Computabilidade

José de Oliveira Guimarães
josedeoliveiraguimaraes@gmail.com
UFSCar, Sorocaba - SP

18 de novembro de 2015

Sumário

Capítulo 1

Introdução

Lógica é uma sub-área da Filosofia que estuda os sistemas de deduções, como algo pode ser deduzido ou não de um conjunto de premissas. A Lógica Matemática é uma subdivisão da Lógica relacionada à Matemática — como técnicas da lógica se aplicam à Matemática e vice-versa. A Lógica Matemática é dividida em Teoria da Prova, Teoria dos Modelos, Teoria dos Axiomática dos Conjuntos e Computabilidade. Esta última era antigamente chamada de Teoria da Recursão.

Este é um livro sobre Computabilidade, que pode ser definida de várias formas equivalentes entre si. Computabilidade é o estudo das funções de $\mathbb{N}$ em $\mathbb{N}$, dos números reais, dos subconjuntos de $\mathbb{N}$, das funções de $\mathbb{N}$ em $\{0,1\}$ ou de linguagens formais (que serão definidas posteriormente). Usando a notação $F^{\mathbb{N}}_{\mathbb{N}}$ para o conjunto das funções de $\mathbb{N}$ em $\mathbb{N}$, temos que os conjuntos $F^{\mathbb{N}}_{\mathbb{N}}$, $\mathbb{R}$, $\mathcal{P}(\mathbb{N})$ (conjunto das partes de $\mathbb{N}$) e $\mathcal{P}(\Sigma^\star)$ (conjunto das linguagens sobre um alfabeto Σ) são equipotentes entre si. Para a Computabilidade, tanto faz estudar um ou outro conjunto.

Em cursos de Ciência da Computação, vários nomes são utilizados para disciplinas sobre este tópico: Linguagens Formais e Autômatos, Linguagens formais, Teoria da Computação, Computabilidade.

Mostraremos inicialmente algumas definições básicas sobre cadeias e linguagens formais. São utilizadas as definições matemáticas dadas no Apêndice A.

Definição 1.0.1. *Um alfabeto Σ é um conjunto finito não vazio de símbolos.*

Note que um alfabeto Σ não é um conjunto qualquer. Além de ser finito e não vazio, ele necessariamente é composto por símbolos, que são elementos que podem ser impressos no papel ou mostrados em um monitor de computador.

Definição 1.0.2. *Uma cadeia (string) x sobre um conjunto Σ de símbolos é*

(a) a cadeia vazia, indicada por ϵ ou

(b) um elemento do produto cartesiano Σ^n para algum n.

Então $x = (s_1, s_2, \ldots, s_n)$ para algum n. Escrevemos $x = s_1 s_2 \ldots s_n$. O tamanho de x, escrito $|x|$, é n, $|x| = n$. Naturalmente, $|\epsilon| = 0$.

A concatenação de duas cadeias x e y, $x = (s_1, s_2, \ldots, s_n)$ e $y = (r_1, r_2, \ldots r_m)$, denotada por $x \cdot y$, é a cadeia

$$x \cdot y = (s_1, s_2, \ldots s_n, r_1, r_2, \ldots r_m) = s_1 s_2 \ldots s_n r_1 r_2 \ldots r_m$$

Usualmente não utilizamos o operador $\cdot$ de concatenação, usamos simplesmente por xy.

Definição 1.0.3. *Dado um conjunto Σ, $\Sigma^\star$ é o menor conjunto contendo ϵ, os elementos de Σ e fechado sobre a operação $\cdot$ de concatenação.*

$\Sigma^\star$ é o conjunto de todas as cadeias de qualquer tamanho sobre um alfabeto Σ. Este conjunto é chamado de fecho de Kleene.

Exemplo 1.0.1. *Se* $\Sigma = \{0, 1\}$, *então* $\Sigma^\star = \{\epsilon, 0, 1, 10, 01, 11, 00, 100, 111, 101, 011, \ldots\}$

Usaremos Σ^n para o conjunto de todas as cadeias de tamanho n sobre o alfabeto Σ. Então, se Σ for finito (neste texto, sempre será), o número de elementos de Σ^n é também finito. De fato, é igual a $|\Sigma|^n$.

Definição 1.0.4. *Dado um conjunto finito* Σ *de símbolos, uma linguagem* L *é qualquer subconjunto de* $\Sigma^\star$.

Neste texto, todos os conjuntos Σ utilizados para definir linguagens serão finitos.

Definição 1.0.5. *O complemento de uma linguagem* $L \subset \Sigma^\star$ *é a linguagem* $L^c = \Sigma^\star - L$.

Usamos x^k, $x \in \Sigma$ para um elemento

$$\overbrace{xx...x}^{k}$$

de $\Sigma^\star$.

Definição 1.0.6. *Assumindo que os elementos de* Σ *estão ordenados por uma relação* $<$, *existe uma relação induzida* $<$ *entre os elementos de* $\Sigma^\star$ *definida da seguinte forma:* $x < y$ *se*

(a) $|x| < |y|$;

(b) $|x| = |y|$, $x =_{def} x_1x_1 \ldots x_n$, $y =_{def} y_1y_2 \ldots y_n$ *e existe* $j \geqslant 1$ *tal que*

$$x_i = y_i \text{ para todo } i < j \text{ e } x_j < y_j$$

A menos de menção contrária, usaremos esta ordem induzida em $\Sigma^\star$ *pela relação* $<$ *em* Σ. *Esta é a chamada ordem lexicográfica.*

Quando os símbolos do alfabeto Σ forem números ou letras, usaremos a ordem dada pela tabela ASCII. Assim, se $\Sigma = \{0, 1, a, b\}$, temos que a ordem dos elementos em $\Sigma^\star$ é:

$$0, 1, a, b, 00, 01, 0a, 0b, 10, 11, 1a, 1b, ...000, 001, \ldots$$

Proposição 1.0.1. *Assumindo* Σ *finito e não vazio,* $\Sigma^\star \sim \mathbb{N}$.

Demonstração. Sendo Σ finito, sempre é possível construir uma relação de ordem total entre os seus elementos. Esta relação induz uma relação de ordem entre os elementos de $\Sigma^\star$. Esta relação de ordem permite enumerar os elementos de Σ^n. Consequentemente, podemos enumerar $\Sigma^0, \Sigma^1, \Sigma^2, \ldots$. Agrupando todas estas enumerações conseguimos uma enumeração para $\Sigma^\star$.

□

Proposição 1.0.2. *Dado qualquer* Σ *finito,* $\Sigma^\star$ *é equipotente a* $\{0, 1\}^\star$.

Demonstração. $\Sigma^\star$ e $\{0, 1\}^\star$ são ambos infinitos e enumeráveis. Portanto, $\Sigma^\star \sim \{0, 1\}^\star$. □

Uma consequência da Proposição acima é que não precisamos mais do que dois símbolos para definir uma linguagem. Podemos fazer todos os teoremas e funções usando apenas o conjunto $\{0, 1\}$.

Proposição 1.0.3. $\mathcal{P}(\Sigma^\star) \sim \mathcal{P}(\mathbb{N})$.

Demonstração. Temos $\Sigma^\star \sim \mathbb{N}$ pela Proposição 1.0.1. Pela Proposição A.1.2, $2^{\Sigma^\star} \sim 2^{\mathbb{N}}$. □

Proposição 1.0.4. $\mathcal{P}(\mathbb{N}) \sim [0, 1]$

Demonstração. Mostraremos uma função $f : 2^{\mathbb{N}} \longrightarrow [0, 1]$ bijetora. Dado $A \in 2^{\mathbb{N}}$, $A \subset \mathbb{N}$ por definição. Então

$$f(A) = 0.\chi_A(0)\chi_A(1)\chi_A(2)\chi_A(3)\ldots$$

Utilizamos a notação binária para $[0, 1]$.

A função f associa a um conjunto $A = \{2, 5, 7\}$ o número 0.00100101 no qual tem o número 1 na posição n se e somente se $n \in A$.

A função f é injetora, pois se $A, B \in 2^{\mathbb{N}}$, e $A \neq B$ então $\chi_A \neq \chi_B$ e portanto $f(A) \neq f(B)$.[1] Como

[1] Usamos $\chi_A \neq \chi_B$ para "para pelo menos um x temos $\chi_A(x) \neq \chi_B(x)$".

usamos a notação binária para $[0,1]$, f é sobrejetora, pois a cada $s \in [0,1]$ corresponde a um conjunto $A = \{n : s_n = 1\}$ no qual s_n é o n-ésimo dígito de s. Logo f é bijetora e $2^{\mathbb{N}} \sim [0,1]$. □

Definição 1.0.7. *Usaremos F_B^A para o conjunto de todas as funções de A (domínio) em B (contra-domínio). Isto é,*

$$F_B^A = \{f : f \text{ é uma função de } A \text{ em } B\}$$

Em teoria dos conjuntos, usa-se B^A para F_B^A.

Proposição 1.0.5. $F_{\{0,1\}}^{\mathbb{N}} \sim [0,1]$

Demonstração. Considere a função $f : F_{\{0,1\}}^{\mathbb{N}} \longrightarrow [0,1]$ que associa a cada função $g \in F_{\{0,1\}}^{\mathbb{N}}$ o número real em binário

$$0.\overline{g(0)}\ \overline{g(1)}\ \overline{g(2)}\ldots$$

que está no intervalo $[0,1]$. $\overline{g(0)}, \overline{g(1)}, \overline{g(2)}, \ldots$ são os símbolos correspondentes a $g(0), g(1), g(2), \ldots$ expressos em binário. Um símbolo colocado ao lado de outro é concatenado com este. Esta função é claramente bijetora (veja a prova da proposição anterior). Logo $F_{\{0,1\}}^{\mathbb{N}} \sim [0,1]$.

□

Dado o número 0.101, a função associada a ele por f^{-1} é uma função $g : \mathbb{N} \longrightarrow \{0,1\}$ tal que $g(0) = 1$, $g(1) = 0$, $g(2) = 1$ e $g(n) = 0$ para $n \geqslant 3$.

Proposição 1.0.6. $F_{\mathbb{N}}^{\mathbb{N}} \sim [0,1]$

Demonstração. Nesta prova usaremos o teorema A.1.1 (Cantor-Schröder-Bernstein). Mostraremos que $F_{\mathbb{N}}^{\mathbb{N}} \leq [0,1]$ e que $[0,1] \leq F_{\mathbb{N}}^{\mathbb{N}}$. Pelo teorema, $F_{\mathbb{N}}^{\mathbb{N}} \sim [0,1]$.

$F_{\mathbb{N}}^{\mathbb{N}} \leq [0,1]$ pois a seguinte função é injetora: $f(g) : F_{\mathbb{N}}^{\mathbb{N}} \longrightarrow [0,1]$ tal que

$$f(g) = 0.\overline{g(0)}2\overline{g(1)}2\overline{g(2)}2\overline{g(3)}2\ldots$$

$\overline{g(0)}, \overline{g(1)}, \overline{g(2)}, \ldots$ são os símbolos correspondentes a $g(0), g(1), g(2), \ldots$ expressos em binário. Um símbolo colocado ao lado de outro é concatenado com este. Isto é, se $g(n) = 2n$,

$$f(g) = 0.\mathbf{0}2\mathbf{1}0\mathbf{2}1\mathbf{0}0\mathbf{2}1\mathbf{1}0\mathbf{2}\ldots$$

Dadas duas funções g_1 e g_2, $g_1 \neq g_2$, temos $f(g_1) \neq f(g_2)$ e f é injetora.

$[0,1] \leq F_{\mathbb{N}}^{\mathbb{N}}$ pois a seguinte função é injetora: $h(r) : [0,1] \longrightarrow F_{\mathbb{N}}^{\mathbb{N}}$ tal que

$$h(0.r_0r_1r_2r_3\ldots) = \{(0,r_0),(1,r_1),(2,r_2),\ldots\}$$

Isto é, $h(0.r_0r_1r_2r_3\ldots)$ é uma função t de $\mathbb{N}$ em $\mathbb{N}$ ($t \in F_{\mathbb{N}}^{\mathbb{N}}$) tal que $t(i) = r_i$.

h é claramente injetora pois dados $r, s \in [0,1]$, se $r \neq s$ e o i-ésimo dígito de r é diferente do i-ésimo dígito de s, então $h(r)(i) \neq h(s)(i)$ e portanto $h(r) \neq h(s)$. Note que $h(r)$ é uma função para a qual é passada um parâmetro i em $h(r)(i)$.

□

A Figura 1.1 mostra as relações entre os conjuntos estudados pela Computabilidade. As linhas pontilhadas mostram as relações de equipotência entre os conjuntos — há uma linha pontilhada entre A e B se $A \sim B$. Esta relação é de equivalência, sendo então:

(a) reflexiva: $A \sim A$;

(b) transitiva: se $A \sim B$ e $B \sim C$, então $A \sim C$;

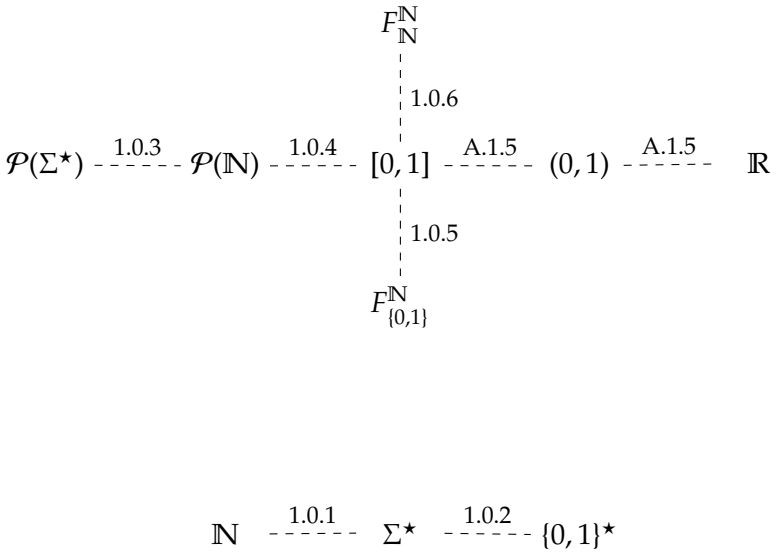


Figura 1.1: Relacionamento entre os conjuntos importantes para a Computabilidade

(c) simétrica: se $A \sim B$, então $B \sim A$.

Então todos os conjuntos mostrados são equipotentes entre si. Para a Computabilidade, podemos escolher com qual conjunto trabalhar. Para alguns teoremas, é mais fácil estudar $\mathcal{P}(\mathbb{N})$. Para outros, é melhor usar $F^{\mathbb{N}}_{\mathbb{N}}$. E assim por diante.

A relação entre os elementos citados neste Capítulo é apresentada na Figura 1.2. Como mostrado pela Figura 1.1, para cada vértice deste pentágono, podemos encontrar, biunivocamente, um outro vértice qualquer. Por exemplo, dado $A \subset \mathbb{N}$, pode-se encontrar, biunivocamente, um $x \in \mathbb{R}$. Isto é, existe uma função bijetora entre o conjunto de todos os subconjuntos de $\mathbb{N}$ e $\mathbb{R}$. Da mesma forma, para cada linguagem $L \subset \Sigma^\star$ podemos encontrar, biunivocamente, uma função $f : \mathbb{N} \longrightarrow \mathbb{N}$.

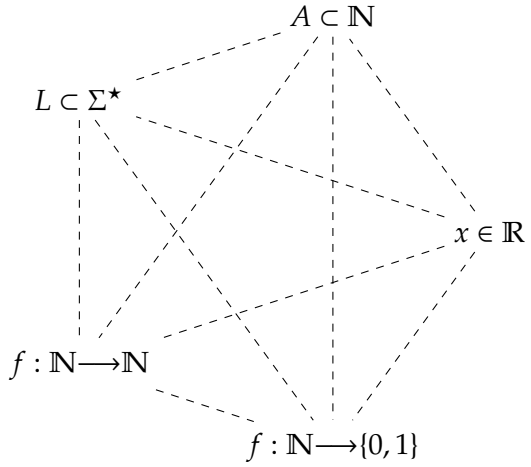


Figura 1.2: Equivalência entre os elementos estudados em Computabilidade

Capítulo 2

Máquinas de Turing

Este Capítulo introduz os principais modelos de computação utilizados nesta pesquisa, as Máquinas de Turing determinísticas e não determinísticas de uma ou mais fitas.

Nas primeiras décadas do século XX, os lógicos procuraram definir o conceito de *computação*. Foram feitas várias tentativas para isto: definição usando a Lógica de Primeira Ordem, funções recursivas, Cálculo Lambda etc. Contudo, nenhum destes sistemas foi amplamente aceito como a definição de *computação*. Havia dúvidas se o conceito de *função mecanicamente computável* era definido corretamente por estas abordagens.

Em 1936, Alan Turing publicou um artigo entitulado "*On Computable Numbers, with an Application to the Entscheidungsproblem*" em que ele descrevia a máquina de Turing (MT). Desde então as máquinas de Turing foram aceitas como uma definição razoável de computação. Verificou-se que os sistemas anteriores eram não só equivalentes entre si como equivalentes às MT´s. Em particular, Church criou um sistema chamado de Cálculo Lambda e pretendia fazê-lo a definição de computação. Então este conceito é definido pela *Tese* abaixo.

> Tese de Church-Turing: toda função mecanicamente computável é computável por uma máquina de Turing.

Como máquinas de Turing e os outros sistemas têm poder equivalente, pode-se substituir "máquinas de Turing" por qualquer um dos sistemas já citados.

Neste mesmo artigo, Turing resolveu o *Problema da Decisão* proposto por David Hilbert em 1900, que é encontrar o algoritmo que verifica se $\Gamma \vDash \phi$ na Lógica de Primeira Ordem. Isto é, se ϕ pode ser logicamente deduzido de um conjunto de fórmulas ϕ. Pelos teoremas da Completude de Gödel e da Correção, isto é o mesmo que $\Gamma \vdash \phi$ (ϕ pode ser deduzido de Γ utilizando os axiomas e as regras de dedução da Lógica de Primeira Ordem). A resolução de Turing foi surpreendente: não existe tal algoritmo. Se existisse, teríamos encontrado um outro algoritmo, que é aquele que verifica se um outro algoritmo pára ou não a sua execução. Mais detalhes sobre este interessantíssimo assunto podem ser encontrados em qualquer livro sobre Computabilidade (Computability, Recursion Theory, ou Recursive Functions).

2.1 Máquinas de Turing Determinísticas

Uma máquina de Turing determinística (MT ou MTD) é uma quadrupla (Q, Σ, I, q) na qual Q e Σ são conjuntos chamados de conjuntos de estados e de símbolos, I é um conjunto de instruções, $I \subset Q \times \Sigma \times Q \times \Sigma \times D$, $D = \{-1, 0, 1\}$ e $q \in Q$ é chamado de estado inicial. Há dois estados especiais: q_s e q_n, ambos elementos de Q e diferentes entre si. Neste texto convenciona-se que o estado inicial será q_0 a menos de menção em contrário. Exige-se que $\{0, 1, \triangleright, \sqcup, \square\} \subset \Sigma$. Uma instrução é da forma (q_i, s_j, q_l, s_k, d)

na qual $s_k \neq \square$ e $q_i \notin \{q_s, q_n\}$. Se $(q, s, q'_0, s'_0, d_0), (q, s, q'_1, s'_1, d_1) \in I$, então $q'_0 = q'_1$, $s'_0 = s'_1$ e $d_0 = d_1$. Q, Σ e I são conjuntos finitos.

O símbolo $\square$ é o branco utilizado para as células ainda não utilizadas durante a execução e $\sqcup$ é utilizado para separar dados de entrada e saída.

A MT definida como mostrado acima corresponde a um programa de computador, software. Para executá-lo, é necessário um hardware que é composto por:

(a) uma fita infinita em ambas as direções composta por células nas quais podem ser escritos e lidos símbolos de Σ;

(b) um mecanismo de executar as instruções I da MT.

A execução da MT é feita em uma fita potencialmente infinita divididas em células. Cada célula funciona como uma posição de memória onde pode ser lidos e escritos, pelas instruções da máquina, os símbolos de Σ (o símbolo $\square$ não pode ser escrito). Há uma cabeça de leitura/gravação que indica a célula corrente sobre a qual são realizadas todas as operações. E há um estado corrente da fita tomado dentre os elementos de Q.

O estado inicial da fita é q_0 mas este estado muda de acordo com a instrução executada (veja adiante). A entrada x da MT é colocada nas primeiras posições da fita tal que a primeira célula depois de x contém um símbolo $\square$. Símbolos $\sqcup$ podem ser utilizados para subdividir x em várias partes diferentes (para passar x e y com entrada, usa-se $x \sqcup y$).

A execução da máquina é feita da seguinte forma: se o estado corrente for q, o símbolo da célula corrente for s e houver uma instrução (q, s, q', s', d), então o estado corrente passará a ser q', será escrito s' na célula corrente e a cabeça de leitura/gravação se moverá de acordo com o valor de d (-1, move para a esquerda, 0 fica na posição atual e 1 move para a direita). Se não houver uma instrução cujos dois primeiros elementos sejam q e s então a máquina pára sem produzir nenhum resultado. Garante-se que não há duas instruções diferentes que comecem com q e s (veja a definição acima da MT). Quando o estado corrente da máquina for q_s ou q_n, a máquina pára. Pela definição de MT, nenhuma instrução (q, s, q', s', d) pode ter $q \in \{q_s, q_n\}$. Note que a execução da máquina é feita por um agente externo e este utiliza um algoritmo para executar a máquina.

Utilizaremos máquinas de Turing para dois propósitos: a) calcular o valor de uma função computável e b) para problemas de decisão. No primeiro caso, o valor da função com entrada x será denotado por $M(x)$ dado que o nome da TM seja M. No segundo caso, exige-se que, quando a máquina pára, o estado corrente seja q_s ou q_n.

Definição 2.1.1. *Em uma computação $M(x)$ (execução da MT M com a entrada x), se o estado final for q_s, consideraremos que M aceita a entrada x e quando o estado final for q_n, que M rejeita x.*

Para tornar os dois tipos de máquinas (retorna o valor e aceita/rejeita) compatíveis, considere que máquinas do tipo b) também colocam um valor 1 ou 0 na fita ao final da computação, conforme o estado final seja q_s ou q_n. Assim pode-se considerar que os dois tipos de máquinas calculam o valor de uma função. E quando uma MT calcula o valor de uma função, o estado final deverá ser q_s.

Em qualquer caso, quando a máquina pára a cabeça de leitura/gravação estará sobre o símbolo mais à esquerda do resultado. O símbolo $\triangleright$ é colocado na primeira célula à direita da saída para indicar o fim desta. Todas as máquinas de Turing utilizadas neste texto terão todas as características dadas no texto acima.

Exemplo 2.1.1. *Seja M uma máquina de Turing que toma dois números unários como entrada e produz como resultado a soma destes números.*

A descrição informal de M é a seguinte:

1. *avance para a direita até encontrar um espaço em branco, $\sqcup$. Este símbolo, por convenção, é utilizado em todas as MT para separar os dados da entrada;*

2. *coloque* 1 *no lugar de* $\sqcup$*;*

3. *mova a cabeça de leitura/gravação para a direita até encontrar um símbolo* $\square$*;*

4. *retroceda duas células para a esquerda;*

5. *escreva* $\triangleright$ *sobre esta célula;*

6. *avance para a esquerda até encontrar um símbolo* $\square$*;*

7. *avance uma célula para a direita e vá para o estado* q_s*. Agora a cabeça de leitura/gravação estará na célula mais à esquerda da resposta, como as nossas convenções exigem.*

As instruções que descrevem os passos acima são:
$(q_0, 1, q_0, 1, 1)$
$(q_0, \sqcup, q_1, 1, 1)$
$(q_1, 1, q_1, 1, 1)$
$(q_1, \square, q_2, \sqcup, -1)$
$(q_2, 1, q_3, \sqcup, -1)$
$(q_3, 1, q_4, \triangleright, -1)$
$(q_4, 1, q_4, 1, -1)$
$(q_4, \square, q_5, \sqcup, 1)$
$(q_5, 1, q_s, 1, 0)$

Esta MT está representada graficamente na Figura 2.1. Usamos $s/s'/D$ *sobre uma transição para representar que, quando o símbolo corrente for* s*, será escrito* s' *e a cabeça de leitura/gravação se moverá na direção dada por* D *(*E *é esquerda,* D*, direita e* N*, neutro).*

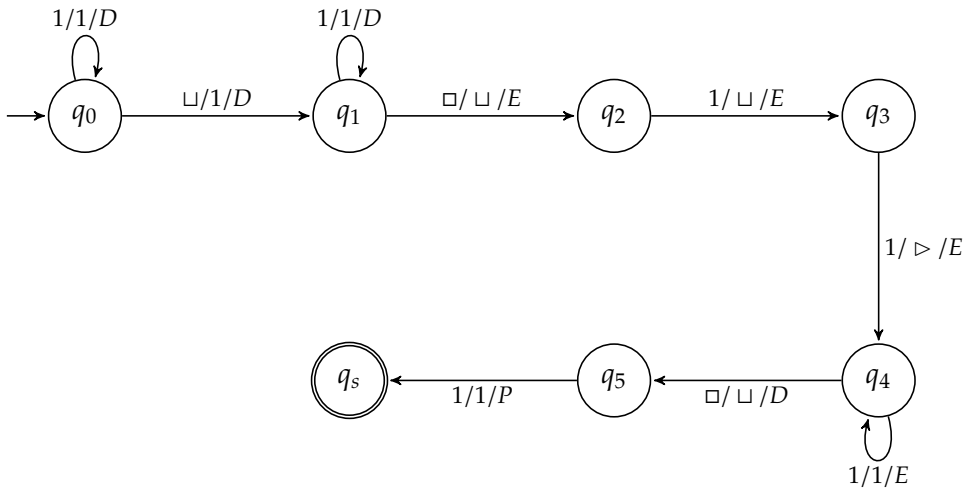


Figura 2.1: Representação gráfica de uma MT que soma dois números unários

Exemplo 2.1.2. *Seja* $M = (Q, \Sigma, I, q_0)$ *uma MT cujas instruções são dadas abaixo. Esta máquina não pára a sua execução quando o primeiro símbolo da entrada for* 0*.*
$(q_0, 0, q_0, 0, 0)$
$(q_0, x, q_s, 1, 0)$ *para todo* $x \in \Sigma, x \neq 0$

Usaremos $M(x) \uparrow$ para "M **nunca termina** a sua execução com a entrada x". E $M(x) \downarrow$ para "M **termina** a sua execução com a entrada x".

Quando duas máquinas de Turing são equivalentes? Isto é relativo. Poderíamos considerar duas máquinas equivalentes se ambas são *isomorfas*. Não definiremos este termo aqui formalmente. Mas considere que duas máquinas são isomorfas se elas são iguais ou apenas dão nomes diferentes para os estados e símbolos. Por exemplo, as duas máquinas abaixo são isomorfas:

$(q_0, 0, q_1, 0, D)$	$(q_a, \#, q_b, \#, D)$
$(q_0, 1, q_s, 1, E)$	(q_a, b, q_s, b, E)
$(q_1, 1, q_s, 0, D)$	$(q_b, b, q_s, \#, D)$

Contudo, a definição de máquinas equivalentes é mais abrangente do que isomorfismo, de certa forma. Confira abaixo.

Definição 2.1.2. *Dizemos que as máquinas de Turing M e M' são equivalentes se, para todo x:*

1. *M e M terminam a execução e* $M(x) = M'(x)$ **ou**;
2. *ambas as máquinas nunca terminam a execução.*

Linguagens de programação, a menos de restrições de memória, devem ter o mesmo poder de calcular funções de $\mathbb{N}$ em $\mathbb{N}$ que máquinas de Turing. Então para provar que dada linguagem é realmente uma linguagem de programação devemos implementar nela um interpretador de máquinas de Turing. Se isto for possível, dizemos que a linguagem é Turing-completa. Para fazer esta prova, pode-se fazer um interpretador de qualquer linguagem que já sabemos que é Turing-completa.

2.2 Armazenando Memória nos Estados

Uma máquina de Turing pode armazenar, nos seus estados, um ou mais números, cada um de tamanho constante. Tudo se passa como se a MT tivesse acesso a um número constante (que pode variar de máquina para máquina) de variáveis x_1, x_2, ..., x_p que poderiam ser utilizados na computação. Esta técnica será utilizada no restante deste texto.

Seja M uma MT que toma uma entrada composta por 0´s e 1´s e implementa o seguinte algoritmo:

1. percorre toda a entrada até o primeiro caráter □ (que sempre está no final da entrada de acordo com a nossa definição de MT). Todos os símbolos 0 são trocados por 1;
2. troca □ por ▷ ao fim da entrada;
3. volta até o início da entrada;
4. pára.

Esta MT é representada graficamente na Figura 2.2.

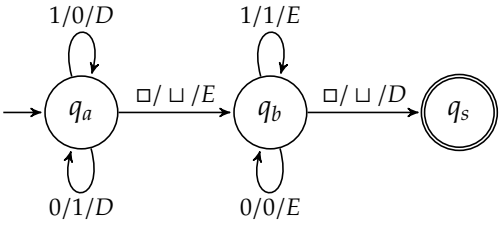


Figura 2.2: Representação de uma MT M

Suponha que seja necessário acrescentar ao final da entrada um símbolo 0 se o primeiro símbolo da entrada for 0 e 1 se for 1. Isto é, a entrada 011 deve ser transformada em 1110 e a entrada 100 em 1111.

Como fazer isto? É necessário guardar o primeiro símbolo enquanto se percorre toda a entrada. Isto pode ser feito utilizando-se estados diferentes para percorrer a entrada. Se o primeiro símbolo for 0, utiliza-se o estado q_0 no lugar de q_a. Se o primeiro símbolo for 1, usa-se q_1. Veja a Figura 2.3.

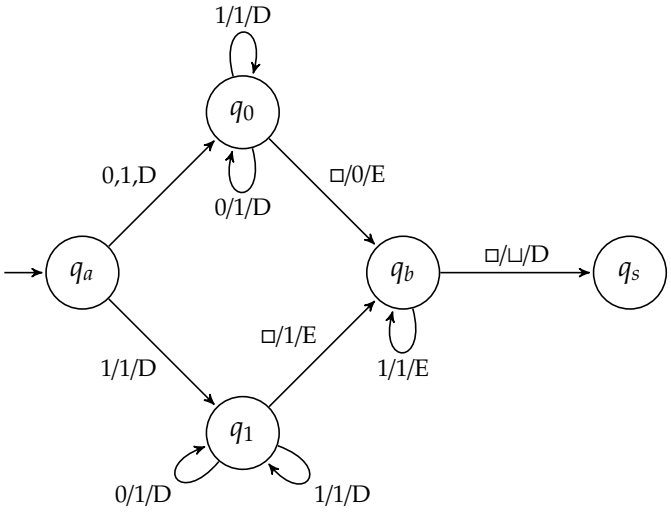


Figura 2.3: Representação de uma MT M com memória de um símbolo

Note que foi necessário duplicar todos os estados entre a obtenção da informação (é 0 ou 1?) e o seu uso. A informação é capturada no estado q_a e utilizada na transição de q_0 ou q_1 para q_b.

A informação se o primeiro símbolo é 0 ou 1 é armazenada no estado corrente após o primeiro símbolo, q_0 ou q_1. Dois símbolos, dois estados. Se fosse necessário guardar os dois primeiros símbolos da entrada, precisaríamos de quatro estados diferentes, assumindo que os dois primeiros símbolos só podem ser 0 e 1. Cada um dos quatro estados significaria 00, 01, 10 ou 11. A MT resultante é mostrada, incompleta, na Figura 2.4.

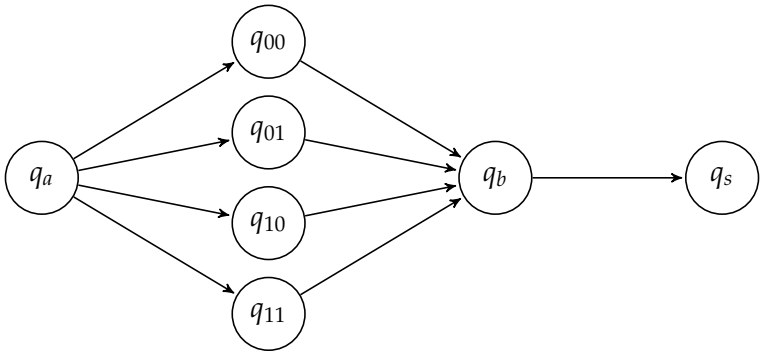


Figura 2.4: Representação incompleta de uma MT com memória para dois símbolos

No caso geral, qualquer número constante, sempre em relação à entrada, de símbolos pode ser armazenado nos estados. Em uma MT $M = (Q, \Sigma, I, q)$, temos $|\Sigma|$ símbolos. Para armazenar m símbolos são necessários $|\Sigma|^m$ novos estados. E todos os estados da máquina entre o armazenamento da informação e o uso devem ser multiplicados. Na Figura 2.4, não foi necessária esta duplicação pois o bit de informação é armazenado em q_0 e q_1 e utilizado logo na transição q_0 ou q_1 a q_b. Mas se houvesse estados intermediários entre q_0 (q_1) e q_b, todos eles deveriam ser duplicados.

Tudo se passa como se pudéssemos utilizar em uma MT variáveis que armazenam uma cadeia de

símbolos de tamanho constante. E estas variáveis poderiam ser utilizadas para decidir se certa instrução está habilitada ou não — estudemos o exemplo da Figura 2.3. Quando o estado corrente for q_0 e a célula corrente contiver $\square$, o estado corrente mudará para q_b e 0 será escrito no lugar de $\square$. O estado corrente só será q_0 neste ponto quando 0 tiver sido "armazenado" implicitamente no estado corrente no início da computação. Da mesma forma, qualquer número constante de símbolos, armazenado nos estados, pode ser utilizado para a decisão de qual instrução utilizar. Isto será utilizado nos algoritmos desta monografia.

2.3 Máquinas de Turing com Múltiplas Fitas

Uma máquina de Turing determinística com k fitas é uma quadrupla (Q, Σ, I, q) na qual Q e Σ são conjuntos de estados e de símbolos, I é um conjunto de instruções, $I \subset Q \times \Sigma^k \times Q \times (\Sigma \times D)^k$ e q é o estado inicial da máquina. A execução desta máquina se faz em k fitas ordenadas de 1 a k, cada qual com a sua própria cabeça de gravação e leitura. Estas cabeças se movimentam, durante a execução, independentemente. Uma instrução $(q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$ é executada quando o estado corrente é q e os símbolos correntes nas fitas são s_1, s_2, ..., s_k. Os símbolos s'_1, ... s'_k são escritos nas posições correntes das k fitas e a cabeça da fita i é movida na direção dada por d_i. Todas as restrições da definição das máquinas de Turing de uma única fita se aplicam a máquinas com k fitas:

1. o estado inicial da máquina é q;
2. $\{0, 1, \triangleright, \sqcup, \square\} \subset \Sigma$;
3. as células das fitas não são numeradas;
4. a entrada é colocada na fita 1 e a cabeça de leitura/gravação está no símbolo mais à esquerda da entrada. Todas as outras células da fita 1 e todas as células de todas as outras fitas contém o símbolo $\square$. A entrada é finita;
5. Q, Σ e $\mathcal{I}$ são conjuntos finitos;
6. há estados q_s e q_n no conjunto Q;
7. em uma instrução $(q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$, $q \notin \{q_s, q_n\}$ e $s'_i \neq \square$ para todo i;
8. dadas as instruções

$$I_1 =_{def} (q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$$

$$I_2 =_{def} (\overline{q}, \overline{s_1}, \overline{s_2}, \ldots \overline{s_k}, \overline{q}', \overline{s_1}', \overline{d_1}, \overline{s_2}', \overline{d_2}, \ldots \overline{s_k}', \overline{d_k})$$

de I, então

$$(q = \overline{q}) \wedge \bigwedge_{i=1}^{k} (s_i = \overline{s_i}) \longrightarrow (q' = \overline{q}') \wedge \bigwedge_{i=1}^{k} (s'_i = \overline{s_i}' \wedge d_i = \overline{d_i})$$

Convenciona-se que:

(a) a menos de menção em contrário, o estado inicial será q_0;

(b) o símbolo $\sqcup$ é utilizado para separar partes da entrada. Por exemplo, uma MT que multiplica dois números em binário pode separar estes números por $\sqcup$: $1010 \sqcup 111$;

(c) a entrada ou saída não podem conter símbolos do conjunto $\{\sqcup, \square\}$. O símbolo $\triangleright$ só pode aparecer como último símbolo da saída;

(d) a saída da MT é sempre colocada na fita k, a última. Quando o estado final for q_s ou q_n, a cabeça de leitura/gravação da fita k estará sobre o símbolo mais à esquerda da saída. Quando a máquina parar porque não foi possível encontrar uma instrução válida para ser executada, esta convenção não vale;

(e) O símbolo $\triangleright$ é colocado na primeira célula à direita da saída para indicar o fim desta. Então este símbolo não pode fazer parte da saída;

(f) todos os números inteiros serão representados em binário. É importante observar que para representar um número $n \neq 0$ em binário são necessários $1 + \lfloor \log_2 n \rfloor$ bits.

Definição 2.3.1. *Chamaremos de posição corrente ou célula corrente de uma fita da MT a posição ou célula que está sob a cabeça de leitura/gravação daquela fita.*

A execução de uma máquina de Turing com k fitas e uma entrada x (que pode ser vazia) é feita segundo o seguinte algoritmo:

1. quando o estado corrente for q e os símbolos nas posições correntes nas fitas forem s_1, s_2, ..., s_k, procura-se em I por uma instrução do tipo

$$(q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$$

   Pela definição deste tipo de MT, no máximo há uma destas instruções em I. Se não houver nenhuma, a máquina pára. Se houver, o estado corrente passa a ser q', os símbolos s'_1, s'_2, ... s'_k são escritos na célula corrente das k fitas (s'_i é escrito na fita i) e as cabeças de leitura/gravação se movem segundo a direção dada por d_1, d_2, ..., d_k;

2. quando o estado corrente se tornar q_s ou q_n, a máquina pára a execução.

Definição 2.3.2. *A execução do item 1 acima é chamado de passo da MT.*

Então a execução de uma MT com certa entrada consiste de uma sequência, possivelmente infinita, de passos. Na definição abaixo, $|x|$ é o tamanho da entrada x.

Definição 2.3.3. *Uma MT M com entrada e saída é uma MT com $k \geqslant 2$ fitas tal que a primeira fita é apenas de leitura e a última é apenas de escrita. Isto é, se $(q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$ for uma instrução de M, então*

(a) $s'_1 = s_1$

(b) se $s_k \neq s'_k$ *então* $d_k = 1$;

(c) se $d_k = 0$ *então* $s_k = s'_k$.

Chamaremos uma MT com entrada e saída de MTES.

Note que a primeira fita nunca é alterada, mas a cabeça de leitura/gravação pode se mover para frente e para trás. A cabeça de leitura/gravação da última fita está sempre sob uma célula que contém $\square$. Sempre que um símbolo é escrito, a cabeça se move para frente e o símbolo escrito nunca será utilizado (nunca será lido).

Uma máquina de Turing (Q, Σ, I, q) com k fitas pode ter no máximo $|Q|\,|\Sigma|^k$ instruções, o que corresponde a todas as possibilidades para os primeiros $k + 1$ símbolos de uma instrução

$$(q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$$

E quantos símbolos não necessários para implementar qualquer TM? A resposta é: dois. Basta 1 e 0, por exemplo. Todas as entradas e todas as computações podem ser convertidas para números, como é

feito em computadores digitais. E números podem ser expressos em notação unária. O 0 seria utilizado para sinalizar o fim da entrada. Uma convenção que torna a programação mais fácil é utilizar números de, por exemplo, três dígitos para representar símbolos. Explicando: suponha que desejamos converter uma MT M que use cinco símbolos em uma MT M' que use apenas 0 e 1. Então representamos os cinco símbolos de M por 000, 001, 010, 011 e 100 em M'. Cada instrução de M é convertida em três instruções de M'. Este é o número de dígitos em que cada símbolo de M foi convertido. Cada passo de M é feito por três passos de M'. Por exemplo, considere a instrução (q, s, q', s', d) de M. O símbolo s foi convertido para 011 e s' para 100. As instruções correspondentes em M' são:

$(q, 0, q_1, 0, 1)$
$(q_1, 1, q_2, 1, 1)$
$(q_2, 1, q_2', 0, -1)$ escreve o último dígito de s'
$(q_2', 1, q_3', 0, -1)$ escreve o segundo dígito de s'
$(q_3', 0, q_4', 1, -1)$ escreve o terceiro dígito de s'

Estas instruções reconhecem 011 e escrevem sobre estas três células o valor 100, em três passos:

$$011 \Longrightarrow 010 \Longrightarrow 000 \Longrightarrow 100$$

Se M tiver $|\Gamma|$ símbolos, estes poderão ser representados por $1 + \lfloor \log_2 |\Gamma| \rfloor$ cadeias compostas por 0 e 1. Cada instrução de M será convertida em $2(1 + \lfloor \log_2 |\Gamma| \rfloor) - 1$ instruções de M' como mostrado acima. $1 + \lfloor \log_2 |\Gamma| \rfloor$ instruções são necessárias para reconhecer o símbolo s codificado em uma cadeia de $1 + \lfloor \log_2 |\Gamma| \rfloor$ zeros e uns. Para escrever a cadeia associada a s' são necessárias outras $1 + \lfloor \log_2 |\Gamma| \rfloor$ instruções, sendo que uma delas pode ser compartilhada com as instruções de reconhecimento (a última instrução do reconhecimento é a mesma que a primeira instrução que escreve o último dígito de s').

Se houver duas instruções que começam com o mesmo estado q mas diferentes símbolos s (como $(q, s_1, ...)$ e $(q, s_2, ...)$) então o número de instruções será menor. Contudo, o número de instruções de M' que simulam uma instrução de M continuará a ser $2(1 + \lfloor \log_2 |\Gamma| \rfloor) - 1$.

Concluímos que apenas dois símbolos são necessários para implementar qualquer *software*, qualquer máquina de Turing. E a sobrecarga em relação a máquinas que utilizam n símbolos é constante e igual a $2(1 + \lfloor \log_2 n \rfloor) - 1$.

Definição 2.3.4. *A configuração de uma máquina de Turing com k fitas é uma sequência*

$$(q, b_1, a_1, b_2, a_2, \ldots, b_k, a_k)$$

na qual os símbolos da fita i estão concatenados, na ordem em que aparecem na fita, em $b_i a_i$ e a cabeça de leitura/gravação da fita i está sobre o último símbolo de b_i. O símbolo $\square$ só pode aparecer em uma configuração como o último símbolo de b_i, o que está sobre a cabeça de leitura/gravação. A string a_i pode ser vazia. Mas b_i contém pelo menos o símbolo da posição corrente da cabeça de leitura/gravação.

Em resumo, $b_i a_i$ contém todos os símbolos que interessam da fita i, sem considerar as infinitas células contendo $\square$ antes e depois das células contendo outros símbolos. A configuração inicial de uma MT (Q, Σ, I, q) com entrada x tal que $|x| > 0$ é:

$$(q, x_1, x_R, \square, \epsilon, \ldots, \square, \epsilon)$$

na qual x_1 é o primeiro símbolo de x e x_R são todos os símbolos de x exceto o primeiro.

Note que após um certo número finito de passos apenas um número finito de células foi tocada. Como a entrada é finita, a configuração é sempre finita.

Proposição 2.3.1. *Depois de t passos em uma máquina de Turing com k fitas, no máximo t células foram tocadas (lidas ou escritas) pela cabeça de leitura/gravação de cada uma das fitas.*

Demonstração. Trivial. Em cada passo, a cabeça de leitura/gravação de cada uma das fitas só pode se movimentar única célula à direita ou à esquerda. Se o movimento for sempre em uma única direção, depois de t passos terão sido tocadas t células. Para cada inversão da direção de movimento uma célula é reutilizada. □

A configuração de um computador, análogo à configuração de uma MT, consiste de todo o estado da memória. Isto inclui toda a memória RAM (principal e cache) e todos os registradores, inclusive aqueles que não podem ser modificados diretamento pelo programador, como o que guarda a próxima posição a ser executada.[1]

Note novamente que a descrição de uma MT com múltiplas fitas corresponde ao software e que é necessário um algoritmo e uma entrada para executá-la. Contudo, normalmente se chama de Máquina de Turing o conjunto de instruções da máquina mais a fita e o maquinário implícito utilizado para executá-la.

Suponha que C_1 é uma configuração de uma MT e depois de um único passo da execução a configuração desta MT seja C_2. Representaremos a relação entre C_1 e C_2 por

$$C_1 \longrightarrow C_2$$

Suponha que, partindo-se de C_1, após n passos se obtém a configuração C. Isto é representado por

$$C_1 \xrightarrow{n} C$$

Se o número de passos n não é conhecido, usa-se

$$C_1 \xrightarrow{\star} C$$

Definição 2.3.5. *Seja M uma máquina de Turing que sempre pára e que retorna 1 ou 0 conforme a sua entrada. Se retorna 1, o estado final é q_s. Se retorna 0, o estado final é q_n. Um MT com estas características será chamada de MT de decisão.*

Uma MT de decisão sempre pára a sua execução e retorna 0 ou 1.

Definição 2.3.6. *Dizemos que uma MT de decisão M decide uma linguagem L se*

$$x \in L \text{ se e somente se } M(x) = 1$$

$M(x)$ é o resultado da execução da MT M com a entrada x. A linguagem que uma MT M de decisão decide é denotada por $L(M)$. Assim, se M decide L, então $L = L(M)$.

Usamos o símbolo M de uma MT para duas finalidades:

(a) para a descrição da MT, como quando dizemos que $M = (Q, \Sigma, I, q)$ e;

(b) para a execução da máquina com certa entrada, como em $M(x)$.

Definição 2.3.7. *Dizemos que uma MT M executa em tempo $f(n)$ se, para entradas de tamanho n, o número de passos que ela toma é menor ou igual a $f(n)$.*

Definição 2.3.8. *Dizemos que uma MT M executa em espaço $f(n)$ se, para entradas de tamanho n, o número de células que ela utiliza é menor ou igual a $f(n)$. Nas máquinas com várias fitas, considera-se apenas o número de células utilizadas nas fitas de trabalho, o que exclui a fita de entrada e a de saída.*

2.4 Simulação de uma MTD com k fitas por uma MTD com uma Fita

Esta seção mostra como simular uma máquina de Turing com várias fitas em uma máquina de uma única fita. Sempre será possível simular qualquer modelo de computação em uma MT com uma única fita. A Tese de Church-Turing afirma que tudo o que é mecanicamente computável é computável por uma máquina de Turing. A única questão é o quanto eficientemente um modelo pode simular outro. Veremos que existem modelos de computação chamados de "razoáveis" em que esta simulação é polinomial entre quaisquer dois deles.

Seja $M = (Q, \Sigma, I, q)$ uma MTD com k fitas. Construiremos uma MTD P com uma única fita equivalente a M. Isto é, $M(x) = P(x)$ para todo x se M pára a sua execução com x ou $P(x) \uparrow$ se $M(x) \uparrow$. Lembre-se de que usamos $M(x) \uparrow$ para "M nunca termina a sua execução com a entrada x".

Na discussão abaixo, assuma que as células das MT´s são numeradas começando com 0 para a posição inicial da cabeça de leitura/gravação. As células à direita da posição inicial têm numeração positiva $(1, 2, \ldots)$ e as da esquerda, negativas $(\ldots, -2, -1)$. As k fitas de M são representadas na única fita de P da seguinte forma: todos os símbolos da posição 0 das k fitas de M são representados por um único símbolo de P, também na posição 0. Para cada símbolo $s \in \Sigma$ criamos outro símbolo $\tilde{x} \in \tilde{\Sigma}$ para indicar que a posição corrente da cabeça de leitura/gravação. Se a codificação de um símbolo s de P na posição j da fita indicar que o símbolo da fita i de M é $\tilde{r}$, então na simulação que P está fazendo o símbolo da fita i é r e a cabeça de leitura/gravação desta fita está sobre esta posição.

O número de símbolos de P necessários para simular M é pelo menos

$$(|\Sigma| + \left|\tilde{\Sigma}\right|)^k = (2\,|\Sigma|)^k$$

A simulação de M por P é feita segundo o seguinte algoritmo:

1. converta a entrada x de M para uma entrada de P. Acrescente \$ depois do último símbolo de x. Acrescente este mesmo símbolo antes do primeiro símbolo de x. Este passo leva $O(n)$, $n = |x|$ passos. A conversão de cada símbolo pode ser feita em tempo constante já que esta só depende do número de símbolos de M, constante em relação à entrada. Caminhar do último símbolo de x para o primeiro, depois de todas as conversões, também gasta n passos;

2. o estado corrente $p \in Q$ de M é armazenado no estado corrente de P. Mas não teremos que o estado de P será simplesmente p, pois P tem as suas próprias computações a fazer. Então p estará armazenado implicitamente no estado corrente de P;

3. a simulação da execução de cada passo de M é feita da seguinte forma: P percorre toda a fita à esquerda e à direita da posição corrente coletando informações sobre os símbolos que estão nas posições correntes das k fitas de M. P precisa de pelo menos $|\Sigma|^k$ estados para armazenar esta informação. Para cada um destes estados, há $|Q|$ possíveis estados de M que também devem estar armazenados. Então são necessários pelo menos $|Q|\,|\Sigma|^k$ estados para este passo. De fato, se P em suas computações usuais utiliza m estados, então o número de estados que esta máquina precisará será de $m\,|Q|\,|\Sigma|^k$. Cada estado de P tem que considerar que a máquina M sendo simulada está em uma particular combinação de estado corrente e símbolos sobre as k células correntes;

4. como as instruções de M estão armazenadas em P, esta máquina tem agora condições de simular a execução da instrução apropriada de M. Se não há nenhuma, a máquina pára. Se há, a instrução é executada. Para isto, a cabeça de leitura/gravação de P deve percorrer a fita à esquerda até encontrar um \$ e então caminhar para a direita simulando a instrução de M. Esta simulação alterará apenas as células de P que codificam células correntes de alguma fita de M ou células vizinhas a esta — em M, uma instrução só altera a célula corrente da fita j mas pode alterar a posição corrente para a esquerda ou direita da posição atual.

2.5 Máquinas de Turing Não Determinísticas

Uma máquina de Turing não determinística (MTND) é definida analogamente à MT determinística exceto que não há restrições quando ao conjunto I de instruções. Isto é, pode-se ter duas instruções com os dois primeiros símbolos q e s iguais. Durante a execução, quando o estado corrente for q e a célula corrente contiver s, a MTND escolhe aleatoriamente uma das instruções que começam com q, s para executar. Máquinas de Turing não determinísticas de múltiplas fitas são definidas analogamente.

Na discussão abaixo, nos referimos a uma MTND de uma única fita. Mas o raciocínio se aplica a máquinas com qualquer número de fitas.

Uma MT determinística é um caso especial de MTND na qual há no máximo uma única instrução que começa com o mesmo estado e símbolo da fita. Uma MTND $N = (Q, \Sigma, I, q)$ que não pode ser considerada uma MT determinística (MTD) possui pelo menos um conjunto

$$\delta_{ps} = \{(p, s, p', s', d) : (p, s, p', s', d) \in I\}$$

tal que $|\delta_{ps}| > 1$. Isto é, há pelo menos duas instruções que começam pelo mesmo p, s.

Seja $N = (Q, \Sigma, I, q)$ uma MTND que não é uma MTD. Na computação de $N(x)$, há diversas escolhas a serem feitas durante a execução da MT. Estas escolhas formam uma árvore de possibilidades. Esta é uma árvore dirigida na qual os vértices são configurações e as arestas são execuções de instruções. A execução de uma instrução transforma uma configuração em outra. A raiz desta árvore é a configuração inicial, com a entrada x na fita, o estado corrente q e a cabeça de leitura/gravação sobre o símbolo mais à esquerda de x. Assumindo que $N(x)$ sempre pára para qualquer entrada, as folhas desta árvore representam os possíveis resultados da computação. Então o estado destas configurações (folhas são configurações) é q_s ou q_n.

Definição 2.5.1. *A árvore dirigida que representa todas as possibilidades da computação $N(x)$ será denotada por $Ar(N(x))$ e chamada de "árvore de computação".*

Um exemplo de árvore $Ar(N(x))$ de configuração é mostrado na Figura 2.5.

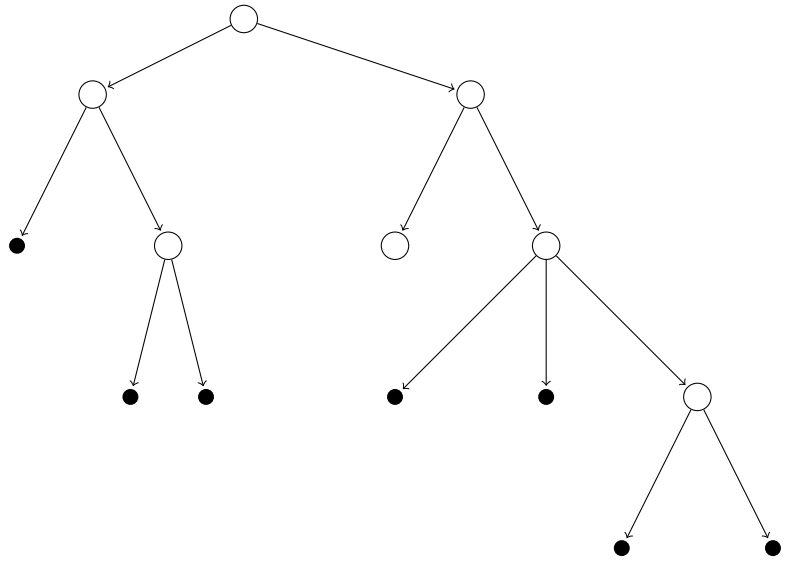


Figura 2.5: Exemplo de uma árvore $Ar(N(x))$

Em uma MT determinística, esta árvore é linear pois a cada configuração apenas uma única instrução se aplica:

$$C_1 \longrightarrow C_2 \longrightarrow \ldots C_t$$

Isto é, os vértices da árvore são (C_1, C_2), (C_2, C_3), ..., (C_{t-1}, C_t). A configuração inicial é C_1 e a final é C_t.

O grau de saída de um vértice v em um grafo G é o número de vértices w tal que (v, w) pertence a G. Em uma MTD, todos os vértices da árvore correspondente a $N(x)$ têm grau de saída 1, exceto um deles, que tem grau de saída 0.

Uma MTND de decisão sempre pára a sua execução e retorna 0 ou 1 qualquer que sejam as escolhas não determinísticas feitas durante a computação.

Definição 2.5.2. *Dizemos que uma MTND de decisão N decide uma linguagem L se*

$$x \in L \text{ sse existe uma sequência de escolhas tal que } N(x) = 1$$

Ou seja, uma MTND N decide uma linguagem L se

$$x \in L \text{ sse existe um caminho da raiz a uma folha em } Ar(N(x)) \text{ tal que } N(x) = 1$$

A exigência $N(x) = 1$ é equivalente a ter o estado final igual a q_s, de aceitação.

Uma MTND também pode ser utilizada para calcular o valor de uma função.

Definição 2.5.3. *Uma MTND N calcula o valor de uma função $g(x)$ se, quando a entrada de N for x, existir uma sequência de escolhas não determinísticas tal que o resultado colocado na fita ao final da computação seja $g(x)$. O estado final neste caso deve ser q_s. Caso contrário exige-se que o estado final seja q_n.*

A árvore de computação $Ar(N(x))$ pode ter várias folhas que correspondem ao estado q_n. Mas para todo x deve existir pelo menos uma folha correspondente a um estado q_s.

Uma máquina de Turing determinística (Q, Σ, I, q) com k fitas pode ter no máximo $|Q|\,|\Sigma|^k$ instruções. Já uma MTND este número é muito maior, igual a

$$|Q|\,|\Sigma|^k\,|Q|\overbrace{|\Sigma|\,3\ldots|\Sigma|\,3}^{k} = 3^k\,|Q|^2\,|\Sigma|^{2k}$$

o que corresponde a todas as possibilidades para todos os símbolos de uma instrução

$$(q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$$

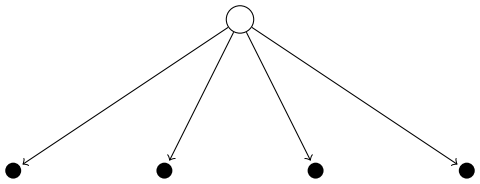


Figura 2.6: Conjunto δ_{ps} com quatro instruções começando com p, s

Opcional

Seja $N = (Q, \Sigma, I, q)$ uma MTND. O conjunto I de instruções pode ser dividido em subconjuntos que começam com os mesmos símbolos p, s para todo $p \in Q$ e $s \in \Sigma$. Estes subconjuntos são agrupados no conjunto S:

$$S = \{\delta_{ps} : |\delta_{ps}| \neq \emptyset\}$$

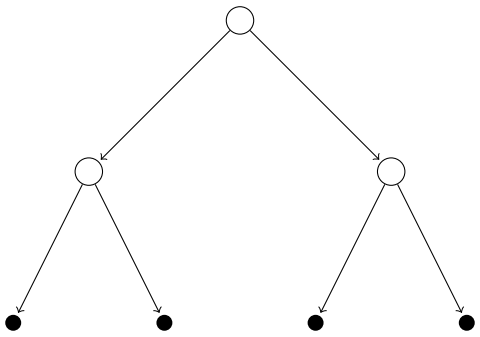


Figura 2.7: Representação das instruções de N' com grau dois em cada vértice

O número $|\delta_{qs}|$ é sempre finito. Na árvore $Ar(N(x))$, o máximo grau de saída d_N de todos os vértices será o tamanho do maior conjunto de S. Isto é,

$$d_N = \text{máx}\{|\delta| : \delta \in S\}$$

Usamos máxX para o maior elemento do conjunto X, assumindo que $X \subset \mathbb{N}$. Observe que o valor d_N depende apenas do conjunto I de instruções de N. Em uma MTD M, $d_M = 1$ e $|S| = |I|$.

Uma MT será considerada não determinística se pelo menos duas instruções começarem com o mesmo estado e símbolo. Não é necessário que para cada instrução (p, s, p', s', d) exista uma outra diferente começando também por p, s. E mesmo em uma MTND N, uma computação $N(x)$ pode ter uma $Ar(N(x))$ linear, sem escolhas não determinísticas. Basta que durante a computação nunca tenha existido a possibilidade de escolha. Então a $Ar(N(x))$ de uma MTND N não é necessariamente do formato apresentado na Figura 2.5: alguns vértices podem ter grau de saída um (quando instruções determinísticas são utilizadas).

Dada uma MTND N sempre é possível transformá-la em uma MTND N' tal que $\left|\delta_{ps}\right|$ é zero, um ou dois. Mostraremos como fazer esta transformação informalmente assumindo que $N = (Q, \Sigma, I, q)$.

Um conjunto δ_{ps} de N tal que $\left|\delta_{ps}\right| = 4$ está representado na Figura 2.6. O vértice acima representa uma configuração na qual o estado corrente é p e a célula corrente contém s. Há quatro possíveis novas configurações, correspondentes aos elementos de δ_{ps}. Na MTND N', criada a partir de N, criam-se novos estados intermediários de tal forma que a grafo se transforma naquele mostrado na Figura 2.7. O grau de saída de cada vértice é zero ou dois.

No caso geral, $\left|\delta_{ps}\right| = n$, $n > 2$ e o grafo é transformado em uma árvore na qual o grau de saída é zero ou dois. Se a árvore original possui $n + 1$ vértices, são criados $n - 2$ vértices intermediários. Isto corresponde à criação em N' de $n - 2$ estados intermediários e $n - 2$ novas instruções que não movimentam a cabeça de leitura/gravação. Como exemplo, suponha que

$$\delta_{ps} = \{(q_0, s_0, q_1, s_1, d_1), (q_0, s_0, q_2, s_2, d_2), (q_0, s_0, q_3, s_3, d_3), (q_0, s_0, q_4, s_4, d_4)\}$$

Este conjunto de instruções de N é transformado em um novo conjunto para N':

$$(q_0, s_0, q_0', s_0, 0), (q_0, s_0, q_0'', s_0, 0), (q_0', s_0, q_1, s_1, d_1), (q_0', s_0, q_2, s_2, d_2)$$
$$(q_0'', s_0, q_3, s_3, d_3), (q_0'', s_0, q_4, s_4, d_4)$$

Note que os estados acrescentados, q_0' e q_0'' foram colocados nos vértices internos da árvore. O tempo de execução de $N'(x)$ é maior do que $N(x)$ se forem utilizados os novos estados q_0' e q_0''. Neste exemplo, no pior caso, uma instrução de N corresponderia a duas instruções de N' (altura da

árvore 2.7 menos um). Mas isto é uma constante. Então existe uma constante k tal que se o número de passos da execução de $N(x)$ for m, o número de passos de $N'(x)$ será km se as mesmas escolhas não determinísticas forem feitas. Mais formalmente,

$$T_{N'}(n) \leqslant kT_N(n)$$

considerando $T_N(n)$ o máximo número de passos que a execução de $N(x)$ leva com entradas de tamanho $n = |x|$.

2.6 Simulação de uma MTND por uma MTD

O não determinismo não acrescenta nenhum poder computacional extra às MTND. Uma MTND $N = (Q, \Sigma, I, q)$ pode ser simulada por uma MTD M de três fitas construída como descrito a seguir. Para simular $N(x)$, M simula todos os caminhos da árvore de computação $Ar(N(x))$. Mais especificamente, seja

$$\delta_{ps} = \{(p, s, p', s', d) : (p, s, p', s', d) \in I\}$$
$$S = \{\delta_{ps} : |\delta_{ps}| \neq \emptyset\}$$
$$d_N = \text{máx}\{|\delta| : \delta \in S\}$$

d_N é o maior grau de saída possível em uma árvore $Ar(N(x))$, qualquer que seja a entrada x. Se N sempre pára com qualquer entrada, a simulação pode ser feita da seguinte forma: M percorre a árvore $Ar(N(x))$ em profundidade realizando todas as computações que N faria. Se uma configuração folha é atingida cujo estado é q_s, M termina a computação no estado q_s. A simulação propriamente de N é feita na terceira fita de M. A segunda fita contém o caminho na árvore de computação, uma sequência $e_1, e_2, \ldots e_t$, na qual t é o número de passos executados até o momento, a altura de $Ar(N(x))$. Quando um estado final q_n é atingido (folha da árvore), a simulação retrocede até o vértice anterior e realiza a próxima escolha não determinística. Estas escolhas são sempre finitas — há no máximo d_N escolhas para N, um número que depende apenas de N e não da entrada. Sempre que um caminho resulta no estado q_n é feito o retrocesso.

A Figura 2.8 mostra uma possível $Ar(N(x))$. Os vértices estão numerados na ordem de visita de uma busca em profundidade que corresponde à ordem de simulação Na primeira instrução há duas escolhas não determinísticas. Então e_1 irá assumir os valores 1 (sub-árvore esquerda) e 2 (sub-árvore direita) durante a simulação (se não for encontrado um estado q_s na sub-árvore esquerda composta pelos vértices $2 - 6$). Quando e_1 assumir 1, a sub-árvore esquerda será simulada e e_2 inicialmente assume o valor 1, indicando que a próxima configuração a ser obtida é a correspondente ao vértice 3. E assim por diante.

A MT M utiliza a primeira fita para manter o número de passos simulados até o momento, digamos t. A segunda fita armazena a sequência $e_1, e_2, \ldots e_t$ de possíveis escolhas. A terceira fita simula a fita de N. É necessário também guardar o estado anterior da fita antes de uma escolha não determística. Se a escolha falhar, deve-se deixar a fita como ela estava antes daquela escolha. Complicado.

Esta simulação só funciona quando N sempre termina a sua execução para qualquer entrada. A próxima simulação termina sempre que uma das folhas corresponde a um estado q_s. Descrevamos então a máquina M que faz esta simulação.

M percorre a árvore $Ar(N(x))$ em profundidade como na simulação anterior, realizando todas as computações que N faria. Mas agora só são simulados t passos de cada vez, sendo que t assume $1, 2, 3, \ldots$. Então com $t = 1$ M simula a execução de todas as possibilidades do primeiro passo de N. Na Figura 2.9, com um passo a MTND N pode estar nos vértices nomeados "1". Se algum estado q_s é atingido, a simulação pára e M fica no estado q_s. Senão M continua a simulação percorrendo a árvore em

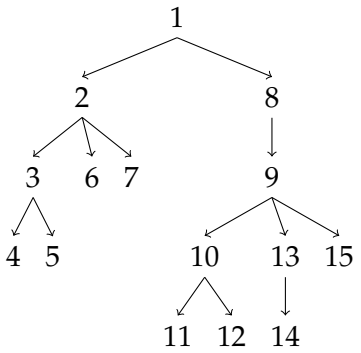


Figura 2.8: Ordem de simulação de uma MTND

profundidade até um número de passos igual a 2. Os vértices nomeados 2 da Figura 2.9 são atingidos. E assim por diante até que um estado q_s é encontrado. Se $Ar(N(x))$ for infinita e não houver nenhum estado q_s em nenhuma folha, M nunca termina a sua execução.

M utiliza um símbolo $\overline{\sqcup}$ para representar o símbolo $\sqcup$ de N. Isto é necessário pois uma nova computação é simulada sobre uma fita já previamente utilizada. Então, após copiar a entrada na fita, deve-se colocar o símbolo $\overline{\sqcup}$ em todas as células restantes que não contenham $\sqcup$.

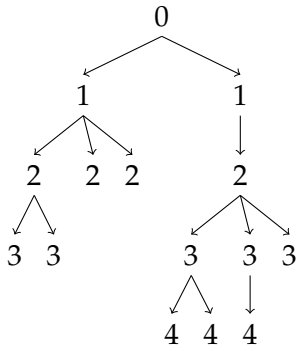


Figura 2.9: Outra simulação de uma MTND

2.7 Codificação de Máquinas de Turing

Codificar é transformar uma descrição qualquer em número inteiro. Mais precisamente, dado um conjunto A qualquer, uma **codificação** de A é uma função $f : A \longrightarrow \mathbb{N}$ tal que

(a) f é injetora. Isto é, dados $x, y \in A$, $x \neq y$, $f(x) \neq f(y)$;

(b) dado $n \in \mathbb{N}$, existe um algoritmo que descobre se existe $x \in A$ tal que $f(x) = n$ ou garante que tal x não existe.

Sendo T o conjunto de todas as máquinas de Turing, nesta seção descrevemos uma função $t : T \longrightarrow \mathbb{N}$ que é uma codificação de T. Isto é, t toma uma MT e retorna um número natural.

Mostraremos como codificar uma MT $M = (Q, \Sigma, I, q)$. Todos os elementos de todos os conjuntos da definição de M serão codificados como números binários entre 1 e $|Q| + |\Sigma| + 3$. Estes conjuntos são Q, Σ e $\{-1, 0, 1\}$, sendo que este último indica o movimento da cabeça de leitura/gravação. Cada número terá

exatamente $1 + \lfloor \log_2(|Q| + |\Sigma| + 3) \rfloor$ dígitos para que não exista ambiguidade — se os números pudessem ter número de dígitos diferentes, não seria possível saber, por exemplo, se 11 representa 11 ou 1 seguido de 1. Lembre-se de que todos os números entre 0 e n podem ser representados em binário com $1 + \lfloor \log_2 n \rfloor$ dígitos. Se tivermos cinco elementos ao todo, estes podem ser representados por

$$000, 001, 010, 011, 100, 101$$

Vejamos como cada parte de M é codificada. Os estados de Q são codificados como números entre 1 e $|Q|$. Os símbolos de Σ como números entre $|Q| + 1$ e $|Q| + |\Sigma|$. Os símbolos utilizados como direção de movimento de M do conjunto $\{-1, 0, 1\}$ são codificados como números entre $|Q| + |\Sigma| + 1$ e $|Q| + |\Sigma| + 3$. Cada instrução (q, s, p, r, d) é codificada como

$$q^c s^c p^c r^c d^c$$

na qual q^c é a codificação em número de q (idem para os outros símbolos). Todas as intruções de uma certa máquina de Turing possuem o mesmo tamanho em número de dígitos (mesmo formato, símbolos representados pelo mesmo número de dígitos). Cada símbolo de M é codificado em $1 + \lfloor \log_2(|Q| + |\Sigma| + 3) \rfloor$ bits ou símbolos 0 e 1. Então cada instrução é codificada em $5(1 + \lfloor \log_2(|Q| + |\Sigma| + 3)) \rfloor$ bits. Se fosse necessário, poderíamos colocar um 2 entre as codificações de uma instrução, resultando em

$$2q^c 2s^c 2p^c 2r^c 2d^c$$

A codificação do conjunto I, denotada por I^c, é a concatenação da codificação de todas as suas instruções.

A codificação da máquina $M = (Q, \Sigma, I, q)$, denotada por $\langle M \rangle$, é

$$|Q|_{bin}\, 2\, |\Sigma|_{bin}\, 2 q^c I^c$$

sendo $|Q|_{bin}$ o número $|Q|$ em binário e I^c a codificação das instruções, que é a concatenação das codificação das instruções. Poder-se ia eliminar q^c da codificação de M, colocando uma instrução que contenha o estado inicial como primeira instrução da codificação de M.

A codificação apresentada acima, a função t, é claramente injetora. Duas máquinas diferentes necessariamente serão codificadas em dois números diferentes. E, dado um $n \in \mathbb{N}$, pode-se descobrir qual máquina M tal que $\langle M \rangle = n$.

Pode-se construir uma função $u : \mathbb{N} \longrightarrow T$ que toma um natural e retorna a MT correspondente a ele. Esta função é definida como:

$$u(n) = \begin{cases} M & \text{se } \langle M \rangle = n \\ M_\emptyset & \text{se, para nenhuma } M,\ t(M) = n \end{cases}$$

Sendo $M_\emptyset$ a seguinte MT: $(\{q_s, q_n\}, \{0, 1, \triangleright, \sqcup, \square\}, q_s, \emptyset)$. Esta máquina, ao iniciar a sua execução, já pára, pois o estado inicial já é um dos estados finais. A escolha desta máquina é arbitrária. Poderia ser qualquer uma.

A função u toma um natural n e retorna a MT M cuja codificação é igual a n. Se não existir tal M, u retorna uma MT arbitrária $M_\emptyset$.

Para descobrir qual a máquina de Turing codificada em um número $n \in \mathbb{N}$ é necessário fazer uma análise sintática em n. A análise sintática é parte de um compilador. Há uma gramática associada à codificação assim como há uma gramática que descreve uma linguagem como C++ ou Java. Pela análise sintática poderíamos descobrir os elementos de M: $|Q|$, $|\Sigma|$. q e I.

2.8 A Máquina de Turing Universal

Mostraremos uma máquina de Turing que pode simular todas as outras MT determinísticas, chamada de *Máquina de Turing Universal* (MTU). A MTU toma como entrada a descrição de uma máquina M e a entrada x desta máquina separados por $\sqcup$. A descrição de M é feita por uma *codificação* das suas instruções: cada instrução desta máquina é transcrita em símbolos da MTU.

A MTU utiliza apenas os símbolos $\{0, 1, \triangleright, \sqcup, \square\}$ no alfabeto. Já uma máquina $M = (Q, \Sigma, I, q)$ pode utilizar muito mais símbolos. Mas isto não importa, pois é sempre possível codificar qualquer quantidade de símbolos utilizando apenas 0 e 1.

A entrada para a MTU é $\langle M \rangle 2x$, sendo x um número binário. A MTU utiliza três fitas. Na primeira fica a entrada $\langle M \rangle 2x$. A segunda fita guarda $|Q|$ seguido de $|\Sigma|$ e o estado corrente. Estas informações estão na primeira fita mas ficam mais facilmente acessíveis na segunda fita. A terceira fita simula a fita de M. O algoritmo utilizado por MTU é:

1. copie $|Q|$, $|\Sigma|$ e o estado inicial da primeira para a segunda fita;

2. copie a entrada x para M da primeira para a terceira fita, deixando a cabeça de gravação no início da entrada (célula mais à esquerda);

3. seja p o estado corrente que está na segunda fita e s o símbolo corrente da terceira fita. Procure na primeira fita por uma instrução que comece por p, s. Se não houver nenhuma, pare a execução da MTU (e consequentemente a simulação de M). Se houver uma instrução (p, s, p', s', d), copie p' para a segunda fita, escreva s' na célula corrente da terceira fita e mova a cabeça de leitura/gravação da terceira fita na direção indicada por d;

4. repita o passo anterior até que o estado corrente seja q_s ou q_n.

Chamando a MTU de U, pela descrição do algoritmo acima fica claro que $U(\langle M \rangle 2x) = M(x)$ para todo x. Então uma MTU é um *interpretador* para máquinas de Turing, o primeiro de todos os interpretadores, projetado por Turing em seu artigo que define as máquinas de Turing [11] [12]. É interessante notar que a descrição de M é um dado para U, sendo este o primeiro caso em que um "programa" é armazenado como dado.

Usualmente substituimos o 2 que aparece nas codificações por $\sqcup$ por uma questão de legibilidade. Então a entrada para a MTU seria $\langle M \rangle \sqcup x$ e a codificação de M, $\langle M \rangle$, seria

$$|Q|_{bin} \sqcup |\Sigma|_{bin} \sqcup q^c I^c$$

Neste caso, a cadeia $\langle M \rangle \sqcup x$ conteria símbolos do conjunto $\Sigma_1 = \{0, 1, \sqcup\}$ e $\langle M \rangle \sqcup x$ seria uma cadeia de Σ_1. Isto é,

$$\langle M \rangle \sqcup x \in \Sigma_1^\star$$

Então há duas formas de codificação: como número ou como cadeia sobre Σ_1. Utilizaremos uma ou outra forma de codificação conforme a proposição ou exemplo.

2.9 Conexões com a Computação

Uma MTD pode ser facilmente implementada em uma linguagem de programação como C++. Mostraremos uma possível implementação. Para facilitar a codificação, admitiremos que não há células à esquerda da célula inical da fita — ela é infinita apenas à direita. A fita é representada por um vetor t de inteiros. Cada símbolo é um número inteiro e os estados são *labels*.

Dada a MT (Q, Σ, I, q), cada instrução (q, s, q', s', d) seria implementada pelo seguinte código:

```
q: if ( t[i] == s ) {
      t[i] = s';
      i = i + d;
      goto q';
   }
```

Se há mais de uma instrução que começa com q, haveria pelo menos um `else` no `if`. Por exemplo, suponha que existam duas instruções que comecem com q, (q, s, q', s', d) e (q, b, q'', b', e). Então o código em C correspondente seria:

```
q: if ( t[i] == s ) {
      t[i] = s';
      i = i + d;
      goto q';
   } else if ( t[i] == b ) {
      t[i] = b';
      i = i + e;
      goto q'';
   }
```

Lembre-se de que é possível guardar um número constante de valores através dos estados de uma MT (Página 11). Então podemos ter um número constante $r_1, r_2, \ldots r_m$ de registros de tamanho fixo armazenados nos estados de uma MT. Pela implementação acima, estes registros seriam guardados pela localização da próxima instrução a ser executada — a informação de qual a próxima instrução a ser executada já revela os valores dos registros.

Como um computador tem um número constante de posições de memória, pode-se simular um computador em uma MT sem o uso da fita. Isto é, pode-se "simular" um computador usando uma Máquina de Estados Finitos (MEF), que é uma Máquina de Turing sem memória. Considere então um computador C. Ele pode ser simulado por uma MEF sem o uso da fita. Esta MEF pode ser simulada em linguagem C como descrito acima (sem usar a fita). Esta simulação pode ser feita em um outro computador C'. A memória de C é simulada em C' pelo registrador que diz qual a próxima instrução a ser executada (e nada mais). Naturalmente este registrador deve ter pelo menos o número de bits da memória de C. Então modificações na memória de C são simuladas em C' pelas instruções `goto q'` como as do código acima.

Pela simulação de uma instrução de uma MT em linguagem C, fica claro que uma instrução de uma MT combina três instruções de linguagens de programação: seleção (*if*), atribuição e salto (`goto`). Além de selecionar a próxima instrução de memória a ser programada. Usualmente as três funções estão presentes em construções diferentes de uma linguagem de programação ou da linguagem de máquina.

Uma máquina de Turing consegue realizar qualquer tipo de computação com um único tipo de instrução, o que não é usual. Usualmente os computadores contam com dezenas ou centenas de instruções diferentes.

Existe uma outra máquina abstrata que também emprega um único tipo de instrução [13]. De fato, a instrução pode ser de dois tipos diferentes:

(a) `subleq a, b, c` com o significado

```
Mem[b] = Mem[b] - Mem[a]
if (Mem[b] <= 0) goto c
```

Subtraia o conteúdo da posição `b` da memória do conteúdo da posição `a` colocando o resultado na posição `b`. Vá para a instrução da posição de memória `c` se o conteúdo da posição `b` é menor ou igual a zero.

(b) `subneg a, b, c` com o significado

```
Mem[b] = Mem[b] - Mem[a]
if (Mem[b] < 0) goto c
```

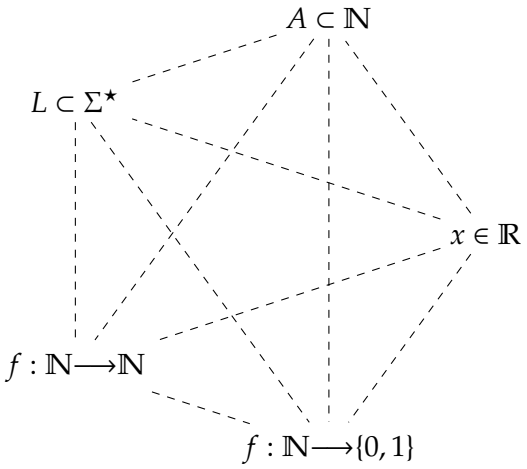
A descrição é similar à acima.

Uma posição de memória específica deve inicialmente conter um certo número maior do que zero, por exemplo 1. E outra posição deve conter zero. Assume-se que cada posição de memória pode conter um inteiro de qualquer tamanho — somente desta forma os números `a`, `b` e `c` podem acessar uma quantidade arbitrariamente grande de posições de memória.

Capítulo 3

Classificação de Linguagens

Este Capítulo apresenta linguagens computáveis ou decidíveis (recursivas) e as computacionalmente enumeráveis (recursivamente enumeráveis). Os nomes entre parênteses foram os mais utilizados até recentemente (2013). O objetivo da Computabilidade é o estudo destas linguagens. Ou um dos conjuntos equipotentes a ele mostrados na Figura abaixo, que já foi mostrada no primeiro Capítulo.



Linguagens não são todas iguais. Linguagens regulares (LREG) são um subconjunto das linguagens livres de contexto (LLC) que são um subconjunto das linguagens *computáveis* (LC) que são um subconjunto das linguagens computacionalmente enumeráveis (LCE). As duas últimas serão vistas neste Capítulo. Veremos inicialmente as definições deste conceitos. A seguir, serão apresentados teoremas sobre estes interessantes tipos de linguagens.

Definição 3.0.1. *Uma linguagem L é computável (recursiva) se existe uma MT M de decisão tal que*

$$x \in L \text{ sse } M(x) = 1$$

Isto é, M decide L.

Definição 3.0.2. *Uma linguagem L é ou computacionalmente enumerável, c.e. (recursivamente enumerável, r.e.) se existe uma MT M tal que se $x \in L$ então $M(x) = 1$. Se $x \notin L$, $M(x) \uparrow$. Dizemos que M aceita L.*

Isto é, se $x \notin L$ a MT M não termina a sua execução. A máquina M não pode ser utilizada para verificar se dado x pertence a L. O motivo é que, ao computar $M(x)$, não sabemos quanto tempo devemos esperar para concluir se $x \in L$ ou não. Se a máquina termina a sua execução, então com certeza $x \in L$. Mas se depois de certo tempo ela ainda não terminou a execução, não podemos concluir nada. Talvez a

máquina nunca termine e então $x \notin L$ ou talvez ela termine daqui a um milhão de anos e teríamos $x \in L$. Em resumo, M é inútil na prática, embora extremamente importante teoricamente.

Outra forma de definir linguagens computacionalmente enumeráveis é o seguinte: uma linguagem L é c.e. se e somente se L é o domínio de uma MT M. O domínio $\text{dom}(M)$ de uma MT M é

$$\text{dom}(M) = \{x : M(x) \downarrow\}$$

Proposição 3.0.1. *Se L é computável, L é computavelmente enumerável.*

Demonstração. Se há uma máquina M que decide L, claramente há uma máquina que aceita L. Basta modificar M de tal forma que, se ela for retornar 0, indicando $x \notin L$, ela entra em laço infinito. □

Proposição 3.0.2. *Se L e L^c são computavelmente enumeráveis, então L e L^c são computáveis.*

Demonstração. Vamos encontrar uma MT M que decide L. Sabemos que existem MT N_L e N_{L^c} que aceitam L e L^c. A máquina M simula a execução de N_L e N_{L^c} com a entrada x, um passo de cada máquina, intercaladamente. Como $x \in L$ ou $x \in L^c$, necessariamente uma das máquinas irá parar. Se for N_L que pára, M retorna 1. Se for N_{L^c}, M retorna 0. Analogamente para a máquina que decide L^c. Então M decide L pois não só sempre retorna a resposta correta como sempre pára a sua execução. □

Definição 3.0.3. *Dizemos que uma MT M enumera os elementos de uma linguagem $L \subset \Sigma^\star$ (ou $A \subset \mathbb{N}$) não vazia se:*

(a) M despresa a sua entrada e imprime na fita um elemento de L, seguido de $\sqcup$, seguido de outro elemento de L, seguido de $\sqcup$ e assim por diante;

(b) dado $x \in L$, em algum momento da execução de M x será impresso na fita;

(c) M nunca pára a sua execução.

Note que os elementos impressos na fita por M podem ser repetidos e estar fora de ordem. Em particular, se L (ou A) for finito, haverá repetições.

Proposição 3.0.3. *$A \subset \mathbb{N}$ (ou $L \neq \emptyset$) é computavelmente enumerável se e somente se existe uma máquina de Turing que enumera todos os elementos de A (ou L).*

Demonstração. Esta proposição é ligeiramente mais fácil de entender se usarmos $A \subset \mathbb{N}$ ao invés de $L \neq \emptyset$.

($\Longrightarrow$) Suponha que uma MT M' de várias fitas enumera os elementos de uma linguagem $A \subset \mathbb{N}$ (ou $L \subset \Sigma^\star$) — os elementos são colocados na fita de saída. Construiremos uma máquina M tal que, se $x \in L$, $M(x) = 1$ e, se $x \notin L$, $M(x) \uparrow$. M tem que parar e escrever 1 na saída se a sua entrada x pertence a L e entrar em laço infinito se não pertencer. Então, dada a entrada x, M simula M' e, à medida que esta produz cada elemento de L, M' compara este elemento que acaba de ser produzido com x. Se forem iguais, M' pára e escreve 1 na saída. Se M' nunca produzir x, M continuará sua busca eternamente sem nunca parar, o que é o comportamento esperado neste caso.

($\Longleftarrow$) Suponha que A seja c.e. Então existe uma MT M tal que, se $x \in A$, $M(x) = 1$ e, se $x \notin A$, $M(x) \uparrow$. Usando M, construiremos uma máquina de Turing de duas fitas M' que enumera todos os elementos de L.

Na primeira fita M' simula a execução de M com todos os elementos $0, 1, 2, 3 \ldots$, mas não todos simultaneamente. Inicialmente, M' simula o primeiro passo da computação $M(0)$, depois o segundo passo da computação de $M(0)$ e o primeiro de $M(1)$, depois o terceiro de $M(0)$, o segundo de $M(1)$ e o primeiro de $M(2)$ e assim por diante. Se uma computação $M(n)$ pára (e neste caso $M(n) = 1$), M' imprime n na segunda fita. Para qualquer $k \in \mathbb{N}$, em algum momento M' irá começar a computação $M(k)$ e, se $k \in \mathbb{N}$, esta computação irá parar. Neste caso M' imprime K na segunda fita. Se a computação $M(k)$

não parar, a simulação continuará para sempre e k não será impresso na segunda fita. A computação de $M(m)$, $m > k$ será mais lenta, mais eventualmente todos os elementos de A serão enumerados.

Para descrever melhor a simulação semi-simultânea de M com todos os elementos de $\mathbb{N}$, definimos $M(n)_j$ como o j-ésimo passo da computação de $M(n)$. A tabela abaixo mostra como M' simula as computações de M com os elementos de $\mathbb{N}$.

$M(0)_1$
$M(0)_2$ $M(1)_1$
$M(0)_3$ $M(1)_2$ $M(2)_1$
$M(0)_4$ $M(1)_3$ $M(2)_2$ $M(3)_1$
...

Cada uma das linhas corresponde a vários passos de M'. A computação de uma coluna $k \geqslant 0$ pára se $M(k)$ terminar a sua execução. $M(n)_j$ pode significar a execução de um número constante de passos de $M(n)$, não necessariamente um único passo.

□

Esta prova para $L \subset \Sigma^\star$ é ligeiramente mais complexa porque os elementos de $\Sigma^\star$ devem ser ordenados. Isto é feito da seguinte forma: como Σ é finito, pode-se atribuir uma ordem qualquer aos seus elementos (se já não existir uma ordem "natural", como a alfabética). Esta ordem induz uma ordem em $\Sigma^\star$ (veja página 4). Então temos uma enumeração $a_0, a_1, a_2 \ldots$ de $\Sigma^\star$ de acordo com esta ordem. A prova daqui em diante é muito semelhante à dada acima.

Proposição 3.0.4. *Um conjunto $A \subset \mathbb{N}$ (ou linguagem $L \neq \emptyset$) é computável se e somente se existe uma máquina de Turing que enumera os elementos de A (ou L) em ordem crescente.*

Demonstração. ($\Longleftarrow$) Seja M' a MT que enumera os elementos de A em ordem crescente. Construiremos uma M que decide se $x \in A$ ou não, sendo x a entrada para M.

M simula a execução de M' até que esta produza um elemento maior do que x, a entrada para M. Se x tiver sido enumerado, M escreve 1 na saída, vai para o estado q_s e pára. Senão M escreve 0, vai para o estado q_n e pára.

($\Longrightarrow$) Suponha A computável. Então existe uma MT M que decide A; isto é:

$$\begin{aligned} x \in A &\iff M(x) = 1 \\ x \notin A &\iff M(x) = 0 \end{aligned}$$

Construiremos uma MT M' de duas fitas que enumera A. M' simula M na primeira fita com as entradas $0, 1, 2, \ldots$. Para cada simulação, M' imprime a entrada na segunda fita se M retorna 1 (e portanto pertence a A). Todos os elementos de A serão enumerados na segunda fita.

Na linguagem C, assumindo que existe uma função M que decide A, o enumerador de elementos de A seria feito pela seguinte função:

```
void enumere() {
    int n = 0;
    while ( true ) {
       if ( M(n) == 1 )
           printf("%d \n", n);
       ++n;
    }
}
```

□

Dada uma máquina de Turing $M = (Q, \Sigma, I, q)$, o conjunto Σ é sempre finito, assim como é o conjunto Σ utilizado para definir as linguagens $L \subset \Sigma^\star$.

A cada MT M de decisão podemos associar a linguagem L decidida por ela, $L = L(M)$. E poderemos associar a cada linguagem de um alfabeto qualquer Σ uma MT que a decide? Isto é, o conjunto de todas as máquinas de Turing de decisão é equipotente ao conjunto das linguagens sobre Σ? A resposta é não. Veja a próxima proposição.

Proposição 3.0.5. *Dado $\Sigma \neq \emptyset$, há uma linguagem $L \subset \Sigma^\star$ que não é decidida por nenhuma máquina de Turing. Equivalentemente, há linguagem L tal que não existe MT M tal que $L = L(M)$.*

Demonstração. O conjunto T de todas as máquinas de Turing é infinito pois podemos sempre acrescentar uma instrução qualquer a ela (mesmo que não tenha utilidade, como $(q_{n+1}, 0, q_{n+1}, 0, 0)$, na qual q_i, $1 \leqslant n \leqslant n$ eram os estados na máquina original).

Na Seção 2.8 foi utilizada uma codificação que converte uma MT em um número inteiro. Sendo T_N o conjunto dos números inteiros correspondentes a máquinas de Turing, temos que T_N é um subconjunto infinito de $\mathbb{N}$, Portanto T_N é enumerável pela Proposição A.1.6. Esta fato é fácil de ser compreendido: basta tomar todos números que correspondem às MT e colocá-los em ordem crescente.[1] Então o primeiro número corresponderá a 0, o segundo a 1 e assim por diante.

Uma linguagem sobre Σ é um subconjunto de $\Sigma^\star$, pela definição de linguagem. O conjunto de todas as linguagens sobre Σ é

$$\{L : L \subset \Sigma^\star\}$$

que é o conjunto de subconjuntos de $\Sigma^\star$, que é $2^{\Sigma^\star}$. Este conjunto é equipotente a $\mathcal{P}(\mathbb{N})$ pela Proposição 1.0.3, que não é enumerável. Então

$$\begin{aligned} 2^{\Sigma^\star} &\sim \mathcal{P}(\mathbb{N}) \\ T_N &\sim \mathbb{N} \end{aligned}$$

Como $\mathcal{P}(\mathbb{N})$ não é equipotente a $\mathbb{N}$, T_N não é equipotente a $2^{\Sigma^\star}$ também.

Mas existe uma função injetora entre $\mathbb{N}$ e $\mathcal{P}(\mathbb{N})$, a saber, $f(n) = \{n\}$. Portanto $\mathbb{N} \preceq \mathcal{P}(\mathbb{N})$. Como $T_N \sim \mathbb{N}$, $T_N \preceq \mathcal{P}(\mathbb{N})$. Podemos concluir que há mais linguagens (ou subconjuntos de $\mathbb{N}$) do que máquinas de Turing disponíveis para decidi-las ou enumerá-las.

□

Sendo *CLC* o Conjunto das Linguagens Computáveis e *CLCE* o Conjunto das Linguagens c.e., temos que $CLC \subset CLCE$ pela Proposição 3.0.1. Mas haverá alguma linguagem *CLCE* que não seja recursiva? A resposta é sim, como garantido pela próxima proposição.

Lembre-se de que $M(x) \downarrow$ significa que M termina a sua execução quando a entrada for x. Usamos $\langle M \rangle \sqcup x$ para codificar uma MT M com a entrada x, ambas em binário — a mesma codificação empregada na Máquina de Turing Universal (MTU) da Seção 2.8.

Proposição 3.0.6. *A linguagem $H = \{\langle M \rangle 2x : M(x) \downarrow\}$ pertence a $CLCE - CLC$.*

Demonstração. A linguagem H consiste de números $\langle M \rangle 2x$ tais que M pára a sua execução quando a entrada for x. A codificação de M, $\langle M \rangle$, foi definida na Seção 2.8 sobre Máquinas de Turing Universais. É um número natural. Então H é um subconjunto de $\mathbb{N}$, $H \subset \mathbb{N}$. Portanto, $H \in \mathcal{P}(\mathbb{N})$.

Trocando-se o 2 utilizado na codificação de $\langle M \rangle$ por $\sqcup$, obtemos uma cadeia sobre o alfabeto $\Sigma = \{0, 1, \sqcup\}$. Da mesma forma, na definição de H podemos trocar o 2 por $\sqcup$, obtendo

$$H = \{\langle M \rangle \sqcup x : M(x) \downarrow\}$$

Então $\langle M \rangle \sqcup x \in \Sigma^\star$.

[1] É possível fazer isto, pois existe um algoritmo que toma um número e faz a análise sintática dele dizendo que ele corresponde a uma MT ou não.

Como já foi dito anteriormente, em Computabilidade podemos estudar tanto os subconjuntos de $\mathbb{N}$ (primeiro caso) como os subconjuntos de algum conjunto $\Sigma^\star$. Tanto faz.

Veremos agora a primeira parte da prova desta proposição, que $H \in CLCE$. Isto que, provaremos que H é computacionamente enumerável.

Pela Proposição 3.0.2, para provar que $H \in CLCE$, basta encontrar uma MT R que, dada a entrada $\langle M \rangle \sqcup x$, dê como saída 1 se $\langle M \rangle \sqcup x \in H$ (ou seja, $M(x) \downarrow$, $M(x)$ termina a execução) ou não pare se $\langle M \rangle \sqcup x \notin H$ (ou seja, $M(x) \uparrow$, $M(x)$ não termina a execução). Esta máquina R é a MTU com uma modificação: ao terminar de simular M com entrada x (se terminar), R imprime 1 na fita de saída e vai para o estado q_s. Se $M(x)$ não terminar a execução, R (que é uma MTU modificada) não terminará a execução também, que é o comportamento esperado de uma MT que aceita uma linguagem c.e.

Provamos então que $H \in CLCE$. Falta provar que $H \notin CLC$; isto é, que H não é computável ou decidível. Ou seja, provaremos que não pode existir uma máquina de Turing que toma $\langle M \rangle \sqcup x$ como entrada e retorna 1 de $M(x) \downarrow$ ou 0 se $M(x) \uparrow$. Em resumo, provaremos que não pode existir uma MT que toma a codificação de uma MT M e um x como entrada e diz se M irá parar ou não com a entrada x.

Provaremos por contradição. Assumiremos que exista uma MT P que decida H e chegaremos a uma contradição. Isto é, suponha que $P(\langle M \rangle \sqcup x)$ retorna 1 se M pára a sua execução com entrada x ou retorna 0 se $M(x) \uparrow$. Máquinas de Turing que decidem linguagens sempre param. Então P sempre pára a sua execução.

Usando P, podemos construir uma MT Q que toma uma entrada $\langle M \rangle$ e faz o seguinte:

(a) a partir da entrada $\langle M \rangle$, produz $\langle M \rangle \sqcup \langle M \rangle$;

(b) P é simulado passando-se $\langle M \rangle \sqcup \langle M \rangle$ como entrada. A simulação pode ser feita colocando-se as instruções de P em Q ou simulando-se P como é feito em uma MT Universal;

(c) se P retornar 1, isto significa que $\langle M \rangle \sqcup \langle M \rangle \in H$. Ou seja, a MT M pára a sua execução com a entrada $\langle M \rangle$. Neste caso, Q entra em um laço infinito, nunca parando a sua execução;

(d) se P retornar 0, isto significa que $\langle M \rangle \sqcup \langle M \rangle \notin H$. Ou seja, a MT M não pára a sua execução com a entrada $\langle M \rangle$. Neste caso, Q pára a sua execução.

Note que Q faz o contrário do que a máquina M com entrada $\langle M \rangle$ faz. Se esta última pára, Q não pára. Se esta última não pára, Q pára.

Sendo Q uma MT, podemos obter a sua codificação $\langle Q \rangle$. O que acontece com a execução de $Q(\langle Q \rangle)$?

Se $Q(\langle Q \rangle)$ pára a sua execução é porque, pela descrição de Q, $Q(\langle Q \rangle)$ não pára a execução (laço infinito).

Se $Q(\langle Q \rangle)$ não pára a sua execução é porque, pela descrição de Q, $Q(\langle Q \rangle)$ pára a execução (laço infinito). Resumindo:

$$\text{se} \begin{cases} Q(\langle Q \rangle) \downarrow & \text{então a descrição de } Q \text{ diz que } \quad Q(\langle Q \rangle) \uparrow \\ Q(\langle Q \rangle) \uparrow & \text{então a descrição de } Q \text{ diz que } \quad Q(\langle Q \rangle) \downarrow \end{cases}$$

Chegamos a uma contradição. O único passo errado desta prova é a suposição da existência de P. Então P não pode existir. Nenhuma MT pode decidir H. Portanto, H não é computável e $H \notin CLC$.

A contradição só apareceu porque usamos `Q` de duas formas diferentes mas equivalentes. Em $Q(\langle Q \rangle)$, o primeiro Q é uma MT e o segundo é a descrição de uma MT. O primeiro é algo a ser executado, dinâmico e o segundo Q é uma *descrição*, estática, um número apenas. Contudo, na simulação de P feita dentro de Q, a entrada $\langle Q \rangle$, um número, é interpretada como uma MT. Esta dualidade de um número ser ambos uma entrada (número) e uma MT é que impossibilita a existência de Q.

Como $H \in CLCE$ e $H \notin CLC$, então $H \in CLCE - CLC$. □

Usualmente imagina-se P como se ele fosse uma MTU que simulasse M com entrada x. Mas não precisa ser assim. P poderia fazer uma análise estática nas instruções de M e deduzir o seu comportamento com a entrada x. Mas como vimos, isto é impossível.

Abaixo é dada uma prova alternativa de que $H \in CLCE$.

Proposição 3.0.7. *O conjunto $H = \{\langle M\rangle 2x : M(x)\downarrow\}$ é computacionalmente enumerável.*

Demonstração. Construiremos uma MT R com várias fitas que enumera as codificações de todas as máquinas de Turing M e entradas nas quais $M(x)\downarrow$. R enumera na primeira fita os números naturais e, para cada $n \in \mathbb{N}$, verifica, usando a segunda fita, se n é a codificação de uma MT seguida de 2 seguida de um número binário. Esta verificação é análoga a uma análise sintática feita por um compilador. Não demostraremos formalmente que uma MT pode fazer esta verificação.

Então, para cada $n \in \mathbb{N}$, R verifica se $n = \langle M\rangle 2x$ para alguma MT M. Se for, R inicia uma simulação de $M(x)$, passo a passo, na terceira fita como é explicado na Proposição 3.0.3. Isto é, a cada número inteiro que corresponde a uma MT M com uma entrada x, R coloca esta máquina com esta entrada em uma lista de máquinas que estão sendo executadas em "paralelo" na fita três. À medida que as máquinas terminam a execução (se terminam), os seus números são colocados na quarta fita.

□

Será a linguagem H^c computável (decidível)? Não pode ser pois, se fosse, H^c também seria decidível (veja exercício a respeito). Será H^c computacionalmente enumerável?

Proposição 3.0.8. *A linguagem H^c não é computacionalmente enumerável.*

Demonstração. Provaremos por contradição. Suponha que H^c seja computacionalmente enumerável; isto é, $H \in CLCE$. Então pela Proposição 3.0.2, ambas H^c e o seu complemento, $(H^c)^c = H$ são computáveis. Mas a Proposição anterior garante que H não é computável. Logo H^c não é computacionalmente enumerável; isto é, $H \notin CLCE$. □

Então temos pelo menos três classificações de linguagens: computáveis, computacionalmente enumeráveis e uma outra que chamaremos de $Co - CLCE$. Mas estas não são as únicas classes de linguagens. Existem infinitas delas e estas compõem uma hierarquia chamada de Hierarquia Aritmética de classes de complexidade crescente. Como cada linguagem corresponde univocamente a um número real (ou subconjunto de $\mathbb{N}$), classes de linguagens correspondem a conjuntos de números reais (conjuntos de subconjuntos de $\mathbb{N}$) e a Hierarquia Aritmética é uma hierarquia de conjuntos de reais que complexidade crescente.

Teorema 3.0.1. *Seja M uma MT que computa uma função binária $f : \mathbb{N}\times\mathbb{N}\longrightarrow\mathbb{N}$. Isto é, M toma dois inteiros como entrada e produz um inteiro como saída. Então existe uma MT R que implementa a função $r : \mathbb{N}\longrightarrow\mathbb{N}$ tal que $r(x) = f(\langle M\rangle, x)$.*

Infelizmente não será dada a prova deste interessantíssimo teorema.

O que este teorema diz é que qualquer MT pode ser modificada para ter acesso ao seu próprio código. Então esta MT modificada, que é R, não só pode simular o seu próprio código, mas também pode saber quantos estados ela tem, qual alfabeto ela usa, quantas são as suas instruções etc.

É possível fazer um programa em qualquer linguagem de programação que imprime o seu próprio código. Por exemplo, em C:

```
char*f="char*f=%c%s%c;main()
{printf(f,34,f,34,10);}%c";
main(){printf(f,34,f,34,10);}
```

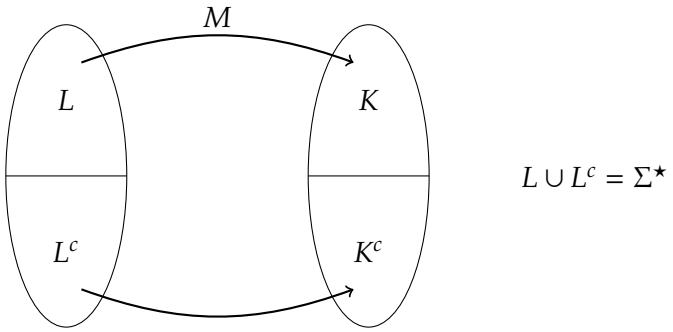


Figura 3.1: m-redução entre L e K

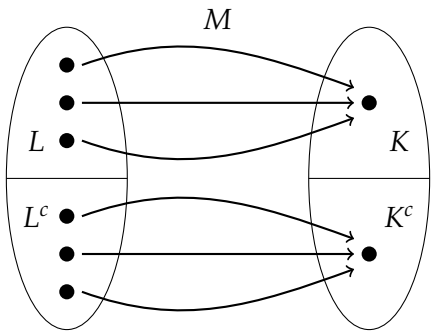


Figura 3.2: Exemplo de m-redução entre L e K que utiliza apenas um elemento de K e um de K^c

Teorema 3.0.2. *Seja $f : \mathbb{N} \longrightarrow \mathbb{N}$ uma função computável (existe uma MT que a computa). Então existe uma MT M tal que $f(\langle M \rangle)$ é a codificação de uma MT R tal que R é equivalente a M.*

Isto é, $\langle R \rangle = f(\langle M \rangle)$ e, para cada entrada x, o valor produzido por $M(x)$ é igual ao de $R(x)$ ou ambas as máquinas não param a sua execução.

Um vírus de computador pode ser considerado uma função f que modifica outros programas. Por este teorema, para cada vírus existe um programa que ele modifica mas que continua a fazer exatamente o que ele fazia antes.

Definição 3.0.4. *(Opcional) Dadas linguagens $L, K \subset \Sigma^\star$, dizemos que L é m-redutível à K se existe uma MT M tal que, para todo $x \in \Sigma^\star$,*

$$x \in L \text{ sse } M(x) \in K$$

M é uma m-redução entre L e K. Usamos a notação $L \leqslant_m K$.

Pela definição de m-redudibilidade,

$$\begin{array}{rcl} x \in L & \Longrightarrow & M(x) \in K \\ x \in L^c & \Longrightarrow & M(x) \in K^c \end{array}$$

Veja a Figura 3.1. Note que a função implementada por M não precisa ser sobrejetora. A MT M pode mapear todos os elementos de L em um único elemento de K. E todos os elementos de $L^c = \Sigma^\star - L$ em um único elemento de $K^c = \Sigma^\star - K$ — veja a Figura 3.2.

Proposição 3.0.9. *(Opcional) Se $L \leqslant_m K$ e L não é decidível, K também não é decidível. Ou equivalentemente, se K é decidível, L é decidível.*

Demonstração. Suponha que $L \leqslant_{\mathrm{m}} K$, L não é decidível, K é decidível. Então a partir do algoritmo de decisão para K podemos construir um algoritmo de decisão para L. Se a m-redução entre L e K for dado pela MT M, para descobrir se $x \in L$ basta testar se $M(x) \in K$. Então se tivermos $L \leqslant_{\mathrm{m}} K$ e K for decidível, L também o será. □

Mostraremos que uma linguagem não é recursiva utilizando a Proposição acima.

Proposição 3.0.10. *(Opcional) A linguagem $HZ = \{\langle M\rangle : M(0)\downarrow\}$ não é recursiva.*

Demonstração. $\langle M\rangle \in HZ$ se M pára a sua execução se a entrada for 0. Reduziremos H a HZ. Se conseguirmos um algoritmo para HZ, teremos um algoritmo para o problema da parada. Absurdo.

Construiremos uma MT P que é uma m-redução de H a HZ. Dado $\langle M\rangle \sqcup x \in H$, P constrói uma MT M' que simula a execução de M com entrada x. A entrada de M' é ignorada por esta máquina. Então

$$\langle M\rangle \sqcup x \in H \text{ sse } \langle M'\rangle \in HZ$$

pois $\langle M\rangle \sqcup x \in H$ sse $M(x)\downarrow$ sse M' pára a sua execução com qualquer entrada sse M' pára a sua execução com entrada zero (já que M' ignora a sua entrada, qualquer entrada e zero produzem o mesmo resultado). Note que a redução aqui é uma MT que toma MT como entrada e produz uma MT. □

Proposição 3.0.11. *(Opcional) Toda linguagem computacionalmente enumerável é redutível a H.*

Demonstração. Seja L uma linguagem c.e. Então existe uma MT P tal que

se $x \in L$ então $P(x) = 1$
se $x \notin L$ então $P(x)\uparrow$

Sempre $x \in L$ $P(x)$ termina a sua execução. Então usaremos H para verificar se $P(x)$ termina a sua execução. A redução de L a H é uma MT M que converte $x \in L$ em $P \sqcup x$. Claramente,

$$x \in L \text{ sse } P(x)\downarrow \text{ sse } P \sqcup x \in H$$

□

Resumo das definições

Linguagem computável Linguagem recursiva Linguagem decidível	Existe uma MT que sempre pára e que retorna 1 ou 0 se a entrada pertence ou não à linguagem
Linguagem computacionalmente enumerável Linguagem recursivamente enumerável	Existe uma MT que retorna 1 se a entrada pertence e que não pára se não pertence

Capítulo 4

Complexidade de Computação

Definição 4.0.5. *A complexidade em tempo de uma MT M é uma função de $n = |x|$ que retorna o máximo número de passos que ela leva do início até a parada com uma entrada de tamanho n. Assume-se que M sempre pára a sua execução para qualquer entrada x. Se M não pára para alguma entrada, qualquer delas, esta definição não se aplica.*

Esta definição se aplica para MT com qualquer número de fitas. Se a complexidade em tempo de uma MT M for $f(n)$, dizemos que M **executa em tempo** $f(n)$. Isto é, o número de passos que M leva até parar com a entrada x é $\leqslant f(|x|)$.

Fixada uma MT M, o número de passos que ela leva do início da execução até à parada varia não só de entrada para entrada mas também varia entre entradas do mesmo tamanho. Por exemplo, M pode levar cinco passos para a entrada 101 e 500 para 111, sendo que estas entradas têm o mesmo tamanho 3.

Mais especificamente, seja $t_M(x)$ uma função $t_M : \Sigma^\star \longrightarrow \mathbb{N}$ que retorna o número de passos que a MT M toma para parar com uma entrada x. A complexidade em tempo para M é uma função $T_M : \mathbb{N} \longrightarrow \mathbb{N}$ com parâmetro n que retorna o máximo número de passos que M levará para uma entrada de tamanho n. Isto é:

$$T_M(n) = \text{máx}\{t_M(x) : |x| = n\}$$

Definição 4.0.6. *A complexidade em espaço de uma MT M com uma fita é uma função de $n = |x|$ que retorna o máximo número de células que ela utiliza do início até a parada com uma entrada de tamanho n. Assume-se que M sempre pára a sua execução para qualquer entrada x. Se M não pára para alguma entrada, qualquer delas, esta definição não se aplica.*

Se a complexidade em espaço de uma MT M for $s(n)$, dizemos que M **executa em espaço** $s(n)$. Isto é, o número de células que M utiliza até parar com a entrada x é $\leqslant s(|x|)$.

Esta definição emprega o conceito de "célula utilizada por uma MT" que não foi definido. Faremos isto agora: uma certa célula C da fita de uma MT M é *utilizada* em uma computação $M(x)$ se a célula C for a célula corrente durante a computação $M(x)$.

Uma definição mais precisa de complexidade em espaço é a seguinte: seja $s_M(x)$ uma função $s_M : \Sigma^\star \longrightarrow \mathbb{N}$ que retorna o número de células que a MT M utiliza com uma entrada x. A complexidade em espaço para M é uma função $S_M : \mathbb{N} \longrightarrow \mathbb{N}$ com parâmetro n que retorna o máximo número de células que M utilizará para uma entrada de tamanho n. Isto é:

$$S_M(n) = \text{máx}\{s_M(x) : |x| = n\}$$

A complexidade em espaço para MT com várias fitas leva em consideração todas as células de todas as fitas. Poderíamos ter utilizado a fita que utiliza *mais* células durante a computação. Se a fita que "toca" mais células em entradas de tamanho n utiliza $f(n)$ células, então a MT utiliza no máximo $kf(n)$ células considerando todas as k fitas. Esta constante não altera os principais resultados de complexidade sendo usualmente desprezada.

Em alguns casos utilizaremos a notação O para descrever a complexidade de uma MT. Esta notação desconsidera detalhes como constantes multiplicadas ou somadas à função de complexidade.

$O(f(n))$ é o conjunto de funções de $\mathbb{N}$ em $\mathbb{N}$ tal que $g \in O(f(n))$ se existem constantes $c, N \in \mathbb{N}$ tal que para $n > N$, temos $g(n) \leqslant cf(n)$. Isto significa que, se desenharmos os gráficos de f e de uma $g \in O(f(n))$, então depois de um certo N, derivada de f será maior ou igual à derivada de g. Isto é, para certa constante c, o gráfico de $cf(n)$ estará acima do gráfico de $g(n)$. Para números n muito grandes teremos $g(n) \leqslant cf(n)$. O símbolo O é lido como "grande O" ou simplesmente "O".

A notação $O(f(n))$ indica o **comportamento assintótico** da função $f(n)$. Isto é, o comportamento da função $f(n)$ quando n assume valores muito grandes.

Como abreviação, usamos as seguintes notações para $g(n) \in O(f(n))$:

- $g(n)$ é $O(f(n))$
- $g(n) = O(n^2)$

Esta última notação é a mais comum. Note que, quando utilizamos a notação $g(n) = O(f(n))$, o símbolo = não é o igual da Matemática. A notação apenas indica que g e f obedecem à definição de O. Em particular, não usamos $O(f(n)) = g(n)$.

Exemplo 4.0.1. *Para $n > 2$, $n < n^2$. Logo*

$$n \in O(n^2) \text{ ou } n \text{ é } O(n^2) \text{ ou } n = O(n^2)$$

Note que "n" em "$n \in O(n^2)$" refere-se à função $f(n) = n$ e "n^2" à função $g(n) = n^2$.

Proposição 4.0.12. *Alguns fatos importantes sobre a notação O são dados abaixo.*

(a) *se $f(n) = kg(n)$, onde $k \in \mathbb{R}$ é uma constante, então $O(f(n)) = O(g(n))$. Isto é, $f(n) = O(g(n))$ e $g(n) = O(f(n))$. Em particular, tomando $g(n) = 1$ para todo n (função constante), temos $O(k) = O(1)$.*

(b) *se $f(n) = O(s(n))$ e $g(n) = O(r(n))$ então*

- $f(n) + g(n) = O(s(n) + r(n))$
- $f(n) \cdot g(n) = O(s(n) \cdot r(n))$

A prova do item 1 é a seguinte: temos que $f(n) < c_s s(n)$ para todo $n > N_s$ e $g(n) < c_r r(n)$ para todo $n > N_r$. Tomando N como o máximo entre N_s e N_r e c como o máximo entre c_s e c_r, temos que

$$f(n) + g(n) < c_s s(n) + c_r r(n) < cs(n) + cr(n) = c(s(n) + r(n))$$

Logo $f(n) + g(n) = O(s(n) + r(n))$. A prova do segundo item é similar.

(c) *se $g(n) = O(f(n))$, então $O(f(n) + g(n)) = O(f(n))$.*

(d)

$$\log_a b = \frac{\log_c b}{\log_c a}$$

Então

$$\log_a n = \frac{\log_c n}{\log_c a}$$

Ou seja, $\log_a n = k \log_c n$ no qual $k = \log_c a$ é uma constante em relação a n. Por (a), $O(\log_a n) = O(\log_c n)$. Não importa a base do logaritmo. Então usamos sempre log, *sem especificar a base.*

É muito diferente afirmar "M executa em tempo $f(n)$" e "a complexidade de M é $O(f(n))$. No primeiro caso, o número de passos que M executa até parar é $\leqslant f(|x|)$ para todo x. No último caso, o número de passos é $\leqslant cf(|x|)$ **apenas** para x maior do que certo tamanho e para uma constante c que não depende de x.

Usaremos apenas linguagens sobre o conjunto $\{0,1\}$. Isto é, todas as linguagens deste Capítulo são subconjuntos de $\{0,1\}^\star$. Um elemento x de uma linguagem L representa uma instância de um problema. Por exemplo, se L é o conjunto de todas as matrizes invertíveis, um $x \in L$ representa uma matriz invertível. Os elementos de L seriam codificações de matrizes invertíveis em um alfabeto Σ tal que $L \subset \Sigma^\star$. Por exemplo, tomando $\Sigma = \{0,1,\square\}$, uma matriz $n \times m$ $A = (a_{ij})$ poderia ser representada da seguinte forma, onde x_b é o número x em binário

$$n_b \square m_b \square a_{11} \square \ldots \square a_{1m} \square a_{21} \square \ldots \square a_{2m} \square \ldots \square a_{n1} \square \ldots \square a_{nm}$$

Note que há elementos de $\Sigma^\star$ que não representam matrizes (por exemplo, $\square$ ou $3_b \square 4_b$). Estes elementos certamente não pertencem a L. Mas pertenceriam a L^c. Contudo, usaremos a convenção de que tanto a linguagem como o seu complemento contém apenas entradas válidas. Isto é, L^c, neste caso, conteria apenas elementos que são codificações válidas de matrizes. Então tanto L como L^c são subconjuntos de um conjunto $\Sigma^V \subset \Sigma^\star$ que contém todas as codificações válidas de matrizes. Usualmente, dado $x \in \Sigma^\star$, pode-se descobrir se $x \in \Sigma^V$ em tempo polinomial (conceito a ser visto adiante).

4.1 Classes de Linguagens

Nesta Seção definimos as mais importantes classes de complexidade. Usamos c para uma constante que depende apenas da linguagem sendo considerada. E n para $|x|$.

Definição 4.1.1. *Uma linguagem L pertence a TIME(f) se existe uma MT M de decisão que decide L e que toma uma entrada x e que termina a sua execução depois de um número de passos menor ou igual a $cf(n)$ tal que $x \in L$ sse $M(x) = 1$. Ou seja, M executa em tempo $cf(n)$.*

Definição 4.1.2. *Uma linguagem L pertence a SPACE(f) se existe uma MT M que decide L e que termina a sua execução depois de utilizar um número de células menor ou igual a $cf(n)$ nas fitas de trabalho (todas exceto a de entrada e a de saída), sendo $n = |x|$, o tamanho da entrada x . Ou seja, M executa em espaço $cf(n)$.*

Definição 4.1.3. *Uma linguagem L pertence a NTIME(f) se existe uma MTND M que decide L e que termina a sua execução depois de um número de passos menor ou igual a $cf(n)$. Isto é, se $x \in L$, então existe uma sequência de escolhas que resulta em $M(x) = 1$. E a computação termina em um número de passos $\leqslant cf(n)$ qualquer que seja o resultado da máquina.*

Definição 4.1.4. *Uma linguagem L pertence a NSPACE(f) se existe uma MTND com entrada e saída M que decide L e que termina a sua execução depois de utilizar um número de células menor ou igual a $cf(n)$. Não se considera o número de células da primeira e da última fitas (entrada e saída).*

Independente das escolhas não determinísticas feitas pelas máquinas M citadas nas definições de NTIME e NSPACE, o número de passos e o número de células utilizadas é menor ou igual a $cf(n)$.

Definição 4.1.5. *Uma classe de linguagens é um conjunto de linguagens. As mais importantes delas são definidas a seguir.*

(a) $L \in P$ sse existe uma MT M que decide L em tempo polinomial.

$$P = \bigcup_{k \in \mathbb{N}} TIME(n^k)$$

(b) $L \in NP$ sse existe uma MTND M que decide L em tempo polinomial.

$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

(c) $L \in EXP$ sse existe uma MT M que decide L em tempo exponencial.

$$EXP = \bigcup_{k \in \mathbb{N}} TIME(2^{n^k})$$

(d) $L \in PSPACE$ sse existe uma MT M que decide L em espaço polinomial.

$$PSPACE = \bigcup_{k \in \mathbb{N}} SPACE(n^k)$$

(e) $L \in NPSPACE$ sse existe uma MTND M que decide L em espaço polinomial.

$$P = \bigcup_{k \in \mathbb{N}} NSPACE(n^k)$$

Além destas classes, existem os seus complementos. Dada uma classe de linguagens C, a classe coC, o complemento da classe C, é definido como

$$\text{coC} = \{L^c : L \in \text{C}\}$$

Proposição 4.1.1. *$TIME(f(n)) \subset NTIME(f(n))$ e $SPACE(f(n)) \subset NSPACE(f(n))$.*

Demonstração. Esta prova é trivial. Se $L \in \text{TIME}(f(n))$, há uma MT M que decide L em tempo $\leqslant cf(n)$, onde c é uma constante. Toda MTD também é uma MTND na qual há apenas uma escolha em cada passo da computação. Então esta mesma M é uma MTND que decide L em tempo $\leqslant cf(n)$ e portanto $L \in \text{NTIME}(f(n))$. Logo $\text{TIME}(f(n)) \subset \text{NTIME}(f(n))$. Raciocínio análogo se aplica ao espaço. □

4.2 A Classe NP

Esta Seção faz um estudo da classe NP e mostra duas linguagens NP-completas. Uma linguagem NP-completa é uma linguagem que, de certa forma, possui uma complexidade maximal dentre todas as linguagens de NP. Antes de apresentar estas linguagens, daremos três definições da classe NP e provaremos que elas são equivalentes. Depois definimos "redução" entre linguagens, conceito essencial para a definição das linguagens NP-completas.

Assumiremos que todas as linguagens são sobre o alfabeto $\{0, 1\}$. Isto é, as entradas e as saídas das MT utilizam apenas os símbolos 0 e 1, sendo $\sqcup$ utilizado apenas para separar partes da entrada. Mas uma MT que decide uma linguagem nunca utiliza $\sqcup$.

Definição 4.2.1. *(Definição I) $L \in NP$ se e somente se existe uma MTND M que decide L em tempo polinomial. Isto é,*

$$NP = \bigcup_{k \in \mathbb{N}} NTIME(n^k)$$

Definição 4.2.2. *(Definição II) $L \in NP$ se e somente se existe uma MTD M que executa em tempo polinomial (veja definição 2.3.7) e um polinômio $p(n)$ tal que*

$$x \in L \text{ sse existe } y, \left|y\right| \leqslant p(|x|) \text{ tal que } M(x \sqcup y) = 1$$

Note que a cada linguagem está associada uma MTD M e um polinômio p diferentes. O y associado a cada x é chamado de testemunha ou certificado.

Definição 4.2.3. *[9] Uma relação $R \subset \Sigma^\star \times \Sigma^\star$ é decidida polinomialmente se a linguagem $\{x \sqcup y : (x, y) \in R\}$ pode ser decidida em tempo polinomial. R é polinomialmente balanceada se $(x, y) \in R$ implica em $\left|y\right| \leqslant |x|^k$ para $k \in \mathbb{N}$, $k > 0$. A constante k depende apenas de R e não do par (x, y) sendo considerado.*

Definição 4.2.4. *(Definição III) $L \in NP$ se e somente se existe um polinômio balanceado R tal que*

$$L = \{x : (x, y) \in R \text{ para algum } y\}$$

Proposição 4.2.1. *As definições 4.3.2 e 4.3.4 são equivalentes.*

Demonstração. Provaremos que a definição 4.3.4 engloba a 4.3.2. O prova $\Longrightarrow$ é basicamente esta lida em ordem inversa. A definição 4.3.4 emprega um polinômio R decidido polinomialmente para cada linguagem $L \in$ NP. Pela definição, existe uma MT M que decide R em tempo polinomial. Isto é, $(x, y) \in R$ se e somente se $M(x \sqcup y) = 1$. Como R também é balanceado, se $(x, y) \in R$ então o tamanho de y é limitado por $|x|^k$, sendo k uma constante específica para cada R (e portanto para cada L). Então a definição 4.3.4 pode ser reescrita como

$$\begin{aligned} L &= \{x : (x, y) \in R \text{ para algum } y\} \\ &= \{x : \text{ existe um } y \text{ tal que } (x, y) \in R\} \\ &= \{x : \text{ existe um } y \leqslant |x|^k \text{ tal que } M(x \sqcup y) = 1\} \end{aligned}$$

A última fórmula é a definição 4.3.2. Note que, pela definição 4.3.4, se $L \in$ NP, existe R e portanto existe M que o decide (exatamente como na definição 4.3.2). □

Proposição 4.2.2. *As definições 4.3.1 e 4.3.2 são equivalentes.*

Demonstração. ($\Longrightarrow$) Suponha que uma linguagem $L \in$ NP pela definição 4.3.1. Provaremos que $L \in$ NP pela definição 4.3.2. Então existe uma MT M que decide L em tempo polinomial. Isto é, dado qualquer $x \in L$, há pelo menos uma sequência de escolhas não determinísticas que resulta em $M(x) = 1$. Como M é polinomial, o número de escolhas é polinomial. Definimos a relação R da seguinte forma: $(x, y) \in R$ se e somente se y é uma sequência de escolhas da máquina M tal que $M(x) = 1$ ($x \in L$). Então y é polinomial em x e sempre que $x \in L$, existe um y (sequência de escolhas) que resulta em $M(x) = 1$.

($\Longleftarrow$) Suponha que uma linguagem $L \in$ NP pela definição 4.3.2. Provaremos que $L \in$ NP pela definição 4.3.1. Então existe uma MT M que executa em tempo polinomial e um polinômio $p(n)$ tal que $x \in L$ sse existe y, $\left|y\right| \leqslant p(|x|)$ tal que $M(x \sqcup y) = 1$. Construiremos uma MTND M' que decide L em tempo polinomial. Dada a entrada x, M' escolhe aleatoriamente um $y \leqslant p(|x|)$ (em tempo polinomial em x) e retorna o valor de $M(x \sqcup y)$. Pela definição 4.3.2, se $L \in$ NP, há pelo menos um y tal que $M(x \sqcup y) = 1$. Então há pelo menos uma escolha aleatória de y, em M', que faz $M'(x)$ retornar 1 (se $L \in$ NP). A máquina M' executa em tempo polinomial, pois o tamanho da entrada para M é $|x| + \left|y\right| + 1 \leqslant |x|^m$ para algum m (polinomial) e M executa em tempo polinomial (se $f(n)$ e $g(n)$ são polinomiais, $f(g(n))$ é polinomial).

□

Exemplo 4.2.1. *As seguintes linguagens estão em NP:*

(a) $L = \{x \sqcup y \sqcup z : z = xy\}$, *multiplicação de dois números;*

(b) $L = \{x \sqcup y \sqcup z : z = x + y\}$, *soma de dois números;*

(c) $L = \{v \sqcup e :$ *elemento e está no vetor v*$\}$

(d) $L = \{(V, E, x, y) :$ *no grafo $G = (V, E)$, há um caminho do vértice x ao y*$\}$
O certificado de (V, E, x, y) é um caminho em $G = (V, E)$ de x até y;

(e) $SAT = \{\phi : \phi$ *está na FNC e é satisfazível* $\}$
O certificado é uma valoração para as variáveis de ϕ *que tornam* ϕ *verdadeira;*

(f) $H = \{(V,E) :$ *existe um caminho hamiltoniano no grafo* $G = (V,E)\}$ *O certificado de* (V,E) *é um caminho hamiltoniano em* $G = (V,E)$.

Agora veremos a definição de redução entre duas linguagens. A redução de certa forma estabelece uma relação entre as linguagens. Uma linguagem mais fácil pode ser reduzida a uma linguagem mais difícil, no qual "fácil" e "difícil" são relativos ao poder da redução — quanto menos poderosa é a redução, em termos de tempo de execução, mais sentido fazem estas palavras.

Definição 4.2.5. *Dizemos que uma linguagem L é polinomialmente redutível ou Karp-redutível a uma linguagem K se existe uma MT R que executa em tempo polinomial tal que*

$$x \in L \text{ sse } R(x) \in K$$

Usaremos $L \leqslant_P K$ *para "L é polinomialmente redutível a R". A MT R é chamada de redução de L para K.*

Proposição 4.2.3. *Qualquer linguagem* $L \in P$ *é polinomialmente redutível a qualquer outra linguagem* $K \in P$ *se* $K \neq \emptyset$ *e* $K \neq \{0,1\}^\star$.

Demonstração. A redução tem o mesmo "poder" computacional que a decisão de L ou K pois ambas são polinomiais. Então podemos implementar a redução usando a MT que decide L. É isto o que fazemos.

Como $K \neq \emptyset$ e $K \neq \{0,1\}^\star$, existe pelo menos um elemento $a \in K$ e pelo menos um $b \in K^c$. Seja M a MT que decide L. A redução R de L para K é definida como

$$R(x) = \begin{cases} a & \text{se } M(x) = 1 \\ b & \text{se } M(x) = 0 \end{cases}$$

Ou, se preferir, $R(x) = aM(x) + b(1 - M(x))$. □

Proposição 4.2.4. *Se* $L_1 \leqslant_P L_2$ *e* $L_2 \leqslant_P L_3$ *então* $L_1 \leqslant_P L_3$.

Demonstração. Queremos provar que, se R_{12} é uma redução polinomial de L_1 para L_2 e R_{23} é uma redução polinomial de L_2 para L_3, então existe uma redução polinomial R_{13} de L_1 para L_3. Esta redução é simplesmente

$$R_{13}(x) = R_{23}(R_{12}(x))$$

Falta provar que R_{13} executa em tempo polinomial. Se R_{12} e R_{23} executam em tempo $p(n)$ e $t(n)$, respectivamente, então R_{13} executará em tempo $t(p(n))$, que é polinomial, pois composição de polinômio é polinômio. E o tamanho máximo da saída de $R_{12}(x)$ será $p(|x|)$. Em outros termos, se R_{12} e R_{23} utilizam um número de passos $\leqslant c_1 n^{k_1}$ e $\leqslant c_2 n^{k_2}$, respectivamente, então R_{13} utilizará um número de passos $\leqslant c_2(c_1 n^{k_1})^{k_2} = c_2 c_1^{k_2} n^{k_1 k_2}$. Falta provar que $x \in L_1$ sse $x \in L3$, o que é trivial:

$$x \in L_1 \text{ sse } R_{12}(x) \in L_2 \text{ sse } R_{23}(R_{12}) \in L3$$

Logo

$$x \in L_1 \text{ sse } R_{23}(R_{12}) \in L3$$

e $R_{13}(x) = R_{23}(R_{12}(x))$ é uma redução de L_1 para L_3.

□

Depois de definida redução polinomial e dadas as suas principais propriedades, podemos definir NP-completude, o que é feito em partes.

Definição 4.2.6. *Dada uma classe de linguagens C, dizemos que uma linguagem L é C-difícil se toda linguagem $K \in C$ pode ser reduzida a L.*

A redução utilizada depende da classe C. Neste texto, será sempre redução polinomial.

Definição 4.2.7. *Dada uma classe de linguagens C, dizemos que uma linguagem L é C-completa se toda linguagem $K \in C$ pode ser reduzida a L e $L \in C$.*

Em resumo, L é C-completa se L é C-difícil e $L \in C$.

Podemos considerar que uma linguagem L que é C-completa é a mais "complexa", considerando a redução utilizada, dentre todas as linguagens de C. Por exemplo, uma linguagem PSPACE-completa é a linguagem mais "complexa" dentre todas as linguagens que executam em espaço polinomial. Uma linguagem NP-completa é a linguagem mais "complexa" dentre todas as linguagens da classe NP. Neste caso, se for encontrado um algoritmo determinístico polinomial para um problema NP-completo, teremos encontrado um algoritmo determinístico polinomial para decidir cada linguagem de NP. Vejamos porquê.

Suponha que L seja NP-completa decidida por uma MT determinística M em tempo polinomial. A classe é definida em termos do não determinismo, mas nada impede que utilizemos uma MT determinística para decidir uma linguagem da classe. Então $x \in L$ sse $M(x) = 1$. Seja $K \in NP$ e R uma redução de K a L. Então

$$x \in K \text{ sse } R(x) \in L \text{ sse } M(R(x)) = 1$$

Logo a MT determinística M' que decide K é definida como $M'(x) = M(R(x))$. Note que as reduções são sempre determinísticas. M' executa em tempo polinomial pois é uma composição de duas máquinas que executam em tempo polinomial.

Mas existirá uma linguagem NP-completa? A resposta é dada pela próxima Proposição.

Proposição 4.2.5. *([2]) A linguagem*

$$TMSAT = \{(\langle M \rangle, x, 1^n, 1^t) : \exists u \in \{0,1\}^n \textit{ tal que } M(x \sqcup u) = 1 \textit{ em } \leqslant t \textit{ passos } \} \tag{4.1}$$

é NP-completa. M é uma MT determinística.

Mostraremos agora uma linguagem NP-completa mais natural do que TMSAT. A linguagem SAT é definida como

SAT = $\{\phi : \phi$ está na FNC e é satisfazível $\}$

Ou seja, SAT é o conjunto das fórmulas do cálculo proposicional que estão na Forma Normal Conjuntiva e sejam satisfazíveis. Uma fórmula é satisfazível se existe uma atribuição de valores às suas variáveis tal que a fórmula seja verdadeira. Ou seja, se fizermos a tabela verdade da fórmula, há pelo menos uma linha da tabela em que o resultado da fórmula é verdadeira (cada linha contém uma associação diferente de valores para as variáveis).

Teorema 4.2.1. *(Cook [3]) SAT é NP-completa.*

Capítulo 5

Outros Modelos de Computação

O cálculo lambda e funções recursivas [7] são modelos de computação não normalmente utilizados para análise de complexidade. Estes modelos são abstratos e suas descrições não utilizam conceitos mecânicos como "escrever um símbolo", "mover a cabeça de leitura/gravação", para a esquerda, "colocar 1 na posição de memória n" e assim por diante. Estes conceitos físicos envolvem o tempo e o espaço, as duas principais medidas de complexidade. Contudo é possível associar medidas de tempo e espaço ao cálculo lambda e às funções recursivas, embora não de uma forma que poderíamos chamar de óbvia ou natural.

5.1 Família Uniforme de Circuitos

Para apresentar um outro modelo de computação, família uniforme de circuitos, precisamos antes de definir o que é um circuito e como avaliá-lo.

Dado um grafo dirigido $G = (V, E)$, o grau de entrada de um vértice $v \in V$ é o número de arestas incidentes a v, ou seja $|R|$, $R = \{w : (w, v) \in E\}$. O grau de saída de v é $|S|$, no qual $S = \{w : (v, w) \in E\}$. Indicaremos o grau de entrada de v por $g_e(v)$ e o grau de saída por $g_s(v)$.

Definição 5.1.1. *Um circuito é um grafo dirigido acíclico $G = (V, E)$ no qual cada vértice v é de um dos seguintes tipos:*

1. *entrada com $g_e(v) = 0$ e $g_s(v) = 1$;*
2. *saída com $g_e(v) = 1$ e $g_s(v) = 0$;*
3. *verdadeiro ou falso (V ou F) com $g_e(v) = 0$ e $g_s(v) = 1$;*
4. *porta NOT com $g_e(v) = 1$ e $g_s(v) = 1$;*
5. *porta AND ou OR com $g_e(v) = 2$ e $g_s(v) = 1$.*

Os vértices do tipo 1 são associados a variáveis $x_1, x_2, \ldots x_n$, que podem assumir V ou F. Os vértices do tipo 2 são associados a $y_1, y_2, \ldots y_m$. Neste caso há n vértices de entrada e m de saída.

Dados valores V e F para os vértices de entrada (variáveis x_i) o circuito calcula valores para os vértices de saída (y_i). Para fazer isto, é necessário associar valores para todos os outros vértices do circuito (pois assume-se que as saídas dependem de todos os outros vértices). Esta associação de valores é feita por uma função $g : V \longrightarrow \{V, F\}$. Inicialmente, sabemos apenas os valores $g(v)$ para os vértices v associados às variáveis x_i. O valor de g para os outros vértices é calculado como se segue. Ordene os vértices de $G = (V, E)$ na ordem topológica obtendo $v_1, v_2, \ldots v_k$, no qual $k = |V|$. A ordenação topológica sempre pode ser feita porque um circuito é um grafo acíclico. Os valores de $g(v)$ serão calculados nesta ordem, o que garante que, ao calcular $g(v)$, sabemos os valores $g(w)$ para todos os vértices w tal que $(w, v) \in E$. Para cada vértice v_i da lista $v_1, v_2, \ldots v_k$, faça:

1. se v_i é vértice V (verdadeiro), $g(v_i) = V$. Idem para vértice F;
2. se v_i é uma porta NOT, $g(v_i) = \neg g(w)$, no qual $(w, v_i) \in E$. Note que sempre existe w, por definição;
3. se v_i é uma porta AND, $g(v_i) = g(w_1) \wedge g(w_2)$, no qual $(w_1, v_i), (w_2, v_i) \in E$;
4. se v_i é uma porta OR, $g(v_i) = g(w_1) \vee g(w_2)$, no qual $(w_1, v_i), (w_2, v_i) \in E$.
5. se v_i é um vértice de saída y_j, $g(v_i) = g(w)$, no qual $(w, v_i) \in E$.

Um circuito C com n entradas $x_1, x_2, \ldots x_n$ e m saídas $y_1, y_2, \ldots y_m$ calcula uma função

$$f : \{0,1\}^n \longrightarrow \{0,1\}^m$$

na qual 0 e 1 representam F e V, respectivamente. O valor $f(x_1, x_2, \ldots x_n)$ é obtido fornecendo-se valores aos vértices de entrada (associados às variáveis x_i) e executando o algoritmo acima para calcular os valores de g. O valor de f será $y_1 y_2 \ldots y_m$ que é

$$g(v_{y_1}) g(v_{y_2}) \ldots g(v_{y_m})$$

no qual v_{y_j} é o vértice associado à variável de saída y_j.

Uma família uniforme de circuitos C é uma sequência $C_1, C_2, \ldots$ de circuitos C_i de tal forma que:

1. C_i possui i entradas;
2. existe uma função $f : \mathbb{N} \longrightarrow \mathbb{N}$ de tal forma que C_i possui $f(i)$ saídas;
3. existe uma MT M_C que, com entrada i, gera C_i. Isto é, M_C gera uma codificação do i-ésimo circuito.

Usaremos C_i para o i-ésimo circuito da família C. Exige-se que exista uma máquina de Turing que gere cada um dos circuitos. Isto é importante pois de outra forma poderíamos ter uma família de circuitos que decide linguagens não computáveis.

Definição 5.1.2. *Dizemos que uma família C de circuitos decide uma linguagem L se cada circuito C_i possui um único vértice de saída v_s e*

$$x \in L \text{ sse o resultado de } C_{|x|} \text{ com entrada } x \text{ é } V$$

Isto é, usando o circuito $C_{|x|}$, o que toma entradas do tamanho de x, com entrada x obtemos V ou F conforme x pertença a L ou não.

Proposição 5.1.1. *Dada qualquer linguagem $L \subset \Sigma^\star$ há uma família de circuitos que a decide.*

Demonstração. Para cada n, o conjunto L_n de elementos de L de tamanho n é finito e portanto pode ser reconhecido por um circuito C_{nk} que simplesmente testa para conferir se a entrada é cada um dos elementos de L_n. Vejamos como isto pode ser feito. Pode-se codificar um elemento x de $\Sigma^\star$ em uma sequência de valores V e F, o que é essencialmente converter o elemento para binário. Esta é a entrada para o circuito C_{nk} no qual $k = 1 + \lfloor \log_2 |\Sigma| \rfloor$. k é então o número de dígitos binários necessários para codificar um único símbolo de Σ.

O circuito C_{nk} implementa função $f : \{V, F\}^{nk} \longrightarrow \{V, F\}$ descrita da seguinte forma, na qual $s'_1 s'_2 \ldots s'_{nk}$ é a codificação em V e F (binário) do elemento $s_1 s_2 \ldots s_n \in \Sigma^\star$:

$$f(s'_1 s'_2 \ldots s'_{nk}) = \begin{cases} V & \text{se} \quad s_1 s_2 \ldots s_n \in L_n \\ F & \text{caso contrário} \end{cases}$$

Pode-se facilmente construir uma tabela verdade a partir desta função e, a partir desta, a fórmula na FNC que a produz e o circuito equivalente. □

Apêndice A

Definições Matemáticas Básicas

Neste capítulo definimos alguns termos, proposições e teoremas que serão utilizados neste texto.

A.1 Teoria dos Conjuntos

Intuitivamente, um conjunto é uma coleção de objetos A com uma relação $\in$ na qual $x \in A$ se x é um objeto pertencente à coleção de objetos A. Um conjunto não possui elementos repetidos. Usamos $B \subset A$ para B subconjunto de A: A contém todos os elementos que B possui. Então, em particular, $A \subset A$ para todo conjunto A. Se $B \subset A$ e $B \neq A$, dizemos que B é um *subconjunto próprio* de A.

Usaremos $\emptyset$ para o conjunto vazio, o conjunto tal que $\forall x \neg(x \in \emptyset)$. Temos que $\emptyset \subset A$ para todo conjunto A e se $A \subset \emptyset$ então $A = \emptyset$.

Definição A.1.1. *O conjunto das partes de um conjunto A é denotado por $\mathcal{P}(A)$ ou 2^A e é definido como o conjunto contendo todos os subconjuntos de A. Então*

$$B \in 2^A \text{ sse } B \subset A$$

Definição A.1.2. *As operações mais importantes entre conjuntos são: união, interseção, diferença, denotados por $\cup$, $\cap$ e $-$. Estas operações são definidadas como*
$A \cup B = \{x : x \in A \text{ ou } x \in B\}$
$A \cap B = \{x : x \in A \text{ e } x \in B\}$
$A - B = \{x : x \in A \text{ e } x \notin B\}$
A operação de união dos elementos de um conjunto é definida como

$$\bigcup S = \{x : x \in A \text{ e } A \in S\}$$

As operações de união e interseção são comutativas, associativas e a união se distribui sobre a interseção (e vice-versa). Então $A \cup (B \cup C) = (A \cup B) \cup C$, $A \cup B = B \cup A$ e $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$. Temos fórmulas similares para $\cap$.

Definição A.1.3. *Um par ordenado com elementos a e b é denotado por (a, b) e representa uma lista com estes dois elementos.*

Um par ordenado pode ser representado em forma de conjunto: $(a, b) = \{\{a\}, \{a, b\}\}$. Desta forma pode-se distinguir quem é o primeiro e quem é o segundo elementos. Note que $(a, b) = (c, d)$ sse $a = c$ e $b = d$. A noção de par ordenado pode ser generalizado para qualquer n: uma n-tupla é uma lista ordenada $(a_1, a_2, \ldots a_n)$ de elementos.

Definição A.1.4. *O produto cartesiano dos conjuntos A_1, A_2, ..., A_n é definido como*

$$A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots a_n) : a_i \in A_i \text{ para } 1 \leqslant i \leqslant n\}$$

Definição A.1.5. *Uma relação R n-ária é qualquer subconjunto de um conjunto $A_1 \times A_2 \times \ldots \times A_n$. Em particular, se $R \subset A$, então R é uma relação unária sobre A.*

Para relações binárias, pode-se usar $a\ R\ b$ para $(a,b) \in R$. Por exemplo, $1 < 2$ ao invés $< (1,2)$.

Definição A.1.6. *Uma relação $f \subset A \times B$ é chamada de função se todo elemento de A estiver relacionado a pelo menos um elemento de B e, para todo $x \in A$ e $y, z \in B$, tivermos $(x,y) \in f$ e $(x,z) \in f$, então $y = z$. Escrevemos $f : A \longrightarrow B$ para uma função $f \subset A \times B$. Se $(a,b) \in f$, então usamos $f(a)$ para b. O conjunto A é chamado de domínio ($dom(f) = A$) e B de contra-domínio de f ($codom(f) = B$). A imagem de f, denotada por $Im(f)$, é definida como*

$$Im(f) = \{b : \text{ existe } a \in A \text{ tal que } b = f(a)\}$$

f será chamada de injetora se $f(x) = f(y)$ implicar em $x = y$. f será chamada de sobrejetora se $Im(f) = B$. Uma função é bijetora se ela é injetora e sobrejetora.

Usaremos as funções especiais $\lceil . \rceil : \mathbb{R} \longrightarrow \mathbb{Z}$ e $\lfloor . \rfloor : \mathbb{R} \longrightarrow \mathbb{Z}$ definidas como o menor inteiro maior ou igual a x e o maior inteiro menor ou igual a x, respectivamente. Ou seja,

$$\lceil x \rceil = min\{n : n \in \mathbb{Z} \text{ e } n \geqslant x\}$$

$$\lfloor x \rfloor = max\{n : n \in \mathbb{Z} \text{ e } n \leqslant x\}$$

Usaremos (a,b) para o conjunto $\{x : x \in \mathbb{R} \text{ e } a < x \text{ e } x < b\}$. Assume-se $a < b$. Usaremos $[a,b]$ para o conjunto $\{x : x \in \mathbb{R} \text{ e } a \leqslant x \text{ e } x \leqslant b\}$. Também se assume $a < b$. Os conjuntos $[a,b)$ e $(a,b]$ são definidos similarmente. O leitor poderá saber pelo contexto se usamos (a,b) para um intervalo real ou para um par ordenado.

Definição A.1.7. *Usamos a notação $A \preceq B$ entre conjuntos A e B para denotar a existência de um função bijetora $f : A \longrightarrow C$ no qual $C \subset B$. Equivalentemente, $A \preceq B$ se existir uma função injetora $g : A \longrightarrow B$.*

Definição A.1.8. *Se existir uma função bijetora $f : A \longrightarrow B$ dizemos que o conjunto A é* **equipotente** *ou* **equipolente** *ao conjunto B. Neste caso usamos a notação $A \sim B$.*

Se $A \sim B$, há uma função bijetora $f : A \longrightarrow B$ e portanto $A \preceq B$ e $B \preceq A$ (toda bijetora é injetora e possui inversa bijetora). O próximo teorema garante que o contrário também é verdadeiro.

Teorema A.1.1. *(Cantor-Schröder-Bernstein) Se $A \preceq B$ e $B \preceq A$, então $A \sim B$.*

Não será feita a prova deste teorema, que está além dos limites deste curso.

Proposição A.1.1. *A relação $\sim$ de equipotência é uma relação de equivalência.*

Demonstração. Dados os conjuntos A, B e C, $A \sim A$ por $f(x) = x$ bijetora. Se $A \sim B$, existe $f : A \longrightarrow B$ bijetora. Portanto existe $f^{-1} : B \longrightarrow A$ bijetora e $B \sim A$ (inversa de bijetora é bijetora). Se $A \sim B$ e $B \sim C$, existem funções $f : A \longrightarrow B$ e $g : B \longrightarrow C$ bijetoras. Logo existe $g \circ f : A \longrightarrow C$ bijetora (pois composição de bijetoras é bijetora) e $A \sim C$. □

Proposição A.1.2. *Se $A \sim B$, então $2^A \sim 2^B$.*

Demonstração. Como $A \sim B$, existe $f : A \longrightarrow B$ bijetora. Construiremos $g : 2^A \longrightarrow 2^B$ bijetora. g toma um elemento de 2^A (um subconjunto de A) e retorna um elemento de 2^B (subconjunto de B) que contém as imagens $f(x)$ dos elementos de A — veja a Figura A.1.

Então dado $C \in 2^A$, $g(C) = \{y : y = f(x) \text{ para algum } x \in C\} = f(C)$. A função g é injetora, pois se $C \neq D$, C tem um elemento que D não possui (ou vice-versa. Assumiremos que existe $z \in C$ e $z \notin D$). Então $f(z) \notin f(D)$. Como $f(z) \in f(C)$ por definição, $g(C) \neq g(D)$. Logo g é injetora. Esta função é também sobrejetora pois dado $C \in 2^B$, o contradomínio de g, temos que $f(D) = C$ no qual

$$D = \{f^{-1}(z) : z \in C\}$$

□

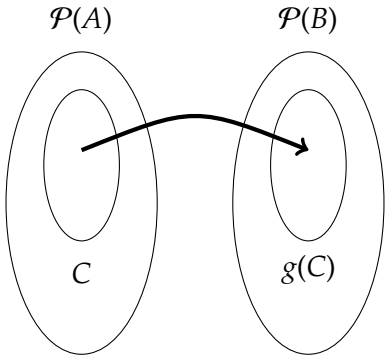


Figura A.1: Função entre 2^A e 2^B bijetora

Proposição A.1.3. $(-1, 1) \sim \mathbb{R}$

Demonstração. A função $f : (-1, 1) \longrightarrow \mathbb{R}$ definida como

$$f(x) = \begin{cases} 0 & \text{se } x = 0 \\ \frac{1-|x|}{x} & \text{se } x \neq 0 \end{cases}$$

é bijetora. Logo $(-1, 1) \sim \mathbb{R}$. □

Proposição A.1.4. *Dados dois intervalos reais quaisquer* (a, b) *e* (c, d), $(a, b) \sim (c, d)$ *e* $(a, b) \sim \mathbb{R}$.

Demonstração. A função $g : (a, b) \longrightarrow (c, d)$ definida como

$$g(x) = c + (x - a)\frac{d - c}{b - a}$$

é bijetora. Logo quaisquer dois intervalos reais abertos são equipotentes.

Temos $(a, b) \sim (-1, 1)$ pela função g e $(-1, 1) \sim \mathbb{R}$ pela função f da Proposição A.1.3. Logo, por composição de funções, temos $(a, b) \sim \mathbb{R}$ para qualquer intervalo real (a, b) — a relação $\sim$ é de equivalência, usamos transitividade neste caso. □

Um intervalo fechado também pode ser equipotente a um intervalo aberto, embora não por uma função contínua.

Proposição A.1.5. $[0, 1) \sim (0, 1)$, $[0, 1] \sim (0, 1)$ *e* $[0, 1] \sim \mathbb{R}$.

Demonstração. A função $f : [0, 1) \longrightarrow (0, 1)$ definida como

$$f(x) = \begin{cases} \frac{1}{2} & \text{se } x = 0 \\ \frac{1}{2^{k+1}} & \text{se } x = \frac{1}{2^k}, k \in \{1, 2, 3, \ldots\} \\ x & \text{se } x \notin \{0, \frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \ldots\} \end{cases}$$

é bijetora. Então $[0, 1) \sim (0, 1)$. Aplicando a mesma técnica uma outra vez podemos provar que $[0, 1] \sim (0, 1)$. Logo $[0, 1) \sim \mathbb{R}$ e $[0, 1] \sim \mathbb{R}$. □

Definição A.1.9. *Um conjunto finito ou equipotente a* $\mathbb{N}$ *é chamado de* **enumerável**. *Um conjunto equipotente a* $\mathbb{N}$ *é chamado de* **denumerável**.

Se o conjunto A é enumerável e finito, podemos listar os seus elementos: $a_0, a_1, \ldots a_n$, sendo n o tamanho de A. Se A é enumerável e infinito, existe uma função bijetora $f : \mathbb{N} \longrightarrow A$ e podemos nomear a_i como $f(i)$. Isto é, $a_0 = f(0)$, $a_1 = f(1)$ e assim por diante. Então os elementos de A são $f(0), f(1), \ldots$ que são $a_0, a_1, a_2, \ldots$. Então podemos enumerar os elementos de A seja ele finito ou infinito.

Proposição A.1.6. *Todo subconjunto infinito de um conjunto enumerável é enumerável.*

Demonstração. Seja A um conjunto enumerável e $B \subset A$, B infinito. Como A é enumerável, existe uma função $f : \mathbb{N} \longrightarrow A$ bijetora e podemos enumerar os elementos de A: $a_0, a_1, a_2, \ldots$, sendo $a_n = f(n)$. Alguns elementos da enumeração não são elementos de B. Eliminando estes elementos ficaríamos com algo como

$$a_0 \quad \not{a_1} \quad \not{a_2} \quad a_3 \quad \not{a_4} \quad a_5 \quad a_6 \quad \ldots$$

Agora podemos chamar os elementos restantes de $b_0, b_1, b_2, \ldots$. Então $b_0 = a_0$, $b_1 = a_3$ e assim por diante (neste exemplo). Formalmente, temos que provar que existe uma função bijetora entre $\mathbb{N}$ e B. Esta função $g : \mathbb{N} \longrightarrow B$ é definida da seguinte forma:

$$g(n) = f(\text{ menor } y \text{ tal que } n \leqslant y \wedge (f(y) \in B))$$

Ou, em notação das funções recursivas,

$$g(n) = f(\mu y[n \leqslant y \wedge (f(y) \in B)])$$

Uma enumeração dos elementos de B pode ser feita enumerando-se os elementos de $A = \{a_0, a_1, a_2, \ldots\}$ e eliminando os elementos não pertencentes a B. A função g é automaticamente obtida por esta enumeração.

$$\begin{array}{cccccccc} a_0 & \not{a_1} & \not{a_2} & a_3 & \not{a_4} & a_5 & a_6 & \ldots \\ \uparrow & & & \uparrow & & \uparrow & \uparrow & \\ g(0) & & & g(1) & & g(2) & g(3) & \ldots \end{array}$$

□

Proposição A.1.7. $\mathbb{N} \sim \mathbb{Z}$

Demonstração. Basta enumerar os elementos de $\mathbb{Z}$ da seguinte forma: $0, 1, -1, 2, -2, 3, -3, \ldots$. □

Proposição A.1.8. $\mathbb{N} \sim \mathbb{Q}$

Demonstração. Construiremos uma função $f : \mathbb{N} \longrightarrow \mathbb{Q}$ bijetora usando a tabela da Figura A.2. As setas da tabela definem uma enumeração dos números racionais se seguirmos a direção dada pelas setas e seguindo da seta menor para as maiores. Esta enumeração é $1/1, -1/1, 2/1, 1/2, -2/1, \ldots$. Os valores de $f(1), f(2), f(3), \ldots$ são associados aos valores desta enumeração ($f(1) = 1/1$, por exemplo). E $f(0) = 0$. Números repetidos que aparecem na tabela não são considerados. Por exemplo, $f(8)$ deveria ser $2/2$, mas $2/2 = 1$ e já temos $f(1) = 1/1 = 1$. Então $f(8) = -3/1$.

Esta função é claramente bijetora e então $\mathbb{N} \sim \mathbb{Q}$.

□

Proposição A.1.9. $\mathbb{N}$ *não é equipotente a* $\mathbb{R}$.

Demonstração. Provaremos por contradição. Suponha que $\mathbb{N} \sim \mathbb{R}$. Como $\mathbb{R} \sim [0, 1)$ e $\sim$ é uma relação de equivalência, $\mathbb{N} \sim [0, 1)$ e os elementos de $[0, 1)$ podem ser enumerados: $r_0, r_1, r_2, r_3, \ldots$. Isto é, a função $f : \mathbb{N} \longrightarrow [0, 1)$ é tal que $f(n) = r_n$. Construiremos uma tabela colocando cada número r_i em uma linha sendo que r_{ij} é o j-ésimo dígito do número r_i. Isto é, $r_i = 0.r_{i0}r_{i1}r_{i2}\ldots$.

$$\begin{array}{cccccccc} r_0 & 0, & r_{00} & r_{01} & r_{02} & r_{03} & \ldots \\ r_1 & 0, & r_{10} & r_{11} & r_{12} & r_{13} & \ldots \\ r_2 & 0, & r_{20} & r_{21} & r_{22} & r_{23} & \ldots \\ r_3 & 0, & r_{30} & r_{31} & r_{32} & r_{33} & \ldots \\ r_4 & 0, & r_{40} & r_{41} & r_{42} & r_{43} & \ldots \\ \ldots & & & & & & \end{array}$$

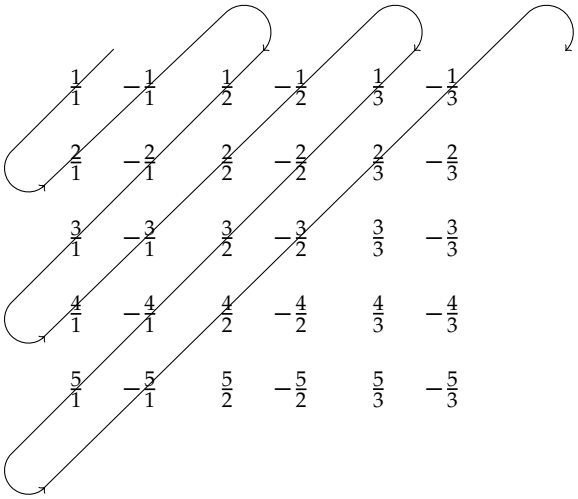


Figura A.2: Uma relação bijetora entre $\mathbb{N}$ e $\mathbb{Q}$

Encontraremos um número s que não está nesta lista (não existe $n \in \mathbb{N}$ tal que $f(n) = s$). Usaremos s_j para o j-ésimo dígito de s. Este número é construído da seguinte forma:

$$s_j = \begin{cases} 0 & \text{se } r_{jj} \neq 0 \\ 1 & \text{se } r_{jj} = 0 \end{cases}$$

Então s é diferente de cada número $r_0, r_1, r_2, \ldots$ pois s tem pelo menos um dígito diferente de cada um destes números (s foi construído assim). Isto é, para todo $j \in \mathbb{N}$, $s_j \neq r_{jj}$ e portanto $s \neq r_j$. Encontramos um número que não está na enumeração $r_0, r_1, r_2, \ldots$ e portanto encontramos $s \in (0,1)$ que não é imagem de nenhum natural n pela função $f : \mathbb{N} \longrightarrow (0,1)$ que assumimos existir (pois $\mathbb{N} \sim (0,1)$).

Como chegamos a uma contradição, $\mathbb{N}$ não é equipotente a $(0,1)$ e consequentemente não é equipotente a $\mathbb{R}$ também.

Assumimos que os números r_i da enumeração de $[0,1)$ não terminam com uma sequência infinita de noves como $0.99999\ldots$. O motivo é que alguns números reais possuem duas representações: uma que termina com uma sequência infinita de noves e outra que não. Para ver isto, considere o número $x = 1.99999\ldots$. Provaremos que $x = 2$:

$$\begin{aligned} 10x &= 19.9999\ldots \\ 10x - x &= 19.9999\ldots - 1.9999\ldots \\ 9x &= 18 \\ x &= 2 \end{aligned}$$

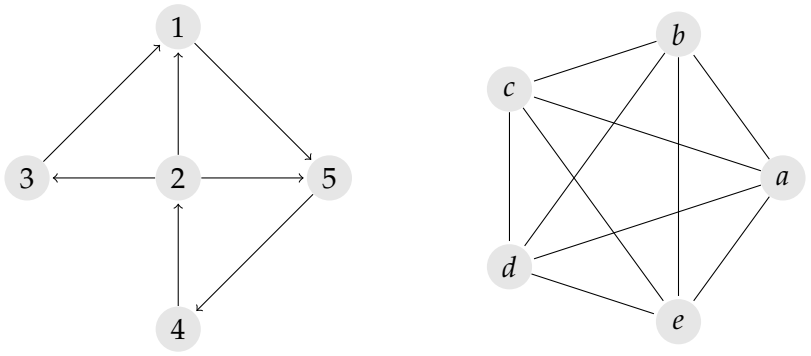
□

Definição A.1.10. *A função característica de um conjunto $A \subset U$ é uma função $\chi_A : U \longrightarrow \{0,1\}$ definida como*

$$\chi_A(x) = 1 \text{ sse } x \in A$$

Algumas vezes a notação c_A é utilizada para a função característica de A.

Os números 0 e 1 em decimal podem ser representados em binário por um dígito. Os números 2 e 3, 10 e 11, por dois dígitos. Os números entre 4 e 7, 100, 101, 110 e 111 por três dígitos. No caso geral, os números entre 2^n e $2^{n+1} - 1$ podem ser representados por n dígitos. Então sendo $N_b(n)$ o número de dígitos binários (bits) necessários para representar n, temos que



(a) (b)

Figura A.3: Exemplo de grafos

$$N_b(n) = \begin{cases} 1 \text{ se } n = 0 \\ 1 + \lfloor \log_2 n \rfloor \text{ se } n \geqslant 1 \end{cases}$$

No sistema de numeração unária, usamos apenas 1 para representar os números naturais. Cada $n \in \mathbb{N}$ é representado pela concatenação de $n + 1$ símbolos 1. Então 0 é representado por 1 e 3 por 1111.

A.2 Teoria dos Grafos

Grafos são utilizados no estudo da complexidade computacional porque as transformações que um programa sofre, durante a sua execução, podem ser representadas por um grafo. Nesta seção definimos os conceitos básicos de teoria dos grafos e um algoritmo, o de busca em largura. Isto é suficiente para a compreenção do restante deste texto.

Definição A.2.1. *Um grafo $G = (V, E)$ é um conjunto V de vértices (vertex em Inglês) e um conjunto E de arestas (edges em Inglês) onde cada aresta é um par de vértices (Ex.: (v, w)).*

Definição A.2.2. *Um grafo pode ser dirigido ou não dirigido. Em um grafo dirigido, a ordem entre os vértices de uma aresta (v, w) é importante. Esta aresta é diferente da aresta (w, v). Em um grafo não dirigido, $(v, w) \in E$ sse $(w, v) \in E$. Isto é, a relação E é simétrica.*

Um grafo dirigido é representado graficamente usando bolinhas para vértices e setas para arestas. Há uma seta do vértice v para o w se $(v, w) \in E$. Em um grafo não dirigido usa-se segmentos de reta para representar as arestas. A Figura A.3 mostra a representação gráfica de (a) um grafo dirigido e (b) um grafo não dirigido. Em (a), $V = \{1, 2, 3, 4, 5\}$ e em (b), $V = \{a, b, c, d, e\}$

Definição A.2.3. *Um* **caminho** *em um grafo entre v_1 e v_n é uma seqüência de arestas $(v_1, v_2), (v_2, v_3), \ldots (v_{n-1}, v_n)$ que abreviamos para $v_1, v_2, v_3, \ldots v_n$. Nenhuma restrição é feita quanto aos vértices do caminho. Podem existir vértices repetidos. O tamanho de um caminho é o número de arestas que o compõem.*

Definição A.2.4. *Um grafo é* **conectado** *se há um caminho entre dois vértices quaisquer. Um grafo não conectado é* **desconectado**.

Definição A.2.5. *Um* **ciclo** *em um grafo é uma seqüência de arestas $(v_1, v_2), (v_2, v_3), \ldots (v_n, v_1)$ que abreviamos para $v_1, v_2, v_3, \ldots v_n, v_1$.*

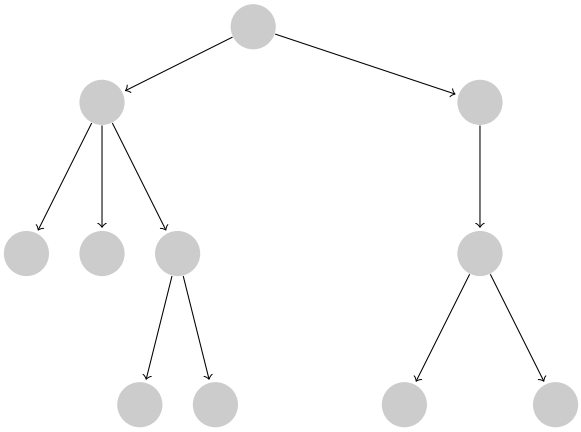


Figura A.4: Exemplo de uma árvore

Por exemplo, no grafo (b) da Figura A.3, temos vários ciclos. Entre eles, $(a, b), (b, c), (c, d), (d, e), (e, a)$ e $(d, b), (b, e), (e, d)$.

Definição A.2.6. *Uma* **árvore** *é um grafo conectado sem ciclos.*

A Figura A.4 mostra um exemplo de árvore.

Definição A.2.7. *Em uma árvore dirigida $G = (V, E)$, se $(v, w) \in E$ então chamamos v de pai e w de filho.*

Definição A.2.8. *Um vértice com zero filhos de uma árvore dirigida é chamado de* **folha**.

Definição A.2.9. *Em uma árvore não dirigida, uma* **folha** *é um vértice ligado a apenas um outro vértice.*

Definição A.2.10. *Uma* **árvore binária** *(AB) é uma árvore dirigida (árvore e grafo dirigido) definido indutivamente como:*

- *uma AB com um único vértice v é uma árvore. A raiz desta árvore é v;*
- *se C e D são duas árvores binárias com raízes w e t, e v um vértice não pertencente a C ou D, então o grafo composto por C, D, v e as arestas (v, w) e (v, t) é uma árvore binária. O vértice v é a raiz desta árvore;*
- *se C é uma AB com raiz w e v um vértice não pertencente a C, então a árvore composta por C, v e a aresta (v, w) é uma AB.*

Dizemos que v é o pai de w e t e estes são os filhos de v. Então cada vértice pode ter zero, um ou dois filhos. As árvores C e D são as sub-árvores da árvore binária completa (v com C e D).

Definição A.2.11. *A* **altura** *de uma árvore binária é definida indutivamente como se segue:*

- *a altura de uma árvore binária com um vértice é 1;*
- *a altura de uma AB com raiz v ligada a árvores C e D é 1 + a maior altura entre C e D.*

Um grafo $G = (V, E)$ pode ser representado por várias estruturas de dados na memória de um computador. A representação que usaremos é a matriz de adjacência. A matriz quadrada $n \times n$ $A = (A_{ij})_{1 \leqslant i,j \leqslant n}$ representa o grafo $G = (V, E)$ se:

(a) os vértices de E pertencem ao conjunto $\{1, 2, \ldots n\}$. Usaremos sempre números entre 1 e $|V|$ para representar os vértices;

```
function Alcancável(G = (V,E), v, w) :  boolean
begin
marque v
for each aresta (v,u) ∈ E do
    if u = w
    then
        return true;
    else
        Alcancável(G, u, w);
end
```

Figura A.5: Algoritmo Alcançável que retorna `true` se há um caminho de v até w

(b) $A_{ij} = 0$ ou $A_{ij} = 1$;

(c) $A_{ij} = 1$ sse $(i, j) \in E$

Claramente, o número de arestas de um grafo com n vértices é no máximo n^2, pois em um vértice (v, w) há n possibilidades para v e n para w. Note que, a matriz de adjacências trás todas as informações sobre um grafo, que são as suas arestas (a própria matriz) e os vértices (o número de linhas da matriz).

Para descobrir se existe um caminho entre os vértices v e w podemos utilizar o algoritmo de busca em largura, dado a seguir. Este algoritmo percorre um grafo G partindo de um vértice inicial v. O percurso é feito "caminhando" sobre as arestas. Para cada vértice visitado pode ser feito um pré-processamento. E para cada aresta visitada pode ser feito um pós-processamento.

```
Algoritmo BL(G = (V,E), v)
begin
marque v
faça o pré-processamento sobre v
for each aresta (v,u) ∈ E do
    if u não foi marcado
    then
        begin
        BL(G, u);
        faça pós-processamento em (v,u)
        end
end
```

A matriz de adjacências de G é passado como parâmetro no lugar de $G = (V, E)$. Para verificar as arestas $(v, u) \in E$, basta verificar quais posições da linha v da matriz possuem o número 1.

Para descobrir se existe um caminho entre v e w em um grafo G, faz-se a busca em largura a partir de v e depois verifica-se se w foi marcado. Só existe um caminho entre v e w se este último foi marcado. Uma otimização do algoritmo é, no `if`, antes da chamada `BL(G, u)`, parar o algoritmo se $u = w$. Chamaremos este algoritmo de Alcancável — veja Figura A.5.

Índice Remissivo

Referências Bibliográficas

[1] Agudelo, Juan C. e Carnielli, Walter. Quantum Computation via Paraconsistent Computation. Disponível em http://arxiv.org/abs/quant-ph/0607100v1

[2] Arora, Sanjeev e Barak, Boaz. Computational Complexity — a Modern Approach. Cambridge University Press, first editon, 2009.

[3] Cook, Stephen A. The complexity of theorem proving procedures. Proc. ACM Symposium on Theory of Computing, 151-158, 1971.

[4] Hedman, Shawn. A First Course in Logic — An Introduction to Model Theory, Proof Theory, Computability, and Complexity. Oxford University Press, 2004.

[5] Fortnow, Lance. Foundations of Complexity. Computational Complexity Blog, `http://blog.computationalcomplexity.org`, may 01, 2009.

[6] Nielsen, Michael e Chuang, Isaac L. Quantum Computation and Quantum Information. Cambridge University Press, 2000.

[7] Odifreddi, Piergiorgio. Classical Recursion Theory — The Theory of Functions and Sets of Natural Numbers. Elsevier Sience Publishers B. V., 1989.

[8] Odifreddi, Piergiorgio. Classical Recursion Theory — The Theory of Functions and Sets of Natural Numbers, Volume 2. Elsevier Sience Publishers B. V., 1999.

[9] Papadimitriou, Christos H. Computational Complexity. Addison-Wesley, 1994.

[10] Savage, John E. Models of Computation — Exploring the Power of Computing. Addison-Wesley, 1998.

[11] Turing, Alan. On Computable Numbers, With an Application to the Entscheidungsproblem. Proceedings of the London Mathematical Society 42 (2), 1936.

[12] Turing, Alan. On Computable Numbers, with an Application to the Entscheidungsproblem: A correction. Proceedings of the London Mathematical Society, 2 43: 544-6, 1937,

[13] One instruction set computer. Wikipedia, disponível em `http://en.wikipedia.org/wiki/One_instruction_se` 2009.