L \in \mathsf{BPP}$ então há um algoritmo M para o qual quase toda sequência de um conjunto $\{0,1\}^m$ é um certificado disso de modo que, para todo $r \in \{0,1\}^m$, 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, $M(w, r \oplus u_i) = 1$. Por outro lado, se $w \notin L$, 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 $M(w, r \oplus u_i) = 0$. Essa afirmação é capturada na sentença (4.5) abaixo que caracteriza L como uma linguagem de Σ_2

$$w \in L \Leftrightarrow \exists u_1, u_2, \ldots, u_m \in \{0,1\}^m \forall r \in \{0,1\}^m, \bigvee_{i=1}^{m} M(w, r \oplus u_i) = 1 \tag{4.5}$$

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

Demonstração do teorema 164. Sejam $L \in \mathsf{BPP}$ 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 159, página 126. Vamos provar que vale a equação (4.5). Suponhamos que $w \in L$. Então

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

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

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

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

portanto, existe $r \in \{0,1\}^m$ tal que $M(w, r \oplus u_1) = M(w, r \oplus u_2) = \cdots = M(w, r \oplus u_m) = 0$, o que prova a equação (4.5). □

A definição de BPP é simétrica, isto é, BPP = coBPP de modo que do teorema acima temos $\mathsf{BPP} \subset \mathrm{co}\Sigma_2$. Definimos $\Pi_2 := \mathrm{co}\Sigma_2$ e, portanto, o teorema 164 garante que

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

Uma visão geral de PH: a **hierarquia polinomial** é definida pela sequências de classes de complexidade: $\Sigma_0 = \Pi_0 = P$ 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 $w \in \Sigma^*$

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

  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;

algoritmos probabilísticos Computação probabilística

- 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 $\Pi_i \cup \Sigma_i \subset \Sigma_{i+1} \cap \Pi_{i+1}$, em particular $\mathsf{NP} \cup \mathsf{coNP} \subset \Sigma_2 \cap \Pi_2$. Notemos que P, NP e BPP são todos subconjuntos de Σ_2.

Uma generalização do problema 1 ($\mathsf{P} \neq \mathsf{NP}$?) é a conjetura $\Sigma_i \neq \Sigma_{i+1}$? E uma generalização do problema 2 ($\mathsf{NP} \neq \mathsf{coNP}$?) 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 7. $\Sigma_i \subsetneq \Sigma_{i+1}$?

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 em 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 $\mathsf{P} \subset \mathsf{P/poly}$, portanto, se $\mathsf{P} = \mathsf{NP}$ então $\mathsf{NP} \subset \mathsf{P/poly}$ (o que não é sabido, problema 3, página 124). Uma questão interessante é saber se $\mathsf{NP} \subset \mathsf{P/poly}$ poderia ocorrer somente se $\mathsf{P} = \mathsf{NP}$. Um resultado nessa direção, num nível acima da hierarquia, é descrito a seguir, mostrando que $\mathsf{NP} \subset \mathsf{P/poly}$ pode ocorrer somente se PH colapsa para o segundo nível.

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

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

Corolário *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 interativos de prova

Num 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

algoritmos probabilísticos — Computação probabilística

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: podemos definir a classe P 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 165. 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 um 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

algoritmos probabilísticos – Computação probabilística

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.

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 et al. (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 and 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[4] polinomial.

Convencionamos que um protocolo interativo é escrito como

[4] Um máquina de Turing tem complexidade de espaço $\mathsf{S}(n)$ se $\mathsf{S}(n)$ 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.

algoritmos probabilísticos - Computação probabilística

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 166 (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.

Problema 8. NONISO ∈ NP?

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 isomórfica 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. No protocolo abaixo essa estratégia é executada duas vezes para diminuir a probabilidade de erro

Entrada: os grafos G_0 e G_1.

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

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
$d \leftarrow_R \{0, 1\}$;
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
$e \leftarrow_R \{0, 1\}$;

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.

algoritmos probabilísticos- Computacao probabilistica

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 $\subset$ IP é própria.

Problema 9. 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 10. coNP $\subset$ IP(2)?

É sabido (Boppana et al., 1987) que se o problema 10 for respondido com sim, coNP $\subset$ 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

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

Exemplo 167 (*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: $r \leftarrow_R \{1,\ldots,p-1\}$;
$b_1, b_2 \leftarrow_R \{0,1\}$;
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,

algoritmos probabilísticos – Computação probabilística

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 mensagem, 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 maios 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 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 fixo. Ainda, Goldwasser and Sipser (1986) mostram a segunda inclusão de

$$\mathtt{AM}(p(n)) \subset \mathtt{IP}(p(n)) \subset \mathtt{AM}(p(n)+2)$$

logo, reunindo várias informações que temos até aqui, podemos escrever

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

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 $\mathsf{V} = \{1, 2, \ldots, n\}$. Sejam G_0 e G_1 dois grafos. A quantidade de grafos isomorfos à G_i é

$$\frac{n!}{|\mathrm{Aut}(\mathsf{G}_i)|} \tag{4.6}$$

em que $\mathrm{Aut}(\mathsf{G}_i)$ é o conjunto de todas as permutações $\pi\colon \mathsf{V} \to \mathsf{V}$ 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 (4.6) é inteiro, pelo Teorema de Lagrange). Por exemplo, no caso do grafo

algoritmos probabilísticos Computação probabilística

$G := (\{1,2,3\},\{12\})$ 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 (4.6).

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

$$S = \{(H,\pi)\colon H \equiv G_0 \text{ ou } H \equiv G_1 \text{ e } \pi \in \mathrm{Aut}(H)\}. \tag{4.7}$$

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 usando o tamanho de S

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

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

a afirmação (4.9) segue imediatamente de (4.6) 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, 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 na seção 3.0.3. Lembremos do exemplo 131, página 103, que a família $\mathcal{H}$ de funções $h_{(a,b)}\colon \{0,1\}^m \to \{0,1\}^m$ dadas por

$$h_{(a,b)}(x) = a \cdot x + b,$$

com as operações + e · relativas ao corpo finito $\mathbb{F}_{2^m}$, é 2-universal e se restringirmos a imagem de cada ponto do domínio aos k primeiros bits temos $h_{(a,b)}\colon \{0,1\}^m \to \{0,1\}^k$ 2-universal. O sorteio de uma função é feita sorteando-se um par $(a,b) \in \{0,1\}^m \times \{0,1\}^m$.

Façamos $m := 2^{\binom{n}{2}}$ e $k := \lceil \log(2n!) \rceil$. A ideia para resolver o nosso problema é que o mapeamento por uma função 2-universal h de $\{0,1\}^m$ para um conjunto pequeno $\{0,1\}^k$ faz com que $h(S)$ seja ou "grande" ou "pequeno" o suficiente para que consigamos fazer a distinção de maneira eficiente. Tomemos

$$K := 2n!, \quad k = \lceil \log(K) \rceil \quad \text{e} \quad \rho := \frac{|S|}{2^k}$$

logo para $H \in_R \mathcal{H}$ o evento $[y \in H(S)]$, dado $y \in \{0,1\}^k$, ocorre com probabilidade

$$\mathop{\mathbb{P}}_{H\in_R\mathcal{H}}[y \in H(S)] = \mathop{\mathbb{P}}_{H\in_R\mathcal{H}}\left[\bigcup_{x\in S}[H(x)=y]\right] \leqslant \sum_{x\in S}\mathop{\mathbb{P}}_{H\in_R\mathcal{H}}[y=H(x)] \leqslant \sum_{x\in S}\frac{1}{2^k} = \rho$$

por um lado, e por outro lado, usando o exercício 15, página 46,

$$\begin{aligned}\mathop{\mathbb{P}}_{H\in_R\mathcal{H}}\left[\bigcup_{x\in S}[H(x)=y]\right] &\geqslant \sum_{x\in S}\mathop{\mathbb{P}}_{H\in_R\mathcal{H}}[H(x)=y] - \frac{1}{2}\sum_{\substack{x,z\in S\\ x\neq z}}\mathop{\mathbb{P}}_{H\in_R\mathcal{H}}([H(x)=y]\cap[H(z)=y]) \\ &= \sum_{x\in S}\frac{1}{2^k} - \frac{1}{2}\sum_{\substack{x,z\in S\\ x\neq z}}\frac{1}{2^{2k}} \geqslant \frac{|S|}{2^k} - \frac{1}{2}\frac{|S|^2-|S|}{2^{2k}} \geqslant \frac{|S|}{2^k}\left(1-\frac{|S|}{2^{k+1}}\right) \geqslant \frac{|S|}{2^k}\left(1-\frac{2^{k-1}}{2^{k+1}}\right) = \frac{3}{4}\rho.\end{aligned}$$

algoritmos probabilísticos — Computação probabilística

Em resumo

$$\frac{3}{4}\rho \leqslant \mathop{\mathbb{P}}_{H\in_R\mathcal{H}}[y \in H(S)] \leqslant \rho. \tag{4.10}$$

Se $|S| = K$ então

$$\rho = \frac{|S|}{2^k} = \frac{K}{2^k} = \frac{K}{2^{\lceil \log(K)\rceil}} > \frac{1}{2}$$

portanto pelo lado esquerdo de (4.10) temos que

$$\mathop{\mathbb{P}}_{H\in_R\mathcal{H}}[y \in H(S)] > \frac{3}{8}.$$

Se $|S| \leqslant K/2$ então pelo lado direito de (4.10) temos que

$$\mathop{\mathbb{P}}_{H\in_R\mathcal{H}}[y \in H(S)] \leqslant \rho = \frac{|S|}{2^k} \leqslant \frac{K}{2^{k+1}} = \frac{K}{2^{\lceil \log(K)\rceil+1}} \leqslant \frac{1}{2}.$$

Vejamos, agora, um protocolo de troca de mensagens com bits aleatórios públicos para testar o tamanho aproximado de um conjunto. Tal protocolo, devido a Goldwasser e Sipser, é para o problema geral de determinar se um conjunto S tem cardinalidade pelo menos K ou no máximo $K/2$. A única restrição em S é que a pertinência tem que ser determinada de forma eficiente. O objetivo do Provador é convencer o Verificador de que $|S| \geqslant K$ e caso $|S| \leqslant K/2$ 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 e vice-versa, isto é, se o Verificador aceita então $x \in S$ e $h(x) = y$.

Entrada: $S \subset \{0,1\}^m$ e um natural K.

V: $h \leftarrow_R \mathcal{H}$;
$y \leftarrow_R \{0,1\}^{\lceil \log(K)\rceil}$;

V→P: h, y;

P: determina $x \in \{0,1\}^m$ 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.

Façamos

$$p := \frac{K}{2^k}$$

e recordemos que $\rho = |S|/2^k$ e $(3/4)\rho \leqslant \mathbb{P}[(V,P)(S,k) = 1] \leqslant \rho$. A pertinência em S para NONISO, definido na equação (4.7), pode ser conferida pelo Verificador com o certificado passado pelo Provador.

O sistema de provas para NONISO repete o protocolo acima por t vezes, para t polinomial no tamanho da entrada. Se $|S| = K/2$, o que representa $G_0 \equiv G_1$, então o probabilidade de uma resposta errada é $\mathbb{P}[(V,P)(S,k) = 1] \leqslant \rho \leqslant p/2$. Se $|S| = K$, o que representa $G_0 \not\equiv G_1$, então o probabilidade de uma resposta certa é $\mathbb{P}[(V,P)(S,k) = 1] \geqslant (3/4)\rho = (3/4)p$.

Seja $X := \sum_{i=1}^{t} \mathbb{1}_{[(V,P)(S,k)=1]}$ a quantidade de respostas 1 em t rodadas. Se $|S| = K/2$, então $\mathbb{E}X \leqslant pt/2$ e se $|S| = K$, então $\mathbb{E}X \geqslant 3pt/4$. Consideremos que após as rodadas o Verificador responde 1 caso a quantidade X de respostas 1 nas t execuções satisfaça

$$X \geqslant \frac{5}{8}pt.$$

Se $G_0 \not\equiv G_1$,

$$\mathbb{P}\left[X < \frac{5}{8}pt\right] \leqslant \mathbb{P}\left[X < \left(1-\frac{1}{6}\right)\frac{3}{4}pt\right] \leqslant \mathbb{P}\left[X < \left(1-\frac{1}{6}\right)\mathbb{E}X\right] \leqslant \exp\left(-\frac{(1/6)^2(3/4)pt}{3}\right)$$

algoritmos probabilisticos - Computacao probabilistica

pela desigualdade de Chernoff, equação (3.20) na página 106, que é menor que 1/4 se $t = 200/p > 200$. Portanto, em t rodadas o sistema de prova *aceita* com probabilidade pelo menos 3/4. Logo a probabilidade de erro é no máximo 1/4 quando a entrada pertence a linguagem.

Se $G_0 \equiv G_1$, então o número esperado de execuções do protocolo acima que terminam em *aceita* é no máximo $pt/2$ e

$$\mathbb{P}\left[X \geqslant \frac{5}{8}pt\right] \leqslant \mathbb{P}\left[X \geqslant \left(1+\frac{1}{4}\right)\frac{pt}{2}\right] \leqslant \mathbb{P}\left[X \geqslant \left(1+\frac{1}{4}\right)\mathbb{E}X\right] \leqslant \exp\left(-\frac{(1/4)^2(1/2)pt}{3}\right)$$

usando a desigualdade de Chernoff, equação (3.20) na página 106, que é menor que 1/4 se $t = 140/p > 140$. Portanto, em t rodadas o sistema de prova *rejeita* com probabilidade pelo menos 3/4, ou seja, erra com probabilidade menor que 1/4.

Resumindo, a execução de 201 rodadas do protocolo garante que se $G_0 \not\equiv G_1$ então o sistema aceita com probabilidade pelo menos 3/4 e se $G_0 \equiv G_1$ então o sistema aceita com probabilidade menor que 1/4. Ainda, as 201 rodadas não estragam o fato de ser um protocolo com 2 rodadas. No primeiro passo o Verificador pode sortear 201 funções e 201 pontos, enviá-los ao Provador que repete sua ação 201 vezes e envia cada resultado numa única mensagem ao Verificador que por sua vez, faz as 201 verificações e emite sua resposta. Essa estratégia foi usada no protocolo IP para NONISO, que executa uma mesma ação em dobro para diminuir a probabilidade de responder errado.

4.4 Criptografia

A Criptografia moderna talvez seja a disciplina que mais se beneficia das técnicas probabilísticas e dos algoritmos aleatorizados e mesmo que venhamos a saber que aleatoriedade não acrescenta poder de computação aos algoritmos a noção moderna de segurança criptográfica assegura um papel importante para a aleatoriedade no projeto de sistemas de criptografia seguros.

Vimos na seção 1.1.4 que um sistema de criptografia $(\mathcal{P},\mathcal{C},\mathcal{K},\mathcal{E},\mathcal{D})$ tem sigilo perfeito se as variáveis aleatórias $E_K(P)$ e $E_K(P')$ com $K \in_R \mathcal{K}$ são identicamente distribuídas para quaisquer textos P e P'. Nos requisitos atuais de segurança, dito *segurança semântica*, as distribuições devem ser *indistinguíveis para algoritmos eficientes*, ou seja, a abordagem moderna para a construção de sistemas de criptografia é baseada na *efetividade* computacional de se extrair informação, ao invés de *possibilidade* de se extrair informação. Nesse sentido, um algoritmo determinístico de criptografia não é seguro (Goldwasser and Micali, 1984). Sistemas probabilísticos foram introduzidos por Goldwasser and Micali (1984), que mostraram que, assumindo a hipótese de existir problema computacionalmente difícil, adversário polinomialmente limitado não obtém informação sobre o texto a partir do texto cifrado para qualquer que seja a distribuição sobre o espaço de textos.

A hipótese da existência de problemas computacionalmente difíceis, em $\mathsf{NP} \not\subset \mathsf{BPP}$ por exemplo, é um princípio importante em Criptografia. Uma linguagem em $\mathsf{NP} \not\subset \mathsf{BPP}$ significa que temos um problema para o qual não é conhecido um algoritmo eficiente mas, conhecido um fato adicional (certificado), o problema tem um Verificador eficiente. A dificuldade de se resolver computacionalmente tal problema é usado como princípio para garantir a segurança de protocolos criptográficos (no sentido de que quebrar a codificação é computacionalmente equivalente a resolver o problema) e a decodificação é viável somente de posse de um segredo (o certificado). Notemos que $\mathsf{NP} \not\subset \mathsf{BPP}$, implica em $\mathsf{P} \neq \mathsf{NP}$, entretanto esse último resultado de pior caso ($\mathsf{P} \neq \mathsf{NP}$) não é suficiente para os propósitos da Criptografia pois não garante que o problema computacional no qual se baseia um sistema criptográfico seja difícil na maioria das instâncias, mais ainda, o pior caso pode ocorrer em instâncias raras do problema e na maioria da outras instâncias o problema pode ser fácil.

Uma primitiva da Criptografia moderna para implementar o princípio descrito acima é chamada de função unidirecional. Uma função unidirecional pode ser computada por algoritmo eficiente porém quase sempre não pode ser invertida por algoritmo eficiente, isto é, qualquer algoritmo probabilístico de tempo polinomial que dado $f(x)$ busca por y tal que $f(y) = f(x)$ terá sucesso apenas com probabilidade insignificante. Especialmente útil é a classe dessas funções que podem

algoritmos probabilísticos - Computação probabilística

ser invertidas facilmente quando temos posse de algum segredo, essas funções são conhecidas por *alçapão* (em inglês *trapdoor function*).

Outra primitiva da Criptografia moderna é o gerador pseudoaleatório. Se o uso de aleatoriedade no projeto de sistemas criptográficos é indispensável, a questão importante que surge é "quanto aleatório" devem ser esses bits. Um gerador pseudoaleatório é um algoritmo que a partir de uma sequência binária genuinamente aleatória produz outra sequência binária, geralmente maior, que tenta imitar uma sequência aleatória. O modo tradicional de apresentar alguma garantia da qualidade de um gerador pseudoaleatório é em função desse ser aceito por testes estatísticos *ad-hoc* (e.g., Knuth, 1981). O uso de geradores pseudoaleatórios em criptografia levou os pesquisadores a procurarem uma definição robusta para geradores, que garantam que a utilização deles mantenha a segurança dos sistemas de criptografia contra quaisquer *testes computacionalmente viáveis*.

4.4.1 Função unidirecional

Formalmente, uma **função unidirecional** é uma função $f\colon \{0,1\}^* \to \{0,1\}^*$ tal que $|f(x)|$ e $|x|$ são polinomialmente relacionados[5] e

- existe um algoritmo de tempo polinomial que computa $f(x)$ para todo x;
- para todo algoritmo probabilístico de tempo polinomial A, para todo inteiro positivo d e todo n suficientemente grande

$$\mathbb{P}\Big[\mathsf{A}(f(X)) \in f^{-1}(f(X))\Big] < \frac{1}{n^d}$$

em que a probabilidade é sobre a escolha aleatória $X \in_R \{0,1\}^n$ e sobre os bits aleatórios usados por A.

Não sabemos se tais funções existem.

Problema 11. Funções unidirecionais existem?

Uma resposta afirmativa para o problema tem consequências profundas como (*i*) $\mathsf{NP} \not\subset \mathsf{BPP}$ e, consequentemente, $\mathsf{NP} \not\subset \mathsf{P/poly}$ e $\mathsf{P} \neq \mathsf{NP}$, (*ii*) a existência de geradores pseudoaleatórios, (*iii*) a existência de provas com conhecimento zero para todo problema em NP, (*iv*) a existência de sistemas criptográficos de chave privada e de assinatura digital seguros, (*v*) a existência de esquemas de empenho de bits, e (*vi*) a não existência de provas naturais para $\mathsf{P} \neq \mathsf{NP}$.

Vejamos alguns candidatos a função unidirecional. A função

$$\textsc{mult}(x,y) := \begin{cases} x \cdot y & \text{se } x,y \neq 1 \\ 1 & \text{caso contrário,} \end{cases}$$

podemos computar de modo eficiente e nem sempre é difícil de inverter, embora possa ser em alguns casos. Notemos que $(1, xy) \notin f^{-1}(f(x,y))$. Se definirmos um algoritmo A que computa

$$\mathsf{A}(z) = \begin{cases} (2, z/2) & \text{se } z \text{ é par} \\ (0,0) & \text{caso contrário} \end{cases}$$

então a probabilidade de sucesso é pelo menos $3/4$ se $z = \langle x, y\rangle$ com $x, y \in_R \{0,1\}^n$. Agora, se definimos $\textsc{multp}_n(x,y)$ como o produto de dois primos de tamanho $n/2$ temos uma família unidirecional sob a *hipótese de fatoração* ser um problema difícil.

[5] Isso é uma exigência técnica para garantir que inversão não seja inviável por razões triviais, como é o caso de $|x|$ ser exponencialmente maior que $|f(x)|$.

algoritmos probabilísticos– Computacao probabilistica

Observação 168 (*hipótese de fatoração*). A hipótese que assumimos é que para qualquer algoritmo probabilístico de tempo polinomial A vale que se p e q são dois primos aleatórios de comprimento n e $N = pq$ então

$$\mathbb{P}[\,A(N) \in \{p, q\}\,] < \frac{1}{n^d}$$

para toda constante $d > 0$, onde a probabilidade é sobre p, q e as escolhas aleatórias do algoritmo A. É possível mostrar que o problema de fatorar o produto de dois primos aleatórios é equivalente a computar *todos* os fatores primos de inteiros aleatórios

> *Problema* 12 (Fatoração). Dado $n \in \mathbb{N}$, existe algoritmo de tempo polinomial em $\log n$ que determina todos os divisores primos de n?

e equivalente a determinar uma representação do produto de dois inteiros aleatórios usando dois números de comprimento no máximo $n/2$. Com isso, podemos definir $\textsc{mult}(x)$ como a multiplicação dos números representados pela metade mais significativa dos bits de x pela metade menos significativa dos bits de x, como possível candidata a função unidirecional.

O problema de extrair raízes também é difícil. É possível calcular $f(x, k, N) = x^k \bmod N$ de modo eficiente (veja o exercício 7, página 156) mas sob certas hipóteses nos parâmetros k e N é difícil inverter, por exemplo, são candidatas

$$\text{Rabin}_N(x) := x^2 \bmod N, \qquad \text{para todo } x \in \{1, 2, \ldots, N\}$$

que sabemos que é difícil de inverter se e somente se fatoração (problema 12, página 142) é um problema difícil. A função Rabin definida acima pode ser generalizada por

$$\text{RSA}_{N,e}(x) := x^e \bmod N, \qquad \text{para todo } x \in \{1, 2, \ldots, N\}$$

e ambas são funções unidirecionais *alçapão* pois podem ser invertidas de modo eficiente se é conhecida a fatoração de N.

Exemplo 169 (*logaritmo discreto*). Em geral, o grupo multiplicativo $(\mathbb{Z}_n^*, \cdot_n)$ dos resíduos invertíveis com respeito a multiplicação módulo n não é cíclico, isto é, não existe $\alpha \in \mathbb{Z}_n^*$ tal que para todo $\beta \in \mathbb{Z}_n^*$, $\beta = \alpha^k$ para algum $k \in \mathbb{N}$. Nos casos em que $\mathbb{Z}_n^*$ é cíclico[6] temos[7] um isomorfismo de grupos $\iota\colon (\mathbb{Z}_n^*, \cdot_n) \to (\mathbb{Z}_{\varphi(n)}, +_n)$, conhecido como **logaritmo discreto**, definindo $\iota(\alpha^k) = k$ para qualquer gerador α de $\mathbb{Z}_n^*$. Por exemplo, a função $\iota\colon \mathbb{Z}_{11}^* \to \mathbb{Z}_{10}$ dada na tabela 4.1 (usando só os representantes das classes dos resíduos) satisfaz $\iota(a \cdot_n b) = \iota(a) +_n \iota(b)$.

A função $f(p, \alpha, x) = (p, \alpha, \alpha^x \bmod p)$, com p primo, α um gerador módulo p e $1 \leqslant x \leqslant p - 1$ pode ser calculada eficientemente mas não se sabe inverter eficientemente, não é conhecido algoritmo eficiente que dados p, α e $\alpha^x \bmod p$, determina x.

> *Problema* 13 (*logaritmo discreto*). Existe um algoritmo que com entrada p primo, α um gerador módulo p e $1 \leqslant b \leqslant p-1$, compute em tempo polinomial em $\log p$ o menor inteiro positivo $0 \leqslant x \leqslant p - 2$ tal que $b = \alpha^x \bmod p$?

Por exemplo, a classe do 3 é um gerador do $\mathbb{Z}_{1999}{}^*$. Sabemos computar rapidamente $3^{789} \mod 1999$ porém não sabemos resolver eficientemente a equação $3^x \equiv 1452 \pmod{1999}$.

Portanto, essa função é candidata a função unidirecional. ◊

O papel chave das funções unidirecionais é dado no seguinte resultado cuja demonstração pode ser encontrada em Arora and Barak (2009).

Teorema *Se existe função unidirecional então para todo $d > 0$ existe um sistema criptográfico semanticamente seguro que usa chaves de tamanho n para codificar mensagens de tamanho n^d.* □

[6] $(\mathbb{Z}_n^*, \cdot_n)$ é cíclico se e somente se $n = 1, 2, 4, p^t$, ou $2p^t$, onde $p > 2$ é primo e $t \geqslant 0$ inteiro (Ireland and Rosen, 1990, capítulo 4).

[7] $\varphi(n) := \big|\{a \in \mathbb{Z}\colon 0 \leqslant a < n \text{ e } \text{mdc}(a, n) = 1\}\big| = n \cdot \prod_{i=1}^{k}\big(1 - \frac{1}{p_i}\big)$, onde $n = p_1^{a_1} \cdots p_k^{a_k}$, é a função de Euler.

algoritmos probabilísticos Computação probabilística

$\mathbb{Z}_{11}^*$	$\mathbb{Z}_{10}$
$1 (= 2^0)$	0
$2 (= 2^1)$	1
$3 (= 2^8)$	8
$4 (= 2^2)$	2
$5 (= 2^4)$	4
$6 (= 2^9)$	9
$7 (= 2^7)$	7
$8 (= 2^3)$	3
$9 (= 2^6)$	6
$10 (= 2^5)$	5

Tabela 4.1: função logaritmo discreto.

4.4.2 Criptografia de chave pública

Os sistemas criptográficos se encaixam em duas grandes classes, os *simétricos*, como o *one-time pad* e os *assimétricos*. Criptossistemas simétricos têm as chaves de codificação e decodificação trivialmente relacionadas (podem ser iguais) e supõem-se secretas. Criptossistemas assimétricos têm as chaves de codificação e decodificação diferentes, a chave de codificação e pode ser feita pública mas a chave $d = d(e)$ de decodificação é secreta *e não pode ser computada eficientemente* a partir das informações públicas.

Nessa seção veremos dois sistemas assimétricos conhecidos. A ideia aqui é mostrar como aleatorização faz parte desses sistemas, não abordaremos aspectos de eficiência de modo aprofundado e tampouco abordaremos os aspectos de segurança desses sistemas. Uma boa referência para esses temas e aspectos práticos é Menezes et al. (1997).

Troca de chaves: o protocolo Diffie–Hellman–Merkle (Diffie and Hellman, 1976) permite que duas partes entrem em acordo sobre uma chave secreta usando um canal de comunicação não seguro. Esse protocolo, publicado em 1976, foi um trabalho pioneiro que lançou as bases para a criptografia de chave pública. Suponha que sejam de conhecimento público um primo p e um gerador γ de $\mathbb{Z}_p$, o grupo multiplicativo dos resíduos módulo n. Alice e Bob estabelecem um *chave* (secreta) comum da seguinte forma

- Alice sorteia uniformemente $a \in \{2, 3, \ldots, p-2\}$, computa $A = \gamma^a \bmod p$ e manda-o para Bob;
- Bob, por sua vez, sorteia uniformemente $b \in \{2, 3, \ldots, p-2\}$, computa $B = \gamma^b \bmod p$ e manda-o para Alice;
- Alice computa B^a e Bob computa A^b e a chave *secreta* comum entre eles é

$$k = A^b \bmod p = B^a \bmod p = \gamma^{ab} \bmod p.$$

Todas as computações podem ser feitas com algoritmos de tempo polinomial em $\log p$. Se um espião, conhecendo γ e p, intercepta A e B então o problema é determinar a tal que $\gamma^a \bmod p = A$, determinar b tal que $\gamma^b \bmod p = B$, e com isso feito basta computar $k = \gamma^{ab} \bmod p$. A parte difícil aqui, que garante a segurança do protocolo, é que determinar a e b é conhecido como o problema de computar o *logaritmo discreto*, ou seja, como vimos acima o problema é computar a inversa de uma função unidirecional.

Um problema nesse protocolo é que um observador malicioso pode assumir o papel de Alice para Bob e o de Bob para Alice e assim conhecer a chave para decodificar alguma mensagem. Isso pode ser evitado usando um protocolo de *assinatura digital* (descrito abaixo).

algoritmos probabilísticos – Computação probabilística

ELGAMAL: é um sistema assimétrico proposto pelo egípcio Taher Elgamal (ElGamal, 1985). O protocolo usado para criar as chaves púbica e privada é o seguinte:

- sorteie um primo grande p e determine um gerador α de $\mathbb{Z}_p^*$;
- sorteie uniformemente $d \in \{2, 3, \ldots, p-2\}$ e compute $A = \alpha^d \bmod p$;
- a *chave pública*, a ser divulgada, é a tripla (p, α, A) e a *chave privada* correspondente, que deve ser mantida em segredo, é d, ou seja, o logaritmo discreto de A com base α.

Para simplificar, vamos assumir que o universo dos textos é o $\mathbb{Z}_p$ e o espaço das cifras (textos codificados) é $\mathbb{Z}_p \times \mathbb{Z}_p$. A codificação (ou cifragem) é feita da seguinte maneira

- dado o texto P e chave pública (p, α, A);
- sorteie uniformemente $b \in \{1, \ldots, p-2\}$;
- compute $E_{(p,\alpha,A)}(b, P) := \left(\alpha^b \bmod p, A^b \cdot P \bmod p\right)$.

Esse é um protocolo probabilístico por causa do parâmetro b na função de codificação, o que acredita-se dificulta muito a criptoanálise. A decodificação que inverte essa codificação é feita do seguinte modo

- dado o texto cifrado (B, C) e considerando a chave privada d;
- compute $D_d(B, C) = B^{p-1-d} \cdot C \bmod p$.

Dessa forma recupera-se o texto original

$$\begin{aligned} D_d\left(E_{(p,\alpha,A)}(b, P)\right) &= D_d\left(\alpha^b \bmod p, A^b \cdot P \bmod p\right) \\ &= (\alpha^b \bmod p)^{p-1-d} \cdot (\alpha^d \bmod p)^b \cdot P \bmod p \\ &= \alpha^{(p-1)b} \cdot P \mod p = P. \end{aligned}$$

Essas funções são computadas eficientemente.

Exemplo 170. Seja $(23, 7, 4)$ uma chave pública, e 6 a chave privada. Para codificar o texto 7, sorteamos $b = 3$ e enviamos $(21, 11)$. O receptor computa $21^{23-1-6} \cdot 11 \mod 23 = 7$. ◊

A segurança do Elgamal é baseada na dificuldade computacional do *Problema de Diffie–Hellman*:

> *Problema* 14 (*Diffie–Hellman*). Existe um algoritmo eficiente que, dados p um primo, α um gerador módulo p e os elementos $\alpha^a \bmod p$ e $\alpha^b \bmod p$, compute $\alpha^{ab} \bmod p$?

Conhecidos um texto cifrado $(\alpha^b \bmod p, (\alpha^d)^b \cdot P \bmod p)$ e a chave pública (p, α, α^d) um algoritmo eficiente que resolve o problema acima pode ser usado para calcular $\alpha^{-db} \bmod p$ e resolver $(\alpha^d)^b \cdot P \bmod p$ para P.

> *Problema* 15. Um algoritmo eficiente para o Problema de Diffie–Hellman implica num algoritmo eficiente para o problema do logaritmo discreto?

ASSINATURA DIGITAL ELGAMAL: Alice quer enviar um documento P com uma assinatura $\text{ass}(P) = (r, s)$ para Bob. Sejam (p, α, A) e d as chaves pública e privada, respectivamente, geradas pelo protocolo do sistema Elgamal de Alice.

A Alice

- sorteia uniformemente $k \in \{2, 3, \ldots, p-2\}$,
- computa $r = \alpha^k \bmod p$ e $s = (P - dr)(k^{-1} \bmod p-1) \bmod p-1$ usando sua chave privada,

e envia $(P, (r, s))$ para Bob. Para a verificação da assinatura o Bob, de posse da chave pública de Alice, verifica se $\alpha^P = A^r r^s \bmod p$.

algoritmos probabilísticos — Computacao probabilistica

RSA: é um criptossistema assimétrico, provavelmente o mais famoso deles, que deve o seu nome aos três professores do MIT responsáveis pela sua criação: Ron Rivest, Adi Shamir e Len Adleman (Rivest et al., 1978), os quais receberam o prestigiado prêmio Alan Turing da ACM em 2002 por "sua engenhosa contribuição que fez a criptografia de chaves públicas úteis na prática".

Para simplificar, vamos assumir que o universo dos textos é o $\mathbb{Z}_n$, onde n é um parâmetro do protocolo. A determinação das chaves é da seguinte maneira

- sorteie dois primo grandes, p e q, com aproximadamente o mesmo tamanho;
- compute $n = pq$ e a função de Euler $\varphi(n) = (p-1)(q-1)$;
- sorteie uniformemente $e \in \{2, 3, \ldots, \varphi(n)-1\}$ com $\text{mdc}(e, \varphi(n)) = 1$;
- compute o único $d \in \{2, 3, \ldots, \varphi(n)-1\}$ tal que $ed \equiv 1 \pmod{\varphi(n)}$;
- a *chave pública*, a ser divulgada, é (e, n) e a *chave privada* correspondente, que deve ser mantida em segredo, é d, o inverso multiplicativo de e módulo $\varphi(n)$.

Dado um texto P, a codificação com a chave (e, n) é dada por $\mathsf{E}_e(\mathsf{P}) := \mathsf{P}^e \bmod n$. A chave de decodificação correspondente é d, o inverso multiplicativo de e módulo $\varphi(n)$. Dado um texto cifrado C, a decodificação é realizada por $\mathsf{D}_d(\mathsf{C}) := \mathsf{C}^d \bmod n$. Essas funções podem ser computadas eficientemente.

Exemplo 171. Tome os primos $p = 61$ e $q = 53$, de modo que $n = pq = 3233$. A função de Euler é $\varphi(n) = (p-1)(q-1) = 3120$. Escolhemos $e = 17$ (chave pública) e obtemos $d = 2753$ (chave privada). Se o texto é $m = 123$ então o texto cifrado é $c = m^{17} \mod 3233 = 855$, que é decifrado por $c^{2753} \mod 3233 = 855^{2753} \mod 3233 = 123$. ◊

Agora, vamos verificar que se $\mathsf{P} \in \mathbb{Z}_n$ então $\mathsf{D}_d(\mathsf{E}_e(\mathsf{P})) = \mathsf{P}$. Temos que $\mathsf{D}_d(\mathsf{E}_e(\mathsf{P}))$ é $\mathsf{P}^{ed} \bmod n$ com $ed \equiv 1 \pmod{\varphi(n)}$, ou seja, $ed = 1 + r\varphi(n)$ para algum $r \in \mathbb{Z}$. Assim,

$$\mathsf{P}^{ed} = \mathsf{P}^{1+r\varphi(n)} = \mathsf{P}(\mathsf{P}^{p-1})^{(q-1)r}$$

e pelo Pequeno Teorema de Fermat $\mathsf{P}^{p-1} \equiv 1 \pmod{p}$, se p não divide P, logo

$$\mathsf{P}\left(\mathsf{P}^{p-1}\right)^{(q-1)r} \equiv \mathsf{P} \pmod{p}. \tag{4.11}$$

A equação (4.11) também vale quando p divide P. Portanto, $\mathsf{P}^{ed} \equiv \mathsf{P} \pmod{p}$. Analogamente, vale que $\mathsf{P}^{ed} \equiv \mathsf{P} \pmod{q}$ e dessas duas congruências temos que $pq | \mathsf{P}^{ed} - \mathsf{P}$, ou seja, $\mathsf{P}^{ed} \equiv \mathsf{P} \pmod{n}$, assim

$$\mathsf{D}_d\left(\mathsf{E}_e(\mathsf{P})\right) \equiv \mathsf{P}^{ed} \equiv \mathsf{P} \pmod{n}.$$

Um algoritmo eficiente para o ao problema da fatoração, problema 12 na página 142, implica que a partir de (e, n) é possível computar $\varphi(n)$, pelo algoritmo de Euclides estendido, e depois computar a chave privada d, quebrando a codificação. Então fatoração eficiente é suficiente para a quebra do RSA, mas não se sabe se é necessária. Pode haver, por exemplo, um algoritmo para computar $\varphi(n)$ sem que se recorra à fatoração de n e que seja eficiente. Porém, a existência desse algoritmo implicaria num algoritmo eficiente que determina p e q.

De fato, a segurança do RSA é baseada na dificuldade computacional do *Problema RSA*, ou seja, inverter uma função candidata a unidirecional.

> *Problema* 16 (*RSA*). Existe um algoritmo eficiente que, dada uma chave pública (e, n) e um inteiro C, determine um inteiro P tal que $\mathsf{P}^e = \mathsf{C} \bmod n$?

Acredita-se que esse problema seja equivalente ao problema da fatoração.

algoritmos probabilísticos – Computação probabilística

Conjectura 172 (Rivest, Shamir e Adleman) *Um algoritmo eficiente para o problema RSA implica num algoritmo eficiente para o* problema da Fatoração.

Sabemos que a partir de (e,n) e d é possível determinar p e q em tempo polinomial (Coron and May, 2007).

Assinatura digital RSA: Alice quer enviar um documento P com uma assinatura ass(P) para Bob. Sejam (e,n) e $d = d(e)$ um par de chaves RSA, pública e privada respectivamente, de Alice. Alice usa sua chave privada e computa $s = \mathsf{P}^d \bmod n$ e envia (P, s) para Bob. Para a verificação da assinatura Bob obtém a chave pública de Alice e verifica se $\mathsf{P} = s^e \bmod n$.

4.4.3 Geradores pseudoaleatórios seguros

O estudo formal de geradores pseudoaleatórios nasceu da necessidade de assegurar que o uso de bits pseudoaleatórios, ao invés de aleatórios, não comprometa a segurança.

Seja p uma função polinomial. As distribuições de $\mathsf{X}_m \in_{\mathsf{D}} \{0,1\}^{p(m)}$ e $\mathsf{Y}_m \in_{\mathsf{E}} \{0,1\}^{p(m)}$ são ditas **computacionalmente indistinguíveis** se para toda constante $d > 0$ e todo algoritmo probabilístico A de tempo polinomial em m

$$\left| \mathop{\mathbb{P}}_{\mathsf{X}_m \in_{\mathsf{D}} \{0,1\}^{p(m)}} [\mathsf{A}(\mathsf{X}_m) = 1] - \mathop{\mathbb{P}}_{\mathsf{Y}_m \in_{\mathsf{E}} \{0,1\}^{p(m)}} [\mathsf{A}(\mathsf{Y}_m) = 1] \right| \leqslant \frac{1}{m^d}$$

para todo m suficientemente grande, em que a medida de probabilidade é sobre as distribuições das sequências e as escolhas internas do algoritmo probabilístico A. Dizemos que $\mathsf{X}_m \in_{\mathsf{D}} \{0,1\}^{p(m)}$ é **pseudoaleatório** se for computacionalmente indistinguível da distribuição uniforme $\mathsf{Y}_m \in_{\mathsf{R}} \{0,1\}^{p(m)}$.

Um **gerador pseudoaleatório seguro** é um algoritmo (determinístico) que para todo inteiro positivo m computa

$$\mathsf{G}_m \colon \{0,1\}^{k(m)} \to \{0,1\}^m$$

em tempo $\mathsf{O}(m^b)$, para alguma constante positiva b, de modo que para todo $m \in \mathbb{N}$ suficientemente grande

1. $m > k(m)$
2. $\mathsf{G}_m(\mathsf{X})$ é pseudoaleatório para $\mathsf{X} \in_{\mathsf{R}} \{0,1\}^{k(m)}$.

Notemos que na definição exigimos que os geradores pseudoaleatórios sejam seguros mesmo contra adversários com poder computacional ligeiramente maior quando comparado aos geradores.

Entendemos que A na definição acima é um algoritmo que decide a aleatoriedade de uma sequência segundo algum critério. Por exemplo, $\mathsf{A}(x) = 1$ se e só se o número de ocorrências do padrão 11 é no máximo 6 desvios padrão da média ou, $\mathsf{A}(x) = 1$ se e só se a maior subsequência de 1 consecutivos é no máximo $6\log_2(n)$, ou ambos os testes. Suponhamos que G_m é um gerador tal que para 3/4 das entradas o primeiro bit da saída é 1. Então, definimos o teste A que com entrada x responde 1 se, e só se, o primeiro bit de x é 1 e

$$\left| \mathop{\mathbb{P}}_{\mathsf{X} \in_{\mathsf{R}} \{0,1\}^k} [\mathsf{A}(\mathsf{G}_m(\mathsf{X})) = 1] - \mathop{\mathbb{P}}_{\mathsf{Y}_m \in_{\mathsf{R}} \{0,1\}^m} [\mathsf{A}(\mathsf{Y}_m) = 1] \right| = \left| \frac{3}{4} - \frac{1}{2} \right| = \frac{1}{4}.$$

Geradores pseudoaleatórios seguros não podem existir incondicionalmente e uma condição que garante a existência de geradores é a existência de funções unidirecionais. Não provaremos o seguinte resultado (Håstad et al., 1999).

Teorema 173 *As seguintes afirmações são equivalentes:*

1. *existe função unidirecional;*
2. *existe gerador pseudoaleatório seguro com* $k(m) = m - 1$;
3. *para todo* $\varepsilon > 0$, *existe gerador pseudoaleatório seguro com* $k(m) = m^\varepsilon$. □

algoritmos probabilísticos Computacao probabilistica

Imprevisibilidade: Se x é uma sequência de bits então denotamos por $x\restriction_{1,\ldots,i}$ a restrição de x aos seus primeiros i bits, isto é, se $x = (x_1, x_2, \ldots, x_m)$ então $x\restriction_{1,\ldots,i} = (x_1, x_2, \ldots, x_i)$ e denotamos por $x\restriction_i$ o i-ésimo bit da sequência. Um vetor aleatório $Z = (Z_1, \ldots, Z_n) \in \{0,1\}^n$ é **imprevisível** para o próximo bit se para qualquer $i \in \{1, 2, \ldots, n\}$ e qualquer algoritmo A probabilístico de tempo polinomial

$$\mathbb{P}[A(Z_1, \ldots, Z_{i-1}) = Z_i] \leqslant \frac{1}{2} + \frac{1}{p(n)}$$

para todo polinômio p.

É natural esperarmos de um gerador pseudoaleatório seguro que os bits gerados não possam ser previstos, isto é, não devem existir um $i \in \{1, 2, \ldots, n\}$ e um algoritmo eficiente A tal que

$$A(G_m(X)\restriction_{1,\ldots,i-1}) = G_m(X)\restriction_i$$

com probabilidade positiva e não desprezível.

Yao (1982) mostrou que um algoritmo que gera uma sequência de bits é um gerador pseudoaleatório se e somente se a sequência gerada é imprevisível para o próximo bit. De fato, seja $G_m \colon \{0,1\}^{k(m)} \to \{0,1\}^m$ um gerador pseudoaleatório e suponhamos existir um algoritmo probabilístico de tempo polinomial A tal que

$$\mathbb{P}[A(G_m(X)\restriction_{1,\ldots,i-1}) = G_m(X)\restriction_i] > \frac{1}{2} + \frac{1}{p(m)}$$

para algum polinômio p, algum i e todo m suficientemente grande. Agora, consideremos um algoritmo B que com entrada x decide se A prevê o i-ésimo bit, isto é, computa

$$B(x) = \begin{cases} 1, & \text{se } A(x\restriction_{1,\ldots,i-1}) = x\restriction_i \\ 0, & \text{caso contrário} \end{cases}$$

e temos que se $Y \in_R \{0,1\}^m$ então $B(Y) = 1$ com probabilidade $1/2$ e para $X \in_R \{0,1\}^{k(m)}$ a probabilidade de $B(G_m(X)) = 1$ é maior que $1/2 + 1/p(m)$, logo

$$\left| \mathop{\mathbb{P}}_{Y \in_R \{0,1\}^m} [B(Y) = 1] - \mathop{\mathbb{P}}_{X \in_R \{0,1\}^{k(m)}} [B(G_m(X)) = 1] \right| > \frac{1}{p(m)}$$

contrariando o fato de $G_m(X)$ ser pseudoaleatório.

Para a recíproca, suponhamos, agora, que A seja um algoritmo probabilístico de tempo polinomial que prova que a sequência $G_m(X)$ não é pseudoaleatória e provemos que ela é previsível. Fixamos m suficientemente grande, $\varepsilon > 0$ pequeno e para $X \in_R \{0,1\}^{k(m)}$ e $Y \in_R \{0,1\}^m$ assumimos que

$$\mathbb{P}[A(G_m(X)) = 1] - \mathbb{P}[A(Y) = 1] > \varepsilon. \tag{4.12}$$

Vamos usar A para prever bits de sequências geradas por G_m da seguinte maneira. Consideremos o algoritmo C que recebe os bits $y_1, \ldots, y_{i-1}$, sorteia os bits $Z_i, \ldots, Z_m \in_R \{0,1\}$, computa

$$a := A(y_1, \ldots, y_{i-1}, Z_i, \ldots, Z_m)$$

e responde

$$C(y_1, \ldots, y_{i-1}) = \begin{cases} Z_i, & \text{se } a = 1 \\ 1 - Z_i, & \text{se } a = 0. \end{cases}$$

Vamos provar que segue de (4.12) que

$$\mathbb{P}[C(G_m(X)\restriction_{1,\ldots,i-1}) = G_m(X)\restriction_i] > \frac{1}{2} + \frac{\varepsilon}{m} \tag{4.13}$$

algoritmos probabilísticos– Computacao probabilistica

em que a probabilidade é sobre escolhas de i, X e C e cuja consequência é o seguinte: se G_m não é gerador pseudoaleatório então a equação (4.12) vale para $\varepsilon = m^{-d}$ para infinitamente muitos valores de m e para cada m temos um algoritmo previsor como descrito acima, ou seja,

se $G_m(X)$ é imprevisível então também é um pseudoaleatório.

Agora, analisaremos o comportamento do algoritmo C descrito acima para provar (4.13). Definimos as seguintes variáveis aleatórias sobre $\{0,1\}^m$, para cada $j \in \{1,\ldots,m-1\}$, $Y_j \in_R \{0,1\}$, $X \in_R \{0,1\}^k$ e

$$\begin{aligned} D_0 &:= (Y_1, Y_2, \ldots, Y_m) \\ D_j &:= (G_m(X)\restriction_{1,\ldots,j}, Y_{j+1}, \ldots, Y_m) \\ D_m &:= G_m(X). \end{aligned}$$

Se $p_i := \mathbb{P}[\,A(D_i) = 1\,]$, para $0 \leqslant i \leqslant m$, então por (4.12) temos

$$\sum_{j=0}^{m-1} (p_{j+1} - p_j) = p_m - p_0 > \varepsilon.$$

Portanto, existe um índice $i \in \{1,\ldots,m\}$ tal que

$$p_i - p_{i-1} > \frac{\varepsilon}{m}.$$

Analisemos $C(G_m(X)\restriction_{1,\ldots,i-1})$ para estimar $\mathbb{P}[\,C(G_m(X)\restriction_{1,\ldots,i-1}) = G_m(X)\restriction_i\,]$ que é a probabilidade de C acertar na previsão de $G_m(X)\restriction_i$. Essa probabilidade é, pela construção do algoritmo C,

$$\frac{1}{2}\mathbb{P}[\,a = 1 \mid Z_i = G_m(X)\restriction_i\,] + \frac{1}{2}\mathbb{P}[\,a = 0 \mid Z_i = 1 - G_m(X)\restriction_i\,] = \frac{1}{2}\mathbb{P}[\,a = 1 \mid Z_i = G_m(X)\restriction_i\,] + \frac{1}{2} - \frac{1}{2}\mathbb{P}[\,a = 1 \mid Z_i = 1 - G_m(X)\restriction_i\,]$$

entretanto

$$\frac{1}{2}\mathbb{P}[\,a = 1 \mid Z_i = 1 - G_m(X)\restriction_i\,] + \frac{1}{2}\mathbb{P}[\,a = 1 \mid Z_i = G_m(X)\restriction_i\,] = \mathbb{P}[\,a = 1\,] = \mathbb{P}[\,A(D_{i-1}) = 1\,] = p_{i-1}$$

e

$$\mathbb{P}[\,a = 1 \mid Z_i = G_m(X)\restriction_i\,] = \mathbb{P}[\,A(D_i) = 1\,] = p_i,$$

substituindo essas igualdades na dedução anterior

$$\begin{aligned} \mathbb{P}[\,C(G_m(X)\restriction_{1,\ldots,i-1}) = G_m(X)\restriction_i\,] &= \frac{1}{2}\mathbb{P}[\,a = 1 \mid Z_i = G_m(X)\restriction_i\,] + \frac{1}{2} - \frac{1}{2}\mathbb{P}[\,a = 1 \mid Z_i = 1 - G_m(X)\restriction_i\,] \\ &= \frac{1}{2}p_i + \frac{1}{2} - p_{i-1} + \frac{1}{2}p_i \\ &\geqslant \frac{1}{2} + p_i - p_{i-1} \\ &> \frac{1}{2} + \frac{\varepsilon}{m}. \end{aligned}$$

4.4.4 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 et al., 1985). Por exemplo, no caso das bolas de bilhar, página 133, 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 capacita as partes de uma interação a provar fatos sem revelar segredos.

algoritmos probabilísticos — Computação probabilística

Exemplo 174 (*Caverna do Ali Babá* por Quisquater et al. 1989). Paula quer provar para Vitor que ela sabe as palavras secretas que abre a porta dentro da Caverna de Ali Babá, mas ela não quer revelar o segredo para Vitor. A caverna tem um túnel em forma de anel com uma porta mágica na posição diametralmente oposta à da entrada, figura[8] 4.6.

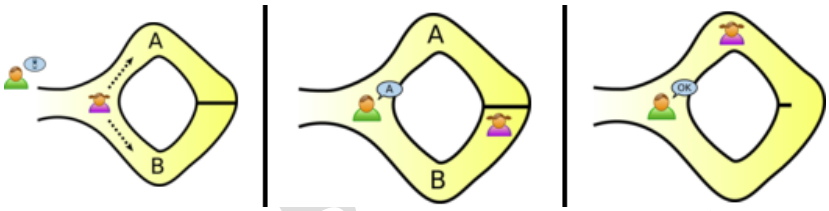


Figura 4.6: caverna do Ali Babá.

Paula usa seus conhecimentos em sistema de provas com conhecimento zero para demonstrar esse fato com o seguinte protocolo:

1. num primeiro momento, Vitor espera fora da caverna enquanto Paula entra e segue pelo lado esquerdo ou pelo lado direito do anel dentro da caverna (caminhos A ou B na esquerda da Figura 4.6);
2. Vitor, que não sabe por qual lado Paula foi, entra na caverna e diz para Paula aparecer por um dos lados, por exemplo A (centro da Figura 4.6), escolhido aleatoriamente;
3. se Paula não conhece as palavras secretas, então existe somente 50% de chance de ela vir pelo caminho certo do túnel. Se Paula conhece as palavras secretas, então ela sempre poderá vir pelo lado que Vitor escolheu.

Se eles repetem o protocolo 20 vezes e Paula não conhece o segredo então a probabilidade com que ela sai pelo caminho correto em todas as vezes é muito pequena, a saber 2^{-20}, de modo se Paula sempre retorna pelo caminho escolhido por Vitor, então esse concluirá que com probabilidade bem alta ela conhece o segredo. Observemos que não importa a quantidade de vezes que se repete o protocolo acima, Vitor nunca irá aprender as palavras secretas. ◊

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 esse mesmo Verificador (ou qualquer outro 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 165 na página 133, 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;

[8] Essa figura tem Creative Commons Atribuição 2.5 e foi legalmente copiada de http://en.wikipedia.org/wiki/Zero-knowledge_proof

algoritmos probabilísticos – Computação probabilística

estatístico – as distribuições são estatisticamente próximas, segundo a distância definida no exercício 13, página 90;

computacional – as distribuições são indistinguíveis por algoritmos eficientes, segundo a definição dada na página 146;

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 et al., 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 175 $\mathsf{BPP} \subsetneqq \mathsf{PZK}$ *e* $\mathsf{SZK} \subsetneqq \mathsf{CZK}$ *e* $\mathsf{CZK} = \mathsf{IP}$.

ISO ∈ PZK: vejamos um sistema de prova interativa com conhecimento zero para a linguagem ISO a qual não sabemos se pertence a BPP.

Entrada: G_0 e G_1 grafos.

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

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, 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. Para tal construiremos um simulador M que responde com a mesma distribuição que (V, P) quando os grafos da entrada são isomorfos. O simulador irá incorporar o papel do V.

algoritmos probabilisticos - Computacao probabilistica

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.

Proposição 176 *Suponha que G_0 e G_1 são grafos isomorfos. Sejam $X \in_R \{0,1\}$ e $Y \in_R \mathbb{S}_n$ variáveis aleatórias independentes. Então, para todo grafo H isomorfo a G_0 e G_1*

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

Disso temos que as variáveis aleatórias X e $Y(G_X)$ são independentes (verifique). 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

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

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.

Demonstração da Proposição 176. Notemos que $\Pi_0 = \{\pi \colon \pi(G_0) = H\}$ e $\Pi_1 = \{\pi \colon \pi(G_1) = H\}$ têm a mesma cardinalidade (justifique). Então $\mathbb{P}[\, Y(G_X) = H \mid X = 1 \,] = \mathbb{P}[\, Y(G_1) = H] = \mathbb{P}[\, Y \in \Pi_1 \,]$. Analogamente, $\mathbb{P}[\, Y(G_X) = H \mid X = 0 \,] = \mathbb{P}[\, Y \in \Pi_0 \,]$, ademais $\mathbb{P}[\, Y \in \Pi_1 \,] = \mathbb{P}[\, Y \in \Pi_0 \,]$. Pelo Teorema de Bayes

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

e $\mathbb{P}[\, X = 0 \mid Y(G_X) = H \,] = 1/2$ pela mesma razão. □

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 se φ lhe é dado como entrada auxiliar. Também, a rigor,

algoritmos probabilísticos — Computação probabilística

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 trabalhosa análise pode ser vista em Goldreich (2001) ou em Goldreich et al. (1991).

4.4.5 NP ⊂ CZK se funções unidirecionais existem

Lembremos que uma linguagem L é NP-completa se está em NP e para todo $I \in \mathsf{NP}$ existe um algoritmo eficiente R tal que $x \in I$ se e só se $R(x) \in L$. Nessa seção mostramos exemplos de sistema de prova com conhecimento zero para as linguagens NP-completas CH e 3-COL; disso temos NP ⊂ CZK. Esse resultado depende da existência de função unidirecional. A discussão nessa seção vai ser informal.

CH ∈ CZK. Se G é um grafo sobre o conjunto de vértices $V := \{1, 2, \ldots, n\}$, então dizemos que G é *hamiltoniano* se existe uma permutação π de V tal que $\{\pi(1), \pi(n)\}$ e $\{\pi(i), \pi(i+1)\}$ são arestas para todo $i \in \{1, 2, \ldots, n-1\}$. Em outras palavras, um grafo é hamiltoniano se seus vértices podem ser ordenados de modo que vértices consecutivos com respeito a tal ordem são adjacentes e também são adjacentes o primeiro e último vértices da sequência. Essa sequência de vértices de G é um *circuito hamiltoniano*. A linguagem CH formada pelos grafos hamiltonianos admite um sistema de provas com conhecimento zero computacional caso exista função unidirecional (Blum, 1986). Como essa linguagem é NP-completa temos NP ⊂ CZK.

Vamos começar com um protocolo: o Provador e o Verificador conhecem um grafo G. O Provador afirma conhecer um um circuito hamiltoniano C. **P** escolhe uma permutação $\sigma \in_R \mathbb{S}_n$, criptografa separadamente cada posição da matriz de adjacências do grafo $\sigma(G)$ e envia a matriz para o Verificador; **V** escolhe $b \in_R \{0, 1\}$ e envia o bit para o Provador. **P** testa se $b = 0$, nesse caso decodifica toda a matriz e a revela junto com σ; no caso $b = 1$ revela somente as n entradas da matriz que correspondem às arestas de $\sigma(C)$; finalmente, envia ao Verificador o que foi revelado; **V**, caso $b = 0$, verifica se $G \equiv \sigma(G)$, caso contrário verifica se recebeu um circuito, nesses casos aceita, senão rejeita.

O protocolo acima é baseado no fato de que o Provador entrega ao Verificador um grafo antes do Verificador decidir o que quer, caso $b = 0$ o Verificador quer verificar se a matriz criptografada é legitima e no caso $b = 1$ verificar se o circuito hamiltoniano é legítimo. Como a escolha do Verificador é aleatória o Provador convence o Verificador com probabilidade 1/2, no caso em que ele (o Provador) não conhece um circuito hamiltoniano. O fato do Verificador ser limitado não permite-o conhecer nada a respeito do que foi criptografado pelo Provador. O fato de usar criptografia torna o protocolo de conhecimento zero *computacional*, ao invés de perfeito. De fato, nesses casos é usado uma técnica chamada de *esquema de empenho de bits* (Goldreich et al., 1991).

Exercício 177. (Goldreich and Levin, 1989) Um predicado $b\colon \{0,1\}^* \to \{0,1\}$ computável em tempo polinomial é dito **hard-core** da função $f\colon \{0,1\}^* \to \{0,1\}^*$ se para todo algoritmo probabilístico de tempo polinomial A e todo inteiro $d > 0$

$$\mathop{\mathbb{P}}_{X \in_R \{0,1\}^n}(A(f(X)) = b(X)) < \frac{1}{2} + \frac{1}{n^d}$$

para todo n suficientemente grande, ou seja, dado $f(x)$ um algoritmo eficiente qualquer consegue prever $b(x)$ com probabilidade no máximo próxima a de responder jogando uma moeda.

Suponha f unidirecional e defina g por $g(x, r) := (f(x), r)$ com $|x| = |r|$ e tome $b(x, r) := \sum_i x_i r_i \pmod 2$. Prove que g é unidirecional e que b é um predicado *hard-core* de g.

Esquema de empenho de bits: (*bit commitment*) é um sistema interativo composto por dois algoritmos probabilísticos de tempo polinomial e de duas fases entre as duas partes: um *Remetente* que compromete-se com um valor $b \in \{0, 1\}$ perante um *Receptor*. A primeira fase é chamada a *fase de comprometimento* — o Remetente guarda um valor b numa

algoritmos probabilísticos Computação probabilística

caixa inviolável — e a segunda é a *fase de revelação* — o Remetente abre a caixa e revela o valor guardado — de modo que são satisfeitos:

1. *Sigilo*: no final da primeira fase o Receptor não ganhou nenhum conhecimento do valor guardado, mesmo no caso de um Receptor desonesto.

2. *Vínculo*: dada a transcrição da interação na primeira fase, há no máximo um valor que o Receptor possa mais tarde aceitar como um valor legal guardado, mesmo no caso do Remetente ser desonesto.

Se ambas as partes seguem o protocolo, então no final da segunda fase o Receptor obtém o valor empenhado pelo Remetente. Exigimos que a fase de comprometimento não produza conhecimento para o Receptor do valor empenhado, enquanto que a fase de revelação vincula ao Remetente um valor único, no sentido de que na fase de revelação o Receptor só pode aceitar esse valor.

Por exemplo, Seja $f\colon \{0,1\}^* \to \{0,1\}^*$ uma função injetiva unidirecional e $b\colon \{0,1\}^* \to \{0,1\}$ um predicado *hard-core* (definido no exercício 177).

1. *Fase de comprometimento*: para empenhar $v \in \{0,1\}$ o Remetente escolhe $x \in_R \{0,1\}^n$ (n é o parâmetro de segurança) e envia $(f(x), b(x) \oplus v)$.

   Após essa fase, por causa do predicado *hard-core*, temos que a probabilidade de um Receptor qualquer prever o valor empenhado v é no máximo $(1/2) + (1/p(n))$.

2. *Fase da revelação*: o Remetente revela v e x e o Receptor verifica se a mensagem recebida (α, β) corresponde a $\alpha = f(x)$ e $\beta = b(x) \oplus v$.

É possível provar que se f é unidirecional e b um predicado *hard-core* de f então o protocolo acima é um esquema de empenho. O sigilo segue da definição de *hard-core* e o vínculo segue da injetividade de f. Para cada mensagem (α, σ) para o Receptor existe um único s tal que $f(s) = \alpha$ e então um único $v \in \{0,1\}$ tal que $b(s) \oplus v = \sigma$.

3-COL $\in$ CZK. O problema de determinar se há uma 3-coloração própria dos vértices de um grafo também é NP-completo. Suponhamos existir uma função unidirecional ou, mais especificamente, de um esquema de empenho de bits.

Vamos mostrar que uma repetição de $n|E|$ vezes o protocolo abaixo resulta numa prova de conhecimento-zero para 3-COL com probabilidade de erro exponencialmente pequena em n para a consistência.

Entrada: Um grafo $G = (V, E)$;

P: P computa $\psi\colon V \to \{1,2,3\}$, se possível uma 3-coloração própria de G;
$\pi \leftarrow_R \mathbb{S}_3$;
$\phi \leftarrow \pi \circ \psi$;

P→V: $\{c_v\}_{v \in V}$ onde c_v é um empenho do valor $\phi(v)$.

V: $\{u, v\} \leftarrow_R E$;

V→P: $\{u, v\}$;

P→V: c_u e c_v;

V: Se os valores revelados $\phi(u)$ e $\phi(v)$ são distintos, responde 1,
senão responde 0.

Claramente, se o Provador conhece uma 3-coloração do grafo então, com probabilidade 1, ele convence o Verificador desse fato; se o grafo não admite uma 3-coloração então o Verificador rejeita com probabilidade pelo menos $1/|E|$. A

algoritmos probabilísticos — Computação probabilística

probabilidade de não haver erro nas $n|\mathsf{E}|$ rodadas é $(1-|\mathsf{E}|^{-1})^{n|\mathsf{E}|} \approx \exp(-n)$ (veja (s.8)). O Verificador executa em tempo polinomial e também o Provador assumindo que ele receba ψ como entrada auxiliar.

O conhecimento-zero é mais difícil de provar, intuitivamente as únicas informações reveladas para o Verificador numa interação são $\pi(\psi(u))$ e $\pi(\psi(v))$, ou seja, um par de cores escolhidas aleatoriamente em $\{1,2,3\}$, pela escolha aleatória de π; qualquer simulador que devolva um par de cores $(i,j) \in_R \{1,2,3\}^2$, $i \neq j$, é computacionalmente indistinguível da informação equivalente no protocolo. Uma demonstração com todos os detalhes pode ser lida em Goldreich et al. (1991).

Um simulador começa com uma 3-coloração aleatória $\psi \in_R \{1,2,3\}^n$ e simula o verificador com um empenho dessas cores. Notemos que essa entrada para a simulação do Verificador tem distribuição diferente da entrada na execução do protocolo iterativo, entretanto, pela segurança envolvida no esquema de empenho, essas distribuições são indistinguíveis. Nesse momento, se o Verificador pede para examinar se uma aresta sorteada está propriamente colorida o simulador revela as cores empenhadas dos vértices.

1. escolhe aleatoriamente $\{i',j'\} \in \mathsf{E}$ e $c_i, c_j \in \{1,2,3\}$, $c_i \neq c_j$; todos os outros vértices recebem cor arbitrária; empenha essas cores;
2. simula V e se o Verificador pede para ver as cores de $\{i',j'\}$, então revela as cores e responde 1; caso contrária reinicia a simulação. Após $n|\mathsf{E}|$ repetições responde 0.

4.5 Provas naturais

Por serem uma estrutura combinatória simples os circuitos booleanos despertaram bastante interesse de estudo pois esperava-se que pudéssemos determinar cotas inferiores não triviais para circuitos que resolvem problemas difíceis já que esses são abundantes, como vimos no exercício 155. Feito isso, uma possível conclusão é que $\mathsf{P} \neq \mathsf{NP}$. Depois de algum tempo sem um resultado significativo nesse sentido, Razborov and Rudich (1997) explicaram porque uma *prova natural* de cota inferior não deve provar que, por exemplo, existe uma linguagem L em NP mas não em $\mathsf{P/poly}$.

Uma estratégia *natural* para mostrar que SAT não pode ser resolvido por circuitos de tamanho polinomial é encontrar alguma propriedade $\mathcal{P}$ que contenha muitas funções booleanas, inclusive SAT, o que se deve poder verificar de modo eficiente, e mostrar, usando algum tipo de argumento indutivo, que nenhuma função computável por circuitos de tamanho polinomial pode ter propriedade $\mathcal{P}$. Razborov e Rudich mostram que o sucesso dessa técnica implicaria a inexistência de funções unidirecionais contra algoritmos de tempo subexponencial. Portanto, sob hipóteses razoáveis de existência de problemas difíceis, as provas naturais não podem ser usadas para provar limites inferiores contra os circuitos de tamanho polinomial. Um exemplo de prova de cota inferior que contornou a barreira da prova natural foi dada por Williams (2014).

Sejam $\mathcal{B}_n$ o conjunto de todas as 2^{2^n} funções booleanas $g\colon \{0,1\}^n \to \{0,1\}$ e Λ, Γ duas classes de funções booleanas (de fato, pensamos em duas classes de complexidade computacional). Chamamos $f\colon \{0,1\}^* \to \{0,1\}$ de uma *família de funções booleanas*, ou simplesmente, **família booleana** e f_n denota a restrição de f a $\{0,1\}^n$. Dada uma família booleana f, o objetivo é mostrar que $f \notin \Lambda$. Uma **propriedade combinatória** é um subconjunto $\mathcal{C} \subset \bigcup_n \mathcal{B}_n$. Dizemos que a família booleana f **tem a propriedade** $\mathcal{C}$ se $f_n \in \mathcal{C}$ para todo n suficientemente grande.

Uma estratégia para provar que $f \notin \Lambda$, é definir um propriedade $\mathcal{C}$ tal que, para n suficientemente grande, f_n tem $\mathcal{C}$ mas que qualquer $g \in \Lambda$, para n suficientemente grande, g_n não tem $\mathcal{C}$. Nesse caso dizemos que $\mathcal{C}$ é uma propriedade **contra** Λ.

A decisão em $\mathcal{C}$ é computada tomando-se como entrada a tabela-verdade de f_n, portanto, a entrada tem tamanho 2^n e a complexidade da decisão é tomada em função desse tamanho. Assim, por exemplo, se o problema de decisão está em

algoritmos probabilísticos Computação probabilística

P/poly significa que a decisão é feita por circuitos de tamanho $2^{O(n)}$.

Uma propriedade combinatória $\mathcal{C}$ é Γ-**natural contra** Λ se satisfaz as seguintes condições:

Amplidão isto é, a propriedade captura em seus elementos os mais "difíceis", a saber os aleatórios: $|\mathcal{C} \cap \mathcal{B}_n| \geq 2^{-O(n)}$, isto é, $H \in_R \mathcal{B}_n$ tem $\mathcal{C}$ com probabilidade pelo menos $2^{-O(n)}$;

Construtividade isto é, a propriedade pode ser decidida pro circuitos não muito grandes: a propriedade pode ser decida em por uma família de circuitos que computa uma função em Γ.

Utilidade isto é, a propriedade é útil contra Λ que, como vimos, significa que se f tem $\mathcal{C}$ então $f \notin \Lambda$.

Um **gerador de função pseudoaleatória** em Λ e seguro contra Γ usa uma semente s de n bits e computa uma função $f_s : \{0,1\}^n \to \{0,1\}$ em Λ, qualquer que seja $s \in \{0,1\}^n$, tal que para todo circuito C em Γ,

$$\left| \mathbb{P}_{Y \in_R \{0,1\}^n}(C(f_Y) = 1) - \mathbb{P}_{H \in_R \mathcal{B}_n}(C(H) = 1) \right| < 2^{-n^2}.$$

Aqui as entradas para os circuitos booleanos são as tabelas verdade das funções booleanas.

Teorema 178 *Se existe um gerador de função pseudoaleatória em Λ e seguro contra Γ, então não existe uma propriedade Γ-natural contra Λ.*

Demonstração. Consideremos o gerador como enunciado, seja f_y uma função pseudoaleatória em Λ e suponhamos, para efeito de contradição, que $\mathcal{C}$ seja uma propriedade Γ-natural contra Λ.

Como $\mathcal{C}$ é Γ-natural, existe um circuito C de complexidade Γ que decide $\mathcal{C}$ e

$$\mathbb{P}_{H \in_R \mathcal{B}_n}[C(H) = 1] \geq 2^{-O(n)} \tag{4.14}$$

e como computar $f_y(x)$ está em Λ e $\mathcal{C}$ é útil contra Λ

$$\mathbb{P}_{Y \in_R \{0,1\}^n}[C(f_Y) = 1] = 0 \tag{4.15}$$

Das equações (4.14) e (4.15) temos que C distingue f_Y de H, contrariando o fato do gerador de função pseudoaleatoria ser Γ-difícil. □

4.5.1 Uma prova natural

4.6 Exercícios

1. Seguindo o exemplo 150, página 114, prove com detalhes que se existe um algoritmo de tempo polinomial para decidir se uma fórmula booleana é satisfazível então existe um algoritmo de tempo polinomial para descobrir uma valoração.

2. Prove que toda função $f : \{0,1\}^* \to \{0,1\}$ 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 $O(\log(|\Gamma|)t(n))$.

3. Seja $I \subset \{0,1\}^*$ uma linguagem indecidível qualquer (por exemplo, a linguagem definida pelo Problema da Parada, definido na página 118). 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.

algoritmos probabilísticos- Computacao probabilistica

4. (**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.

5. (Adleman, 1978) Defina *circuito booleano aleatorizado* como um circuito booleano que além das n portas da entrada contém portas que recebem um bit aleatório uniformemente; esse circuito computa $f\colon \{0,1\}^n \to \{0,1\}$ se quando $f(x_1, \ldots, x_n) = 0$ o circuito com $(x_1, \ldots, x_n)$ responde 0 e quando $f(x_1, \ldots, x_n) = 1$ o circuito com $(x_1, \ldots, x_n)$ 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).

6. (Arora and Barak, 2009, Lema 9.2) Assumamos P = NP e suponhamos um sistema de codificação tal que $D_k(E_k(p)) = p$ com $|p| < |k|$ ($|\cdot|$ é tamanho), com D_k, E_k computáveis em tempo polinomial. Mostre que existe um algoritmo de tempo polinomial tal que para todo w existem $x_0, x_1 \in \{0,1\}^{|w|}$ com

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

em que $n < |w|$.

7. Prove que o algoritmo 17 abaixo computa $a^b \bmod n$ em tempo $O(\log(b)\log^2(n))$.

Instância : inteiros não-negativos a, b e $n > 1$.
Resposta : $a^b \bmod n$.

```
1  c ← 0
2  d ← 1
3  Seja b_k b_{k-1} ... b_1 b_0 a representação binária de b
4  para i ← k to 0 faça
5  |  c ← 2 · c
6  |  d ← d · d mod n
7  |  se b_i = 1 então
8  |  |  c ← c + 1
9  |  |  d ← d · a mod n
10 responda d.
```

Algoritmo 17: Exponenciação modular

algoritmos probabilísticos– Computação probabilística