Classes de Complexidade

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

22 de setembro de 2011

Sumário

Prefácio

Este é um texto introdutório sobre complexidade computacional desenvolvido como parte do projeto "Computação Quântica" aprovado pela trigésima sexta reunião do Conselho Científico do CLE (Centro de Lógica, Epistemologia e História da Ciência) em 13 de setembro de 2007. Além de uma introdução à complexidade computacional há um Capítulo dedicado à pesquisa desenvolvida durante este projeto. Originalmente o projeto seria sobre Computação Quântica e Classes de Complexidade de linguagens. Contudo o foco foi mudado para o estudo de classes de linguagens, que são classificadas em termos da complexidade das máquinas necessárias para reconhecê-las ou decidi-las.

Este texto é uma introdução à Complexidade Computacional. Mas contém praticamente todos os tópicos que seriam importantes em um curso de pós-graduação de introdução ao assunto (veja uma lista elaborada por Lance Fortnow [5]). O Capítulo 1 define os principais conceitos, proposições e teoremas matemáticos que serão necessários nos Capítulos seguintes. O Capítulo 2 apresenta os principais modelos de Computação: a máquina de Turing e suas principais variações (determinística, não determinística, com várias fitas, de entrada e saída), a Random Access Machine (RAM) e uma linguagem de programação de alto nível. Para justificar o uso da máquina de Turing como modelo de computação no restante do texto, o Capítulo 3 mostra que cada modelo de computação pode simular qualquer outro. E os modelos considerados mais "razoáveis" podem simular outros em tempo polinomial. O Capítulo 4 mostra que nem todos os problemas podem ser solucionados por uma máquina de Turing, um modele de computação abstrato. Neste Capítulo são estudados alguns resultados básicos de Computabilidade. Estes resultados são importantes porque a muitas técnicas utilizadas em Complexidade Computacional foram herdadas de Computabilidade. O Capítulo 5 define as principais classes de linguagens de espaço e tempo e mostra as relações conhecidas entre elas. O Capítulo 6 mostra duas linguagens NP-completas e o teorema de Cook, provavelmente o mais importante teorema da Teoria da Computação.

Todas as provas deste texto são originais exceto aquelas em que há uma referência a um livro ou artigo na proposição ou teorema. Há poucas provas neste último caso e mesmo estas não são cópias ipsis litteris das provas das referências. Elas são apresentadas de modo diferente e com muito mais detalhes do que no original Os tópicos aqui abordados estão disponíveis em vários livros sobre complexidade da computação, são tópicos já sedimentados nesta área. Como é comum nestes livros, a referência ao artigo que originou cada proposição ou teorema não é citado.

José de Oliveira Guimarães

Capítulo 1

Definições Matemáticas Básicas

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

1.1 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 1.1.1. *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 1.1.2. *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 1.1.3. *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 1.1.4. *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 1.1.5. *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 1.1.6. *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.

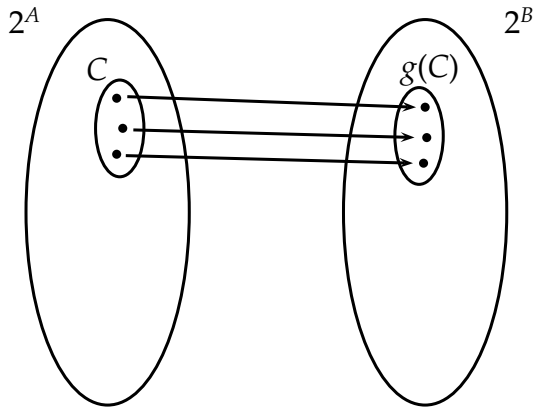

Figura 1.1: Função entre 2^A e 2^B bijetora

Definição 1.1.7. *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 1.1.8. *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).

Teorema 1.1.1. *(Cantor-Schröder-Bernstein) Se $A \preceq B$ e $B \preceq A$, então $A \sim B$.*

Proposição 1.1.1. *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 1.1.2. *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 1.1.

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

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

□

Proposição 1.1.3. $(-1,1) \sim \mathbb{R}$

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 1.1.4. *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 1.1.3. 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 1.1.5. $[0,1) \sim (0,1)$, $[0,1] \sim (0,1)$ *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 1.1.9. *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, seja ele finito ou não, podemos listar os seus elementos: $a_0, a_1, a_2, a_3, \ldots$ pois existe uma função bijetora $f : \mathbb{N} \longrightarrow A$ e $a_i = f(i)$.

Proposição 1.1.6. *Todo subconjunto infinito de um conjunto enumerável é enumerável.*

Demonstração. Seja $B \subset A$, B infinito. Como A é enumerável, existe uma função $f : \mathbb{N} \longrightarrow A$ bijetora. Uma função bijetora $g : \mathbb{N} \longrightarrow B$ 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 1.1.7. $\mathbb{N} \sim \mathbb{Z}$

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

Proposição 1.1.8. $\mathbb{N} \sim \mathbb{Q}$

Demonstração. Construiremos uma função $f : \mathbb{N} \longrightarrow \mathbb{Q}$ bijetora usando a tabela da Figura 1.2. 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 1.1.9. $\mathbb{N}$ *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, $r_i = r_{i0} r_{i1} r_{i2} \ldots$.

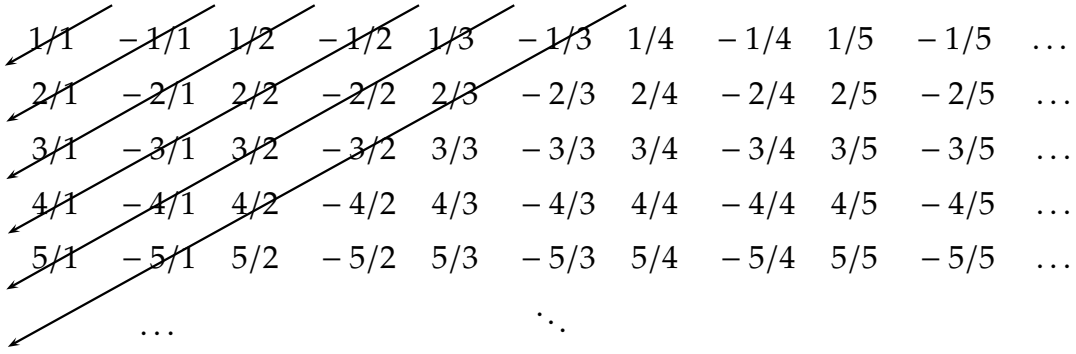

Figura 1.2: Uma relação bijetora entre $\mathbb{N}$ e $\mathbb{Q}$

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

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 $x = 2$:

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

□

Definição 1.1.10. *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.

Proposição 1.1.10. $2^{\mathbb{N}} \sim \mathbb{R}$

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 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 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 1.1.11. *A notação B^A, no qual A e B são conjuntos, denota o conjunto de todas as funções com domínio A e contra-domínio B. Isto é,*

$$B^A = \{f : f \subset A \times B \text{ é função }\}$$

Proposição 1.1.11. $2^{\mathbb{N}} \sim \mathbb{R}$

Demonstração. Em teoria axiomática dos conjuntos, o número 2 é representado pelo conjunto $\{0,1\}$ no qual $0 =_{def} \emptyset$ e $n+1 =_{def} n \cup \{n\}$. Então $2^{\mathbb{N}}$ também denota o conjunto de todas as funções da forma $g : \mathbb{N} \longrightarrow \{0,1\}$. Este conjunto é equipotente a $[0,1]$ pela função $f : 2^{\mathbb{N}} \longrightarrow [0,1]$ que associa a cada função $g \in 2^{\mathbb{N}}$ o número real em binário

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

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 $2^{\mathbb{N}} \sim \mathbb{R}$. □

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

Proposição 1.1.12. $\mathbb{N}^{\mathbb{N}} \sim \mathbb{R}$

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

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

Usamos $\overline{g(n)}$ com o mesmo significado que na Proposição 1.1.11. Isto é, se $g(n) = 2n$, $f(g) = 0.0210210021102\ldots$. 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.

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

Então provamos que $\mathbb{N}^{\mathbb{N}} \leq [0,1]$ e $[0,1] \leq \mathbb{N}^{\mathbb{N}}$. Pelo Teorema 1.1.1, $\mathbb{N}^{\mathbb{N}} \sim [0,1]$. Como $[0,1] \sim \mathbb{R}$, $\mathbb{N}^{\mathbb{N}} \sim \mathbb{R}$. □

1.2 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 1.2.1. *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 1.2.2. *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 1.3 mostra a representação gráfica de (a) um grafo dirigido e (b) um grafo não dirigido. Em ambos os casos, $V = \{1, 2, 3, 4, 5\}$.

Definição 1.2.3. *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.*

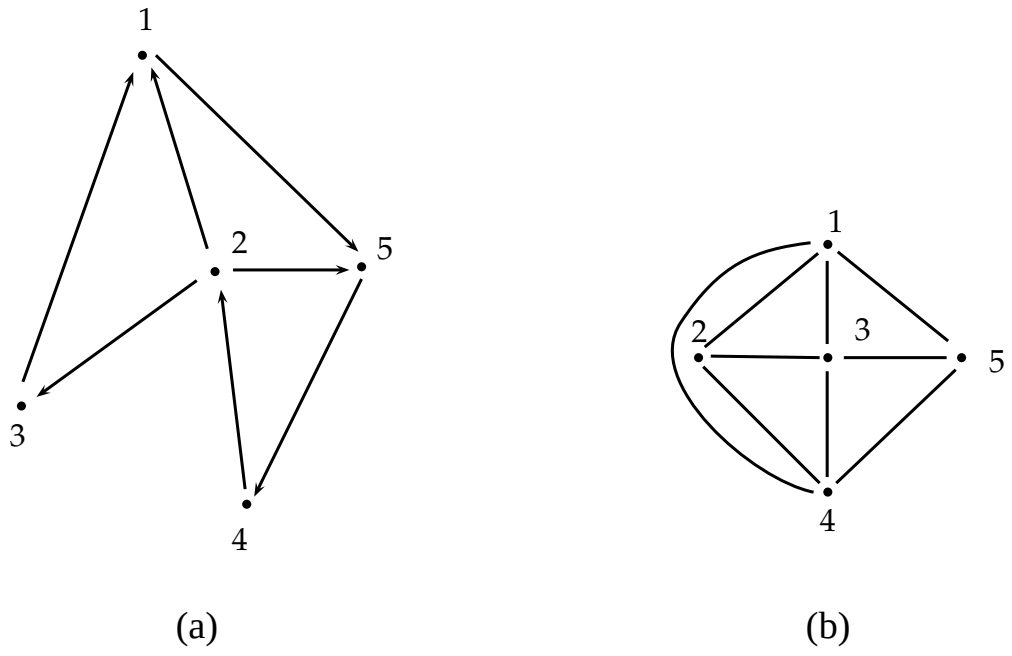

Figura 1.3: Exemplo de um grafo

Definição 1.2.4. *Um grafo é* **conectado** *se há um caminho entre dois vértices quaisquer. Um grafo não conectado é* **desconectado**.

Definição 1.2.5. *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* $v_1, v_2, v_3, \ldots v_n, v_1$.

Por exemplo, no grafo (b) da Figura 1.3, temos ciclos $1, 2, 4, 5, 1$ e $1, 3, 2, 1$.

Definição 1.2.6. *Uma* **árvore** *é um grafo conectado sem ciclos.*

A Figura 1.4 mostra um exemplo de árvore.

Definição 1.2.7. *Em uma árvore dirigida* $G = (V, E)$*, se* $(v, w) \in E$ *então chamamos v de pai e w de filho.*

Definição 1.2.8. *Um vértice com zero filhos de uma árvore dirigida é chamado de* **folha**.

Definição 1.2.9. *Em uma árvore não dirigida, uma* **folha** *é um vértice ligado a apenas um outro vértice.*

Definição 1.2.10. *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;*

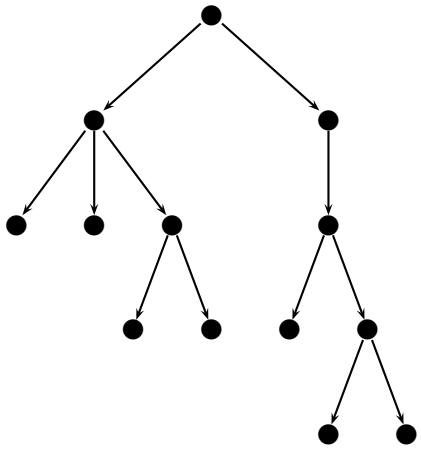

Figura 1.4: Exemplo de uma árvore

- *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 1.2.11. *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.*

Definição 1.2.12. *Uma* **árvore binária cheia** *(ABCh) é uma árvore binária onde cada vértice tem zero ou dois filhos.*

Definição 1.2.13. *Uma* **árvore binária completa** *(ABC) é uma árvore binária na qual as sub-árvores ligadas a um mesmo vértice possuem a mesma altura.*

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;

(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 1.5.

1.3 Linguagens Formais

A teoria da computação utiliza largamente linguagens formais. Uma linguagem é um conjunto de cadeias de símbolos de um certo alfabeto (como todas as palavras em português, em que o alfabeto latino é utilizado). Uma linguagem formal é uma linguagem que é definida formalmente. Vejamos algumas definições sobre este tópico.

```
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 1.5: Algoritmo Alcançável que retorna `true` se há um caminho de v até w

Definição 1.3.1. *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 1.3.2. *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 1.3.1. *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 1.3.3. *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 1.3.4. *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 1.3.5. *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) $|x| < \left|y\right|$;

(b) $|x| = \left|y\right|$, $x =_{def} x_1x_1 \dots x_n$, $y =_{def} y_1y_2 \dots y_n$ *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:

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

Proposição 1.3.1. *Assumindo Σ finito, $\Sigma^\star \sim \mathbb{N}$.*

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 $\Sigma^0, \Sigma^1, \Sigma^2, \dots$. Agrupando todas estas enumerações conseguimos uma enumeração para $\Sigma^\star$.

□

Proposição 1.3.2. *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 1.3.3. $2^{\Sigma^\star}$ *é equipotente a* $[0, 1]$.

Demonstração. Temos $\Sigma^\star \sim \mathbb{N}$ pela Proposição 1.3.1. Pela Proposição 1.1.2, $2^{\Sigma^\star} \sim 2^{\mathbb{N}}$. Como $2^{\mathbb{N}} \sim [0, 1]$, pela transitividade de $\sim$ temos $2^{\Sigma^\star} \sim [0, 1]$. Como $[0, 1] \sim \mathbb{R}$, também temos $2^{\Sigma^\star} \sim \mathbb{R}$. □

Uma 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).

Os números 0 e 1 em decimal pode 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

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

Capítulo 2

Modelos de Computação

Este Capítulo introduz os principais modelos de computação utilizados nesta pesquisa, as Máquinas de Turing determinísticas e não determinísticas de uma ou mais fitas. Além disto são apresentados as máquinas de acesso aleatório e as famílias uniformes de circuitos.

2.1 Máquinas de Turing

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 três estados especiais: q_f, q_s e q_n, todos elementos de Q e todos 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 $q_i \notin \{q_f, q_s, q_n\}$. 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_f, q_s ou q_n, a máquina pára. Pela definição de MT, nenhuma instrução (q, s, q', s', d) pode ter $q \in \{q_f, q_s, q_n\}$. 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 uma célula 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. 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:

$(q_0, 1, q_0, 1, 1)$
$(q_0, \sqcup, q_1, \sqcup, 1)$
$(q_1, 1, q_1, 1, 1)$
$(q_1, \square, q_2, \sqcup, -1)$
$(q_2, 1, q_3, \triangleright, -1)$
$(q_3, 1, q_3, 1, -1)$
$(q_3, \square, q_4, \sqcup, 1)$
$(q_4, 1, q_f, 1, 0)$

Esta MT está representada graficamente na Figura 2.1.1. 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).

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*.*

$(q_0, 0, q_0, 0, 0)$
$(q_0, x, q_f, x, 0)$ *para todo* $x \in \Sigma, x \neq 0$

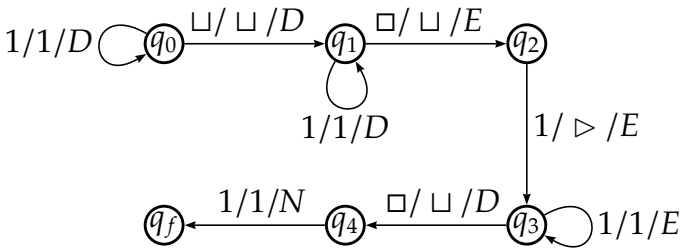

Figura 2.1: Representação gráfica de uma MT que soma dois números unários

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".

2.2 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 $s'_1, \ldots s'_k$ 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, q_n e q_f no conjunto Q;
7. em uma instrução $(q, s_1, s_2, \ldots s_k, q', s'_1, d_1, s'_2, d_2, \ldots s'_k, d_k)$, $q \notin \{q_f, q_s, q_n\}$ 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 multipla 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_f, 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 2.2.1. *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 $d_1, d_2, \ldots, d_k$;

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

Definição 2.2.2. *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 2.2.3. *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* $s_k = s'_k$ *então* $d_k = 0$.

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 □. 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 $2(1 + \lfloor \log_2 n \rfloor) - 1$.

Proposição 2.2.1. *Dada uma MT M que utiliza um alfabeto $\{r_1, r_2, \ldots r_m\}$ e um número $n < m$, há uma MT M' equivalente a M que utiliza um alfabeto $s_1, s_2, \ldots s_n$ e na qual cada instrução de M é simulada por $c \log_n m$ instruções de M', c uma constante.*

Demonstração. Vários símbolos de M' são utilizados para representar um único símbolo r_i de M. Quantos são necessários? Se cada símbolo r_i for representado por

$$s_{i_1} s_{i_2} \ldots s_{i_k}$$

o número k deve ser tal que o número de combinações possíveis de símbolos s_j seja maior do que m. O número de combinações é

$$\overbrace{nnn\ldots n}^{k} = n^k$$

Então devemos ter $n^k \geqslant m$ e $k \geqslant \log_n m$. Usaremos $k = \lceil \log_n m \rceil$. A MT M' gasta $\lceil \log_n m \rceil$ passos para reconhecer um símbolo de M e outros tantos para escrever um símbolo. Então cada passo de M será simulado por $2\lceil \log_n m \rceil$ em M'. □

Definição 2.2.4. *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 □ 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 □ 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:

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

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 2.2.2. *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 2.2.5. *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 2.2.6. *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:

[1] Um nome comum para este registrador é IP, instruction pointer.

(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 2.2.7. *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 2.2.8. *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.*

Nem todas as funções podem ser utilizadas para medir a complexidade em tempo ou espaço de uma máquina de Turing. Algumas funções não são computáveis e outras possuem um comportamento estranho que invalidaria a prova de muitos teoremas. Este tipo de função não mede a complexidade dos algoritmos encontrados na prática e não deve ser permitido. Mas ao invés de definir quais funções não são permitidas, definiremos quais são.

Definição 2.2.9. *Uma função $f : \mathbb{N} \longrightarrow \mathbb{N}$ é uma* **função adequada de complexidade** *se:*

- *f é não decrescente; isto é, $f(n+1) \geqslant f(n)$ para todo n;*
- *existe uma MT com entrada e saída M_f com k fitas que toma uma entrada x e que imprime $f(n)$ símbolos $\sqcup$ na última fita, onde n é o tamanho da entrada. Esta máquina gasta um número de passos exatamente igual a $c_1(n + f(n)) + d_1$ e utiliza um número de células exatamente igual a $c_2 f(n) + d_2$ na qual c_i e d_i são constantes. O número de passos e o número de células utilizadas depende apenas do tamanho da entrada x.*

O número de passos é igual a $c_1(n + f(n))$ e não $c_1 f(n)$ porque algumas funções podem ser tais que $f(n) \leqslant n$ para todo n. Todas as funções normalmente utilizadas para complexidade de tempo e espaço são *funções adequadas de complexidade*: n, n^2, n^k, 2^n, $\log n$, $\log^k n$, $n \log n$, etc.

Faremos a prova de que $f(n) = n$ é uma função adequada de complexidade. A MTES de duas fitas associada a f, M_f, simplesmente escreve na segunda fita um símbolo $\sqcup$ para cada dígito da entrada e volta as cabeças de leitura/gravação para o início das fitas. As instruções de M_f são:

$(q_0, 0, \square, q_0, 0, \sqcup, 1, 1)$
$(q_0, 1, \square, q_0, 1, \sqcup, 1, 1)$
$(q_0, \square, \square, q_1, \sqcup, \triangleright, -1, -1)$
$(q_1, 0, \sqcup, q_1, 0, \sqcup, -1, -1)$
$(q_1, 1, \sqcup, q_1, 1, \sqcup, -1, -1)$
$(q_1, \square, \square, q_f, \sqcup, \sqcup, 1, 1)$

O número de passos que esta máquina leva com entrada x é igual a $|x|+2$. O espaço utilizado é $|x| + 2$ também.

Para facilitar a prova de vários teoremas, é importante a utilização de máquinas de Turing que são precisas, que terminam a sua execução depois de exatamente um certo número de passos dado por uma função e que utilize exatamente o número de espaços dados por outra função. Isto é, para todas as entradas de certo tamanho n, o número de passos será exatamente o mesmo. E o número de células utilizadas também.

Definição 2.2.10. *Uma MT M é* **precisa** *se existem funções $t : \mathbb{N} \longrightarrow \mathbb{N}$ e $s : \mathbb{N} \longrightarrow \mathbb{N}$ tal que, para qualquer entrada x de tamanho n, o número de passos que M leva até parar é exatamente $t(n)$ e o número de células utilizadas é exatamente $s(n)$. Em uma MT de entrada e saída não se considera em $s(n)$ as células utilizadas na fita de entrada e de saída.*

Será sempre possível converter uma MT em uma outra equivalente e que seja precisa? A resposta é sim. Mas antes veremos uma definição de equivalência entre MTs.

Definição 2.2.11. *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.*

Proposição 2.2.3. *Dada uma MT M, existe uma MT M' equivalente a M que é precisa.*

Demonstração. Faremos a prova apenas para uma MT determinística M de uma única fita. Suponha que M executa em tempo $f(n)$ e em espaço $g(n)$. Então o número de passos executados com entrada x tal que $n = |x|$ é menor do que $f(n)$ e o espaço utilizado é menor do que $g(n)$. O que M' fará é simular M e contar o número de passos e o espaço utilizado. Se a computação terminar antes de $f(n)$ passos, M' continua até completar os $f(n)$ passos. Idem para o espaço.

M' utiliza duas fitas. A primeira conta o número de passos de M que já foram simulados. A segunda simula a fita de M.

Inicialmente, M' escreve $f(|x|)$ símbolos $\sqcup$ na primeira fita e $g(|x|) - |x|$ $\sqcup$ na segunda fita, depois da entrada x. Assume-se que $g(n) \geqslant n$. A cabeça de leitura/gravação é deixada no símbolo mais à esquerda das duas fitas (primeiro $\sqcup$ da primeira fita e primeiro símbolo da entrada na segunda fita).

M' prossegue simulando M, o que pode ser feito em tempo constante. A cada passo da simulação de M, a cabeça de leitura/gravação da primeira fita avança para a direita. Como isto a primeira fita efetivamente conta o número de passos da simulação de M. Observe que o avanço é feito depois que a execução de uma instrução de M é simulada, não depois da execução de uma instrução de M' (contudo usualmente a simulação de uma instrução tomará apenas uma instrução de M'). Quando a simulação de M termina, M' avança a cabeça de leitura/gravação da primeira fita até encontrar um $\square$ que sinaliza o fim dos símbolos válidos da fita. Como M executa em tempo $f(n)$, o número de passos será $\leqslant f(n)$ para entrada x tal que $n = |x|$. Então

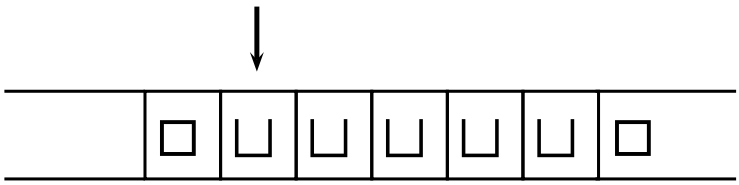

Figura 2.2: Primeira fita de M'

a cabeça de leitura/gravação da primeira fita M' estará no máximo sobre o primeiro símbolo $\square$ depois das células com $\sqcup$. A Figura 2.2 mostra a configuração da primeira fita depois de dois passos. Neste caso, o número de passos que M deverão ser simulados é cinco.

A simulação de M na segunda fita utilizará no máximo $g(|x|)$ células (pois M executa em espaço $g(n)$). M' utiliza exatamente $g(|x|)$ células da segunda fita ($g(|x|) - |x|$ símbolos $\sqcup$ mais $|x|$ símbolos da entrada). Então na segunda fita serão utilizados exatamente $g(|x|)$ células. Na primeira fita serão utilizadas exatamente $f(|x|)$ células. Para calcular as funções f e g é necessário utilizar células da fita, que devem ser somadas ao número de células utilizadas na simulação de M e na contagem dos passos desta máquina. Estas funções são calculadas por MT M_f e M_g, já que assume-se que elas sejam *funções adequadas de complexidade*. O número de células utilizadas por M_f e M_g depende apenas de $|x|$ e não de x (o parâmetro para f e g é n, não x). Isto é, dadas duas entradas de mesmo tamanho, o tempo de execução e o espaço utilizado por M_f (e M_g) são iguais.

Sendo $s_f(n)$ e $s_g(n)$ o espaço utilizado para o cálculo das funções f e g, o espaço total utilizado por M' é exatamente igual a

$$f(|x|) + g(|x|) + s_f(n) + s_g(n)$$

Assumiu-se que as células utilizadas por M_f e M_g não são reaproveitadas na simulação de M.

Similarmente, o tempo de execução é constante para entradas de mesmo tamanho. As máquinas M_f e M_g gastam um tempo de execução que depende apenas de $|x|$ e que portanto independe do valor específico de x. Então o tempo de execução de M' é exatamente igual ao tempo de execução de $M_f(x)$ e $M_g(x)$ mais o tempo de escrever os símbolos $\sqcup$ nas duas fitas mais o tempo de simulação. Todos estes tempos dependem apenas de $|x|$. Portanto, M' é precisa.

Provamos que, para cada MT M com uma única fita, existe uma MT M' equivalente que é precisa. Para uma máquina com várias fitas, a prova é idêntica. Para uma MT M não determinística, utiliza-se M' não determinística. Todos os possíveis caminhos da computação (caminhos em $Ar(M(x))$) utilizam exatamente o mesmo número de passos e células, pois este número depende apenas de $|x|$ e não das escolhas feitas durante a computação não determinística. $\square$

2.3 Complexidade de Computação

Definição 2.3.1. *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 2.3.2. *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.

Definição 2.3.3. *A complexidade em espaço de uma MT com entrada e saída com k fitas não considera as células utilizadas nas fitas de entrada e saída.*

Só não consideradas as células utilizadas nas fitas 2 a $k-1$ (se $k > 2$). Com esta definição uma MTES pode utilizar menos espaço do que n, o tamanho da entrada. Como exemplo considere uma MT que retorna o índice do menor número de uma sequência de números em binário, inicialmente separados por $\sqcup$. Em pseudo-código, o algoritmo seria:

```
function Menor(v : vetor, n : integer) : integer
  begin
  menor = 1
  for j = 2 to n do
     if v[j] < v[menor] then menor = j
  return menor;
  end
```

O algoritmo utiliza índices `j` e `menor`. Cada índice utiliza um número de células $1 + \lfloor \log_2 n \rfloor$ pois o maior tamanho do índice é n (um número $n \neq 0$ representado em binário ocupa $1 + \lfloor \log_2 n \rfloor$ dígitos). Então no total a MT implementando este algoritmo utilizará $2(1 + \lfloor \log_2 n \rfloor)$ células.

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 2.3.1. *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 2.3.1. *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.

2.4 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 NDTM 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_f, q_s ou q_n.

Definição 2.4.1. *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 2.3.

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.

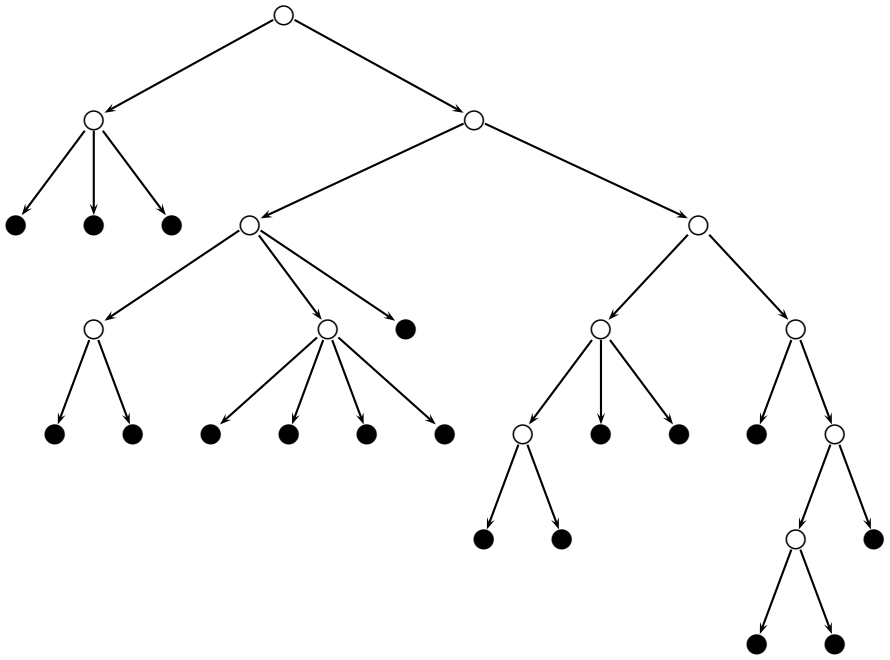

Figura 2.3: Exemplo de uma árvore $Ar(N(x))$

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:

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

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 2.3: 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 $N = (Q, \Sigma, I, q)$.

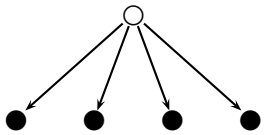

Figura 2.4: Conjunto δ_{ps} com quatro instruções começando com p, s

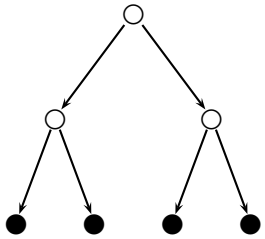

Figura 2.5: Representação das instruções de N' com grau dois em cada vértice

Um conjunto δ_{ps} de N tal que $\left|\delta_{ps}\right| = 4$ está representado na Figura 2.4. 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 2.5. 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 2.5 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 $n = |x|$.

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 2.4.2. *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 2.4.3. *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 a estados q_n e q_f. 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

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

A definição 2.2.10 define MT precisas, que são máquinas que sempre param e que utilizam um tempo e espaço dados por duas funções t e s que tomam um parâmetro $n = |x|$, o tamanho da entrada. Definiremos este mesmo conceito para MTND.

Definição 2.4.4. *Uma MTND N é* **precisa** *se existem funções $t : \mathbb{N} \longrightarrow \mathbb{N}$ e $s : \mathbb{N} \longrightarrow \mathbb{N}$ tal que, para qualquer entrada x de tamanho n, o número de passos que M leva até parar é exatamente $t(n)$ e o número de células utilizadas em todos os caminhos da raiz a uma folha da $Ar(N(x))$ é exatamente $s(n)$.*

Então em uma árvore $Ar(N(x))$ de uma MT precisa N, as sub-árvores de um vértice são sempre do mesmo tamanho.

2.5 RAM

As máquinas de Turing são dispositivos muito diferentes dos computadores convencionais, que possuem memória de acesso aleatório (RAM - *random access memory*). A n-ésima posição de memória pode ser usada sem que o computador use as $n-1$ primeiras posições. Em uma MT, para usar a n-ésima célula (considerando 0 como a posição inicial) é necessário fazer a cabeça de leitura/gravação percorrer todas as células entre 0 e n.

Uma máquina RAM (*random access machine*) permite o acesso aleatório à memória como um computador convencional, tendo uma descrição mais próxima deste do que as máquinas de Turing. Uma máquina RAM possui uma quantidade ilimitada de células de memória, cada uma capaz de guardar um inteiro positivo qualquer, incluindo zero. Esta memória é manipulada usando-se o símbolo m indexado por uma constante ou outra posição de memória. Os tipos de instrução da máquina RAM são dados abaixo, na qual i, j e k são constante inteiras. $m[i]$ é a posição de memória i.

$m[i] = 1$
$m[i] = m[j] + m[k]$
$m[i] = m[j] - m[k]$
$m[i] = m[j]/2$
$m[i] = m[m[j]]$
$m[m[i]] = m[i]$
goto $m[i]$
goto i
if$m[i] < 0$ then goto j
if$m[i] = 0$ then goto j
stop

O significado da maioria das instruções é claro. Por exemplo, a instrução $m[i] = 1$ coloca 1 na posição de memória i. A instrução $m[m[i]] = m[i]$ coloca o valor da posição i na posição de memória $m[i]$. stop pára a máquina. A instrução $m[i] = m[j] - m[k]$ poderia produzir um número negativo. Mas quando $m[j] - m[k] < 0$, o valor armazenado em $m[i]$ será 0.

Um programa na RAM consiste de uma ou mais instruções que são armazenadas começando na posição zero de uma memória m'. A memória que contém as instruções é diferente da que contém os dados — escolhemos esta configuração pois ela facilita a simulação de uma RAM por uma MT. Cada instrução é armazenada em uma única posição de memória. Há um registrador implícito IP que contém o endereço da próxima posição de memória que deve ser executada (supõe-se que $m'[IP]$ contenha uma instrução. Se não contiver, a máquina pára.). Inicialmente, $IP = 0$. A execução do programa começa na instrução $m'[0]$ e prossegue na instrução $m'[1]$ a menos que a instrução contida em $m'[0]$ contenha um goto habilitado. Um goto desvia o fluxo de execução para uma outra instrução; isto é, goto j faz a atribuição $IP = j$. As instruções
if $m[i] < 0$ then goto j e

if $m[i] = 0$ then goto j
atribuem j a IP só se $m[i] < 0$ ou $m[i] = 0$, respectivamente.

A entrada de uma RAM é colocada na memória m começando em $m[1]$. O número de posições da entrada é armazenado em $m[0]$. O tamanho da entrada é o número de dígitos binários necessários para expressar a entrada.

Uma RAM necessariamente tem que ter posições de memória $m[i]$ de tamanho ilimitado. Se cada posição fosse de um tamanho constante, como nos computadores digitais, as instruções `m[i] = m[m[j]]` e `m[m[i]] = m[i]` não poderiam acessar toda a memória. Se $m[i]$ tivesse o tamanho de 32 bits, por exemplo, o número máximo de posições de memória que a RAM poderia manipular seria 2^{32}, um número finito. Então uma RAM não teria o poder computacional de uma máquina de Turing, podendo ser simulada por uma máquina de estados finitos (MEF).

Mas um modelo de máquina RAM mais parecido com os computadores digitais utilizaria um tamanho constante para cada posição de memória. E como permitir uma memória m com infinitas posições? Simples: substituiríamos a instrução $m[i] = m[m[j]]$ pela instrução $m[i] = m[m^\star[j]]$ na qual $m^\star[j]$ é definido como

$$m^\star[j] = \begin{cases} m[j] + m^\star[m[j]] & \text{se } m[j] \text{ contém um endereço válido} \\ m[j] & \text{caso contrário} \end{cases}$$

Haveria uma diferença entre uma posição $m[j]$ contendo um número qualquer e um endereço válido. Para indicar que $m[j]$ contém um endereço teríamos uma instrução toaddr $m[j]$ (*change address*, em Inglês). Esta instrução transforma um número em endereço. tovalue $m[j]$ transformaria $m[j]$ em um valor que não pode ser utilizado como endereço. A cada posição de memória haveria um bit indicando se a posição guarda um endereço ou um valor que não pode ser utilizado como endereço. toaddr e tovalue simplesmente mudaria este bit para 1 e 0.

O efeito da definição de $m^\star[j]$ é o mesmo que o seguinte algoritmo: $m^\star[j]$ é $m[j]$ se $m[j]$ é um endereço válido. Senão, procure nas posições $m[m[j]]$, $m[m[m[j]]]$, $m[m[m[m[j]]]]$, ... até encontrar uma posição me memória que é um endereço. Este seria o valor de $m^\star[j]$. Este algoritmo é ilustrado na Figura 2.6 na qual as posições de memória são colocadas como células da MT, cada uma delas capaz de guardar um número constante de bits. Assume-se que o endereço destas posições cresce da esquerda para a direita.

As posições com conteúdos 3, n_2 e n_3 são endereços. A posição com o número 9 não contém um endereço. As setas indicam o endereço de memória que o endereço se refere, já com a soma apropriada do valor do endereço corrente. Por exemplo, a posição com 3 se refere à posição n_2 que está três posições adiante na memória. Se n_2 está na posição i de m ($m[i] = n_2$), então a n_3 está na posição $i + n_2$ e 3 está na posição $n_2 - 3$. Então o endereço j da posição que contém 9 é $i + n_2 + n_3$. Obrigatoriamente $n_3 < 0$. Neste tipo de máquina deve ser permitido números negativos.

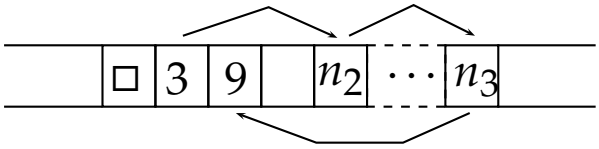

Figura 2.6: Uso de $m^\star[j]$

2.6 Uma Linguagem de Alto Nível

Implementar algoritmos em uma máquina de Turing geralmente é uma tarefa muito difícil. Apesar de adotarmos as MT´s para classificar as linguagens em relação a classes de complexidade, em alguns casos isto não é necessário. Quando o importante é saber se certa linguagem pertence a P não interessa muito qual modelo de computação utilizar para implementar um algoritmo que decide esta linguagem. Afinal, dado um algoritmo polinomial em qualquer um dos modelos "razoáveis" de computação, existe um algoritmo polinomial em qualquer dos outros modelos "razoáveis". Para estes casos, utilizaremos uma linguagem baseada em Pascal que contém apenas os comandos mais importantes: `for`, `while`, `if`, comando composto, procedimentos e funções, atribuição, etc. Para implementar não determinismo, utilizaremos o seguinte comando:

```
choose
  comando-1;
  comando-2;
  ...
  comando-n;
end
```

O comando `choose` escolhe aleatoriamente um dos comandos e o executa. Se for necessário um comando com mais de uma instrução, deve-se utilizar o comando composto `begin-end`.

Chamaremos esta linguagem de PS, Pascal Simples. Não definiremos precisamente a semântica das instruções desta linguagem, que deve ser clara pelo contexto. Uma questão mais importante é se um programa em PS pode ser traduzido para uma MT de tal forma que t passos no programa PS correspondam a t^k passos na MT, k uma constante. A resposta é sim, pois um programa em PS pode ser traduzido para uma RAM e esta pode ser simulada em tempo polinomial por uma MT. As simulações entre os diversos modelos de computação são feitas no Capítulo 3.

2.7 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.

Para expor 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 2.7.1. *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 2.7.2. *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 2.7.1. *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. □

Capítulo 3

Simulações entre os Modelos de Computação

Este Capítulo mostra como simular alguns modelos de computação em outros. Isto é, dada uma máquina em certo modelo, como construir uma máquina em outro modelo que faz a mesma coisa. Isto é sempre possível: a Tese de Church-Turing afirma que tudo o que é 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. Antes de estudar as simulações, veremos que uma máquina de Turing pode armazenar, nos seus estados, um número 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 $x_1, x_2, \ldots, x_p$ 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. volta até o início da entrada;
3. pára.

Esta MT é representada graficamente na Figura 3.

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

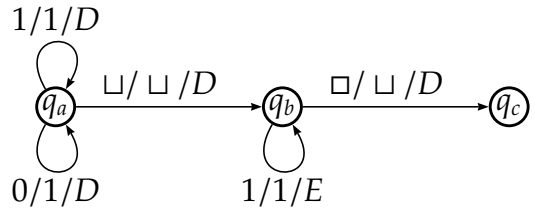

Figura 3.1: Representação de uma MT M

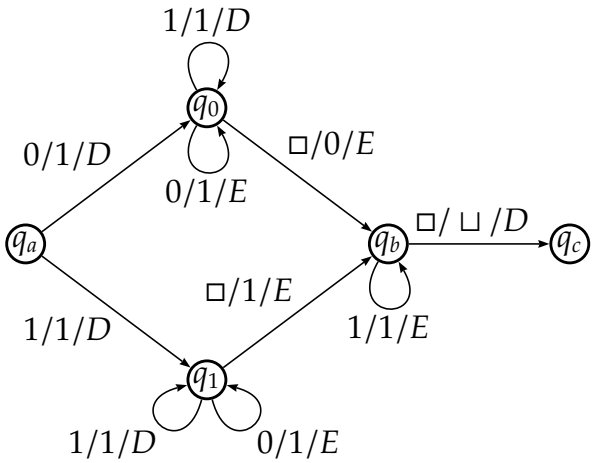

Figura 3.2: Representação de uma MT M com memória de um símbolo

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 3.2.

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 3.3.

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 3.3, 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.

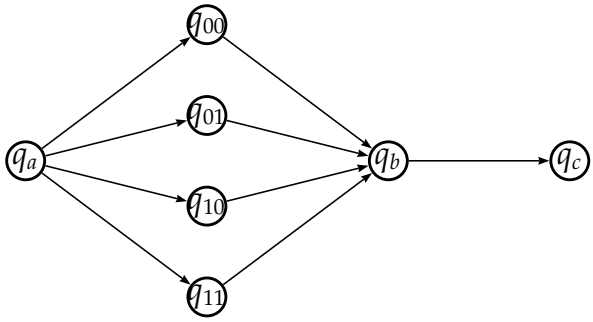

Figura 3.3: Representação incompleta de uma MT com memória para dois símbolos

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 3.2. 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.

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

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.

3.2 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 ou q_f 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 em estados q_n ou q_h é feito o retrocesso.

A Figura 3.4 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 e 2 durante a simulação (se não for encontrado um estado q_s na sub-árvore esqueda 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.

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

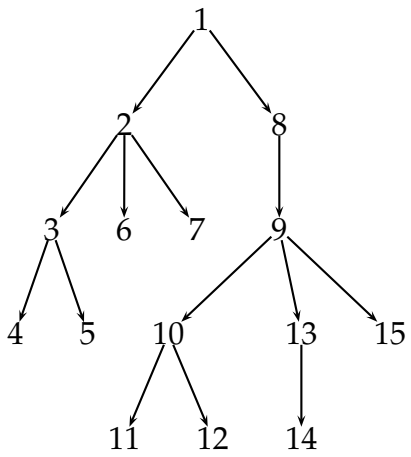

Figura 3.4: Ordem de simulação de uma MTND

passo de N. Na Figura 3.5, 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 profundidade até um número de passos igual a 2. Os vértices nomeados 2 da Figura 3.5 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.

3.3 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. 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

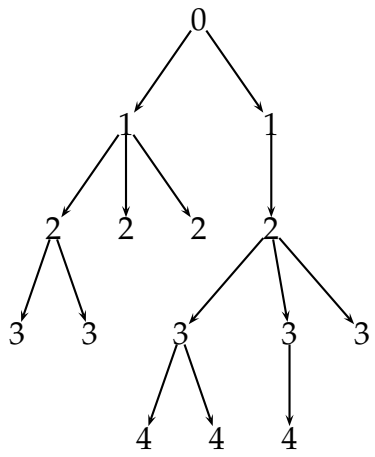

Figura 3.5: Outra simulação de uma MTND

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). Os parênteses pertencem ao alfabeto da MTU e não são realmente necessários, já que todas as intruções possuem o mesmo tamanho em número de dígitos (mesmo formato, símbolos representados pelo mesmo número de dígitos).

A descrição da máquina $M = (Q, \Sigma, I, q)$, denotada por $\langle M \rangle$, contém, na ordem, a codificação de (, $|Q|$, $|\Sigma|$, q, das instruções e), sendo que estes códigos são separados por $\sqcup$. A entrada para a MTU é $\langle M \rangle \sqcup x$. Poder-se ia eliminar q^c na descrição de M, colocando uma instrução que contenha o estado inicial como primeira instrução da codificação de M.

A MTU utiliza três fitas. Na primeira fica a entrada $\langle M \rangle \sqcup x$. 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, páre 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, q_n ou q_f.

Chamando a MTU de U, pela descrição do algoritmo acima fica claro que $U(\langle M\rangle \sqcup x) = M(x)$ 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.

3.4 Simulação de uma RAM por uma MT

Um programa em uma RAM pode ser simulado em uma máquina de Turing M de três fitas. Não tentaremos descrever M detalhadamente, o que seria uma tarefa trabalhosa e que não nos interessa. O ponto importante é a complexidade da simulação do programa da RAM por M.

A máquina M representa as posições de memória e os índices destas usando números binários. A primeira fita de M descreve o conteúdo da memória m da RAM. A memória da RAM é potencialmente infinita. Contudo, em uma computação, apenas um número finito de posições de memória é utilizado e cada posição contém um número finito. Então a primeira fita guarda, durante uma computação, *apenas as posições de memória que foram utilizadas até o momento*. As células contém pares $(i, m[i])$ como mostrado na Figura 3.6. Os números são separados pelo símbolo ●. Esta fita mostra as posições de memória 4 e 7.

A fita da Figura 3.6 simula a memória de uma RAM em que foram utilizadas, até este ponto da computação, apenas as posições 4 e 7 da memória. Cada célula mostrada neste desenho representa várias células da MT. O valor $m[4]$, por exemplo, seria representado por 100 (binário), o que ocuparia três células da MT M.

Para calcular o número máximo de células da MT necessárias para armazenar a memória da RAM, precisamos considerar duas coisas: a) o número de posições de memória da RAM a serem armazenadas em M e b) o número de símbolos necessários para armazenar cada célula. O número máximo de células de M necessárias para armazenar m será o produto destes dois valores. Inicialmente, há uma única entrada de tamanho n na RAM que ocupa n bits (consideraremos que a entrada é colocada em uma única posição de memória). A configuração

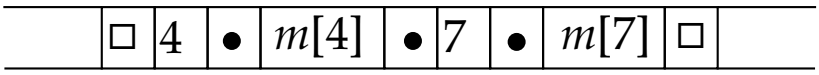

Figura 3.6: Fita da MT utilizada para armazenar a memória de uma RAM

inicial da primeira fita de M é

$$\square \bullet 0 \bullet m[0]_b \square$$

na qual $m[0]_b$ é o número $m[0]$ em binário.

Cada instrução da RAM utiliza no máximo uma posição de memória ainda não utilizada (assumiremos se uma posição de memória for lida antes de ter sido inicializada haverá um erro de execução na RAM. Ler uma posição de memória é utilizá-la do lado direito de uma instrução RAM, como $m[j]$ e $m[k]$ em $m[i] = m[j] + m[k]$). Então a execução de uma instrução RAM adiciona no máximo uma posição de memória na lista de posições de M. Depois de t passos, teremos $1 + t$ posições armazenadas na primeira fita de M.

E qual o tamanho máximo de cada uma das posições de memória $m[i]$? Suponha que no passo t da execução o tamanho máximo seja r (sempre em binário, tanto para a RAM tanto quanto para a MT M). No passo $t + 1$, este tamanho pode aumentar no máximo de um bit. As únicas instruções capazes de modificar este tamanho são as de soma e subtração. Somando ou subtraindo dois número de r bits no máximo obteremos um número de $r + 1$ bits. Como inicialmente a entrada possui n bits, depois de t passos o tamanho máximo de cada posição de memória será de $n + t$ bits.

O número de células da primeira fita de M utilizadas para armazenar a memória da RAM é aproximadamente igual ao tamanho em bits desta memória, pois ambos são armazenados em binário. Este número é $(1 + t)(n + t)$ mais o número de símbolos utilizados para separar os números, que são t, pois este é o máximo número de posições de memória armazenadas na fita menos um.[1] Então o número de células da primeira fita de M no passo t é

$$n + t + nt + t^2 + t = t^2 + 2t + nt + n = t(t + n + 2) + n$$

Assume-se que cada instrução da RAM executa em tempo constante, independente do tamanho dos números manipulados. Veremos agora, em alto nível como algumas das instruções da RAM são simuladas por M. Para simular $m[i] = 1$, M percorre a primeira fita procurando por um índice i que está armazenado nos estados de M — afinal, i é uma constante e portanto pode ser armazenda em estados (veja página 42). A comparação pode ser feita em um número de passos menor ou igual ao número de células utilizadas da fita um. Isto é, $t(t + n + 2) + n$. Se este índice não estiver na fita, ele é acrescentado com o valor 1, o que é feito em tempo constante.

Para simular $m[i] = m[j] + m[k]$, M faz uma busca pelos valores de $m[j]$ e $m[k]$ na primeira fita (em tempo $t(t + n + 2) + n$), copia estes valores para a segunda e terceira fita e os soma,

[1] há um símbolo ● para cada símbolo, exceto o último. Então são utilizados $1 + t - 1$ símbolos.

colocando o resultado na segunda fita. Os números $m[j]$ e $m[k]$ têm tamanho menor ou igual a $t(t+n+2)+n$. A soma é feita em tempo $t(t+n+2)+n+1$. A simulação desta instrução toma um tempo igual a $ct(t+n+2)+n$, no qual c é uma constante. A simulação das outras instruções é feito de forma semelhante. A MT M gasta $ct(t+n+2)+n$ para simular uma instrução no passo t.

Para simular t instruções o tempo total é

$$\sum_{i=1}^{t} i(i+n+2)+n+1 \leqslant kt^2(t+n+2)+n+1 = O(t^3)$$

para alguma constante k.

3.5 Outras Simulações

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 42). 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.

3.6 Conclusão

Este Capítulo apresentou a simulação entre diversos modelos de computação, mas não entre todos eles. Algumas simulações mais simples não foram feitas:

(a) uma MT com uma fita simulada por uma MT com k fitas, $k > 1$. Trivial, basta não usar apenas a primeira fita;

(b) uma MT simulada por uma RAM. As células de memória da RAM poderiam simular as células da MT. Cada instrução da MT poderia ser facilmente ser simulada pelas instruções da RAM.

O fato importante com relação a estes modelos de computação é que a simulação de um por qualquer outro pode ser feita em tempo polinomial. De fato, considera-se um modelo como "razoável" se ele pode ser simulado polinomialmente por uma MT. Do contrário ele não pode ser implementado fisicamente com a tecnologia atual. Possivelmente os computadores quânticos [6] não obedecem a esta regra.

Capítulo 4

Computabilidade

Computabilidade estuda o que um modelo de computação pode ou não fazer. Neste texto, usaremos as máquinas de Turing como modelo de computação para o estudo da computabilidade. Este Capítulo é importante para a compreensão dos Capítulos seguintes porque muitas das técnicas utilizadas em Classes de Complexidade se originaram em Computabilidade, um campo mais antigo. De fato, Odifreddi [8] considera o campo da complexidade computacional como Computabilidade.

Definição 4.0.1. *Uma linguagem L é recursiva se existe uma MT M de decisão tal que*

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

Isto é, M decide L.

Definição 4.0.2. *Uma linguagem L é 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 recursivamente enumeráveis é o seguinte: uma linguagem L é r.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 4.0.1. *Se L é recursiva, L é recursivamente 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 4.0.2. *Se L e L^c são recursivamente enumeráveis, então L e L^c são recursivas.*

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. □

Proposição 4.0.3. *$L \neq \emptyset$ é recursivamente enumerável se e somente se existe uma máquina de Turing que enumera todos os elementos de L.*

Demonstração. ($\Longrightarrow$) Suponha que uma MT M' de várias fitas enumera os elementos de uma linguagem $L \subset \Sigma^\star$ — 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 L seja r.e. Então existe uma MT M tal que, se $x \in L$, $M(x) = 1$ e, se $x \notin L$, $M(x) \uparrow$. E L é um subconjunto de $\Sigma^\star$ para algum alfabeto Σ finito. Então os elementos de Σ podem ser ordenados e esta ordem induz uma ordem em $\Sigma^\star$ (veja página 15). Seja $a_1, a_2, a_3 \ldots$ uma enumeração de $\Sigma^\star$ de acordo com esta ordem. A máquina de Turing de várias fitas M', descrita a seguir, enumera todos os elementos de L.

M' simula a execução de M com todos os elementos $a_1, a_2, a_3 \ldots$, mas não todos simultaneamente. Inicialmente, M' simula o primeiro passo da computação $M(a_1)$ (ou um número constante de passos), depois o segundo passo da computação de $M(a_1)$ e o primeiro de $M(a_2)$, depois o terceiro de $M(a_1)$, o segundo de $M(a_2)$ e o primeiro de $M(a_3)$ e assim por diante. Se uma computação $M(a_i)$ pára (e neste caso $M(a_i) = 1$), M' imprime a_i na fita de saída. Para qualquer $a_j \in \Sigma^\star$, em algum momento M' irá começar a computação $M(a_j)$ e, se $a_j \in L$, esta computação irá parar. Neste caso M' imprime a_j na fita de saída. Se a computação $M(a_j)$ não parar, a simulação continuará para sempre e nada será escrito (a_j não será enumerado). A computação de $M(a_k)$, $k > j$ será mais lenta mais eventualmente todos os elementos de L serão enumerados.

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

$M(a_1)_1$
$M(a_1)_2$ $M(a_2)_1$
$M(a_1)_3$ $M(a_2)_2$ $M(a_3)_1$
$M(a_1)_4$ $M(a_2)_3$ $M(a_3)_2$ $M(a_4)_2$
...

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

□

Proposição 4.0.4. *Uma linguagem $L \neq \emptyset$ é recursiva se e somente se existe uma máquina de Turing que enumera os elementos de L em ordem crescente.*

Demonstração. ($\Longleftarrow$) Aqui assumimos que $L \subset \Sigma^\star$ e a ordem crescente é a ordem induzida da ordem dos elementos de Σ (que supomos existir, veja página 15). Seja M' a MT que enumera os elementos de L em ordem crescente. Faremos uma M que decide se $x \in L$ 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 L recursiva. Há um algoritmo *elem*(i) que retorna o i-ésimo elemento de $\Sigma^\star$ na ordem lexicográfica. A MT que implementa o seguinte algoritmo enumera os elementos de L em ordem crescente.

```
i = 0;
while true do
  begin
  if elem(i) pertece a L
  then
     imprima elem(i);
  i = i + 1;
  end
```

Neste algoritmo em PS, `imprima n` imprime `n` na saída. □

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?

Proposição 4.0.5. *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 3.3 foi utilizada uma codificação que converte uma MT em um número inteiro. Assim T é um subconjunto infinito de $\mathbb{N}$ e portanto T é enumerável (Proposição 1.1.6).

O conjunto de todas as linguagens sobre Σ, que é $2^{\Sigma^\star}$, é equipotente a $[0,1]$ pela Proposição 1.3.3, que não é enumerável. Então

$$\begin{array}{rcl} 2^{\Sigma^\star} & \sim & [0,1] \\ T & \sim & \mathbb{N} \end{array}$$

Como $[0,1]$ não é equipotente a $\mathbb{N}$, T não é equipotente a $2^{\Sigma^\star}$ também.

Existe uma função injetora entre $\mathbb{N}$ e $[0,1]$, a saber, $f(n) = 0.\overline{n}2$, no qual $\overline{n}$ é uma cadeia de símbolos representando n em binário ($f(5) = 0.1012$, $f(10) = 11102$). Portanto $\mathbb{N} \preceq [0,1]$. Mas como $\mathbb{N}$ não é equipotente a $[0,1]$, não temos $[0,1] \preceq \mathbb{N}$ (se tivéssemos, pelo Teorema 1.1.1 teríamos $\mathbb{N} \sim [0,1]$). Logo o conjunto $\mathbb{N}$ é de certa forma "menor" do que $[0,1]$. Logo, qualquer função $f : [0,1] \longrightarrow \mathbb{N}$ não pode ser injetora. Como $T \sim \mathbb{N}$ e $2^{\Sigma^\star} \sim [0,1]$, não existe função injetora

$$g : 2^{\Sigma^\star} \longrightarrow T$$

Há mais linguagens do que máquinas de Turing disponíveis.

Cada máquina de Turing calcula uma função de $\mathbb{N}$ em $\mathbb{N}$. Então podemos associar cada MT a uma destas funções. Mas o conjunto das máquinas de Turing é enumerável e o conjunto de todas as funções $f : \mathbb{N} \longrightarrow \mathbb{N}$ é equipotente a $\mathbb{R}$ (Proposição 1.1.12). Logo há mais funções do que MT disponíveis para calculá-las. □

Sendo R o conjunto das linguagens recursivas e RE o conjunto das linguagens r.e., temos que $R \subset RE$ pela Proposição 4.0.1. Mas haverá alguma linguagem RE que não seja recursiva? A resposta é sim.

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 3.3.

Proposição 4.0.6. *A linguagem $H = \{\langle M \rangle \sqcup x : M(x) \downarrow\}$ pertence a $RE - R$.*

Demonstração. Provaremos que $H \in RE$. Para tanto, pela Proposição 4.0.2, basta encontrar uma MT U 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 U é a MTU com uma modificação: ao terminar de simular M com entrada x (se terminar), M imprime 1 na fita de saída e vai para o estado q_s. Se $M(x)$ não terminar a execução, a MTU não terminará a execução também, que é o comportamento esperado de uma MT que aceita uma linguagem r.e.

A prova de que $H \notin R$ é mais complexa. Provaremos por contradição. Assumiremos que exista um algoritmo *Para* que decida H e chegaremos a uma contradição. Para simplificar a prova, usaremos a linguagem de alto nível PS descrita na Seção 2.6. O algoritmo *Para* é uma função

```
function Para(<A>; x) : boolean
```

que, dada a descrição de um procedimento ou função `A` com uma entrada `x`, retorna `true` se `A` pára a sua execução com a entrada `x` e `false` caso contrário. Usamos `<A>` para a descrição do procedimento ou função `A`. Um procedimento ou função pode ser facilmente codificado em um número inteiro. O texto do procedimento/função pode ser expresso utilizando a codificação ASCII que associa a cada caráter um número entre 0 e 127 e que pode ser representado por sete bits. Colocando-se os valores ASCII de cada caráter de um procedimento/função obtemos um número que identifica univocamente esta subrotina. Por exemplo,
`procedure Q(T) ... end` é representado pelo número

$$11100001110010110111 \ldots 10100010101000101010001010 01 \ldots 1100100$$

Confira:

p	r	o	cedure	Q	(	T	)	...	d
1110000	1110010	1101111	...	1010001	0101000	1010100	0101001	...	1100100

Assumindo a existência de `Para`, podemos construir um procedimento `Q`:

```
procedure Q( T )
   begin
   while Para(T, T) do
      ;
   end
```

Este é um procedimento completamente válido em PS. O `;` depois da instrução `while` é um comando vazio — o `while` repete enquanto `Para(T, T)` for `true` sem executar nenhum comando.

O que acontece quando chamamos `Q` passando a sua própria descrição?

```
Q(<Q>)
```

Neste caso, o comando `while` em `Q` significa simplesmente "não pare a execução desta chamada de `<Q>`, `Q(<Q>)`, se e somente se `Q(<Q>)` termina a sua execução". Ou seja, "não pare se e somente se pare". De qualquer forma, `Q` parando ou não, há uma contradição.

A contradição só apareceu porque usamos `Q` de duas formas diferentes mas equivalentes. Na chamada `Q(<Q>)`, o primeiro `Q` é uma chamada de procedimento, que acontece durante a execução do programa. O segundo `Q`, em `<Q>`, é a descrição do procedimento, que pode ser mesmo o seu texto, "`procedure Q(T) begin ... end`". O parâmetro `T` é `<Q>` e o seu primeiro uso na chamada de `Para`, `Para(T, T)` é utilizado por `Para` para descobrir se `Q` pára ou não — ou seja, `<Q>` é tratado como um procedimento, não como um texto. O segundo uso de `Q` em `Para` é o parâmetro para o primeiro uso e espelha `<Q>` na chamada `Q(<Q>)`. Então a descrição de `Q`, `<Q>`, em `Q(<Q>)` é utilizado pela execução de `Para(<Q>, <Q>)` como se fosse o próprio `Q`.

Usualmente imagina-se `Para` como se ele fosse uma MTU que simulasse `Q(<Q>)` e então `<Q>` é empregado como se fosse o próprio procedimento — dado utilizado como se fosse programa. Mas `Para` não precisaria ser assim. Ele poderia fazer uma análise do procedimento ou função empregando outro tipo de algoritmo para decidir se ele pára ou não. □

Este tipo de prova empregou sutilmente o método da diagonalização. Para ficar mais claro como a diagonalização foi utilizada, modificaremos o procedimento `Q`:

```
procedure Q( T, V )
   begin
   while Para(T, V) do
      ;
   end
```

`Q` agora é chamado como `Q(<Q>, <Q>)`. Se `Para(<Q>, <Q>)` retornar `true` indicando que `Q` pára com a entrada `<Q>`, então `Q` não pára. Se `Para(<Q>, <Q>)` retornar `false` indicando que `Q` não pára a sua execução o `while` falha e `Q` pára.

A diagonalização é feita com linhas com procedimentos e colunas com entradas. Ambos são enumeráveis, pois o conjunto de todas as entradas é $\mathbb{N}$ e o conjunto dos números de todos os procedimentos/funções é infinito e um subconjunto de $\mathbb{N}$.

	0	1	2	3	...
P_0	$P_0(0)$	$P_0(1)$	$P_0(2)$	$P_0(3)$	...
P_1	$P_1(0)$	$P_1(1)$	$P_1(2)$	$P_1(3)$	...
P_2	$P_2(0)$	$P_2(1)$	$P_2(2)$	$P_2(3)$	...
P_3	$P_3(0)$	$P_3(1)$	$P_3(2)$	$P_3(3)$	...
...	...	...	...	...	$\ddots$

O conjunto $RE - R$ não é vazio, temos pelo menos H neste conjunto. Então existem pelo menos duas classes de linguagens: RE e R. E quanto à linguagem H^c? Ela não pode ser recursiva, pois se fosse H também o seria. Não pode ser r.e. também, pois pela Proposição 4.0.2 H seria recursiva. Então $H^c \notin R$ e $H^c \notin RE$. Esta linguagem pertence à classe *coRE*, o complemento da

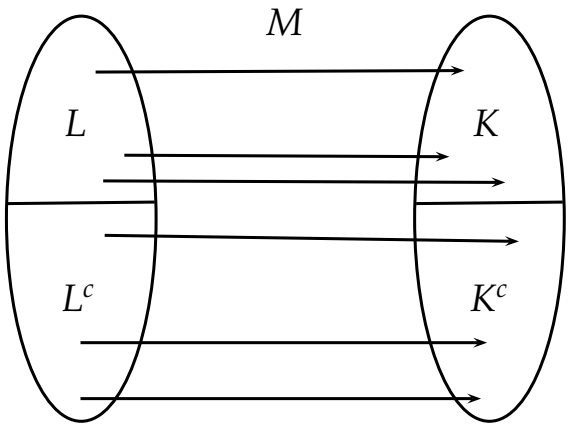

Figura 4.1: m-redução entre L e K

classe das linguagens recursivamente enumeráveis:

$$coRE = \{L : L^c \in RE\}$$

Definição 4.0.3. *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,

$$\begin{aligned} x \in L &\implies M(x) \in K \\ x \in L^c &\implies M(x) \in K^c \end{aligned}$$

Veja a Figura 4.1. 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 4.2.

Proposição 4.0.7. *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.

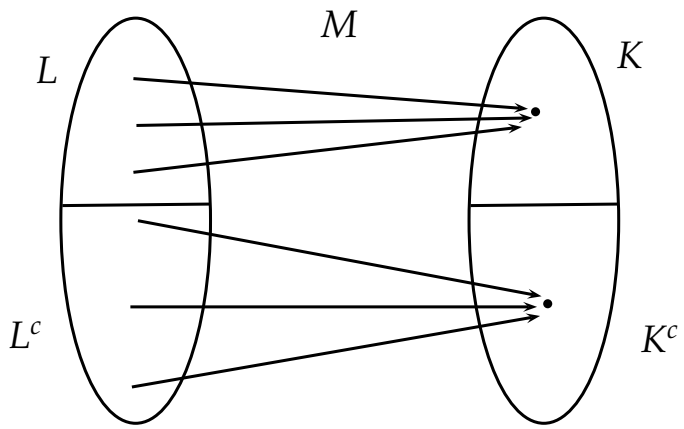

Figura 4.2: Exemplo de m-redução entre L e K que utiliza apenas um elemento de K e um de K^c

Proposição 4.0.8. *A linguagem* $HZ = \{M : M(0) \downarrow\}$ *não é recursiva.*

Demonstração. $M \in HZ$ 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 $M \sqcup x \in H$, P constrói uma MT M' que simula a execução de M com entrada x. A entrada de M' é ignorada por esta máquina. Então

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

pois $M \sqcup x \in H$ sse $M \sqcup x \downarrow$ 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 4.0.9. *Toda linguagem recursivamente enumerável é redutível a H.*

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

$$\begin{array}{ll} \text{se } x \in L & \text{então } P(x) = 1 \\ \text{se } x \notin L & \text{então } P(x) \uparrow \end{array}$$

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,

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

□

A m-redução entre linguagens L e K não é uma técnica poderosa. Como se permite que a função de redução seja implementada por qualquer máquina de Turing, qualquer transformação computável é valida, sendo possível anular quaisquer diferenças entre L e K. Isto é o que basicamente afirma a próxima proposição.

Proposição 4.0.10. *Se $K \neq \emptyset$ e $K \neq \Sigma^\star$, então qualquer linguagem decidível sobre Σ é m-redutível a K.*

Demonstração. Como $K \neq \emptyset$ e $K \neq \Sigma^\star$, há pelo menos um elemento em K e pelo menos um elemento em K^c (que é $\Sigma^\star - K$). Este fato é utilizado para mapear todos elementos de L em um elemento de K e todos os elementos de L^c ($\Sigma^\star - L$) em um elemento de K^c. Seja x_{in} um elemento de K e x_{out} um elemento de K^c. A função computável de redução é

$$f(x) = \begin{cases} x_{in} & \text{se } x \in L \\ x_{out} & \text{se } x \in L^c \end{cases}$$

Observe que podemos usar $x \in L$ e $x \in L^c$ nesta definição porque L é decidível. □

Capítulo 5

Classes de Espaço e Tempo

Este Capítulo apresenta as definições das mais importantes classes de complexidade de espaço e tempo e as relações entre elas. Também é demonstrado que existe uma hierarquia entre as linguagens em relação à complexidade: para cada função $f(n)$, há um problema que não pode ser resolvido em tempo $f(n)$.

Usaremos apenas linguagens sobre o conjunto $\{0, 1\}$. Isto é, todas as linguagens deste capítulo são subconjuntos de $\{0, 1\}^\star$.

5.1 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 5.1.1. *Uma linguagem L pertence a TIME(f) se existe uma MT M de decisão 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 5.1.2. *Uma linguagem L pertence a SPACE(f) se existe uma MT M que toma uma entrada x 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). Ou seja, M executa em espaço $cf(n)$.*

Definição 5.1.3. *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 5.1.4. *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 5.1.5. *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

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

A próxima proposição diz que se uma MT que executa em tempo $f(n)$, então existe uma MT equivalente que executa em tempo $f(n)/k + n + 2$. Por exemplo, se tivermos uma MT que executa em tempo $f(n) > n$, como $f(n) = 5n^3$, sempre é possível fazer uma MT equivalente a esta que executa em tempo n^3. Em resumo, as constantes multiplicativas não são importantes: se $f(n) > n$ e $L \in \text{TIME}(cf(n))$, c constante, então $L \in \text{TIME}(f(n))$. Contudo, para a função

$f(n) = cn$, a constante c pode ser abaixada arbitrariamente para perto de 1 sem contudo nunca atingir este limite. Isto é, dada uma MT M que executa em tempo $\leqslant n$, pode-se construir uma MT M' que executa em tempo $(1 + \epsilon)n$ para qualquer $\epsilon > 0$ (dado o ϵ desejado, construimos a M' apropriada).

Proposição 5.1.1. *([9]) Se $L \in TIME(f(n))$, então $L \in TIME(f(n)/k + n + c)$ para qualquer constante k, no qual c é uma constante.*

Demonstração. Dada uma MT M com uma única fita que decide $L \in \text{TIME}(f(n))$, faremos uma MT M' com duas fitas que é equivalente a M e que executa em tempo $f(n)/2$ (a generalização para k qualquer e M com várias fitas é direta e não será feita). A idéia da prova é simular várias instruções de M em um único passo de M'. Mas quantas instruções devem ser simuladas em um único passo para que M' execute em tempo $f(n)/2$? Aparentemente, seria necessário simular duas de M em M'. Mas isto não funciona, pois não considera os passos auxiliares que M' tem que fazer para simular M. Vejamos como é feita a simulação.

Considere $M = (Q, \Sigma, I, q_0)$ e $M' = (Q', \Sigma', I', q)$. m símbolos de M serão comprimidos em um único símbolo de M' (o valor de m será definido posteriormente). Para tanto, M' utiliza $|\Sigma|^m$ símbolos, cada qual representando uma combinação de símbolos de M. Estes símbolos são agrupados em um conjunto Σ_m. Por exemplo, se m células em sequência contém os símbolos $s_1 s_2 \ldots s_m$, estes símbolos serão representados por um símbolo $s \in \Sigma_m$ que representaremos pela m-tupla $(s_1, s_2, \ldots s_m)$. O número de símbolos necessários para representar todas as tuplas é $|\Sigma|^m = |\Sigma_m|$. Além disso, M' tem que ter símbolos iguais aos de M pois a entrada para ambas deve ser codificada igualmente. Então Σ' terá $|\Sigma| + |\Sigma|^m$ elementos e $\Sigma' = \Sigma \cup \Sigma_m$.

Inicialmente M' copia a entrada (que contém apenas símbolos de Σ) da primeira fita para a segunda fita, mas trocando os símbolos de m células consecutivas por um único símbolo de Σ_m. Sendo $n = |x|$, esta cópia toma $n + c$ passos. Esta constante c é a que aparece no enunciado da proposição. A cada movimento da cabeça de leitura/gravação da primeira fita, o estado de M' muda para armazenar (no estado) o conjunto de símbolos lido até agora (veja como armazenar um número constante de símbolos nos estados na página 42. Aqui, m é constante, independe de x, depende apenas de M'). Para armazenar m símbolos nos estados, M' deve ter um número de estados igual a

$$\sum_{i=0}^{m} |Q|\,|\Sigma|^i$$

Cada estado de M' pode ser representado por uma tupla $(q, s_1, s_2, \ldots s_m)$ no qual $q \in Q$ e $s_i \in \Sigma \cup \{\epsilon\}$. O símbolo $\epsilon \notin \Sigma$ é utilizado no caso em que nenhum símbolo da entrada está sendo armazenado no estado de M'.

M' armazena m símbolos de M em um único símbolo de Σ'. A fita desta última máquina é dividida em blocos de m símbolos que são manipulados de uma única vez por M', que simula m instruções de uma única vez. E m instruções de M podem mover a cabeça de leitura/gravação para qualquer célula entre dois limites:

(a) m células à esquerda da primeira célula do bloco (se a cabeça de leitura/gravação está na primeira célula do bloco e todas as instruções de M movem a cabeça para a esquerda);

(b) m células à direita da última célula do bloco (se a cabeça de leitura/gravação está na última célula do bloco e todas as instruções movem a cabeça para a direita).

Contudo, a posição corrente da cabeça de leitura/gravação de M deve ser armazenada em algum lugar em M'. Não pode ser na posição corrente da cabeça de M', pois vários símbolos de M são comprimidos em um único de M' — é como se a cabeça de leitura/gravação se movesse saltando sempre m células da fita de M. A solução é armazenar esta posição no estado corrente de M'. Para tanto usaremos $m\,|Q|$ novos estados que serão representados por tuplas (q, i), no qual $1 \leqslant i \leqslant m$ é a posição da cabeça de leitura/gravação de M dentro do bloco representado pela célula corrente de M'.

Agora descreveremos como M' simula M. m movimentos de M são simulados por t movimentos de M', no qual t é constante e menor do que m. Se fizermos $m = 2t$, teremos efetivamente uma máquina M' que executa em metade do tempo que M. Um símbolo de M' representa m símbolos de M e simularemos m passos de M tem t passos de M'. Mas m passos de M podem levar a cabeça de leitura/gravação para qualquer célula m posições à esquerda ou à direita da célula corrente. Em M', depois de simulados m passos de M a cabeça de leitura/gravação estará na posição atual ou uma posição à esquerda ou uma à direita. Para descobrir quais alterações têm que ser feitas na fita de M, esta máquina armazena as informações da célula à esquerda, da célula corrente e da célula à direita da célula atual. Isto pode ser feito em quatro movimentos da cabeça de leitura/gravação de M': esquerda, direita, direita, esquerda. Estas informações são suficiente para M' predizer quais os próximos m movimentos de M. A simulação destes movimentos são feitos modificando-se a célula corrente e possivelmente a célula esquerda ou direita, mas não ambas. Não é possível que sejam ambas porque em m movimentos a cabeça da máquina M não conseguiria ir do bloco corrente de m símbolos para o bloco da direita e depois voltar para o bloco da esquerda. Seriam necessários pelo menos $m + 1$ movimentos para isto. É possível que seja necessário modificar apenas a célula corrente de M' (se a cabeça de leitura/gravação de M' permanecer no bloco corrente). No máximo, a alteração das células de M' tomará dois passos. Somando-se aos quatro passos necessários para guardar informações sobre o que fazer, temos um total de 6 passos, no máximo.

E como as informações armazenadas nas células corrente de M' e à esquerda e à direita desta são armazenadas para que se possa prever os m próximos movimentos de M'? Estas informações podem ser armazendas no estado corrente de M', já que o número de bits necessários é constante (veja página 42). Então são necessários criar novos estados para isto. Estudemos primeiro que informações serão guardadas. É necessário guardar o estado corrente de M e o contéudo de três células de M' ($3m$ células de M, cada uma podendo guardar um símbolo de Σ). E mais a posição da célula corrente de M na célula corrente de M'. A célula corrente de M' está representada exatamente na célula corrente de M', mas ela pode estar em qualquer das m possíveis posições.

Então para armazenar todas estas informações no estado corrente de M' são necessários

$$|Q|\,\Sigma^{3m}m$$

novos estados.

□

5.2 Hierarquia de linguagens

Dada uma função $g(n) \geqslant n$, haverá problemas que não podem ser resolvidos em tempo $g(n)$? As próximas proposições provam que não [9].

Lema 5.2.1. *Dada uma função $f(n) \geqslant n$, a linguagem*

$$H_f = \{\langle M\rangle \sqcup x : M \text{ aceita entrada } x \text{ em um número de passos } \leqslant f(|x|)\}$$

pertence a TIME($f(n)^3$). Lembre-se de que $\langle M\rangle$ é a descrição 3.3 de M.

Demonstração. Mostraremos apenas o esboço da construção uma MT U_f que decide H_f. Isto deverá ser suficiente para convencer o leitor de que esta linguagem pode ser decidida em $cf(n)^3$ passos, com c constante.

U_f toma $M \sqcup x$ como entrada e simula M com a entrada x como foi feito na máquina de Turing Universal. Cada passo de M é simulado por $O(f(n)^2)$ passos de U_f. Como M deve executar no máximo por $f(n)$ passos, a máquina U_f tomará no máximo $O(f(n)^3)$ passos. Só resta saber como U_f decide parar e retornar 0 (não pertence) ou 1 (pertence à H_f).

Se a simulação da execução de $M(x)$ terminar antes de $f(|x|)$ passos e $M(x) = 1$, então U_f retorna 1. Se a simulação gastar mais do que $f(|x|)$ passou ou $M(x) = 0$, U_f retorna 0. Para U_f saber quantos passos já foram simulados de M, U_f preenche inicialmente uma de suas fitas com $f(|x|)$ símbolos $\sqcup$, voltando a cabeça de leitura/gravação desta fita para a posição inicial. Depois da simulação de um passo de M a cabeça desta fita move-se para a direita. Quando esta cabeça estiver em uma célula com o símbolo □, então U_f terá simulado $f(|x|)$ passos de M. □

Lema 5.2.2. $H_f \notin f(\lfloor n/2 \rfloor)$

Demonstração. Fixado uma função f e portanto uma linguagem H_f, iremos supor que existe uma MT M que decide H_f. Intuitivamente M deveria pelo menos $f(n)$ passos, pois ela precisa de alguma forma simular a execução das MT que executam em um número de passos $\leqslant f(n)$.

Suponha então que exista M que decide H_f em tempo $f(\lfloor n/2 \rfloor)$:

$$N \sqcup x \in H_f \text{ sse } M(N \sqcup x) = 1$$

e M gasta um número de passos $\leqslant f(\lfloor n/2 \rfloor)$. Então

$$H_f \in \text{TIME}(f(\lfloor n/2 \rfloor)) \subset \text{TIME}(f(n))$$

Construiremos uma MT D da seguinte forma:

$$D(\langle N \rangle) = 1 - M(\langle N \rangle \sqcup \langle N \rangle)$$

Fato: D executa em tempo $f(n)$; isto é o número de passos que D gasta com qualquer entrada x é $\leqslant f(|x|)$.

Prova: se o tamanho da entrada $\langle N \rangle$ é k, o tamanho da entrada passada à chamada $M(\langle N \rangle \sqcup \langle N \rangle)$ é $2k+1$. Como M executa em tempo $f(\lfloor n/2 \rfloor)$, D executa em tempo $f(\lfloor \frac{2k+1}{2} \rfloor) = f(k)$. Renomeando k para n, temos que D executa em tempo $f(n)$ (o número de passos é menor ou igual a $f(n)$). Agora estudaremos a chamada a D passando D como parâmetro, a diagonalização de M.

Se $D(\langle D \rangle) = 0$, então $M(\langle D \rangle \sqcup \langle D \rangle) = 1$, o que quer dizer que $\langle D \rangle \sqcup \langle D \rangle \in H_f$; isto é, D aceita $\langle D \rangle$ em um número de passos $\leqslant f(|\langle D \rangle|)$. Isto é o mesmo que $D(\langle D \rangle) = 1$, o que é o contrário da hipótese inicial.

Se $D(\langle D \rangle) = 1$, então $M(\langle D \rangle \sqcup \langle D \rangle) = 0$, o que quer dizer que $\langle D \rangle \sqcup \langle D \rangle \notin H_f$. Há dois casos em que isto pode ocorrer:

(a) D não aceita $\langle D \rangle$ ($D(\langle D \rangle) = 0$). Então temos uma contradição, pois supomos inicialmente que $D(\langle D \rangle) = 1$;

(b) D aceita $\langle D \rangle$ em um número de passos $> f(|\langle D \rangle|)$. Contradição, pois pelo Fato D gasta $\leqslant f(n)$ passos para qualquer entrada de tamanho n.

□

Teorema 5.2.1. *Se $f(n) \geqslant n$, então TIME($f(\lfloor n/2 \rfloor)$) $\subsetneq$ TIME($f(n)^3$). Ou equivalentemente, TIME($f(n)$) $\subsetneq$ TIME($f(2n+1)^3$).*

Demonstração. $H_f \in \text{TIME}(f(n)^3)$ pelo Lema 5.2.1 e $H_f \notin \text{TIME}(f(\lfloor n/2 \rfloor))$ pelo Lema 5.2.2. Como $\text{TIME}(f(\lfloor n/2 \rfloor)) \subset \text{TIME}(f(n)^3)$, temos $\text{TIME}(f(\lfloor n/2 \rfloor)) \subsetneq \text{TIME}(f(n)^3)$. □

Proposição 5.2.1. *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. □

Proposição 5.2.2. *NTIME($f(n)$) $\subset$ SPACE($f(n)$).*

Demonstração. Esta proposição afirma que uma linguagem que é decidida por uma MTND em tempo $f(n)$ também é decidida por uma MTD em espaço $f(n)$. Isto é, se $L \in$ NTIME($f(n)$), então $L \in$ SPACE($f(n)$). Se $L \in$ NTIME($f(n)$), existe uma MTND N tal que

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

A altura máxima da árvore $Ar(N(x))$ é $cf(|x|)$ para certa constante c. Uma execução não determinística $N(x)$ percorre um único caminho entre a raiz e uma folha.

Construiremos uma MTD M que decide L e que utiliza um número de células $\leqslant cf(|x|)$. M, com entrada x, simula todas as computações da árvore $Ar(N(x))$, o que é equivalente a fazer uma busca, começando na raiz, por um vértice que corresponda ao estado q_s de aceitação. Se esta busca falha (nenhum q_s é encontrado em nenhuma folha), então M vai para o estado q_n e retorna 0. Utilizando busca em profundidade, o máximo número de vértices empilhados na pilha de busca é igual à altura da árvore, $cf(|x|)$ (esta é a altura máxima porque a MTND N pertence a NTIME($f(n)$) e cada passo de execução de N corresponde a um vértice a mais na árvore).

Ao fazer a busca em profundidade, M não precisa armazenar os vértices de $Ar(N(x))$, que são configurações de N. Basta armazenar as escolhas não determinísticas feitas em cada passo. Aqui admitiremos que N possui exatamente duas escolhas não determinísticas em cada passo — veja página 33. Cada máquina não determinística pode ser transformada em uma com exatamente duas escolhas e o tempo de execução desta nova máquina é uma constante vezes o tempo de execução da máquina original.

M armazena números 0 ou 1 que correspondem às escolhas não determinísticas feitas por N em cada passo. São necessários $cf(n)$ destes números, pois esta é a altura da árvore. Então M utiliza $cf(|x|)$ células na simulação de N (porque este é o tempo máximo de execução e cada passo utiliza no máximo uma nova célula) mais $cf(n)$ para guardar os números 0 ou 1 correspondentes às escolhas não determinísticas. Então M utiliza no máximo $2cf(n)$ células e $L \in$ SPACE($f(n)$).

□

Proposição 5.2.3. *NSPACE($f(n)$) $\subset$ TIME($c^{f(n)+\log n}$).*

Demonstração. A proposição diz que, se $L \in$ NSPACE($f(n)$), então $L \in$ TIME($c^{f(n)+\log n}$). Dada uma MTND N que decide L em tempo $f(n)$, construiremos uma MTD M que decide L em tempo $c^{f(n)+\log n}$ para algum k. Dado x, temos uma árvore de configuração $Ar(N(x))$ que utiliza no máximo espaço $f(|x|)$ em cada uma das computações possíveis de $N(x)$. Cada computação possível com entrada x, em cada possível escolha não determinística, é um caminho da raiz de $Ar(N(x))$ até uma folha. Sabemos que este caminho utiliza no máximo $cf(n)$ células, mas não sabemos quantos passos são utilizados. Pode ser um número arbitrariamente grande, muito

maior do que $cf(n)$, pois as células da MT N podem ser reutilizadas. Contudo, o número de passos tem um limite bem definido. Vejamos que limite é este.

A MTND N sempre pára a sua computação com qualquer entrada — esta é uma condição para que L pertença a NSPACE($f(n)$). Então duas configurações de N com entrada x nunca se repetem. Como o espaço utilizado por $N(x)$ é finito, o número de configurações é finito também. Uma máquina determinística que simula N com entrada x só tem que considerar todas estas configurações. É isto que faremos.

Construiremos uma MTD M que, com entrada x, percorre a árvore $Ar(N(x))$ até encontrar uma configuração com estado q_s de aceitação em N. Se encontrar, M vai para o estado q_s e pára. Se percorrer toda a árvore e não encontrar estado q_s em N, M vai para o estado q_n e pára. Mas quantos passos, no máximo, M deverá executar? O pior caso é percorrer toda a árvore e não encontrar estado q_s em N. Então o pior caso depende do número de configurações de $Ar(N(x))$. É este número que calcularemos.

Uma configuração (veja 2.2.4) 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. A string a_i pode ser vazia mas b_i contém pelo menos o símbolo da posição corrente da cabeça de leitura/gravação. A fita 1 é a de entrada e k a de saída. A classe NSPACE é definida em termos de máquinas de Turing com entrada e saída. O espaço utilizado não considera a entrada e a saída. Como utilizamos máquinas de decisão, a saída utiliza uma única célula e pode ser desconsiderada no cálculo do espaço. O espaço utilizado pode até ser menor do que n, pois não se considera a fita de entrada. Mas para armazenar a configuração, temos que guardar a posição da cabeça de leitura/gravação na primeira fita. Há n posições possíveis, assumindo que a cabeça está sempre sob uma célula da entrada e $n = |x|$. Então uma configuração será representada por $(q, i, b_2, a_2, \ldots b_{k-1}, a_{k-1})$.

Sendo $N = (Q, \Sigma, I, q)$, o número de configurações NC é igual a

$$|Q|\, n \Sigma^{(k-2)f(n)}$$

Temos que $n = \Sigma^{\log_\Sigma n} = \Sigma^{\frac{\log n}{\log \Sigma}}$. Colocando $c_1 = 1/\log \Sigma$, temos

$$NC = |Q|\, n \Sigma^{(k-2)f(n)} = |Q|\, \Sigma^{c_1 \log n} \Sigma^{(k-2)f(n)} = |Q|\, \Sigma^{c_1 \log n + (k-2)f(n)}$$

Colocando $c_2 = \text{máx}\{c_1, k-2\}$, temos

$$NC \leqslant |Q|\, \Sigma^{c_2(\log n + f(n))} = |Q|\, c_2^{\log n + f(n)}$$

A máquina M determinística fazer uma busca em profundidade em uma árvore $Ar(N(x))$ com no máximo $|Q|\, c_2^{\log n + f(n)}$ vértices e encontrar um vértice com estado q_s (se existir). O tamanho

de cada configuração, correspondente a um vértice, é no máximo $c'cf(n)$, pois N utiliza no máximo $cf(n)$ células (porque L pertence a NSPACE($f(n)$)). A codificação destas células em uma configuração $(q, i, b_2, a_2, \ldots b_{k-1}, a_{k-1})$ é no máximo uma constante c' versus o número de células $cf(n)$ (se o número de símbolos utilizados na codificação for m e esta for feita em binário, cada símbolo terá $1 + \lfloor \log_2 m \rfloor$ dígitos).

A máquina M necessita de uma estrutura de dados que contém a árvore $Ar(N(x))$. Esta estrutura pode ser construída sob demanda. Sempre que a configuração (vértice) atual for C, M podemos construir os próximos dois vértices, se C não for final. E em quanto tempo isto pode ser feito? Dada uma configuração não final C (estado diferente de q_s, q_n e q_f), os próximos dois estados dependem das instruções de N (tamanho constante em relação a x), do estado q da configuração e das células correntes de todas as fitas de C. As células correntes de todas as fitas estão representadas em C,[1] exceto a da primeira fita, a da entrada. Tudo o que temos é um índice i a partir do qual o valor da célula corrente tem que ser recuperado da entrada de M (que é a mesma entrada de N). Para recuperar o valor da célula corrente da entrada, contamos as células do início da entrada até a posição i. Como $1 \leqslant i \leqslant |x|$, este passo toma um tempo linear no tamanho da entrada x.

O cálculo das configurações que se seguem a C pode ser feito claramente em tempo linear no tamanho de C. Vejamos porque. M percorre C coletando as células correntes de cada fita de N (tempo linear). Então, usando o estado corrente de N e estas células, M consulta as instruções de N para descobrir quais instruções estão habilitadas. A modificação da configuração é feita percorrendo-se novamente a configuração C e modificando-a para produzir duas novas configurações C' e C''.

Como o máximo tamanho de C é $c'cf(n)$, o cálculo das duas próximas configurações pode ser feito em tempo $\leqslant c_3 n + c_1 f(n)$ (o n corresponde à busca pela célula corrente da entrada x). A busca em $Ar(N(x))$ toma um tempo linear no número de vértices, $|Q|\, c_2^{\log n + f(n)}$, sendo que a construção dos dois próximos vértices toma $c_3 n + c_1 f(n)$. Então o número de passos é menor ou igual a

$$(c_3 n + c_1 f(n))\, |Q|\, c_2^{\log n + f(n)} \leqslant c^{f(n) + \log n}$$

□

Proposição 5.2.4. *Dado um grafo $G = (V, E)$ e dois vértices $s, t \in V$, existe uma MT que descobre se existe um caminho entre s e t em espaço* $\log^2 |V|$.

Demonstração. A entrada para a MT é $\langle G \rangle \sqcup s \sqcup t$ na qual $\langle G \rangle$ é a representação de G em forma de matriz de adjacência ($A_{ij} = 1$ sse a aresta $(i, j) \in E$). Sendo $n = |V|$, o número de vértices, o número de arestas $|E|$ é $\leqslant n^2$. Então o tamanho da entrada é $|G| + 1 + \log n + 1 + \log n \leqslant n^2 + 2 \log n + 2 = O(n^2)$.

[1] O valor da célula corrente da fita de saída será 0, 1 ou □ se o estado corrente for q_n, q_s ou diferentes destes dois, respectivamente.

Cada vértice é representado por um número entre 1 e $|V| = n$ e portanto são necessários $\log n$ bits para representar um vértice.

Contruiremos uma MT M tal que $M(G, x, y, i) = 1$ se e somente se há um caminho dirigido em G de x para y de tamanho $\leqslant 2^i$. Não nos importaremos no tempo que o algoritmo tomará. O objetivo é economizar espaço. Note que, se há um caminho entre x e y de tamanho $\leqslant 2^i$, há um vértice z tal que há um caminho entre x e z de tamanho $\leqslant 2^{i-1}$ e um caminho entre z e y de tamanho $\leqslant 2^{i-1}$. Podemos escolher um z qualquer dentre todos os vértices e testar se há um caminho entre x e z e depois se há um caminho entre z e y. É isto o que o algoritmo faz.

M utiliza uma pilha para armazenar os vértices intermediários — mas não todo o caminho, o que tomaria espaço $n \log n$ (n vértices cada um ocupando $\log n$ bits). M armazena apenas os vértices intermediários que estão nas posições $2^m \leqslant n$ do caminho. Então, se o caminho tem tamanho $k \leqslant n$, são armazenados os vértices da posição $k, k/2, k/2^2, \ldots 2, 1$. Então são armazenados $\log k \leqslant \log n$ vértices, cada um ocupando $\log n$ bits, o que dá $\log^2 n$ bits no total. Vejamos o algoritmo em Pascal Simples:

```
function ConectadoLimitado(G, x, y, i) : boolean
begin
   { usamos E(x, y) para (x, y) ∈ E, sendo G = (V, E) }
if i == 0 then return E(x, y) = 1;
for z = 1 to n do
   if z <> x and z <> y then
       if ConectadoLimitado(G, x, z, i - 1) and
          ConectadoLimitado(G, z, y, i - 1)
       then
          return true;
return false;
end
```

Este algoritmo faz no máximo $\log_2 n$ recursões pois a cada chamada recursiva o número máximo de vértices permitido no caminho cai pela metade. Então o tamanho máximo da pilha utilizada é $\log n$. Esta função passa G como parâmetro em cada chamada, uma operação desnecessária que não é utilizada na máquina de Turing M que implementa este mesmo algoritmo.

A MT M utiliza uma pilha contendo apenas os parâmetros (x, y, i). Após executar o algoritmo com $(x, z, i-1)$, esta tripla é apagada e o mesmo espaço é utilizado na execução do algoritmo com $(z, y, i-1)$. Cada tripla ocupa um espaço de $\log n + \log n + \log\log n \leqslant 3 \log n$. A MT M deve ser chamada inicialmente com

$$M(G, x, y, 1 + \lfloor \log n \rfloor)$$

pois a distância máxima entre dois vértices em G é $n-1$ e portanto i deve assumir valores apenas entre 0 e $1 + \lfloor \log n \rfloor$. Então no máximo haverá $c' \log n$ triplas na pilha. Isto é fácil de verificar na

função `ConectadoLimitado` acima: a cada chamada recursiva i diminui de um e o valor inicial desta variável é $1 + \lfloor \log n \rfloor$.

Como cada tripla utiliza $3 \log n$ bits e há no máximo $c' \log n$ triplas simultaneamente na pilha, o espaço utilizado será $\leqslant c'' \log n \log n = c'' \log^2 n$. O tamanho da entrada, $m = |\langle G \rangle \sqcup s \sqcup t|$, é $O(n^2)$. Então $n = c''' \sqrt{m}$. O espaço utilizado em função de m é

$$c'' \log^2 \sqrt{m} = c \log^2 m$$

Renomeando o tamanho da entrada de m para n, temos que a linguagem decidida por esta máquina de Turing pertence a SPACE($\log^2 n$).

□

Proposição 5.2.5. *(Teorema de Savitch) Se $f(n) \geqslant \log n$, NSPACE($f(n)$) $\subset$ SPACE($f^2(n)$).*

Demonstração. Seja $L \in$NSPACE($f(n)$). Então existe uma MT N que decide L em espaço $f(n)$. Construiremos uma MT M que decide L em espaço $f^2(n)$.

Dada a entrada x, M constrói um grafo de configurações $Ar(N(x))$ como na Proposição 5.2.3 e então utiliza o algoritmo da Proposição 5.2.4 para descobrir se existe um caminho entre dois vértices. Estes dois vértices são a configuração inicial e uma configuração em que o estado de N seja q_s. Se houver tal caminho, M vai para o estado q_s, senão vai para q_n e pára. Podemos considerar que N seja um tipo especial de máquina que, quando completa a sua computação, executa os seguintes passos:

1. apaga todas as fitas exceto a de entrada e a de saída;
2. escreve 0 ou 1 na fita de saída (existe o caminho ou não);
3. retrocede as cabeças de leitura/gravação das fitas de entrada e saída para a primeira posição;
4. vai para o estado q_s ou q_n conforme existe um caminho ou não entre os vértices.

Assim há apenas uma configuração final de aceitação da entrada; a saber,

$$(q_s, x_1, x_{2n}, \sqcup, \epsilon, \ldots, \sqcup, \epsilon)$$

na qual x_1 é o primeiro símbolo da entrada x e x_{2n} os símbolos restantes.

O algoritmo da proposição 5.2.4 utiliza um número de células $\leqslant c(\log n)^2$ no qual n é o número de vértices. Na máquina M definida acima, n é igual a

$$c^{f(n)+\log n} \leqslant c^{f(n)+f(n)} = c_1^{f(n)}$$

Então o número de células utilizadas por M é

$$\leqslant c(\log c_1^{f(n)})^2 = cf^2(n)$$

Logo a linguagem decidida por M pertence a SPACE($f^2(n)$).

□

Antes de apresentar o próximo teorema, recordaremos a definição da classe complementar de uma classe de linguagens. NSPACE($f(n)$) é a classe das linguagens que podem ser decididas por uma MTND em espaço $f(n)$. Isto é, se $L \in$ NSPACE($f(n)$), então existe uma MTND N tal que

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

E coNSPACE($f(n)$) é definida como

$$\text{coNSPACE}(f(n)) = \{ L^c : L \in \text{NSPACE}(f(n)) \}$$

Isto é, se $L^c \in$ coNSPACE($f(n)$), então

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

no qual N é a MTND que decide a linguagem L.

Apresentaremos um exemplo, na lógica, de uma linguagem e seu complemento. Antes disto precisaremos definir alguns termos. Um **literal** é uma variável do cálculo proposicional ou a sua negação. As variáveis proposicionais são $A, B, C, \ldots$ e V_i para $i \in \mathbb{N}$. Então são literais: A, $\neg B$, V_5, $\neg V_3$. Uma **cláusula** é uma disjunção de literais. Então são cláusulas: $A \vee V_3$, $V_1 \vee \neg C \vee \neg V_7 \vee D$, A (apenas um literal é uma cláusula), $\neg V_3$. Uma fórmula na FNC (forma normal conjuntiva) é uma conjunção de disjunções de literais. Então estão na FNC as seguintes fórmulas: A (uma única variável está na FNC), $\neg B$, $A \vee \neg B$ (uma cláusula é uma fórmula na FNC), $(\neg V_4 \vee V_2) \wedge (V_1 \vee \neg V_3)$, $A \wedge \neg B$ (as duas cláusulas são literais), $(A \vee \neg B \vee \neg C) \wedge (\neg A \vee \neg D) \wedge (B \vee \neg D)$. SAT é a linguagem que contém todas as fórmulas na FNC (forma normal conjuntiva) que são satisfazíveis. Uma fórmula é satisfazível se existe pelo menos uma atribuição de valores para as variáveis que torna a fórmula verdadeira. Isto é, se fizermos a tabela verdade da fórmula, pelo menos uma linha possui resultado V (verdadeiro). Definindo formalmente,

$$\text{SAT} = \{\phi : \text{ existe uma valoração } v \text{ tal que } v(\phi) = V\}$$

na qual v é uma atribuição de valores para as variáveis que ocorrem em ϕ e $v(\phi)$ é o valor verdade de ϕ com esta valoração (usamos o mesmo nome para duas coisas diferentes).

A linguagem SAT^c é definida como

$$\text{SAT}^c = \{\phi : \text{ para qualquer valoração } v \text{ temos } v(\phi) = F\}$$

Ou seja, uma fórmula ϕ pertence a SAT^c se ela é uma contradição.

Sabemos que SAT pertence a NP. Portanto SAT^c pertence a coNP.

Lema 5.2.3. *Existe uma MTND que executa em espaço* $\log n$ *e que, dada como entrada um grafo dirigido* G *e um vértice* v *deste grafo, calcula o número de vértices atingíveis a partir de* v*.*

Demonstração. Usaremos, como usual, uma MTND com estrada e saída. O espaço $\log n$ se refere apenas às fitas de trabalho (exclui a fita de entrada e a de saída). Uma MTND N calcula 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 $g(x)$ e o estado final seja q_s. Se o valor $g(x)$ não for calculado, o estado final deve ser q_n. O fato importante a notar é que deve existir pelo menos uma sequência de escolhas que leve ao resultado correto para cada entrada. Mas se a sequência escolhida não for adequada, podemos descartá-la e ir para o estado q_n.

Seja N_k o número de vértices que estão a uma distância $\leqslant k$ do vértice v. A distância de um vértice w a v é o número de arestas do menor caminho entre v e w. Então $N_0 = 1$, pois apenas v está a uma distância 0 de v. Sendo $G = (V, E)$, $N_1 = 1 + |\{w : (v, w) \in E\}|$.

Construiremos uma MTND N tal que $N(G, v)$ retorna o número de vértices atingíveis a partir de v (ou entra no estado q_n). Esta máquina de Turing utiliza várias subrotinas. Vejamos a primeira delas.

O algoritmo caminhoMenork retorna 1 se, a partir de escolhas não determinísticas, ele conseguir um caminho de tamanho $\leqslant k$ de v para x. Caso contrário o algoritmo entra no estado q_n terminando a execução de N.

Algoritmo caminhoMenork(x, k)

Por uma sequência de escolhas não determinísticas, escolha um caminho de tamanho k começando em v. Se x for o último vértice deste caminho, retorne 1 e vá para o estado q_s. Senão entre no estado q_n. Este caminho pode ter vértices repetidos, o que é natural, já que a escolha dos vértices é não determinística. Isto não tem importância, pois estamos interessados em saber se x pertence a caminhos de tamanho $\leqslant k$, não a caminhos de tamanho exatamente k.

Usaremos S_k para o conjunto dos vértices cuja distância a v é menor ou igual a k. Veremos agora como determinar se dado vértice w pertence a S_k. Podemos usar N_{k-1}, que é o número de vértices que estão a distância $\leqslant k$ de v. A forma como N_{k-1} é utilizado pelo algoritmo é muito original. Um vértice w está a uma distância $\leqslant k$ de v se está a uma distância $\leqslant k-1$ de v ou se w está ligado a um vértice que está a uma distância $\leqslant k-1$ de v. Isto é, $u \in S_k$ se $u \in S_{k-1}$ ou temos uma aresta de x para u ($(x, u) \in E$) sendo que $x \in S_{k-1}$.

Algoritmo CalculaTodosS_k(G, v)

$N_k = |S_k|$. Então $S_0 = \{v\}$ e $N_0 = 1$. n é $|V|$.

$N_0 = 1$

```
for k = 1 to n − 1 do
   begin
   N_k = 0
   for u = 1 to n do
      begin
      upertenceS_k = 0
      sizeS_{k−1} = 0
      for x = 1 to n do
         if caminhoMenork(x, k − 1)
         then
            begin
            sizeS_{k−1} = sizeS_{k−1} + 1;
            if x = u or (x, u) ∈ E
            then
               upertenceS_k = 1
            end
      if sizeS_{k−1} < N_{k−1}
      then
         reject;
      N_k = N_k + upertenceS_k;
      end
   end
```

A operação `reject` muda o estado corrente para q_n e a máquina pára.

O algoritmo não guarda os conjuntos S_k que contém todos os vértices cuja distância de v é $\leqslant k$. Ele guarda apenas o valor de N_{k-1} no cálculo de N_k. Todos os outros valores armazenados nas variáveis estão entre 1 e n, o número de vértices do grafo. Então o espaço utilizado é $\leqslant c \log n$ para alguma constante c, que é o número de variáveis necessárias simultaneamente no algoritmo acima.

A entrada do algoritmo é $G = (V, E)$ e v (um número). G é representada por uma matriz $n \times n$ na qual cada entrada ocupa $\log n$ células (ou bits), sendo $n = |V|$. Então o tamanho da entrada é $m = |G| + |v| = n^2 \log n + \log n \leqslant 2n^3$. E $n^2 \leqslant m$. Então o espaço utilizado pelo algoritmo, $c \log n$, expresso em função do tamanho m da entrada é

$$c \log n \leqslant c \log \sqrt{m} = c'' \log m$$

Portanto o espaço utilizado pelo algoritmo é $c'' \log n$, renomeando n para o tamanho da entrada $m = |G| + |v|$.

□

Teorema 5.2.2. *(Teorema de Immerman-Szelepcsényi) Se $f(n) \geqslant \log n$, então*

NSPACE($f(n)$) = coNSPACE($f(n)$).

Demonstração. Este teorema afirma o seguinte:
$L \in$ NSPACE($f(n)$) se e somente se $L \in$coNSPACE($f(n)$)
Ou seja, existe uma MTND N que decide L em espaço $f(n)$ se e somente se existe uma MTND N' que decide L^c em espaço $f(n)$ (esta é a definição de coNSPACE). Expandindo a frase, ficamos com
existe uma MTND N tal que

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

e uma MTND N' tal que

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

Ou seja, para todo x, N aceita x se e somente se N' rejeita x.

Dada a MTND N, mostraremos como construir N'. A idéia é utilizar a árvore $Ar(N(x))$ e contar quantas configurações de aceitação existem. Se existir uma, N' deve rejeitar a entrada (ir para o estado q_n). Se não existir nenhuma, N' aceita a entrada. Uma configuração de aceitação em um grafo $G = (V, E)$ está a uma distância máxima $|V| - 1$ da raiz de $Ar(N(x))$. Então N' executa o algoritmo descrito no Lema 5.2.3 com a árvore $Ar(N(x))$ que possui $c_1^{f(n)}$ vértices (pela prova da Proposição 5.2.3, seria $c^{f(n)+\log n}$, mas como assumimos que $f(n) \leqslant \log n$, o número de vértices é $\leqslant c_1^{f(n)}$).

Se algum vértice correspondente a uma configuração de aceitação é encontrado em algum conjunto S_k, para qualquer valor de k, N' rejeita a entrada (vai para o estado q_n). Se nenhum vértice de aceitação é encontrado em S_{m-1}, no qual m é o número $c_1^{f(n)}$ de vértices, então N' aceita a entrada (vai para o estado q_s e retorna 1). O espaço utilizado, de acordo com o Lema 5.2.3, é $\log m^2 = \log c_1^{2f(n)} = c_2 f(n)$. Portanto N' executa em espaço $f(n)$ e $L \in$ coNSPACE($f(n)$).

□

Proposição 5.2.6. *$P = coP$*

Demonstração. Provaremos que $L \in$ P sse $L^c \in$ P. Por definição, $L^c \in$ P sse $L \in$ coP. Então teremos provado que $L \in$ P sse $L \in$ coP.

$L \in$ P sse existe MTD M polinomial tal que $x \in L$ sse $M(x) = 1$
sse existe MTD M polinomial tal que $x \notin L$ sse $M(x) = 0$
sse existe MTD M' polinomial tal que $M'(x) = 1 - M(x)$ e $x \in L^c$ sse $M'(x) = 1$
sse $L^c \in$ coP

□

Capítulo 6

A Classe NP

Este Capítulo 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.

Neste Capítulo 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 6.0.1. *(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 6.0.2. *(Definição II) $L \in NP$ se e somente se existe uma MTD M que executa em tempo polinomial (veja definição 2.2.7) 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 6.0.3. *[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 6.0.4. *(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 6.0.7. *As definições 6.0.2 e 6.0.4 são equivalentes.*

Demonstração. Provaremos que a definição 6.0.4 engloba a 6.0.2. O prova $\Longrightarrow$ é basicamente esta lida em ordem inversa. A definição 6.0.4 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 6.0.4 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 6.0.2. Note que, pela definição 6.0.4, se $L \in$ NP, existe R e portanto existe M que o decide (exatamente como na definição 6.0.2). □

Proposição 6.0.8. *As definições 6.0.1 e 6.0.2 são equivalentes.*

Demonstração. ($\Longrightarrow$) Suponha que uma linguagem $L \in$ NP pela definição 6.0.1. Provaremos que $L \in$ NP pela definição 6.0.2. 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 6.0.2. Provaremos que $L \in$ NP pela definição 6.0.1. 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 6.0.2, 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 6.0.1. *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) $L = \{v \sqcup e : \textit{ elemento } e \textit{ está no vetor } v\}$

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

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

(f) $H = \{(V, E) : \textit{ existe um caminho hamiltoniano no grafo } G = (V, E)\}$ *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 6.0.5. *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*

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

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

Definição 6.0.6. *Dizemos que uma linguagem L é log-redutível a uma linguagem K se existe uma MT R que executa em espaço menor ou igual a* $c \log n$ *tal que*

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

Usaremos $L \leqslant_L K$ *para "L é log-redutível a R".*

É interessante observar que uma redutibilidade de L para K por uma redução R mapeia uma linguagem L em um subconjunto $R(L)$ de K — veja a figura 6.1. Os dois quadrados da direita representam o conjunto $\{0, 1\}^\star$, já que estamos considerando apenas linguagens sobre o conjunto $\{0, 1\}$. A redução R pode mapear toda a linguagem R a um subconjunto pequeno de K. Em particular, pode mapear L para um conjunto de um único elemento. Os elementos de $K - R(L)$ e $K^c - R(L^c)$ são os que tornam possível que K seja uma linguagem possivelmente mais difícil de decidir do que L. Afinal, para responder a todas as questões do tipo $x \in L$ basta responder às questões $x \in R(L)$ e $x \in R(L^c)$ que envolvem **subconjuntos** K e K^c.

Proposição 6.0.9. *Se* $L \leqslant_L K$ *então* $L \leqslant_P K$*.*

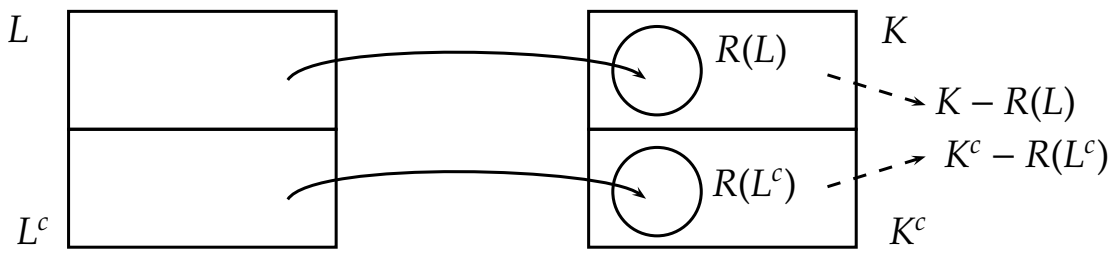

Figura 6.1: redução

Demonstração. Se $L \leqslant_L K$, existe uma MT R que executa em espaço $\leqslant c' \log n$. O número de configurações desta máquina (veja Proposição 5.2.3), assumindo que estamos utilizando logaritmo na base 2 e que $e = \log_d 2$, é

$$d^{\log n + c' \log n} = d^{c \log n} = d^{c \frac{\log_d n}{\log_d 2}} = d^{\log_d n^{c/e}} = n^{c/e}$$

Ou seja, R executa em tempo polinomial e portanto $L \leqslant_P K$. □

Proposição 6.0.10. *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 6.0.11. *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.

□

Proposição 6.0.12. *Se $L_1 \leqslant_L L_2$ e $L_2 \leqslant_L L_3$ então $L_1 \leqslant_L L_3$.*

Demonstração. Queremos provar que existe uma redução R_{13} em espaço $\log n$ de L_1 para L_3 dadas que existem reduções R_{12} e R_{23} em espaço $\log n$ de L_1 para L_2 e de L_2 para L_3, respectivamente.

Não podemos definir R_{13} como na Proposição anterior, $R_{13}(x) = R_{23}(R_{12}(x))$, pois $R_{12}(x)$ pode ter um número polinomial de símbolos 0´s e 1´s. Como esta redução é em espaço logarítmico, ela executa em tempo polinomial (Proposição 6.0.9) e portanto pode produzir um número polinomial, em x, de símbolos de saída. Lembre-se de que a definição de execução em espaço (página 79) considera apenas o espaço utilizado nas fitas de trabalho, não considerando as fitas de entrada e saída.

Definiremos uma máquina R_{13} que é uma redução de L_1 a L_3 e que executa em espaço $c \log n$. $R_{13}(x)$ é basicamente $R_{23}(R_{12}(x))$ mas o cálculo de $R_{12}(x)$ não é feito completamente antes de passar o resultado a R_{23}. Como R_{12} e R_{23} são máquinas com entrada e saída, a cabeça de leitura/gravação de R_{12} da primeira fita, sob o x da entrada, pode se mover para frente e para trás. Já a cabeça de escrita só pode se mover para frente. Os símbolos produzidos por R_{12} na fita de escrita são passados como entrada para R_{23} cuja cabeça de leitura/gravação pode se mover para frente e para trás. Então temos uma incombatibilidade aqui: os símbolos de saída de uma máquina, R_{12}, são produzidos um por vez com a cabeça se movimentando sempre para frente e estes mesmos símbolos são lidos por uma cabeça de leitura/gravação de R_{23} que pode se movimentar em ambas as direções.

Quando R_{23} movimenta a cabeça de leitura/gravação da fita de entrada para frente, basta invocar R_{12} para produzir o próximo símbolo da saída. Quando esta cabeça se movimenta para trás, temos um problema, pois a saída de R_{12} não é armazenada — nem poderia, pois ela pode ocupar um espaço polinomial. A solução é invocar R_{12} para fazer todos os cálculos novamente, partindo do início, até chegar à célula anterior à atual. Desta forma não é necessário armazenar a saída de R_{12}.

A máquina R_{13} definida como no parágrafo anterior produz exatamente o mesmo resultado que a máquina $R_{23}(R_{12}(x))$. Então, pela Proposição anterior,

$$x \in L_1 \text{ sse } R_{13} \in L3$$

E ambas R_{12} e R_{23} utilizam espaço $\leqslant c \log n$ (estamos usando a mesma constante para ambas). Logo a sua composição utiliza espaço $\leqslant 2c \log n$. □

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

Definição 6.0.7. *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 6.0.8. *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 6.0.13. *([2]) A linguagem*

$$TMSAT = \{(\langle M \rangle, x, 1^n, 1^t) : \exists u \in \{0,1\}^n \textit{ tal que } M(x \sqcup u) = 1 \textit{ em } \leqslant t \textit{ passos } \} \tag{6.1}$$

é NP-completa. M é uma MT determinística.

Demonstração. TMSAT utiliza em sua definição os requerimentos necessários para uma linguagem pertencer a NP. Então não é surpreendente que ela seja NP-completa. As tuplas

$(\langle M\rangle, x, 1^n, 1^t)$ de TMSAT são aquelas tais que u tem tamanho n, M executa em um número de passos $\leqslant t$ com entrada $(x \sqcup u)$ e $M(x \sqcup u) = 1$.

Vejamos como uma linguagem $L \in$ NP pode ser reduzida a TMSAT. Pela definição de NP, associado a L existem:

(a) uma MT determinística M que executa em tempo $\leqslant q(n)$ no qual n é o tamanho da entrada e $q(n)$ um polinômio;

(b) um polinômio $p(n)$

M, p e q devem satisfazer os seguintes requerimentos:

$$x \in L \text{ sse existe } u \in \{0,1\}^{p(|x|)} \text{ tal que } M(x,u) = 1 \text{ em núm. passos } \leqslant q(|x| + p(|x|))$$

Provaremos que $L \in$ NP é polinomialmente reduzível a TMSAT. A redução é

$$R(x) = (\langle M\rangle, x, 1^{p(|x|)}, 1^{q(|x|+p(|x|))})$$

Temos que provar que $x \in L$ sse $R(x) \in$ TMSAT. Vejamos:

$$\begin{array}{lll} x \in L & \text{sse} & \text{existe } u \in \{0,1\}^{p(|x|)} \text{ tal que } M(x,u) = 1 \text{ em núm. passos } \leqslant q(|x| + p(|x|)) \\ & \text{sse} & (\langle M\rangle, x, 1^{p(|x|)}, 1^{q(|x|+p(|x|))}) \in \text{ TMSAT} \\ & \text{sse} & R(x) \in \text{ TMSAT} \end{array}$$

Falta provar que TMSAT $\in NP$. Pela definição 6.0.1 de NP, isto acontece se existe uma MTND N que executa em tempo polinomial e que decide TMSAT. Isto é,

$$(\langle M\rangle, x, 1^n, 1^t) \in \text{ TMSAT sse existe uma sequência de escolhas tal que } N(\langle M\rangle, x, 1^n, 1^t) = 1 \tag{6.2}$$

Definiremos esta máquina N. Dada a entrada $(\langle M\rangle, x, 1^n, 1^t)$, a MTND N, através de uma sequência de escolhas não determinísticas, escolhe $u \in \{0,1\}^n$ e simula a execução de $M(x \sqcup u)$ até um máximo de t passos. O resultado de M será o resultado de N se M produzir um resultado. Se M demorar mais do que t passos, N retorna 0. Se $(\langle M\rangle, x, 1^n, 1^t)$ pertencer a TMSAT, existirá um u de tamanho n tal que $M(x \sqcup u) = 1$ em um número de passos $\leqslant t$, por definição da linguagem TMSAT (equação 6.1). Então neste caso N retornará 1. Se $(\langle M\rangle, x, 1^n, 1^t)$ não pertencer a TMSAT, nenhuma sequência escolhida por N fará M retornar 1 (todas elas farão M retornar 0) ou M demorará mais do que t passos. Portanto N também retornará 0. Isto é a conclusão que esperávamos para concluir que N, definida desta forma, satisfaz a equação 6.2.

Falta provar que N executa em tempo polinomial. Sabemos que

$$\begin{array}{lll} n = & |1^n| & \leqslant \left|(\langle M\rangle, x, 1^n, 1^t)\right| \\ t = & \left|1^t\right| & \leqslant \left|(\langle M\rangle, x, 1^n, 1^t)\right| \end{array}$$

Então a produção de u, que toma n passos, é feita em tempo linear na entrada e que a simulação de M por N toma tempo polinomial (porque o número de passos de N deve ser $\leqslant t$). Logo N executa em tempo polinomial. Isto conclui a prova.

Note que, se tivéssemos definido os elementos de TMSAT como $(\langle M\rangle, x, n, t)$, então a produção de u e a simulação de M por N poderiam tomar um tempo exponencial. Se uma MT toma x como entrada e executa por x passos, ela executa em tempo exponencial: o tamanho da entrada é $n = |x| = 1 + \lfloor \log_2 x \rfloor$ e o número de passos é $x = O(2^n)$.

□

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

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

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

Demonstração. Temos que provar que: a) SAT $\in$ NP e b) qualquer linguagem de NP é polinomialmente redutível a SAT.

A parte a) é simples: $\phi \in$ SAT sse existe uma atribuição de valores para as variáveis que a tornam verdadeira. Definiremos uma MTND N que toma uma fórmula ϕ como parâmetro e que executa em tempo polinomial tal que $\phi \in$ SAT sse existe uma sequência não determinística de escolhas tal que $N(\phi) = 1$. N escolhe não deterministicamente valores para todas as variáveis de ϕ e depois calcula o seu valor verdade. Se $\phi \in$ SAT, haverá alguma sequência de escolhas que a tornam verdadeira. A escolha não determinística dos valores verdade para as variáveis de ϕ é feita em tempo linear pois o número de variáveis de ϕ é $\leqslant |\phi|$, que é o tamanho da entrada. Assuma que os valores verdade para as variáveis foram colocados em uma fita de N em pares (X, v) no qual X é o nome da variável e v é o valor verdade escolhido para X. N percorre a fórmula substituindo cada variável pelo seu valor verdade, o que é feito em tempo cn^2, $n = |\phi|$. Depois N calcula o valor verdade da fórmula, o que é feito em tempo linear, pois cada ocorrência de uma variável na fórmula é utilizada uma única vez (e o número de ocorrências é também $\leqslant n$)

Provaremos agora a parte b). Suponha que $L \in$ NP. Então existe uma MTND N que decide L em tempo polinomial. Construiremos uma redução R tal que $x \in L$ sse $R(x) \in$ SAT; isto é, $R(x)$ é uma fórmula na FNC satisfazível. Então existe uma sequência de escolhas tal que $N(x) = 1$ se e somente se existe uma escolha de valores para as variáveis de $R(x)$ tal que $R(x)$ é satisfazível. Assumiremos que N possua algumas restrições:

(a) tenha uma única fita;

(b) seja potencialmente infinita apenas para a direita;

(c) cada passo tenha zero, uma ou duas escolhas não determinísticas.

. Estas restrições não são importantes para esta prova pois uma MT com várias fitas e infinita em ambas as direções pode ser simulada por outra com uma única fita e infinita apenas à direita em tempo polinomial. A restrição (c) também não tira a generalidade da prova, pois qualquer MTND pode ser transformada em uma máquina com esta restrição e que executa em um número de passos $cf(n)$, se a original executa em tempo $f(n)$ (veja página 33).

A idéia da construção de $R(x)$ é fazer uma fórmula que simule a execução da MTND N. As variáveis da fórmula $R(x)$ corresponderão às escolhas não determinísticas da execução de $N(x)$. Como N executa em tempo polinomial, $R(x)$ terá um número polinomial de variáveis. É importante notar que, dado x, a execução de $N(x)$ está completamente determinada a menos das escolhas não determinísticas: sabemos quantos passos, no máximo, a execução tomará e no máximo quantas células serão utilizadas. Contudo, a máquina de Turing R construirá inicialmente uma fórmula com muito mais variáveis que aquelas que correspondem às escolhas não determinísticas (estas variáveis serão chamadas de y_i). Esta fórmula será chamada de ϕ_x. A MT R eliminará variáveis de ϕ_x deixando-a apenas com as variáveis y_i que correspondem às escolhas não determinísticas.

Suponha que N execute em tempo cn^k. Então o número de passos da execução $N(x)$ será $\leqslant cn^k$ ($n = |x|$) e o número de células de memória utilizadas será $\leqslant cn^k$ (pois a cada passo se pode utilizar no máximo uma célula que não tinha sido utilizada antes). A execução $N(x)$ será representada por uma matriz na qual as linhas representam as configurações da fita de N — assumiremos que esta máquina possui apenas uma fita. Assim, se a matriz é A, A_1, a primeira linha, corresponde à configuração inicial em que a fita contém apenas a entrada x. A segunda linha de A, A_2, contém a configuração depois da execução da primeira instrução (que pode ter sido escolhida não deterministicamente). E assim por diante. Como o número de passos será $\leqslant cn^k$, o número de linhas de A será $\leqslant cn^k$ também. Como o número de células utilizadas será $\leqslant cn^k$, o número de colunas de A será $\leqslant cn^k$. Assumiremos que A tem cn^k linhas e cn^k colunas.

As linhas de A correspondem a um caminho na árvore $Ar(N(x))$ da raiz até uma folha. Cada linha contém uma configuração que é um vértice de $Ar(N(x))$. Ao percorrer esta árvore, pode ser necessário fazer uma escolha não determinística — a árvore contém todas as escolhas possíveis para dado x mas a matriz A mostra apenas uma das escolhas. Esta escolha será feita pelas variáveis y_i da fórmula $R(x)$. Note que a altura máxima de $Ar(N(x))$ é cn^k, que é o número de linhas de A.

Uma configuração de $N = (Q, \Sigma, I, q_0)$ será representada por uma cadeia $w = w_1 w_2 \dots w_m$ tal que $w_i \in \Sigma \cup \Sigma'$ no qual Σ' contém símbolos para cada um dos pares $(u, q) \in \Sigma \times Q$. Um par (u, q) indica que o estado corrente de N é q e a cabeça de leitura/gravação está sob um símbolo

$u \in \Sigma$. Então exatamente um único símbolo de w pertence a Σ'. Todos os outros pertencem a Σ. O símbolo correspondente a (u, q) é u_q.

Vejamos agora como construir, dado x, uma fórmula ϕ_x que simula a execução não determinística de $N(x)$. A primeira observação é que o valor de uma posição $A_{i,j}$ depende apenas dos valores de $A_{i-1,j-1}$, $A_{i-1,j}$ e $A_{i-1,j+1}$. A linha A_{i-1} representa a configuração do passo anterior de N. Se a cabeça de leitura/gravação de N está na posição j no passo $i-1$, no passo i ela poderá estar em $j-1$, j ou $j+1$ sendo que a célula alterada será a da posição j. Na tabela A, o estado é armazenado junto com um símbolo na célula corrente. Assim, se $A_{i-1,j}$ contém s_q (símbolo corrente é s e o estado corrente é q), então $A_{i,j-1}$, $A_{i,j}$ e $A_{i,j+1}$ poderão ser modificados. Por exemplo, se a cabeça se move para a esquerda e $A_{i-1,j-1}$ é σ, o valor de $A_{i,j-1}$ será σ_q.

A fórmula ϕ_x contém m variáveis para cada célula $A_{i,j}$, no qual $m = 1 + \lfloor \log_2(|\Sigma \cup \Sigma'|) \rfloor$. Isto é, conjuntamente, estas m variáveis definem precisamente qual símbolo está em $A_{i,j}$. Este símbolo pertence a $\Sigma \cup \Sigma'$. Usamos uma representação em binário para cada um dos elementos. Estas variáveis serão $x_{i,j,p}$ no qual $1 \leqslant i, j \leqslant cn^k$ e $0 \leqslant p < m$. São $cn^k cn^k m = mc^2 n^{2k}$ variáveis ao todo, um número polinomial. Cada variável $x_{i,j,p}$ pode ser definida em função das variáveis $x_{i-1,j-1,r}$, $x_{i-1,j,r}$ e $x_{i-1,j+1,r}$, para todos os valores de $0 \leqslant r < m$. Esta definição é feita utilizando-se o conjunto I de instruções de N, que diz como um passo é feito. Mas como N é não determinística, as variáveis da linha i não podem ser definidas somente em termos de variáveis da linha $i-1$. É necessário acrescentar uma variável y_i que informa qual escolha não determinística foi feita, se for necessário fazer uma escolha deste tipo. A máquina N pode ter zero, uma ou duas escolhas não determinísticas a cada passo (ela é limitada a no máximo duas escolhas). Terá zero se a máquina pára. Neste caso a configuração da próxima linha será igual à da linha anterior. Se N tiver duas escolhas, haverá uma variável y_i associada à linha i. Quando a computação está em um vértice v da árvore $Ar(N(x))$ correspondente à linha i, a variável y_i escolhe em qual sub-árvore do vértice v a computação continuará. Então o número destas variáveis é $\leqslant cn^k$. Com isto temos uma fórmula ϕ_x que contém toda a computação de $N(x)$. Mas esta fórmula pode ser simplificada. Como fazer isto é explicado a seguir.

As variáveis $x_{1,j,p}$ da primeira linha da tabela são definidas em função da cadeia x da entrada e do estado inicial de N. As variáveis que nos interessam são realmente as da última linha da tabela, que contém o resultado da computação — lembre-se de que, se a computação termina antes da última linha (posição cn^k), as linhas restantes ficam com valores iguais à da linha onde a computação terminou. As variáveis $x_{i+1,j,p}$ de cada linha $i+1$ são definidas em função das variáveis $x_{i,j,r}$ da linha anterior e da variável y_i (que não depende de outras variáveis). Então as variáveis da linha cn^k, a última linha, podem ser expressos como uma expressão que contenha apenas as variáveis $x_{1,j,r}$ da primeira linha e as variáveis y_i. Mas os valores das variáveis $x_{1,j,r}$ são definidos em função da entrada. Sabemos precisamente qual deles é V ou F (estamos usando V para 1 e F para 0). Então as variáveis da última linha podem ser expressas como uma expressão booleana que contém apenas as variáveis y_i. Mas para tanto, é necessário simplificar as expressões correspondentes às variáveis da última linha. Há um número polinomial destas variáveis, $mc^2 n^{2k}$, e um número polinomial de substituições (o número de linhas, cn^k). A

simplificação de cada uma das variáveis (baseadas nas variáveis da linha anterior) também pode ser feita em tempo polinomial. Então a expressão para cada variável da última linha pode ser obtida em tempo polinomial. Falta uma fórmula que considere todas as variáveis da última linha e que seja V ou F caso o estado final seja q_s ou q_n. Esta fórmula simplesmente verifica se algum dos símbolos da última linha é s_{q_s}, o que pode ser feito por uma fórmula com um número polinomial de termos e que pode ser produzida em tempo polinomial. É esta fórmula simplificada e com a verificação do estado final de N que é retornada pela MT R.

A MT $R(x)$ toma a entrada x e produz a fórmula final que é satisfazível se e somente se $N(x) = 1$ (estado final q_s). Assumimos que o estado final é sempre q_s ou q_n. A fórmula produzida simula a execução de N com entrada x. Ela é simplesmente a simplificação de um circuito que simula a execução de $N(x)$. As variáveis y_i selecionam qual caminho na árvore $Ar(N(x))$ a computação deve seguir.

□

Capítulo 7

f-g-simulação

Este Capítulo descreve parte da pesquisa desenvolvida. Os conceitos e proposições aqui definidos estão sendo refinados para serem apresentados em um artigo em Model-Based REasoning in Science and Technology — Abduction, Logic, and Computational Discovery, Brazil, 2009.

7.1 Motivação

A máquina de Turing Universal (UTM) toma como entrada dois argumentos: a descrição de uma máquina de Turing M e a entrada x para esta máquina. A UTM simula então a execução de M com a entrada x. A UTM é capaz de simular a execução de qualquer máquina de Turing com qualquer entrada. Então o seu tempo de execução é indeterminado, com certas entradas ela não pára nunca. Isto é uma conseqüência da generalidade da UTM.

É possível construir uma MT M_N que simule a execução de uma única MT N. Tanto M_N como N tomam uma entrada x. A máquina que M_N simula é fixa, N, mas a entrada pode variar. O tempo de execução de $M_N(x)$ será dependente do tempo de execução de N com entrada x. Se M_N for uma MT com quatro fitas e N tiver uma única fita, o tempo de execução de M_N será $kf(n)$ (k constante) se o tempo de execução de N for $f(n)$ (os detalhes da simulação são irrelevantes para o objetivo da pesquisa. Mas note que a) uma das fitas de M_N pode representar a fita de N e b) a codificação de N pode ser colocado na segunda fita e encontrar a próxima instrução de N a ser executada toma um tempo constante). Então uma máquina de quatro fitas pode simular uma MT de uma fita na mesma ordem de complexidade ($O(kf(n)) = O(f(n))$).

Imagine agora uma MT M que toma uma entrada x dentro da qual algumas partes são na verdade instruções a serem simuladas por M. Utilizando uma notação em que d representa um dado (informação, não instrução), i é instrução, uma entrada x poderia ser

```
ddiddi
```

Durante a execução de M, uma instrução `i` seria utilizado, por exemplo, para ordenar que M copie certo valor de uma célula para outra ou para modificar o estado corrente para outro valor. Na UTM, já temos instruções junto com dados. A entrada para uma UTM é algo do tipo "$D_N \sqcup x$", onde D_N é a descrição de uma MT N, x é a entrada de N e "$\sqcup$" separa os dois campos.

Uma questão que surge naturalmente é: é possível diferenciar dados (informações que não não instruções) de instruções na entrada como foi feito acima? A resposta é não. Examinando-se as instruções da MT M descrita acima, que utiliza dados e instruções na entrada, não é possível afirmar que há instruções na terceira e última posições da entrada (`ddiddi`). Então pode-se considerar que uma máquina de Turing M, com a interpretação apropriada, está considerando parte ou toda a sua entrada como instrução. Estas instruções formam uma outra MT N_0 que está sendo simulada por M. Mas o que é considerado instrução é relativo. Pode-se mudar a interpretação da entrada e de M de tal forma que outras partes da entrada sejam consideradas instruções de uma outra MT N_1. Da mesma forma, há interpretações em que M simula máquinas N_2, N_3, etc.

Podemos concluir que qualquer MT está simulando outras máquinas de Turing, infinitas delas, em geral. O ponto principal desta pesquisa é descobrir quais MT´s uma certa máquina está simulando. Para isto, é necessário definir o que é "simulação". Claramente, não podemos empregar o conceito de simulação empregado na UTM. Este conceito é geral demais. Não é possível fazer um algoritmo que toma duas MT´s M e N e que retorna "sim" ou "não" caso M simule N ou não. Esta questão é indecidível e pode ser reduzida à questão "dadas duas MT´s M e N, elas são equivalentes (produzem sempre o mesmo resultado) ?".

A definição de simulação que empregamos é mais adequada a uma pesquisa sobre Classes de Complexidade. Ela define simulação em termos de tempo de execução das máquinas de Turing envolvidas. O objetivo final é, dada uma MT M, caracterizar as máquinas de Turing que

(a) simulam M;

(b) são simuladas por M.

Com isto, pretende-se relacionar a classe de complexidade da linguagem que M decide com as classes das linguagens decididas pelas máquinas dos ítens (a) e (b). Este é um objetivo ambicioso que ainda está longe de ser concluído.

7.2 Descrição da Pesquisa

Esta seção define a "simulação" descrita na seção anterior. Antes de apresentar a definição, precisaremos recordar como foi feita a prova de que de que SAT é NP-completa (Teorema 6.0.3). Dada uma linguagem $L' \in$ NP, existe uma MTND M' que decide L'. A redução R de L' a SAT

deve ser tal que

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

A MTD R toma x como entrada e, utilizando a definição de M', simula uma execução não determinística de $M'(x)$. Isto é, R produz uma fórmula que é satisfazível se e somente se existe uma sequência de escolhas não determinísticas em $M'(x)$ tal que $M'(x) = 1$ (o estado final é q_s, de aceitação). R de fato produz uma fórmula que simula a execução de M', e é necessário o conhecimento desta máquina para se projetar R.

Há duas maneiras de se provar que uma linguagem L' se reduz a uma linguagem L:

1. simulando-se a máquina M' que decide L' em uma máquina M que decide L. A máquina de Turing R que é o redutor de L' para L converte x para uma entrada de M e se baseia em M'. Esta técnica é utilizada na prova de que SAT é NP-completo. E novamente é importante observar que a MT R depende explicitamente dos detalhes de M';

2. fazendo-se um redutor R de L' para L que **não** se baseia em M', a MT que decide L'. É possível que a MT M' que decida L' não seja do conhecimento do criador de R ou de M.

Em qualquer dos dois casos, devemos ter

$$M'(x) = M(R(x))$$

O ponto principal desta pesquisa é unir as abordagens 1) e 2) pela definição de f-g-simulação. Uma simulação usual de uma máquina de Turing por outra apresenta a característica de que parte da fita da máquina que simula é igual à fita da máquina simulada durante todo o tempo. Por causa disto, no final da simulação o resultado de uma é igual à de outra. O conceito de f-g-simulação estende o conceito de simulação para quaisquer duas máquinas que produzem sempre o mesmo resultado dadas as mesmas entradas.

Assumiremos que cada máquina de Turing citada a seguir possua as seguintes características:

- a máquina é uma variação da MTES de quatro fitas. Na primeira fita é colocada a entrada da máquina, que chamaremos de x. À direita das células da entrada devem vir células com o símbolo □. A entrada é apenas de leitura. Nenhum símbolo pode ser escrito na primeira fita. A segunda fita é chamada de "área de rascunho", abreviada para *ar*. Apenas um número constante de células desta fita é utilizada em cada computação, independente da entrada. A terceira fita é chamada de "área de trabalho", *at*. É nesta fita que são feitas as computações. A quarta fita é a saída da máquina na qual inicialmente todos os símbolos são □. Qualquer instrução que escreve um símbolo diferente do símbolo corrente na quarta fita deve mover a cabeça da fita para a direita. O símbolo corrente é sempre □. Em cada instrução executada, a cabeça da fita ou permanece no mesmo lugar (neste caso, exige-se que o símbolo escrito seja igual a □) ou se move para a direita. Somente o espaço utilizado na terceira fita é contado no espaço utilizado pela máquina ;

- se o estado final da máquina for q_s ou q_n, é escrito 1 ou 0 (respectivamente) na fita de saída;
- a máquina pára depois de exatamente um certo número de passos dado por uma função. A função pode variar de máquina para máquina. Isto é, a MT é precisa (veja a Definição 2.2.10).

Definição 7.2.1. *Dizemos que uma MT M_1 $f(n)$-$g(n)$-simula uma MT M_2 se existem funções f, g e h de $\mathbb{N}$ em $\mathbb{N}$ tal que, na computação de $M_1(x)$ e $M_2(x)$, depois de $mf(n)$ passos de M_1 e $mg(n)$ passos de M_2, $1 \leqslant m \leqslant h(n)$:*

(a) áreas de trabalho at´s de ambas as máquinas são iguais ($n = |x|$) e as fitas de saída também (inclusive a posição da cabeça de leitura/gravação);

(b) a cabeça de leitura/gravação nas duas at´s está na primeira célula de cada at. Isto é, a célula à esquerda contém $\square$;

Chamando de at_1 e at_2 as áreas de trabalho de M_1 e M_2, $at_1 = at_2$ a cada $f(n)$ passos de M_1 e $g(n)$ passos de M_2 (idem para a saída). A intenção é que $f(n)$ passos de M_1 simule $g(n)$ passos de M_2 e vice-versa. Chamaremos de "grande passo" de M_1 a execução de $f(n)$ instruções de M_1. Um grande passo sempre começa, em M_1, em um passo que é múltiplo de $f(n)$: 0, $f(n)$, $2f(n)$, ... Grande passo de M_2 é definido similarmente.

Algumas conclusões podem ser tiradas da definição de f-g-simulação.

1. Se M_1 termina a execução em exatamente $F(n)$ passos e M_2 em $G(n)$ passos, então

$$F(n) = h(n)f(n)$$

$$G(n) = h(n)g(n)$$

   Neste projeto utiliza-se, sempre, máquinas de Turing exatas, que terminam a sua execução precisamente depois de um número de passos dados por uma função $t(n)$. A função, naturalmente, pode variar de máquina para máquina.

2. É possível que duas máquinas M_1 e M_2 produzam resultados idênticos para todo x de entrada e que as áreas de trabalho fiquem iguais diversas vezes durante a execução simultânea das máquinas. Contudo, não necessariamente uma f-g-simulará a outra. A f-g-simulação só acontecerá se existirem funções $f(n)$ e $g(n)$ que indiquem os passos em que as áreas de trabalho ficam iguais. Contudo, se as funções não existirem, em alguns casos pode-se modificar as máquinas para que existam. Por exemplo, suponha que as máquinas M_1 e M_2 produzam áreas de trabalho idênticas depois dos seguintes números de passos:

$$\begin{array}{lcccccccc} M_1 & n & n^3 & n & n^3 & n & n^3 & \ldots \\ M_2 & \log n & n\log n & \log n & n\log n & \log n & n\log n & \ldots \end{array}$$

Isto é, depois de iniciada a computação simultânea de M_1 e M_2 com uma entrada x qualquer, $n = |x|$, após n passos de M_1 e $\log n$ passos de M_2 as áreas de trabalho são idênticas. Este é o primeiro "grande passo". No segundo grande passo, são executados n^3 passos de M_1 e $n \log n$ passos de M_2. Novamente, $at_1 = at_2$ e assim por diante. Note que apesar das máquinas produzirem áreas de trabalho idênticas em intervalos previsíveis, M_1 não f-g-simula M_2 pois os intervalos não são dados por funções $f(n)$ e $g(n)$ — deveríamos utilizar duas funções para cada máquina (por exemplo, $f_1(n) = n$ e $f_2(n) = n^3$ para M_1). Neste caso, podemos modificar M_1 de tal forma que cada "grande passo" tome exatamente n^3 passos. Isto é, após os primeiros n passos, M_1 executa $n^3 - n$ instruções que não fazem nada, apenas para completar as n^3 instruções de cada grande passo. Então teríamos $f(n) = n^3$. A mesma técnica deveria ser aplicada a M_2 e teríamos $g(n) = n \log n$.

Esta solução foi escolhida ao invés de se ter funções $h(n, m)$ onde m é o número do grande passo (ao invés de uma função $h(n)$). Em M_1 deste exemplo, $h(n, m) = n$ se m é par e $h(n, m) = n^3$ se m é ímpar.

3. A definição de f-g-simulação emprega máquinas de Turing precisas com quatro fitas. Esta exigência não diminui a generalidade da definição pois para qualquer MT com k fitas, $k \geqslant 1$, existe uma MT com quatro fitas que a simula. E vice-versa. E qualquer MT pode ser transformada em uma precisa (veja Proposição 2.2.3).

4. As áreas de rascunho de M_1 e M_2 podem ser de tamanhos diferentes.

5. Se M_1 é um simulador usual de M_2, então a área de rascunho de M_2 tem tamanho zero e a área de rascunho de M_1 contém a codificação de M_2, além de espaço para índices que percorrem a área de rascunho para procurar a próxima instrução a ser executada. Quando M_1 simula M_2, a área de trabalho de M_1 fica igual à área de trabalho de M_2. Necessariamente M_1 tem que executar vários passos para simular um único passo de M_2. Mas este número de passos é menor ou igual a uma constante m. Como podemos sempre inserir instruções inúteis na máquina, podemos assumir que cada passo de M_2 é simulado com exatamente m passos de M_1. Então, a cada m passos de M_1 e um passo de M_2, $at_1 = at_2$. Logo $f(n) = m$ e $g(n) = 1$.

6. Se M_1 $f(n)$-$g(n)$-simula M_2, então M_1 deve terminar a sua execução depois de $F(n)$ passos. Logo M_1 não pode simular qualquer máquina de Turing, pois há máquinas que não terminam a sua execução. Então por definição se uma máquina f-g-simula alguma outra ela é menos poderosa do que a máquina de Turing Universal.

7. Se M_1 $f(n)$-$g(n)$-simula M_2, então M_2 $g(n)$-$f(n)$-simula M_1.

8. Se duas máquinas M_1 e M_2 produzem resultados idênticos para toda entrada, então podemos afirmar que M_1 f-g-simula M_2. No pior caso, $h(n) = 1$, $F(n) = f(n)$ e $G(n) = g(n)$. Isto é, há apenas um grande passo no qual é realizado toda a computação de M_1 e toda a computação de M_2.

9. É possível retirar a exigência de que as fitas de saída sejam iguais. Isto pode ser feito eliminando-se qualquer possibilidade de transferência de informação da primeira e segunda fitas (entrada e rascunho) para a quarta fita (saída). Isto pode ser feito exigindo-se que, se

$$(q, s_1, s_2, s_3, s_4, q', s'_1, d_1, s'_2, d_2, s'_3, d_3, s'_4, d_4)$$

for uma instrução onde $s_4 \neq s'_4$ (um símbolo s'_4 é escrito na fita, sendo que s_4 é sempre $\square$), então há instruções

$$(q, \tilde{s}_1, \tilde{s}_2, s_3, s_4, q', s'_1, d_1, s'_2, d_2, s'_3, d_3, s'_4, d_4)$$

com todas as combinações possíveis dos símbolos $\tilde{s}_1, \tilde{s}_2 \in \Sigma$ no qual Σ é o alfabeto utilizado pela máquina. Isto é, a decisão sobre se a instrução deve ser executada ou não no próximo passo da execução. A instrução acima só depende do valor da célula corrente da terceira fita, a área de trabalho — não interessa os valores das células correntes das duas primeiras instruções. Com isto nenhuma informação flui das fitas um e dois para a fita de saída;

10. se a exigência de que as máquinas parem depois de precisamente um certo número de passos, então teremos uma definição de f-g-simulação que se aplica mesmo para MT´s que nunca param. Então podemos ter máquinas M_1 e M_2 tal que $M_1(x) \uparrow$ e $M_2(x) \uparrow$ para todo x e M_1 não f-g-simula M_2. Então temos uma forma de diferenciar máquinas que nunca páram.

As duas grandes questões que esta pesquisa quer responder é: dada uma máquina M_1, quais outras máquinas ela está f-g-simulando? E quais máquinas a simulam?

Antes de tentar responder a estas questões, definiremos um tipo de máquina alternativo ao tipo de MT utilizado na definição de f-g-simulação. Este tipo de máquina possui as seguintes características:

(a) além dos símbolos Σ do alfabeto, são utilizados mais $3\,|\Sigma|$ símbolos. Para cada símbolo $s \in \Sigma$, há símbolos s_1, s_2 e s_{12}. Este símbolos novos servirão para que a máquina recorde a posição corrente da cabeça de leitura/gravação da primeira fita, a de entrada. A célula corrente sempre possuirá um símbolo do tipo s_1. Como simularemos uma outra máquina com a mesma fita de entrada, se a cabeça de leitura/gravação desta outra máquina estiver sob uma célula com símbolo que deveria ser s, então o símbolo de fato será s_2. Se as cabeças de leitura/gravação das duas máquinas estiverem sob uma mesma célula cujo símbolo deveria ser s, então o símbolo será s_{12};

(b) somente células alternadas da área de trabalho são utilizadas. Numerando as células em pares e ímpares, a máquina utiliza apenas as células de numeração par. Esta mesma técnica é empregada na fita de saída e na área de rascunho. As células ímpares serão empregadas na simulação de uma outra máquina.

Dada uma máquina de Turing M, o conjunto das máquinas M' tal que $M(x) = M'(x)$ para todo x é indecidível. Contudo, é possível listar o conjunto das máquinas de quatro fitas M' que f-g'-simulam M, se f e g são tomados de um conjunto enumerável de funções e a definição de máquina for aquela dada acima. A função g' é definida em termos de f e g como será mostrado na Proposição seguinte.

Proposição 7.2.1. *Seja C_{fg} um conjunto enumerável de funções adequadas de complexidade e M uma máquina de Turing. Então pode-se enumerar o conjunto*

$$\{M'' : M f\text{-}g'\text{-simula } M'' \text{ sendo } f, g \in C_{fg}\}$$

A função g' será definida em termos de f e g.

Demonstração. Seja E uma enumeração de todas as máquinas de Turing de quatro fitas possíveis. O produto cartesiano P de E com o conjunto C^2_{fg}, sendo ambos os conjuntos enumeráveis, é enumerável. Cada elemento do produto cartesiano P é da forma (M', f, g). Para cada um destes elementos, construiremos uma máquina M'' que simula M e M' de tal forma que M'' f-g'-simulará M. A idéia da prova é obrigar M' a ter a mesma área de trabalho e saída a cada $f(n)$ passos de M e $g(n)$ passos de M'. Assim necessariamente esta simulação de M' produzirá o mesmo resultado que M. Esta máquina M'' faz o seguinte:

(a) simula $f(n)$ passos de M e depois $g(n)$ passos de M', no qual $n = |x|$. M é simulado nas células pares da área de trabalho de M'' e M' nas células ímpares. Idem para a fita de saída. Os símbolos do tipo s_1, s_2 e s_{12} são utilizados na entrada para guardar a posição corrente da cabeça de leitura/gravação de M e M'. Então pode-se escrever na fita de entrada;

(b) copia a *at* de M para M'. Se $|at_M| < |at_{M'}|$, as posições extras da $at_{M'}$ (a área de trabalho de M') são preenchidas com o símbolo $\square$ (esta é a única ocasião em que se pode escrever este símbolo em alguma fita). Depois da cópia, a cabeça de leitura/gravação é deixada sempre na primeira posição da área de trabalho (os $f(n)$ passos de M começam a ser simulados sempre com a cabeça de leitura/gravação na posição 0 da *at* de M''. Idem para M' com $g(n)$ passos e posição 1);

(c) é feita uma cópia idêntica à copia da *at* na fita de saída.

A cópia da área de trabalho e da saída de M para M' obriga esta última máquina a f-g-simular M. De fato, é M'' que está f-g'-simulando M, pois são células ímpares das fitas de M'' que têm o mesmo valor que as células da M sendo simulada. Assim M'' está produzindo o mesmo resultado que M. Mas esta não é uma f-g-simulação, pois M'' precisa de muito mais do que $g(n)$ passos para ter a suas fitas com conteúdo igual às de M. Vejamos qual o valor de g', considerando que M'' tem o trabalho extra de simular M, M', contar o número de passos ($f(n)$ de M e $g(n)$ de M') e ainda fazer a cópia das fitas.

Assumiremos que M executa em tempo $F(n)$ e M' executa em tempo $G(n)$.

A cópia da área de trabalho e da fita de saída pode ser feita em $c_1 h(n)$ passos no qual c_1 é uma constante e $h(n) = \text{máx}\{f(n), g(n)\}$. A simulação de $f(n)$ passos de M pode ser feita em no máximo $c_2 f(n)^2$ passos de M'' (é o que gasta uma ineficiente MTU). Para simular $g(n)$ passos de M', M'' gastará $c_3 g(n)^2$ passos. Mas deve-se simular a execução de exatamente $f(n)$ instruções de M. A área de rascunho é utilizada para guardar quantos passos de M (ou M') já foram executados até o momento. Para controlar este número assumiremos que cada passo é controlado em $c_4(f(n) + g(n))$ passos de M''. No total, M'' gastará um número de passos

$$g'(n) = c_1 h(n) + (c_2 f(n)^2 + c_3 g(n)^2) c_4 (f(n) + g(n)) \leqslant c h(n)^3$$

para cada $f(n)$ passos de M. Então M'' f-g' simula M. Para cada elemento (M', f, g) da enumeração de P conseguimos uma máquina que f-g'-simula M. □

Se M_1 f-g-simula M_2, então $mf(n)$ instruções de M_1 deve produzir o mesmo resultado (área de trabalho e saída) que $mg(n)$ M_2 para $1 \leqslant m \leqslant h(n)$. A partir deste requerimento, é possível deduzir uma relação entre $f(n)$ e $g(n)$, o que é feito no próximo parágrafo.

A execução das $f(n)$ instruções iniciais de M_1 só pode utilizar $f(n)$ células da área de trabalho (pois cada instrução de uma MT só pode mover a cabeça de leitura/gravação uma célula, seja para a esquerda ou direita). Idem para $g(n)$ e M_2. Depois de $f(n)$ instruções de M_1 e $g(n)$ de M_2, as áreas de trabalho e saída devem ser iguais. Isto implica em que $f(n)$ e $g(n)$ não podem ser funções quaisquer, a menos que uma delas entre em um laço infinito. Vejamos porque. Pode ser provado que

$$\text{SPACE}(t(n)) \subseteq \text{TIME}(k^{\log n + t(n)})$$

Logo se uma MT N utiliza espaço $t(n)$, há uma outra máquina N' que produz o mesmo resultado e que utiliza tempo $k^{\log n + t(n)}$. A constante k depende apenas da máquina (tamanho do alfabeto e número de estados). De fato, $k^{\log n + t(n)}$ é o número de configurações possíveis para uma máquina de Turing que utiliza $t(n)$ células. Se uma máquina utiliza $t(n)$ células e gasta mais do que $k^{\log n + t(n)}$ passos, então com certeza ela repetiu uma configuração uma vez. Se repetiu uma vez, repetirá sempre e entrará em um laço infinito. Mas só estamos interessados em máquinas que executam um número de passos dado exatamente por uma função de $\mathbb{N}$ em $\mathbb{N}$. Então $f(n)$ e $g(n)$ devem estar relacionadas de alguma forma — veja o próximo exemplo.

Se M_1 n-c^{n^2}-simula M_2, c constante, então há uma MT que produz o mesmo resultado que M_1 em tempo $k^{\log n + n}$. Mas $k^{\log n + n}$ é assintoticamente menor do que c^{n^2}. Então M_2 não é a máquina "ideal" para simular M_1: há uma que toma um menor número de passos.

Uma das linhas de pesquisa possível de ser explorada é tentar relacionar as funções f e g por meio do espaço que elas utilizam. Possivelmente esta linha de pesquisa necessitará de modificações na definição de f-g-simulação.

7.3 Conclusão

O objetivo desta pesquisa é caracterizar de alguma forma os conjuntos das MT que f-g-simulam ou são f-g-simuladas por uma dada MT M. Com isto pode-se também relacionar classes de complexidade da seguinte forma: suponha que uma linguagem L seja decidida por uma máquina de Turing M que executa em tempo $F(n)$. L pertence a uma classe de complexidade C das máquinas que executam em tempo $G(n)$ no qual $F(n) = O(G(n))$. Se for possível conseguir informações sobre o conjunto das MT´s que f-g-simulam M, pode ser possível encontrar ou provar a inexistência de uma MT que simula M em tempo menor do que $F(n)$ (aqui provavelmente será necessário restringir as funções $g(n)$ das máquinas que simulariam M). Em qualquer dos dois casos, teria-se uma informação sobre a classe de complexidade de L. Se existir uma máquina que f-g-simula M em tempo menor do que $F(n)$, possivelmente L pode ser enquadrada em uma classe de complexidade que está contida em C, a classe original de L. Se tal máquina não existir, nada se pode afirmar, exceto que não existe uma MT que produz o mesmo resultado que M e que obedece a algumas condições (como a função $g(n)$ da simulação pertencer a certo conjunto de funções).

Note que para conseguir este objetivo é muito provável que seja necessário criar outras definições de simulação, definições mais adequadas para responder a perguntas mais específicas (sobre determinada linguagem ou classe de complexidade).

O conjunto de máquinas que são equivalentes a uma MT M é indecidível. E não há algoritmo que tome a codificação de uma MT M e retorne a mais eficiente máquina, em termos do tempo de execução, que é equivalente a M. Se isto fosse possível, todos os problemas relacionados a classes de complexidade estariam resolvidos. Poder-se-ia, por exemplo, tomar uma máquina de Turing que decide a linguagem SAT (o problema da Satisfabilidade) e encontrar a mais eficiente máquina que a decide. Se esta máquina executasse em tempo polinomial, teríamos P = NP. Se executasse em tempo exponencial, teríamos P ≠ NP. Como isto não é possível, foi criado o conceito de f-g-simulação que não é tão geral quanto o conceito de "máquinas equivalentes", mas sobre o qual é mais fácil de pesquisar. Este conceito de simulação não nos leva à mais eficiente máquina equivalente a uma máquina M dada, mas possivelmente poderá nos levar às características de todas as máquinas equivalentes à M que obedecem a certas restrições. E sobre estas características poderemos tirar conclusões importantes.

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_set_computer, 2009.