4 | Computação probabilística

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

4.1 MODELOS DE COMPUTAÇÃO

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

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

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

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

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

Em um problema de decisão $\mathcal{O} = \{\text{sim}, \text{não}\}$ e, usualmente, especificamos um subconjunto $\mathsf{L} \subseteq \mathcal{I}$ tal que $\mathsf{A}(x) = \text{sim}$ se e somente se $x \in \mathsf{L}$; desse modo o pro-

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

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

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

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

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

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

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

4.1.1 MÁQUINAS DE TURING

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

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

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

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

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

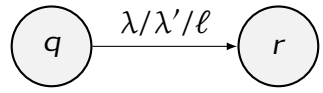

Figura 4.1: legenda para um diagrama de estados: se a máquina está no estado $q \in Q$ e lê $\lambda \in \Gamma$ na fita, então escreve $\lambda' \in \Gamma$ nessa posição e movimenta-se, de acordo com $\ell \in \{\leftarrow, —, \rightarrow\}$ ou uma posição para a esquerda, ou uma para a direita ou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

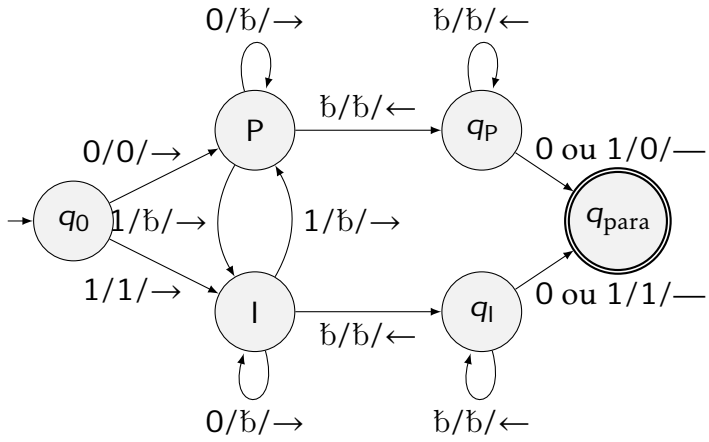

Figura 4.2: diagrama de estados da máquina de Turing para o problema da paridade.

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

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

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

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

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

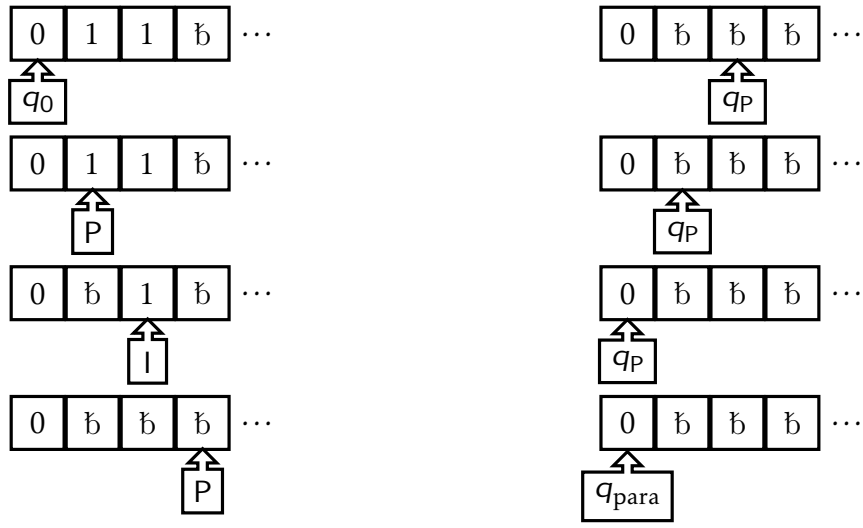

Figura 4.3: a fita na execução da máquina de Turing para a paridade com entrada 011.

todo $w \in \Sigma^*$

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

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

Codificação Vamos assumir que as instâncias e respostas dos problemas computacionais estão codificadas usando símbolos de um alfabeto Σ finito e com pelo menos dois símbolos. De fato, assumimos, sem perda de generalidade (veja o Exercício 4.19, página 238) que

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

existem linguagens que não são decidíveis

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

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

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

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

Algoritmo 38: $P(x)$

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

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

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

4.1.2 MODELO RAM E ALGORITMOS EFICIENTES

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

Um conjunto típico de instruções é dado na Tabela 4.1.

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

Tabela 4.1: Instruções básicas do modelo RAM.

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

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

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

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

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

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

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

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

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

4.1.3 CIRCUITO BOOLEANO

Um **circuito booleano** é representado por um grafo orientado acíclico em que cada vértice está associado a: (i) ou a uma porta lógica dentre *e*, *ou*, *não*; (ii) ou a uma variável x_i para $i \in \{1, 2, \ldots, n\}$ para algum $n \in \mathbb{N}$; (iii) ou uma saída s_i, $i \in \{1, 2, \ldots, m\}$ para algum $m \in \mathbb{N}$. O grau de entrada dos vértices pode assumir um de três valores: 0 (variáveis), 1 (porta *não* e saídas) e 2 (portas *e* e *ou*). Um circuito computa uma

função binária (o caso $m = 1$ é chamado de **função booleana**)

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

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

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

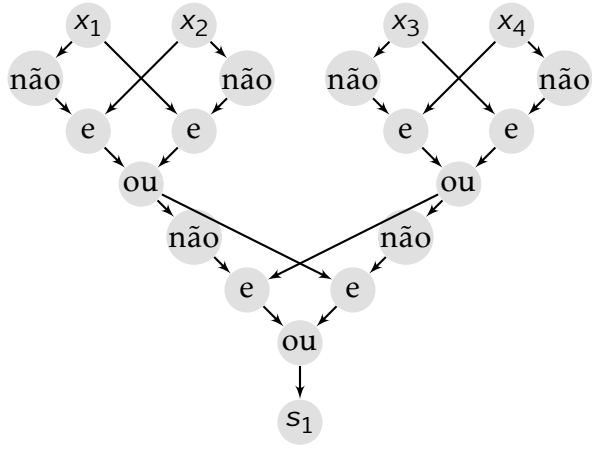

Figura 4.4: circuito booleano que computa $s_1(x_1, x_2, x_3, x_4) = x_1 + x_2 + x_3 + x_4 \bmod 2$.

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

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

$$C_n(x) = L(x)$$

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

O **tamanho do circuito** C, denotado por $|C|$, é o número de vértices do grafo subjacente a definição de circuito e, em geral, estamos interessado no tamanho dos circuitos de uma família $\mathcal{C}$ em função do tamanho da entrada. Outro parâmetro usado para medir a complexidade de um circuito é a sua **profundidade** que é o número de arestas no caminho mais longo através do circuito.

Seja $\mathcal{C} = (C_n)_{n\in\mathbb{N}}$ uma família de circuitos e $s\colon \mathbb{N} \to \mathbb{N}$ uma função. Dizemos que $\mathcal{C}$ é uma família de **tamanho** $s(n)$ se $|C_n| \leqslant s(n)$ para todo n. A **complexidade de**

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

A recíproca dessa proposição não vale pois há linguagens indecidíveis por máquina de Turing que admitem circuito de tamanho polinomial (veja o Exercício 4.20, página 238). Circuito booleano e máquina de Turing não são computacionalmente equivalentes.

Ao contrário da máquina de Turing, com circuito booleano temos um dispositivo de computação para cada tamanho de entrada, nesse caso dizemos que circuito booleano é um **modelo não-uniforme** de computação enquanto que máquina de Turing é um **modelo uniforme** de computação. A não uniformidade faz com que seja possível *desaleatorizar* eficientemente circuitos probabilísticos, ou seja é possível evitar o uso de bits aleatórios sem aumentar substancialmente o tamanho dos circuitos (veja o Exercício 4.22, página 238).

Circuito aritmético Um circuito aritmético é como um circuito booleano no qual as entradas são variáveis ou constantes, as portas *ou* são substituídas pela *soma* e as portas *e* são substituídas pela *multiplicação*. De modo mais geral, em um **circuito aritmético sobre um corpo** $\mathbb{F}$ as constantes e as operações aritméticas são as do corpo. Por exemplo, o circuito na Figura 4.5 abaixo computa $x_1x_2 + 1$. Circuito

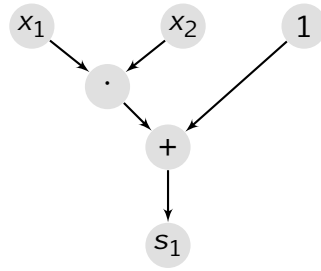

Figura 4.5: Circuito aritmético que computa $s_1(x_1, x_2) = x_1 \cdot x_2 + 1$.

aritmético é o modelo padrão para computação sobre polinômios, a saída de um circuito aritmético é um polinômio (ou um conjunto de polinômios) nas de variáveis entrada. As medidas de complexidade associadas com tais circuitos são *tamanho*, que é a quantidade de vértices no grafo subjacente, e *profundidade*, que é a maior distância entre uma entrada e uma saída.

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

4.2 CLASSES DE COMPLEXIDADE DE TEMPO POLINOMIAL

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

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

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

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

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

2. se $w \notin L$ então $M(w, B) = 0$ qualquer que seja $B \in \{0,1\}^{p(|w|)}$, nesse caso, se $w \notin L$, M responde *não* qualquer que seja a *testemunha*.

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 existe uma testemunha c que convence M de que a fórmula é satisfazível. Ainda, qualquer sequência c não satisfaz uma fórmula $\Phi \notin \textsc{sat}$, logo o algoritmo responde não. Ademais, o tamanho de $\langle c \rangle$ é polinomial no tamanho de $\langle \Phi \rangle$.

Uma sequência c como no exemplo acima também é chamada de **certificado**. Se $w \in \mathsf{L}$ e $\mathsf{L} \in \mathsf{NP}$ então há um certificado curto, de tamanho polinomial em $|w|$, que certifica tal fato, senão, caso $w \notin \mathsf{L}$, não há um certificado curto de que $w \in \mathsf{L}$. Por exemplo, um inteiro $d > 1$ que divide outro inteiro positivo n é um certificado curto de que n é composto. O problema de decidir se um inteiro positivo é composto está em 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 $\mathsf{P} \subseteq \mathsf{NP}$: se $\mathsf{L} \in \mathsf{P}$ e D é um algoritmo polinomial para L então um algoritmo verificador de tempo polinomial M para L simula D e ignora a entrada auxiliar de modo que $\mathsf{M}(w, \mathsf{B}) = \mathsf{D}(w)$ para todo w e todo B.

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

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

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

Exercício 4.5 (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 diferente. Essa máquina não é um modelo realista de computação (veja mais em Sipser, 1996). Prove que a classe NP é a mesma para qualquer definição dentre as descritas acima.

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 5.

O famoso Teorema de Cook–Levin estabelece que SAT é NP-completo. 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 $L \in P$ 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 $\mathcal{P} := (\mathcal{I}, \{\text{sim}, \text{não}\}, \mathcal{R})$ é um problema computacional de decisão, então $\mathcal{R}$ particiona $\mathcal{I}$ de modo natural em $\mathcal{I}_{\text{sim}}$, o conjunto das instâncias sim, e $\mathcal{I}_{\text{não}}$, o conjunto das instâncias não. Como vimos acima, a codificação dos elementos em $\mathcal{I}_{\text{sim}}$ é a linguagem L associada ao problema. O **complemento de um problema de decisão**, denotado $\text{co}\mathcal{P}$, é obtido trocando-se os papéis das intâncias *sim* e *não*. Com respeito à linguagem associada, temos que coL é a linguagem definida pela codificação de $\mathcal{I}_{\text{não}}$. O **complemento de uma linguagem** $L \subseteq \Sigma^*$ é a linguagem $\overline{L} = \Sigma^* \setminus L$.

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

Se C é uma classe de complexidade de problemas de decisão o **complemento da classe de complexidade**, denotada por coC, é a classe dos problemas de decisão formada pelos complementos dos problemas em C.

Pode-se deduzir facilmente da definição que P = coP: se $L \in P$ então existe um algoritmo polinomial M que decide L. O algoritmo $\overline{M}(w) := 1 - M(w)$ de tempo polinomial decide $\overline{L}$, portanto $L \in \text{coP}$. A recíproca desse argumento vale, ou seja, $\text{coP} \subseteq P$, 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 NP = coNP. De fato, atualmente, não é sabido se vale tal igualdade.

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

Por exemplo, a linguagem TAUT formada pelas fórmulas booleanas verdadeiras (tautologias) está em coNP. O complemento da linguagem é dado pelas fórmulas booleanas que não são tautológicas e para uma dada fórmula dessa linguagem um

certificado é uma valoração que faça fórmula falsa, portanto em NP. É fácil verificar que se NP ≠ coNP então P ≠ NP.

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

O principal motivo para a definição da classe P/poly foi a esperança de poder separar P de NP. Do Exercício 4.4 acima nós deduzimos que uma cota inferior para o tamanho dos circuitos que computam um determinado problema implica numa cota inferior para o tempo de um algoritmo que computa o mesmo problema. Esse fato fornece uma estratégia de prova de que P ≠ 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 P ≠ NP.

Ademais, funções booleanas que precisam de circuitos grandes abundam. O número de funções booleanas em n variáveis é 2^{2^n} e a fração delas que são computadas com um circuito com $t = 2^n/n$ portas lógicas tende a 0 quando n cresce. A quantidade de circuitos com t portas lógicas pode ser estimada, de fato super estimada, com a seguinte descrição: cada uma das t portas pode ser de 3 tipos (e, ou, não) e recebe bits de, no máximo, outras duas portas dentre as $n + t - 1$. Se $C(n,t)$ é a quantidade de circuitos com n entradas e t portas, então

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

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

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

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

ou seja, a quantidade de circuitos com $2^n/n$ portas lógicas é assintoticamente muito menor que a quantidade de funções booleanas em n variáveis, assim concluímos

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

que, assintoticamente, quase todas as funções (uma fração $(1 - o(1))$ delas) necessitam de circuitos maiores que $2^n/n$ para serem computadas.

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

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

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

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

4.2.1 CLASSES PROBABILÍSTICAS

O estudo sistemático da computabilidade e complexidade de modelos probabilísticos, bem como das classes de complexidade, começou com Leeuw *et al.*, 1970 e Gill, 1974. Este último prova algumas propriedades básicas de máquinas de Turing probabilísticas, como a enumerabilidade efetiva, a universalidade e outros resultados. Podemos associar a cada modelo de computação um modelo aumentado que permite elementos aleatórios, por exemplo, podemos modificar a classe de programas RAM introduzindo uma instrução especial chamada RANDOM(N) que atribui para a variável N um bit com igual probabilidade e de forma independente de chamadas anteriores. Como já dissemos antes, dentro das classes de tempo polinomial, os resultados relatados neste texto não dependem de máquina.

Uma **máquina de Turing probabilística** M é definida como uma máquina em que cada transição tem dois movimentos legais δ_0 e δ_1, entretanto, a cada transição durante uma computação só um deles é escolhido como o próximo passo; dado um estado q e um símbolo da fita σ a máquina executa a transição $\delta_i(q, \sigma)$ para uma escolha aleatória $i \in \{0, 1\}$. A resposta da máquina para uma entrada w é uma variável aleatória que depende de w; nesse caso, consideramos a distribuição da variável aleatória $\mathsf{M}(w)$ 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}. Para uma tal sequência de comprimento m associamos a proba-