UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Tiago Royer

MÁQUINAS DE TURING NÃO-DETERMINÍSTICAS COMO COMPUTADORES DE FUNÇÕES

Florianópolis

2015

Tiago Royer

MÁQUINAS DE TURING NÃO-DETERMINÍSTICAS COMO COMPUTADORES DE FUNÇÕES

Trabalho de Conclusão de Curso submetido ao Programa de Graduação em Ciência da Computação para a obtenção do Grau de Bacharel em Ciência da Computação.
Orientadora: Profa. Dra. Jerusa Marchi

Florianópolis

2015

Tiago Royer

MÁQUINAS DE TURING NÃO-DETERMINÍSTICAS COMO COMPUTADORES DE FUNÇÕES

Este Trabalho de Conclusão de Curso foi julgado aprovado para a obtenção do Título de "Bacharel em Ciência da Computação", e aprovado em sua forma final pelo Programa de Graduação em Ciência da Computação.

Florianópolis, 01 de novembro 2015.

Prof. Dr. Mário Antônio Ribeiro Dantas
Coordenador do Curso

Banca Examinadora:

Profa. Dra. Jerusa Marchi
Orientadora
Universidade Federal de Santa Catarina

Profa. Dra. Karina Girardi Roggia
Universidade do Estado de Santa Catarina

Prof. Dr. Ricardo Azambuja Silveira
Universidade Federal de Santa Catarina

Prof. Dr. Rosvelter João Coelho da Costa
Universidade Federal de Santa Catarina

À minha mãe, Venida

AGRADECIMENTOS

Agradeço muito minha família, em especial à minha mãe, por sempre terem me apoiado, guiado e auxiliado ao longo dos últimos 20 anos.

Agradeço minha orientadora, Jerusa, que me acolheu logo no início do curso, por todas as conversas e orientações, que muitas vezes extrapolaram os limites da Academia.

Agradeço aos professores do Programa Avançado de Matemática (Melissa, Gilles, Fernando), que me ajudaram a construir a maturidade matemática que me foi extremamente útil nos quatro anos de graduação.

Agradeço as discussões com Santana, Schultz, Gabriel e Abouhatem sobre os algoritmos da Maratona de Programação.

Agradeço todas as discussões produtivas (e improdutivas) com o pessoal do laboratório IATE.

E, por último, agradeço o companheirismo dos meus amigos ao longo desta jornada.

Even if one works basically all week trying to prove $\mathsf{P} \neq \mathsf{NP}$, one should set aside Friday afternoon for trying to prove $\mathsf{P} = \mathsf{NP}$.

(John Edward Hopcroft)

RESUMO

Os conceitos de complexidade de tempo e espaço para máquinas de Turing determinísticas e não-determinísticas, embora compartilhem muitos teoremas, costumam ser definidos individualmente. Blum (1967, p. 324) define um "recurso computacional" como sendo algo que satisfaz dois axiomas; desta forma, podemos tratar da complexidade computacional de maneira axiomática e unificar os teoremas que costumam ser demonstrados separadamente.
Entretanto, Blum (1967, p. 324) não define a noção de recurso computacional para decisores, mas sim para computadores de funções de inteiros (isto é, máquinas de Turing que computam funções $f : \mathbb{N} \to \mathbb{N}$ (HOPCROFT; ULLMAN, 1979, p. 151)). Portanto, para definir as classes P e NP em termos dos axiomas de Blum, precisamos interpretar um decisor como um computador de uma função de inteiros. Para o caso determinístico é simples, pois a máquina determinística possui apenas uma sequência de computação. O caso não-determinístico é mais complexo, e é o objeto de estudo deste TCC.
Pretendo definir o conceito de "função não-determinística" de forma que a composição destas funções corresponda, de alguma forma, à noção de composição de funções determinísticas.

Palavras-chave: Complexidade computacional, funções não-determinísticas, composição de funções.

ABSTRACT

Normally, the concepts of time and space complexity for deterministic and nondeterministic machines are defined separately, although they share several similar theorems. Blum (1967, p. 324) define a "computational resource" using two axioms; this allows for an axiomatic treatment of the computational complexity, enabling us to provide a single proof for these analogous theorems.

However, the Blum axioms are not defined in terms of deciders, but for integer function computers (that is, Turing machines that compute functions of the form $f : \mathbb{N} \to \mathbb{N}$ (HOPCROFT; ULLMAN, 1979, p. 151)). Therefore, to define P and NP via Blum axioms, we need to see a decider as an integer function computer. The case where the machine is deterministic is easy, because it have a single sequence of computation. The nondeterministic case, which is more complex, is studied in this text.

In this text, we define the concept of "nondeterministic function" in a way that the composition of nondeterministic functions correspond to the composition of the analogous deterministic functions.

Keywords: Computational complexity, nondeterministic functions, function composition.

LISTA DE SÍMBOLOS

$f(x) \simeq g(x)$ f concorda com g em x 24
$\langle x \rangle$ Codificação em binário do objeto matemático x 26
f_w Função computada pela máquina codificada pela palavra w 28
$\mathcal{T}$ Complexidade de Tempo 33
$\mathcal{S}$ Complexidade de Espaço 34
$\mathcal{NT}$ Complexidade de Tempo não-determinística 34
$\mathcal{NS}$ Complexidade de Espaço não-determinística 34
$\mathcal{B}$ Conjunto das funções booleanas 47
M^A Máquina de Turing M com oráculo A 61
$L^A(M)$ Linguagem da máquina M, quando equipada com o oráculo A . 61
ϕ^A Enumeração das funções recursivas em A 62
L_u^1 Problema da parada 63
S_1 Problema da vacuidade 63
L_u^n Problema da parada, generalizado via oráculos 64
S_n Problema da vacuidade, generalizado via oráculos 64
Σ_n Linguagens decidíveis usando um oráculo para L_u^n 64
$\mathsf{P}^{\mathcal{A}}$ Classe P relativa à $\mathcal{A}$ 68
$\mathsf{NP}^{\mathcal{A}}$ Classe NP relativa à $\mathcal{A}$ 68
$\mathsf{coNP}^{\mathcal{A}}$ Classe coNP relativa à $\mathcal{A}$ 68
Δ_n^p Generalização de P na hierarquia polinomial 69
Σ_n^p Generalização de NP na hierarquia polinomial 69
Π_n^p Generalização de coNP na hierarquia polinomial 69
FP Funções determinísticas polinomiais 77
FNP Funções não-determinísticas polinomiais 77
coFNP Funções co-não-determinísticas polinomiais 77
$T(M,x,n)$ "Respostas" da computação de M em x por n passos. 79
HaltFNP Versão de FNP do problema da parada 79
d Função de decodificação de subpalavras 81
FdSAT SAT funcional "composto" com d 81
CallbackSAT Versão funcional de SAT com um callback 83
$\mathsf{FP}^{\mathcal{A}}$ FP usando um oráculo de $\mathcal{A}$ 84
$\mathsf{FNP}^{\mathcal{A}}$ FNP usando um oráculo de $\mathcal{A}$ 84

$\mathsf{coFNP}^{\mathcal{A}}$ coFNP usando um oráculo de $\mathcal{A}$ 84
Σ_n^f Versão funcional de Σ_n^p 86
Δ_n^f Versão funcional de Δ_n^p 86
Π_n^f Versão funcional de Π_n^p 86
$\mathcal{F}^\circ$ Fecho compositivo de $\mathcal{F}$ 90
OptP Classe polinomial de problemas de otimização 100
$\delta(E)$ Ação tomada no estado de computação E 109
$A(E)$ Efeito da ação A no estado de computação E 110

SUMÁRIO

1	**INTRODUÇÃO**	19
1.1	OBJETIVOS	21
1.1.1	**Objetivos específicos**	21
1.2	ESTRUTURAÇÃO DO TEXTO	22
2	**ENUMERAÇÃO DE FUNÇÕES RECURSIVAS**	23
2.1	MODELO DE MÁQUINA UTILIZADO	23
2.2	FUNÇÕES RECURSIVAS	24
2.2.1	**Codificação de objetos**	25
2.2.2	**Funções de várias variáveis**	27
2.2.3	**Funções de inteiros**	27
2.3	ENUMERAÇÃO DE FUNÇÕES RECURSIVAS	28
2.4	TEOREMA DA MÁQUINA UNIVERSAL	29
2.5	TEOREMA S_{MN}	29
2.6	ENUMERAÇÕES DE GÖDEL ACEITÁVEIS	30
3	**COMPLEXIDADE COMPUTACIONAL**	33
3.1	TEORIA DE COMPLEXIDADE AXIOMÁTICA: AXIOMAS DE BLUM	33
3.1.1	**Classes de Complexidade**	40
3.1.2	**Teorema da União**	43
3.2	MEDIDAS DE COMPLEXIDADE COMPUTACIONAL	47
3.2.1	**Relações entre medidas de Complexidade**	55
3.2.2	**Teorema de Savitch**	56
3.2.3	**Principais Classes de Complexidade Computacional**	58
4	**ORÁCULOS**	61
4.1	DEFINIÇÃO	61
4.2	PROBLEMAS INDECIDÍVEIS E EQUIVALÊNCIA DE ORÁCULOS	63
4.3	HIERARQUIA ARITMÉTICA	64
4.4	REDUÇÕES E ORÁCULOS	66
4.5	HIERARQUIA POLINOMIAL	68
5	**FUNÇÕES NÃO-DETERMINÍSTICAS**	73
5.1	MOTIVAÇÃO	73
5.2	DEFINIÇÃO	74
5.3	COMPLEXIDADE FUNCIONAL	76
5.4	FUNÇÕES FNP-COMPLETAS	78
5.5	ORÁCULOS FUNCIONAIS	83
5.6	HIERARQUIA POLINOMIAL	86

5.7 COMPOSIÇÃO DE FUNÇÕES 90
6 COMPARAÇÃO COM OUTROS TRABALHOS 97
6.1 PROBLEMAS DE BUSCA VS PROBLEMAS DE DECISÃO (PAPADIMITRIOU) 97
6.2 CONJUNTO POTÊNCIA COMO CONTRADOMÍNIO (ANDREEV) 98
6.3 DEFINIÇÃO DE HOPCROFT E ULLMAN 99
6.4 PROBLEMAS DE OTIMIZAÇÃO (KRENTEL) 100
7 CONSIDERAÇÕES FINAIS 101
REFERÊNCIAS 103
APÊNDICE A – MÁQUINAS DE TURING 107

1 INTRODUÇÃO

A teoria da computabilidade classifica os problemas computacionais entre aqueles que podem e não podem ser resolvidos mecanicamente. Existem diversos dispositivos matemáticos que tentam capturar esta noção de "resolvidos mecanicamente"; o modelo que usaremos neste trabalho é a máquina de Turing.

Dizemos que um problema é *decidível* se ele puder ser resolvido por uma máquina de Turing que sempre para. Embora as máquinas de Turing não se pareçam com os computadores modernos, estes dois dispositivos são equivalentes, no sentido de que todos os problemas resolvíveis por um deles também é resolvível pelo outro.[1] Portanto, podemos pensar nos problemas decidíveis como aqueles que podem ser resolvidos com o uso de um computador (HOPCROFT; MOTWANI; ULLMAN, 2001, p. 307). Embora a maior parte dos problemas encontrados no dia-a-dia seja decidível, existem alguns problemas de importância teórica e prática que não o são. Um exemplo notável é determinar se dois programas são equivalentes.[2]

Entretanto, apenas ser decidível não é o suficiente. Existem alguns problemas (de fato, categorias inteiras deles) que, por mais rápido que seja o computador que esteja tentando resolvê-los, o tempo necessário torna-se excessivamente grande muito rapidamente. Chamamos estes problemas de *intratáveis* (HOPCROFT; MOTWANI; ULLMAN, 2001, p. 1). Por mais que eles sejam decidíveis, pode ser que demore até o Sol esfriar (evento que, estima-se, vai ocorrer em 5 bilhões de anos) para que o computador devolva a solução.

É aqui que entra a teoria de complexidade computacional. Essencialmente, *complexidade* é a quantidade de recursos que são gastos para obter a solução de determinado problema. Geralmente nos concentramos em um único recurso; os dois principais são o tempo de execução e a quantidade de memória usada (HOPCROFT; ULLMAN, 1979, p. 285).

A complexidade computacional classifica os problemas de acordo com os recursos necessários para que sejam resolvidos. Por exemplo, a classe de complexidade P é o conjunto de todos os problemas que podem ser resolvi-

[1] Eles são equivalentes até certo ponto. Máquinas de Turing são objetos matemáticos com "memória infinita", enquanto que os computadores estão limitados a uma quantidade finita de memória; esta limitação impõe restrições no tamanho das entradas, o que não ocorre com as máquinas de Turing.

[2] Este problema possui importância prática: caso existisse um algoritmo para determinar equivalência entre programas, poderíamos usar um computador para saber se determinado refatoramento ou otimização de código não alterou acidentalmente o significado do programa; isto é, a mudança não introduziu um bug.

dos por uma máquina de Turing determinística em tempo polinomial; isto é, alguma máquina de Turing determinística resolve qualquer instância de tamanho n usando no máximo $p(n)$ unidades de tempo, para algum polinômio p. Similarmente, a classe NP contempla os problemas resolvíveis em tempo polinomial por máquinas de Turing *não* determinísticas.

Estes dois conjuntos protagonizam o maior e mais importante problema da teoria de complexidade computacional, e um dos mais importantes problemas em aberto de toda a matemática. Ei-lo:

$$\mathsf{P} \stackrel{?}{=} \mathsf{NP}$$

(SIPSER, 2006, p. 270). Sabemos que $\mathsf{P} \subseteq \mathsf{NP}$, pois toda máquina determinística pode ser vista como uma máquina não-determinística que escolheu não "usar" seu não-determinismo. A maior parte dos pesquisadores acredita que esta inclusão é estrita (GASARCH, 2012, p. 54).

Os modelos determinístico e não-determinístico de máquinas de Turing são equivalentes, no sentido de que qualquer problema que pode ser resolvido por um também pode ser resolvido por outro. Entretanto, o modelo não-determinístico aparenta ser exponencialmente mais rápido que o modelo determinístico; de fato, para um subconjunto vasto de NP, os problemas NP-completos (considerados os mais difíceis da classe NP), são conhecidos apenas algoritmos determinísticos exponenciais.

Curiosamente, o ganho exponencial de desempenho só é atingido se considerarmos o tempo de execução. A complexidade de espaço permite apenas um ganho quadrático; este resultado é conhecido como teorema de Savitch (ARORA; BARAK, 2009, p. 86).

Apesar desta discrepância entre o ganho de eficiência via não-determinismo, complexidade de tempo e complexidade de espaço compartilham muitos teoremas semelhantes; isso sugere que elas são casos particulares de uma noção abstrata, a de "recurso computacional". Blum (1967, p. 324) formalizou a noção de "medida de complexidade" através de dois axiomas, aplicáveis a qualquer modelo de máquina que compute funções de inteiros.[3] Usando apenas estes dois axiomas, muitos teoremas a respeito de complexidade computacional, que costumam ser apresentados separadamente em livros didáticos sobre o assunto, podem ser demonstrados de maneira universal.

Para máquinas de Turing determinísticas, os resultados são facilmente aplicados; isto é, é intuitivo interpretar uma máquina de Turing determinística como um computador de alguma função. Podemos, por exemplo, entender o

[3] Embora usamos o termo "funções de inteiros", as funções tratadas possuem $\mathbb{N}$ como domínio e contradomínio. (Eu mantive a terminologia "funções de inteiros" para ser compatível com o termo em inglês *integer functions*, utilizado por Hopcroft e Ullman (1979, p. 151).)

conteúdo inicial da fita como a entrada da função e o conteúdo da fita quando a máquina para como a saída da função.

Entretanto, para máquinas de Turing não-determinísticas, há um obstáculo elementar: os axiomas de Blum dizem respeito a funções. O que é uma "função não-determinística"?

Hopcroft e Ullman (1979, p. 313) dão uma possível definição: uma máquina não-determinística computa uma função se, e somente se, todos os ramos de computação encerram com a mesma palavra na fita; neste caso, esta palavra é a saída da função para a respectiva entrada.

Esta definição não é satisfatória; isto é, ela não generaliza a noção de não-determinismo para decisores. Se todos os ramos de computação encerram com a mesma palavra, poderíamos simplesmente fixar uma das possíveis escolhas na função de transição; afinal, o resultado sempre será o mesmo. Portanto, em essência, temos o poder computacional de uma máquina determinística.

Isso ignora a noção de que máquinas não-determinísticas aparentemente possuem velocidade de execução exponencialmente maior do que máquinas determinísticas, impedindo que relacionemos axiomas de Blum ao problema P vs NP.

Neste trabalho, proporei uma nova definição de função não-determinística, de maneira a preservar este aparente ganho exponencial de tempo de execução, permitindo, portanto, um paralelo com o problema P vs NP.

1.1 OBJETIVOS

Desenvolver uma nova definição de funções não-determinísticas, de modo a generalizar naturalmente o aparente aceleramento exponencial que obtemos ao conceder não-determinismo a máquinas de Turing.

1.1.1 Objetivos Específicos[4]

1. Manter compatibilidade com os axiomas de Blum.
2. Comparar a definição dada com os trabalhos de Papadimitriou (1994, p. 229), Andreev (1994, p. 3), Hopcroft e Ullman (1979, p. 313), e Krentel (1988, p. 493).
3. Desenvolver a noção de composição de funções não-determinísticas.

[4] Como o objetivo geral é definir um novo conceito, os objetivos específicos são voltados a validá-lo.

4. Relacionar a composição de funções não-determinísticas com a hierarquia polinomial.

1.2 ESTRUTURAÇÃO DO TEXTO

Os axiomas de Blum definem a noção de complexidade computacional não apenas para máquinas de Turing, mas sim para qualquer modelo de computação que satisfaça algumas restrições. O capítulo 2 contém a maquinaria matemática para formular, precisamente, quais são estas restrições. Este capítulo discorre sobre funções recursivas, formas de enumerá-las e características desejáveis destas enumerações.

O capítulo 3 apresenta os axiomas de Blum e o conceito associado de classes de complexidade computacional. Também são definidas formalmente as duas principais classes de complexidade computacional — P e NP. Neste capítulo aparece o teorema da união, que justifica (do ponto de vista da definição axiomática de "complexidade") chamar as classes P, NP, PSPACE etc. de "classes de complexidade".

A noção de "oráculo computacional" é explorada no capítulo 4. Aqui, são definidas as hierarquias aritmética e polinomial.

Finalmente, no capítulo 5, é definida "função não-determinística". Podemos finalmente estender a noção de complexidade computacional para máquinas de Turing não-determinísticas. Também é definido "oráculo funcional", que permite construir uma hierarquia de funções análoga às hierarquias aritmética e polinomial. E, atingindo um dos objetivos do TCC, demonstramos que o fecho compositivo da classe Σ_n^f é a classe Δ_{n+1}^f; de certa forma, encontramos uma relação na hierarquia polinomial através da composição de funções não-determinísticas.

Na maior parte do texto, o conceito de "máquina de Turing" é tratado informalmente. Entretanto, algumas afirmações feitas no texto (por exemplo, a de que uma representação em binário identifica unicamente uma máquina de Turing) depende da definição formal deste dispositivo — isto é, dos "detalhes de implementação". Estes detalhes estão no apêndice A.

2 ENUMERAÇÃO DE FUNÇÕES RECURSIVAS

A teoria de complexidade axiomática proposta por Blum (1967, p. 324), que estudaremos na seção 3.1, é *independente de máquina*; isto é, ela é aplicável a qualquer modelo de máquina "bem comportado". Nesta seção, veremos como generalizar a noção de "modelo de máquina" e estudaremos as condições para que ele seja "bem comportado"; isto é, que propriedades o modelo de máquina precisa satisfazer para que possamos definir os axiomas de Blum.

Entenderemos um modelo de máquina como uma maneira de atribuir um significado a cada cadeia binária.[1] A ideia é que cada cadeia represente o "código-fonte" de alguma função recursiva; o modelo de máquina, então, dirá como devemos interpretar este código-fonte.

Precisaremos, antes, definir a noção de *função recursiva*. Para isso, usaremos um modelo específico de máquina de Turing, que será definido na seção 2.1.

A definição de *função recursiva* está na seção 2.2. As seções 2.4 e 2.5 contém teoremas S_{mn} e da máquina universal, que são as características que exigiremos de um modelo de máquina para "merecer" o título de "bem comportado". Esta amarração está na seção 2.6.

2.1 MODELO DE MÁQUINA UTILIZADO

Para os propósitos deste trabalho, assumiremos que todas as máquinas de Turing possuem três fitas.

- A primeira fita será utilizada apenas para a entrada da máquina; ela sempre conterá uma palavra de $\{0,1\}^*$, e o seu conteúdo não poderá ser alterado.
- A segunda fita será a fita de trabalho da máquina. A máquina poderá ler e escrever símbolos arbitrários nesta fita infinita, que começará totalmente em branco.
- A terceira fita será utilizada apenas para saída. O cabeçote desta fita poderá realizar apenas três movimentos:
  - Escrever um 0 e mover-se para a direita;
  - Escrever um 1 e mover-se para a direita;
  - Manter-se parada.

[1] Estamos identificando uma máquina com sua representação em binário.

A função de transição da máquina poderá depender apenas das duas primeiras fitas. Como estaremos interessados no cálculo de funções, não precisaremos de estados de aceitação ou rejeição; basta não haver transição partindo do estado atual para que a máquina pare. (Podemos codificar "aceitação" como sendo a escrita de um único 1 da fita de saída e "rejeição" como a escrita de um único 0.)

Observe que não há perda de generalidade ao fazer com que a fita de saída seja somente-escrita, pois a máquina poderia construir a saída toda na máquina de trabalho, e escrevê-la de uma só vez na fita de saída.

Em relação à definição usual de "máquina de Turing", nossa versão não possui a definição de Σ, o alfabeto de entrada. Em todo o TCC, assumiremos que $\Sigma = \{0,1\}$; isto é, todas as máquinas trabalharão com o sistema binário. Observe que isto não é uma restrição significativa, pois podemos codificar outros alfabetos em binário.

Este modelo precisará ser ajustado no capítulo 4, quando adicionaremos mais uma fita à máquina, que será usada para comunicação com o oráculo.

O apêndice A contém a definição formal deste modelo.

2.2 FUNÇÕES RECURSIVAS

Máquinas de Turing podem nem sempre parar ao computar uma função; esta é uma "deficiência" de todos os modelos de máquina "bem comportados". Portanto, podem haver duas máquinas de Turing tais que, sempre que ambas as máquinas param, o "valor de retorno" destas máquinas é o mesmo, mas existem entradas em que uma máquina para e outra não. De certa forma, as duas "funções" calculadas por estas máquinas estão definidas em domínios diferentes. Não faz sentido, portanto, perguntar se

$$f(x) = g(x)$$

para todos os x, pois pode existir algum x tal que um lado está definido e outro não. Estamos interessados nos casos em que ambas as funções estão definidas *e* são iguais; a seguinte notação condensa esta exigência.

Notação. Dadas duas funções $f, g : A \to B$, escreveremos

$$f(x) \simeq g(y)$$

para expressar o fato de que f está definida em x se, e somente se, g está definida em y; e, caso ambas estejam definidas, $f(x) = g(y)$. (Esta notação

foi extraída de Epstein e Carnielli (2008, p. 124). Nos casos em que $x = y$, podemos ler esta notação como "f e g concordam em x".)

Esta notação simplificará o enunciado de alguns teoremas.

Exemplo 2.1. Defina $f, g : \mathbb{R} \to \mathbb{R}$, dadas por

$$f(x) = \sqrt{x^2} \qquad\qquad g(x) = (\sqrt{x})^2.$$

Então, para $x \geq 0$,

$$f(x) \simeq g(x),$$

mas isso não é válido para $x < 0$. Além disso, por exemplo,

$$g(-1) \simeq g(-3)$$

pois tanto $g(-1)$ quanto $g(-3)$ não existem. □

Notação. Seja M uma máquina de Turing determinística e $x \in \{0,1\}^*$ uma palavra. Se M parar ao computar x (isto é, ao colocar a palavra x na fita de entrada de M, a sequência de passos de computação que se seguem é finita), denotaremos por $M(x)$ a palavra deixada na fita de saída. Caso M não pare ao computar x, deixaremos $M(x)$ indefinido.

Observe que a notação $M(x)$ coincide com a notação de chamada de função. É exatamente esta a analogia que pretendemos fazer.

Definição 2.2. Seja M uma máquina de Turing. A *função computada por M* é a função parcial $f : \{0,1\}^* \to \{0,1\}^*$ tal que

$$f(x) \simeq M(x)$$

para todo x.[2]

Uma função f é *recursiva parcial* se for computada por alguma máquina M. Se f estiver definida para todo $x \in \{0,1\}^*$, dizemos que f é *recursiva total*. [3]

2.2.1 Codificação de objetos

Conforme observado por Arora e Barak (2009, p. 2), não perdemos generalidade ao restringir funções a palavras de $\{0,1\}^*$, pois podemos codificar

[2] Esta definição é uma variação da definição de Hopcroft e Ullman (1979, p. 151); em particular, utilizaremos funções que mapeiam palavras para palavras em vez de funções que mapeiam números naturais para números naturais.

[3] Não confundir com o conceito de *recursão* em linguagens de programação.

objetos matemáticos complexos usando o sistema binário — de fato, é exatamente isso o que fazemos nos computadores modernos. Podemos, por exemplo, codificar inteiros usando sua representação binária, e grafos podem ser representados por sua matriz de adjacência ou pelos pares de nodos adjacentes. Garey e Johnson (1979, p. 10) argumentam que não é necessário se ater aos detalhes da representação dos objetos, pois quaisquer duas codificações "razoáveis" destes objetos diferirão apenas polinomialmente de tamanho.

Neste trabalho, dois objetos aparecerão mais frequentemente: máquinas de Turing e tuplas. Uma máquina de Turing pode ser representada em binário listando todos elementos da função de transição (HOPCROFT; ULLMAN, 1979, p. 182); os detalhes estão no apêndice A. Denotaremos por $\langle M \rangle$ a representação em binário da máquina M.

Já uma tupla $(x_1, x_2, \ldots, x_n)$ pode ser codificada combinando as representações em binário de $x_1, \ldots, x_n$. A ideia é trocar, nas representações em binário, 0 por 00 e 1 por 01; então, a cadeia 1 faz o papel de separador. Denotaremos por $\langle x_1, x_2, \ldots, x_n \rangle$ a representação binária da tupla $(x_1, x_2, \ldots, x_n)$.[4] Observe que esta codificação é não-ambígua; podemos recuperar a representação em binário de cada um dos x_i a partir de $\langle x_1, \ldots, x_n \rangle$.

Exemplo 2.3. Podemos representar o número 2 por 10 e o número 5 por 101, em binário. A transformação mencionada acima troca 10 por 0100 e 101 por 010001; portanto, a tripla ordenada $(2,5,2)$ é representada por

$$\langle 2,5,2 \rangle = 0100\ 1\ 010001\ 1\ 0100. \qquad \square$$

É importante notar que codificaremos números naturais sempre usando sua expansão binária. Hopcroft e Ullman (1979, p. 151), por exemplo, utilizam uma representação unária; isto é, o número n é representado por 0^n, Entretanto, esta codificação faz com que o tamanho da palavra resultante seja proporcional a n. Já a expansão binária de n possui tamanho $\lfloor \log_2 n \rfloor + 1$; ou seja, o tamanho da expansão binária é proporcional ao logaritmo do número.

Em outras palavras, a representação em unário é exponencialmente maior que a representação em binário. Isto será importante ao considerarmos classes de complexidade; portanto, seguiremos a convenção de Arora e Barak (2009, p. 2) de usar sempre a expansão binária para representar números naturais.

[4] Em geral, $\langle x \rangle$ denotará a representação em binário do objeto matemático x. Arora e Barak (2009, p. 2) usa $\llcorner x \lrcorner$ para o mesmo fim; usaremos a notação $\langle x \rangle$ por conformidade com a notação de Hopcroft e Ullman (1979, p. 182).

2.2.2 Funções de várias variáveis

Observe que o termo "função recursiva" foi definido apenas para funções de uma variável. Entretanto, em alguns pontos do texto, precisaremos nos referir a funções de mais de uma variável.

É possível estender as máquinas de Turing para trabalhar diretamente com funções de várias variáveis. Por exemplo, Hopcroft e Ullman (1979, p. 151) fazem a extensão no contexto de funções de inteiros. Dada uma máquina M, para calcular o valor de $M(i)$, executamos M na entrada 0^i. Os vários argumentos são separados por caracteres 1; isto é, para calcular $M(i_1, i_2, \ldots, i_k)$, executamos M na entrada $0^{i_1}10^{i_2}1\ldots10^{i_k}$.

Para não precisar alterar a definição de máquinas de Turing, recorreremos à codificação $\langle\cdot\rangle$, definida na seção 2.2.1.

Definição 2.4. Uma função parcial $f : (\{0,1\}^*)^k \to \{0,1\}^*$ é *recursiva parcial* se existir alguma função recursiva parcial $g : \{0,1\}^* \to \{0,1\}^*$ tal que

$$g(\langle i_1, i_2, \ldots, i_k\rangle) \simeq f(i_1, i_2, \ldots, i_k).$$

Funções recursivas totais são definidas analogamente.

Observe que esta definição não é autorreferencial pois a recursividade de g é estabelecida na definição 2.2.

2.2.3 Funções de inteiros

Embora na maior parte deste TCC usaremos funções da forma

$$f : \{0,1\}^* \to \{0,1\}^*,$$

em alguns pontos do texto precisaremos usar números naturais para contar coisas — por exemplo, quantos passos uma máquina de Turing fez ao computar uma função. Além disso, queremos usar máquinas de Turing para manipular estas funções; portanto, é importante definir o que significa para uma função dessas ser "recursiva".

Definição 2.5. Uma função parcial $f : \mathbb{N} \to \mathbb{N}$ é *recursiva* se existir uma função recursiva parcial $g : \{0,1\}^* \to \{0,1\}^*$ tal que, para todo n,

$$g(\langle n\rangle) \simeq \langle f(n)\rangle.$$

O caso em que o domínio ou o contradomínio é $\{0,1\}^*$, ou f for uma função de várias variáveis, é definido analogamente.

Em outras palavras, f é recursiva quando existir uma função g que calcula f usando a representação em binário dos números.

Exemplo 2.6. A função de soma,

$$+ : \mathbb{N} \times \mathbb{N} \to \mathbb{N},$$

é recursiva. Podemos construir uma máquina de Turing M que, ao receber na entrada o valor $\langle m,n \rangle$ — isto é, a representação em binário de m e n — soma estes dois números (usando adição com *carry*, o "vai um") e escreve a soma na fita de saída. Esta máquina satisfaz

$$M(\langle m,n \rangle) = \langle m+n \rangle,$$

portanto, a operação de soma é recursiva. □

Todas as operações matemáticas que são realizáveis num computador correspondem a funções recursivas; portanto, nos limitarmos a funções de inteiros recursivas não é uma restrição significativa.

2.3 ENUMERAÇÃO DE FUNÇÕES RECURSIVAS

A existência de uma codificação em binário de máquinas de Turing permite, de certa forma, enumerar todas as funções recursivas parciais — basta listar todos os códigos possíveis para máquinas de Turing. Estritamente falando, não são as funções que estão sendo listadas (isso seria impossível pois funções são objetos matemáticos infinitos), mas sim códigos em binário para máquinas que as computam.

A construção da codificação das máquinas de Turing não é "injetora"; isto é, existem máquinas M e N diferentes tais que

$$\langle M \rangle = \langle N \rangle.$$

Entretanto, a construção foi feita de forma que as funções computadas por M e N coincidam sempre que a equação acima for satisfeita. Isto é, se $\langle M \rangle = \langle N \rangle$, então, para todo x,

$$M(x) \simeq N(x).$$

Ou seja, M e N computam exatamente a mesma função parcial. Portanto, dada uma cadeia $w \in \{0,1\}^*$, existe uma única função recursiva parcial que é computada pelas máquinas cuja codificação em binário é w. Chamaremos esta função de f_w.

Notação. Se $w = \langle M \rangle$, então f_w é a função parcial computada por M.

Toda máquina de Turing é representada por uma palavra de $\{0,1\}^*$, portanto o mapeamento $w \mapsto f_w$ enumera todas as funções recursivas. Além disso, toda palavra de $\{0,1\}^*$ representa alguma máquina; assim, f_w sempre está bem definido. As seções 2.4 e 2.5 contém teoremas que nos permitem trabalhar com as funções f_w e a seção 2.6 define o conceito de "Enumeração de Gödel aceitável", que generaliza este mapeamento.

2.4 TEOREMA DA MÁQUINA UNIVERSAL

Durante a demonstração da indecidibilidade do problema da parada, é construída uma máquina de Turing universal, que é capaz de simular qualquer outra máquina de Turing, bastando apenas lhe ser fornecida uma representação da máquina de Turing simulada. (De certa forma, os computadores modernos são máquinas de Turing universais.) Podemos sintetizar esta construção através do mapeamento de funções recursivas definido na seção anterior.

Teorema 2.7. *Existe uma função recursiva parcial g de duas variáveis tal que, para todo x e y,*

$$g(x,y) \simeq f_x(y).$$

Demonstração. A função g é a função computada por uma máquina de Turing universal. Em nosso caso, a máquina universal U, na entrada $\langle x,y \rangle$, usará a tabela de transições disponível em x para executar as etapas de computação de alguma máquina M que satisfaz $\langle M \rangle = x$. x é a codificação binária de M; portanto, ao simular as etapas de computação de M em y, U estará, efetivamente, calculando o valor de f_x (que é a função computada por M).

$f_x(y)$ só não está definida se M nunca parar ao processar y; neste caso, U também não parará ao processar $\langle x,y \rangle$. Caso contrário, após a simulação chegar a um estado final, basta que U escreva a "resposta" de M na saída. Como a resposta de M é $M(y) = f_x(y)$, obtemos a igualdade $g(x,y) = f_x(y)$.■

2.5 TEOREMA S_{MN}

Teorema 2.8. *Existe uma função recursiva total σ de duas variáveis tal que, para todo $x,y,z \in \{0,1\}^*$,*

$$f_{\sigma(x,y)}(z) \simeq f_x(\langle y,z \rangle).$$

Isto é, dado um índice x para a função de duas variáveis f_x e uma palavra y qualquer, a função σ devolve uma máquina de Turing M (quer dizer, $\sigma(x,y) = \langle M \rangle$) que, na entrada z, calcula o valor $f_x(\langle y,z \rangle)$.

Em outras palavras, a função σ deixa o primeiro argumento da função f_x fixo em y. Esta versão do teorema (mencionada por Blum (1967, p. 324)) lida apenas com o caso de fixar uma entrada, y, e deixar uma entrada livre, z. A "versão completa" do teorema está presente no livro de Rogers Jr. (1987, p. 23); ele diz como transformar uma função de $m+n$ variáveis numa função de n variáveis fixando as m primeiras para valores pré-estabelecidos.

Entretanto, para nossos propósitos, esta versão será suficiente.

Demonstração. Dado x e y na entrada, σ terá de retornar uma representação de uma máquina de Turing. A função σ é total, o que significa que $\sigma(x,y)$ sempre existirá; entretanto, pode ser que a máquina $\sigma(x,y)$ não pare em todas as entradas.

Se $\langle M \rangle = \sigma(x,y)$, a ideia é fazer com que M substitua a entrada z por $\langle y,z \rangle$, e execute a máquina representada por x no resultado. Para isso, basta embutir em M o valor y e as transições de x.

Esta construção pode ser executada para todo x e y, tornando σ uma função recursiva total. A máquina M representada por $\sigma(x,y)$ executará, na entrada z, a máquina x em $\langle y,z \rangle$; portanto, $M(z)$ estará definido se, e só se, $f_x(\langle y,z \rangle)$ está definido; e, neste caso,

$$M(z) = f_x(\langle y,z \rangle).$$

Mas como $\langle M \rangle = \sigma(x,y)$, a função calculada por M é, exatamente, $f_{\sigma(x,y)}$, o que prova o teorema. ■

2.6 ENUMERAÇÕES DE GÖDEL ACEITÁVEIS

De posse dos teoremas da máquina universal e S_{mn}, podemos definir as restrições sob as quais definiremos os axiomas de Blum.

Definição 2.9. Seja $\mathcal{P}$ o conjunto de todas as funções recursivas parciais. Uma *enumeração de Gödel aceitável* é uma função $\phi : \{0,1\}^* \to \mathcal{P}$, sobrejetora, que satisfaz às conclusões do teorema da máquina universal e do teorema S_{mn}. (ROGERS JR., 1987, p. 41; BLUM, 1967, p. 324)

Uma enumeração de Gödel aceitável é uma função ϕ que associa cada cadeia de $\{0,1\}^*$ a uma função recursiva parcial. Como ϕ é sobrejetora, o contrário também acontece: toda função recursiva parcial é "enumerada" por ϕ. Denotaremos $\phi(w)$ por ϕ_w.

Por "satisfazer às conclusões do teorema da máquina universal" quer-se dizer que, para esta enumeração ϕ, existe alguma função recursiva parcial g, de duas variáveis, tal que

$$g(x,y) \simeq \phi_x(y).$$

Similarmente, por "satisfazer ao teorema S_{mn}" queremos dizer que existe alguma função recursiva total σ de duas variáveis tal que

$$\phi_{\sigma(x,y)}(z) \simeq \phi_x(\langle y,z\rangle).$$

Exemplo 2.10. O mapeamento $w \mapsto f_w$, definido na seção 2.3, é uma enumeração de Gödel aceitável, pois enumera todas as funções recursivas (por definição) e satisfaz aos teoremas da máquina universal e S_{mn}. □

Podemos pensar nas enumerações de Gödel aceitáveis como mapeamentos semânticos; são regras que atribuem a cada cadeia de $\{0,1\}^*$ uma função de $\mathcal{P}$ — isto é, a função ϕ_x é o que a cadeia x significa. Todos os modelos de computação equivalentes à máquina de Turing induzem enumerações de Gödel aceitáveis. Em particular, o formalismo das máquinas de Turing com oráculos, que definiremos na seção 4.1, pode ser traduzido numa enumeração de Gödel aceitável.

Exemplo 2.11. Toda linguagem de programação Turing-completa pode, de certa forma, ser vista como uma enumeração de Gödel aceitável. Para isso, precisamos restringir os programas a serem não-interativos. Por exemplo, se restringirmos os programas da linguagem `C` a se comunicar com o mundo externo usando apenas `scanf`, `printf` e similares (isto é, sem arquivos, sockets, prompts interativos, etc.), definimos o mapeamento

$$\texttt{file.c} \mapsto C_{\texttt{file.c}},$$

em que $C_{\texttt{file.c}}$ é a função calculada pelo arquivo `file.c` depois de compilado.

(De certa forma, o mapeamento C é um "compilador".) □

Computacionalmente, não podemos trabalhar diretamente com as enumerações, pois são estruturas infinitas; as restrições adicionais (possuir uma função universal e satisfazer ao teorema S_{mn}) nos permite trabalhar com estas enumerações indiretamente.

3 COMPLEXIDADE COMPUTACIONAL

Complexidade é a quantidade de recursos que uma máquina de Turing gasta para computar determinada função ou para decidir pertinência a uma linguagem (HOPCROFT; ULLMAN, 1979, p. 285). Os recursos mais importantes para a teoria de complexidade computacional são o espaço e o tempo, em máquinas de Turing determinísticas e não-determinísticas.

A seção 3.1 contém os axiomas de Blum, que são uma formalização matemática da noção de complexidade. A seção 3.2, então, utiliza estes axiomas para construir as medidas de complexidade padrão.

3.1 TEORIA DE COMPLEXIDADE AXIOMÁTICA: AXIOMAS DE BLUM

Definição 3.1. Seja ϕ uma enumeração de Gödel aceitável. Uma *medida de complexidade* é uma função $\Phi : \{0,1\}^* \times \{0,1\}^* \to \mathbb{N}$, que satisfaz aos seguintes axiomas:[1]

Axioma 1 $\Phi(w,x)$ está definido se, e somente se, $\phi_w(x)$ está definido.

Axioma 2 A função $R(w,x,m)$, definida como 1 se $\Phi(w,x) = m$, e 0 caso contrário, é recursiva total.

O axioma 2 nos dá um semialgoritmo para calcular $\Phi(w,x)$. Entretanto, pelo axioma 1, não podemos ir muito além disso, pois $\phi_w(x)$ não está definido para todo w e x. (De fato, decidir se $\Phi(w,x)$ existe é exatamente o problema da parada para o modelo de computação de ϕ.)

Nos exemplos que se seguem, quando estivermos usando a enumeração de Gödel f_w, utilizaremos a notação $\Phi(\langle M \rangle, x)$ para nos referir a $\Phi(w,x)$, em que w é a codificação (em binário) da máquina M. Não há problemas de ambiguidade, pois a codificação $w = \langle M \rangle$ como definida identifica a função f_w unicamente.

Exemplo 3.2. A *complexidade de tempo*, que denotaremos por $\mathcal{T}$, é a função que diz quantos movimentos uma máquina de Turing faz até retornar uma

[1] A definição original de Blum (1967, p. 324) é dada no contexto do cálculo de funções de inteiros. Em particular, as enumerações de Gödel aceitáveis são indexadas por números, não por cadeias de $\{0,1\}^*$. Esta adaptação é baseada na definição de Papadimitriou (1994, p. 156).

resposta. Isto é,

$$\mathcal{T}(\langle M\rangle,x) = \begin{cases} k, & \text{se } M \text{ executa exatamente } k \text{ passos em } x \text{ antes de parar.} \\ \text{indefinido}, & \text{caso } M \text{ nunca pare de computar } x. \end{cases}$$

Para determinar se $\mathcal{T}(\langle M\rangle,x) = k$, execute a máquina M por k passos e veja se é a primeira vez que M atinge um estado aceitador. E, como $\mathcal{T}(\langle M\rangle,x)$ só está definido se M para ao computar x, $\mathcal{T}$ satisfaz aos dois axiomas de Blum. □

Exemplo 3.3. Para a *complexidade de espaço*, que denotaremos por $\mathcal{S}$, assumiremos que M possui uma fita somente-leitura específica para a entrada.

$$\mathcal{S}(\langle M\rangle,x) = \begin{cases} k, & \text{se } M \text{ para ao computar } x, \text{ tendo lido exatamente } k \text{ células de sua fitas de trabalho.} \\ \text{indefinido}, & \text{se } M \text{ nunca parar ao computar } x. \end{cases}$$

Claramente o axioma 1 é satisfeito. Para o axioma 2, o algoritmo é um pouco mais complicado.

Comece executando M em x. Caso M extrapole k células lidas, podemos rejeitar a entrada (isto é, $\mathcal{S}(\langle M\rangle,x) = k$ é falso). Caso contrário, existirá um número finito de configurações da máquina.

Argumentaremos que existe um número finito de diferentes estados de computação em que a máquina lê, no máximo, k células da fita.

Para $M = (Q,\Gamma,\delta,q_0)$, existem $|\Gamma|^k$ diferentes fitas de tamanho k. (Como o caractere branco pertence a Γ, este também é a quantidade de fitas de tamanho menor ou igual a k.) Existem k diferentes posições para o cabeçote de trabalho, $|x|$ diferentes posições para o cabeçote de leitura, e $|Q|$ diferentes estados.

Como estas informações são a única coisa que importa para determinar o próximo estado de computação da máquina, existem

$$k * |x| * |Q| * |\Gamma|^k \tag{3.1}$$

diferentes estados de computação para M. Portanto, se a máquina executar mais movimentos do que este número, significa que ela entrou em loop. Podemos "rejeitar" a entrada[2]; isto é, $\mathcal{S}(\langle M\rangle,x)$ não está definido.

E, por último, caso M pare, precisamos nos assegurar que, de fato, k células foram lidas. □

[2] Este cuidado adicional é imprescindível para garantir que o predicado "$\Phi(\langle M\rangle,x) = k$" seja decidível, não apenas semidecidível.

Note que precisamos rejeitar a entrada caso a máquina entre em loop porque $\Phi(\langle M\rangle,x)$ só está definido quando $M(x)$ está — não é o caso se M entra em loop ao computar x.

Exemplo 3.4. Podemos adaptar $\mathcal{T}$ e $\mathcal{S}$ para máquinas de Turing não-determinísticas.[3]

A complexidade de tempo não-determinística, que denotaremos por $\mathcal{NT}(\langle M\rangle,x)$, definiremos como sendo a maior quantidade de movimentos tomadas por M ao computar x dentre todas as escolhas de transições possíveis.

Analogamente, definiremos a complexidade de espaço não-determinística, que denotaremos por $\mathcal{NS}(\langle M\rangle,x)$, como sendo a maior quantidade de células lidas em qualquer dos ramos da computação de M em x. Aqui, precisamos tomar o mesmo cuidado que tomamos com $\mathcal{S}$ para demonstrar o axioma 2. □

Nos exemplos 3.2 e 3.3, a enumeração utilizada é a f_w definida na seção 2.3. No exemplo 3.4, ficou faltando definir a enumeração de Gödel associada; isto é, jogamos o problema de explicar como uma máquina não-determinística computa uma função para "debaixo do tapete". Voltaremos a este problema no capítulo 5. Por ora, será suficiente nos restringirmos às funções booleanas, usando a mesma definição usada para decisores[4].

Exemplo 3.5. Escolher $\Phi(w,x) = 0$ para todo M e x satisfaz ao axioma 2, mas não ao axioma 1, pois $\Phi(w,x)$ está definida mesmo quando $f_w(x)$ não está. Já definir $\Phi(w,x) = |f_w(x)|$ satisfaz ao axioma 1, mas não ao axioma 2, pois poderíamos resolver o problema da parada: dada uma máquina M, podemos modificá-la para apagar sua fita logo antes de parar. Então, para esta M', $\Phi(\langle M'\rangle,x) = 0$ se, e somente se, a M original para ao computar x. Estes dois exemplos mostram que os axiomas são independentes (BLUM, 1967, p. 324). □

Podemos ver que as medidas $\mathcal{T}$ e $\mathcal{S}$ estão relacionadas. Para ler uma posição da fita, é necessário gastar ao menos uma unidade de tempo. Ou seja,

$$\mathcal{S}(\langle M\rangle,x) \leq \mathcal{T}(\langle M\rangle,x).$$

E, de acordo com o raciocínio do exemplo 3.3, para todo M existe algum c tal que

$$\mathcal{T}(\langle M\rangle,x) \leq c^{\mathcal{S}(\langle M\rangle,x)}.$$

De fato, podemos relacionar quaisquer duas medidas de complexidade.

[3] Tecnicamente, as denominações usuais, "complexidade de tempo/espaço não-determinística" ou "tempo/espaço não-determinístico" estão erradas; não é o tempo ou espaço ou a complexidade que são não-determinísticos, mas sim o modelo de máquina ao qual nos referimos. Entretanto, toleraremos este abuso de nomenclatura neste texto.

[4] Dado um decisor não-determinístico M, dizemos que M aceita a entrada w se algum ramo de computação aceita, e M rejeita a entrada w se todos os ramos rejeitam. Da mesma forma, para uma função booleana f computada por uma máquina não-determinística M, $f(w) = 1$ se algum ramo da computação de M em w retorna 1, e $f(w) = 0$ se todos os ramos retornam 0.

Teorema 3.6. *Dada uma enumeração de Gödel aceitável ϕ e duas medidas de complexidade Φ e Φ' para ϕ, existe uma função recursiva r tal que*

$$\Phi(w,x) \leq r(x,\Phi'(w,x))$$

para todo w e quase todo x.[5]

Demonstração. Defina

$$r(x,k) = \max\{\Phi(w,x) \mid |w| \leq |x| \wedge \Phi'(w,x) = k\}$$

Fixado x, existe um número finito de máquinas de Turing cuja descrição é menor que $|x|$. O conjunto na definição acima é um subconjunto desta lista (pois, além da exigência $|w| \leq |x|$, exigimos que $\Phi'(w,x) = k$.)

O predicado $\Phi'(w,x) = k$ é recursivo. Quando este predicado é verdadeiro, $\phi_w(x)$ está definido, pelo axioma 1, portanto $\Phi(w,x)$ também está definido e pode ser calculado. Concluímos que r é recursiva.

Agora, para todos os x que são mais longos que w, $\Phi(w,x)$ será um dos elementos do conjunto acima para $r(x,\Phi'(w,x))$, portanto é menor ou igual a $\Phi(w,x)$. ■

Blum (1967, p. 325) demonstra uma versão ligeiramente mais forte deste teorema. Ele prova que r pode ser tal que, simultaneamente,

$$\Phi(w,x) \leq r(x,\Phi'(w,x))$$

e

$$\Phi'(w,x) \leq r(x,\Phi(w,x))$$

Podemos construir uma função dessas pegando o máximo de duas funções obtidas usando o teorema 3.6.

O teorema, assim como provado, não pode ser fortalecido para que r seja uma função de apenas uma variável. Considere A uma máquina de Turing que opere como um autômato finito. $\mathcal{T}(\langle A\rangle,x) = |x|$ para toda palavra x, enquanto que $\mathcal{S}(\langle A\rangle,x) = 1$ para toda palavra x.[6] Se r pudesse depender apenas da segunda variável, isto é, $r(x,k) = r'(m)$ para alguma função r',

[5] Um predicado é "verdadeiro para quase todo n" quando ele é falso para apenas uma quantidade finita de números n. Equivalentemente, é quando existe algum n_0 tal que o predicado é válido para todo $n > n_0$.

[6] A complexidade de espaço não é 0 pois A é obrigada a ler ao menos a célula inicial da sua fita de trabalho, embora a máquina não use aquela célula.

teríamos

$$\begin{aligned} |x| &= \mathcal{T}(\langle A \rangle, x) \\ &\leq r(x, \mathcal{S}(\langle A \rangle, x)) \\ &= r(x, 1) \\ &= r'(1) \end{aligned}$$

que é falso para todo x suficientemente comprido.

Caso r pudesse depender apenas da primeira variável, isto é, $r(x,k) = r''(x)$ para alguma função r'', teríamos, para todas as máquinas de Turing,

$$\mathcal{T}(\langle M \rangle, x) \leq r''(x).$$

Mas, como r'' é recursiva (pois r o é), podemos construir uma máquina que calcula $r''(x)$, desperdiça $r''(x)$ movimentos, e aceita a entrada. Para esta M',

$$\mathcal{T}(\langle M' \rangle, x) > r''(x),$$

contradizendo a equação anterior.

No parágrafo anterior, construímos uma máquina de Turing que deliberadamente desperdiça tempo ao computar determinada função. Blum (1967, p. 325) demonstrou que é sempre possível desperdiçar recursos computacionais, quaisquer que sejam estes recursos. Precisamos de um lema, vindo direto da teoria das funções recursivas.

Lema 3.7 (Teorema da Recursão). *Para qualquer enumeração de Gödel aceitável ϕ e qualquer função recursiva total σ, existe um valor w tal que*

$$\phi_w(x) = \phi_{\sigma(w)}(x)$$

para todo x. (Tal valor é chamado de ponto fixo *para σ.)*

Esta demonstração foi adaptada de (HOPCROFT; ULLMAN, 1979, p. 208).

Demonstração. Primeiro, construiremos uma função recursiva total f tal que $f(x)$ é uma máquina equivalente a $\phi_x(x)$. Isto é, para todo y,

$$\phi_{f(x)}(y) \simeq \phi_{\phi_x(x)}(y), \tag{3.2}$$

se o valor $\phi_x(x)$ estiver definido.

A maior dificuldade será contornar a possibilidade de $\phi_x(x)$ estar indefinido, pois queremos que a função f seja recursiva total.

Usaremos a função universal e a função S_{mn} para isso. Mais especificamente, sejam U e S estas duas funções; isto é,

$$U(x,y) \simeq \phi_x(y)$$
$$\phi_{S(x,y)}(z) \simeq \phi_x(\langle y,z\rangle)$$

Sabemos que estas duas funções existem pois ϕ é uma enumeração de Gödel aceitável. Além disso, S é recursiva total.

Construa uma máquina de Turing M que, na entrada $\langle x,y\rangle$, calcula o valor

$$z = \phi_x(x) = U(x,x), \tag{3.3}$$

e em seguida calcula

$$f_z(y) = U(z,y) \tag{3.4}$$

Como U é uma função recursiva parcial, M calcula uma função recursiva parcial. Mais especificamente, se $f_x(x)$ e $f_{f_x(x)}(y)$ ambos existirem, então

$$M(\langle x,y\rangle) = f_{f_x(x)}(y).$$

Caso contrário (se $f_x(x)$ ou $f_{f_x(x)}(y)$ não existirem), $M(\langle x,y\rangle)$ também não existirá.

Usando esta máquina M, podemos construir nossa função f. Seja m um índice[7] para a função computada por M; a função recursiva total desejada é dada por

$$f(x) = S(m,x).$$

Mostraremos que, de fato,

$$\phi_{f(x)}(y) \simeq \phi_{\phi_x(x)}(y)$$

Primeiro, a condição de existência.

Caso o valor de $\phi_x(x)$ não exista, então a máquina M jamais sairá do primeiro "estágio" de computação (a equação 3.3), portanto, apesar de $f(x)$ existir, $\phi_{f(x)}(y) = \phi_{S(m,x)}(y)$ não existirá para nenhum y. Caso $\phi_x(x)$ exista,

[7] Um *índice* para uma função recursivamente enumerável f é uma cadeia w que representa uma máquina de Turing para f; isto é, $\phi_w = f$ (EPSTEIN; CARNIELLI, 2008, p. 130). Todas as funções recursivas parciais possuem índices, mesmo que a enumeração de Gödel aceitável ϕ em questão não esteja relacionada com máquinas de Turing, pois exigimos que ϕ (cujo contradomínio é o conjunto de todas as funções recursivas parciais) seja uma função sobrejetora.

(Como estamos lidando com cadeias de caracteres binários, talvez seria melhor chamar de "programa" em vez de "índice"; isto é, w seria um *programa* para f_w.)

mas $\phi_{\phi_x(x)}(y)$ não, então M passará pelo primeiro "estágio", calculando o z da equação 3.3, mas M nunca terminará o segundo "estágio", que é calcular $f_z(y)$ (equação 3.4). Portanto, novamente, $\phi_{f(x)}(y)$ não existe. No caso oposto, em que tanto $\phi_x(x)$ quanto $\phi_{\phi_x(x)}(y)$ existem, a máquina M eventualmente encerrará a computação, passando pelos dois "estágios" com sucesso. Concluímos que $\phi_{f(x)}(y)$ existe se, e somente se, $\phi_{\phi_x(x)}(y)$ existe.

Agora, à condição de igualdade. Se tanto $\phi_x(x)$ quanto $\phi_{\phi_x(x)}(y)$ existem, então

$$\begin{aligned}\phi_{f(x)}(y) &= \phi_{S(m,x)}(y)\\ &= \phi_m(\langle x,y\rangle)\\ &= M(\langle x,y\rangle)\\ &= U(U(x,x),y\rangle)\\ &= U(\phi_x(x),y)\\ &= \phi_{\phi_x(x)}(y).\end{aligned}$$

Isso prova a equação 3.2. Como f foi definida com base na função recursiva total S, sabemos que f é recursiva total, independentemente de $\phi_x(x)$ existir ou não. [8]

De posse da função f, podemos, finalmente, demonstrar o teorema. Escolha k como sendo um índice para $\sigma \circ f$. Isto é,

$$\phi_k(x) = \sigma(f(x)) \tag{3.5}$$

para todo x. Afirmamos que $w = f(k)$ satisfaz às exigências do teorema.

De fato, para todo x,

$$\begin{aligned}\phi_w(x) &\simeq \phi_{f(k)}(x)\\ &\simeq \phi_{\phi_k(k)}(x) && \text{Pela equação 3.2}\\ &\simeq \phi_{\sigma(f(k))}(x) && \text{Pela equação 3.5}\\ &\simeq \phi_{\sigma(w)}(x) && \text{Pela definição de } w.\end{aligned}$$ ■

Proposição 3.8. *Seja ϕ uma numeração de Gödel aceitável, Φ uma medida de complexidade para ϕ, e $f : \{0,1\}^* \to \{0,1\}^*$ e $g : \mathbb{N} \to \mathbb{N}$ funções recursivas totais. Então existe um índice w para a função f tal que*

$$\Phi(w,x) > g(|x|)$$

[8] De certa forma, a tarefa de computar $\phi_x(x)$ é delegada à "máquina" $f(x)$; portanto, f sempre para ao processar x, mesmo que $\phi_x(x)$ não exista — neste último caso, a "máquina" $f(x)$ pode até não parar, mas a função f em si finaliza a computação.

para todo x (BLUM, 1967, p. 324).

Ou seja, o programa w, ao computar f, desperdiça mais de $g(n)$ unidades do recurso medido por Φ ao processar uma palavra de tamanho n. Em outras palavras, código ruim pode ser feito em qualquer linguagem.

Demonstração. Defina a função h, de duas variáveis, por

$$h(w,x) = \begin{cases} \phi_w(x) \cdot 1 & \text{se } \Phi(w,x) \leq g(|x|) \\ f(x) & \text{caso contrário.} \end{cases}$$

($\phi_w(x) \cdot 1$ é o valor de $\phi_w(x)$ concatenado com o símbolo 1; o importante aqui é que $\phi_w(x) \cdot 1$ seja diferente de $\phi_w(x)$, pois queremos fazer algo parecido com diagonalização.)

Observe que h é uma função computável, pois Φ, g e f o são, e, caso $\Phi(w,x)$ esteja definido, $\phi_w(x)$ também estará.

Seja S a função S_{mn} da enumeração ϕ, e v um índice para h. Construa a função σ que, na entrada w, calcula $S(v,w)$ — isto é, $\sigma(w) = S(v,w)$. Portanto,

$$\phi_{\sigma(w)}(x) = \phi_{S(v,w)}(x) = \phi_v(\langle w,x \rangle) = h(w,x).$$

Observe que σ é recursiva total, pois S o é.

Pelo teorema da recursão (lema 3.7), σ possui um ponto fixo v. Demonstraremos que v satisfaz às exigências do teorema.

Caso a função h, ao computar o valor de $h(v,x)$ para algum x, tenha escolhido a primeira opção (isto é, $\Phi(w,x) \leq g(|x|)$), a saída final de $\sigma(v)$ teria sido $\phi_v(x)$ concatenado com 1, que é diferente de apenas $\phi_v(x)$. Portanto, v não seria um ponto fixo de σ, contradizendo o teorema da recursão.

Portanto, h nunca seleciona a primeira opção ao computar $h(v,x)$, para qualquer x. Isto significa que $\Phi(v,x) > g(|x|)$ para todo x, o que garante a exigência de complexidade, e que

$$\phi_{\sigma(v)}(x) = f(x).$$

Mas, como v é um ponto fixo de σ, a própria v já computava f antes de passar por σ, o que prova a exigência da função. ■

3.1.1 Classes de Complexidade

Definição 3.9. (KOZEN, 2006, p. 242) Dada uma enumeração de Gödel aceitável ϕ, uma medida de complexidade Φ para ϕ, e uma função recursiva total

$f : \mathbb{N} \to \mathbb{N}$, a *classe de complexidade* f com relação a Φ é o conjunto

$$\mathcal{C}_\Phi(f) = \{\phi_w \mid \Phi(w,x) \leq f(|x|) \text{ para quase todos os } x\}.$$

Permitiremos que a complexidade de ϕ_w possa ser maior que f para um número finito de elementos para generalizar algumas demonstrações.[9] Como esta permissão existe apenas para um número finito de elementos, para todo w, o valor de $\Phi(w,x)$ eventualmente sempre estará definido; quer dizer, existe algum k (dependente de w) tal que $\Phi(w,x)$ está definido para todo x com $|x| > k$. Portanto, todas as funções ϕ_w que pertençam a alguma classe de complexidade estão definidas para todo x suficientemente longo (o quão longo depende de w). Isto é, o domínio destas funções é *cofinito*.

Embora faça sentido definir $\mathcal{C}_\Phi(f)$ para funções f arbitrárias, a exigência de f ser recursiva total torna as classes de complexidade suscetíveis a argumentos por diagonalização.

Por exemplo, podemos mostrar que nenhuma classe de complexidade contém todas as linguagens recursivas. Precisamos de um lema.[10]

Lema 3.10. *Existe uma função recursiva total $h : \{0,1\}^* \to \{0,1\}^*$ tal que, para todos x, existem infinitos y tais que $h(y) = x$. Isto é, as imagens inversas por h são sempre infinitas.*

Demonstração. Defina h por

$$h(\langle x_1, x_2, \ldots, x_n \rangle) = x_1;$$

se x não representa uma tupla, conforme definido na seção 2.2.1, defina $h(x) = \varepsilon$ (ou qualquer outra cadeia fixa).

A função h assim definida é recursiva total; além disso, para todo x e todo z, $h(\langle x, z \rangle) = y$, portanto todos os x são $g(y)$ para infinitos y. ■

[9] Por razões didáticas, Hopcroft e Ullman (1979, p. 285) definem classes de complexidade individualmente para cada medida de complexidade. Para tempo e espaço, que são as medidas levadas em consideração por estes autores, não precisamos nos preocupar com este número finito de elementos que extrapola o limite de complexidade, pois sempre podemos embutir suas respostas no controle finito da máquina. Desta forma, a complexidade resultante desta nova máquina satisfaz à restrição.

Entretanto, como estamos trabalhando com medidas de complexidade arbitrárias, não há garantia de que possamos fazer esta modificação — a medida de complexidade poderia, por exemplo, depender também da quantidade de estados da máquina. Portanto, para que estes resultados possam ser generalizados, embutimos esta liberdade adicional na própria definição de classe de complexidade (MCCREIGHT; MEYER, 1969, p. 80).

[10] A maior parte dos teoremas não precisou ser alterada significativamente para ser adaptada a esta noção de classe de complexidade. Este teorema é a exceção, pois a diagonalização agora precisa provocar uma quantidade infinita de falhas.

Teorema 3.11. *Seja $\mathcal{C}_\Phi(f)$ uma classe de complexidade com relação à Φ. Então existe uma função recursiva total g que não pertence à esta classe de complexidade.*[11] *Além disso, g pode ser construída de forma que sua imagem seja o conjunto $\{0,1\}$.*

Demonstração. Construiremos uma função g que discorda de todas as funções que gastam menos de $f(n)$ recursos ao computar uma palavra de tamanho n.

Usaremos a função h do lema anterior. Defina a função g por

$$g(x) = \begin{cases} 1, & \text{se } \phi_{h(x)}(x) \text{ existe, } \Phi(h(x),x) \leq f(|x|) \text{ e } \phi_{h(x)}(x) = 0 \\ 0, & \text{caso contrário.} \end{cases}$$

Observe que g é uma função total. Para computar g, primeiro use o fato de que o predicado $\Phi(w,x) = n$ é decidível para descobrir se $\Phi(h(x),x) \leq f(|x|)$. Caso isso seja falso, então independente de $\phi_{h(x)}(x)$ existir ou não, $g(x)$ será igual a 0. Agora, se $\Phi(h(x),x) \leq f(|x|)$, sabemos automaticamente que $\phi_{h(x)}(x)$ existe, e podemos computar este valor usando a função universal da enumeração ϕ; então, usamos o resultado desta computação para calcular $g(x)$. Concluímos que g é recursiva.

Agora, tome ϕ_w uma função de $\mathcal{C}_\Phi(f)$. Mostraremos que $g \neq \phi_w$.

Como $\phi_w \in \mathcal{C}_\Phi(f)$, existe algum n_0 tal que todos os x com $|x| > n_0$ satisfaz $\Phi(w,x) \leq f(|x|)$. Escolha algum y com $|y| > n_0$ tal que $h(y) = w$. Sabemos que tal y existe pois a imagem inversa de w por h é infinita.

Argumentaremos que $\phi_w(y) \neq g(y)$. Sabemos que $\Phi(w,y) \leq f(|y|)$, pois $|y| > n_0$; portanto, $\phi_w(y)$ está definido. Mas $w = g(y)$; portanto, das três condições do primeiro caso da definição de $g(y)$, duas já foram atendidas. Se a terceira também for atendida, isto é, $\phi_w(y) = 0$, o primeiro caso é selecionado e $g(y) = 1 \neq \phi_w(y)$. Entretanto, se a terceira não for atendida, temos $\phi_w(y) \neq 0 = g(y)$, pois o segundo caso será escolhido. Ou seja, $\phi_w \neq g$.

Como g é diferente de todas as funções de $\mathcal{C}_\Phi(f)$,[12]

$$g \notin \mathcal{C}_\Phi(f). \qquad \blacksquare$$

[11] Observe que este teorema possui uma afirmação mais forte do que o teorema 3.8; lá, nós encontramos um índice (para uma função) que desperdiça recursos ao computá-la. Aqui, construiremos uma função que, não importa qual técnica seja usada, a quantidade de recursos gastos eventualmente superará f.

[12] Este argumento é, de certa forma, uma versão limitada (por f) do problema da parada.

3.1.2 Teorema da União

Na seção 3.2.3, definiremos algumas classes de complexidade computacional como sendo a união de infinitas classes de complexidade. O teorema da união diz que, sob condições apropriadas, esta união é uma classe de complexidade de acordo com nossa definição de classe de complexidade.

Teorema 3.12 (Teorema da União). *Seja* $\{h_1, h_2, h_3, \dots\}$ *uma lista infinita de funções computáveis tais que, para todo* i *e* n*,*

$$h_i(n) \leq h_{i+1}(n).$$

Então existe uma função computável g *tal que*

$$\bigcup_i \mathcal{C}_\Phi(h_i) = \mathcal{C}_\Phi(g).$$

A demonstração foi retirada do texto de Hopcroft e Ullman (1979, p. 310).

Demonstração. Construiremos uma máquina de Turing que enumerará os valores de $g(n)$.

Observe que

$$\mathcal{C}_\Phi(h_1) \subseteq \mathcal{C}_\Phi(h_2) \subseteq \mathcal{C}_\Phi(h_3) \subseteq \mathcal{C}_\Phi(h_4) \subseteq \dots;$$

se alguma função ϕ_w pertence a união destas classes de complexidade, então $\phi_w \in \mathcal{C}_\Phi(h_k)$ para algum k, mas também que $\phi_w \in \mathcal{C}_\Phi(h_i)$ para todas as classes de complexidade com $i \geq k$.

Enumere as palavras de $\{0,1\}^*$ por $w_1, w_2, \dots, w_n, \dots$, ordenando lexicograficamente. Para definir o valor de $g(n)$, analisaremos as funções $\phi_{w_1}, \dots, \phi_{w_n}$ e manteremos uma lista de "chutes" da forma $\phi_{w_i} \in \mathcal{C}_\Phi(h_j)$, que denotaremos por $C[i] = j$.[13] Isto é, tentaremos "adivinhar" em qual dos conjuntos a função ϕ_{w_i} entra.[14]

[13] A ideia é que C represente um vetor infinito, indexado por $i \in \mathbb{N}$. Na i-ésima iteração do algoritmo, na qual imprimiremos o valor de $g(i)$, apenas os valores de $C[1], C[2], \dots, C[i]$ estarão definidos, portanto apenas trabalharemos com uma porção finita de $\mathcal{C}$ (que crescerá conforme o algoritmo enumera novos valores de g).

[14] Embora estejamos usando os termos "adivinhar" e "chute", não há não-determinismo envolvido. O problema é que não sabemos de antemão qual é o valor correto $C[i]$, então precisamos estabelecer algum valor arbitrário e reestabelecer este valor conforme descobrimos que ele não funciona.

Observe também que não precisamos "acertar na mosca"; basta algum conjunto $\mathcal{C}_\Phi(h_k)$ ao qual ϕ_{w_i} pertença.

Ao computar o valor de $g(n)$, um chute errado é um chute $C[i] = j$, em que $\Phi(w_i, x) > h_j(|x|)$ para algum x com $|x| = n$. Isto é, a complexidade do programa w_i é maior que $h_j(n)$ para alguma palavra de tamanho n. Conforme computamos $g(n)$, sempre que descobrirmos que nosso chute, digamos, para w_i, está errado, definiremos g para um valor menor que $\Phi(w_i, x)$ (para $|x| = n$), e faremos um novo chute, desta vez "aumentando as apostas". Dessa forma, o valor de g propositalmente ficará abaixo da complexidade de w_i para palavras de tamanho n. Se a complexidade de w_i for maior do que todas as funções h_j, todo chute feito sempre será "desmascarado" como errado, portanto o valor de g ficará infinitas vezes *abaixo* da complexidade de g. Entretanto, sempre que descobrirmos que um chute está errado, aumentaremos o valor do chute; assim, se $\phi_{w_i} \in \mathcal{C}_\Phi(h_k)$ para algum k (isto é, a complexidade de w_i é menor que h_k), assim que o chute $C[i]$ for ajustado para algum valor acima de k, nunca "desmascararemos" o chute como errado — portanto, o valor de $g(n)$ sempre ficará acima da complexidade de w_i.

O restante da demonstração formaliza o algoritmo do parágrafo acima.

Defina $g(1)$ para $h_1(1)$ (ou qualquer valor arbitrário) e inicialize a lista com o chute $C[1] = 1$. A lista terá sempre tamanho $n-1$ ao começar a decidir o valor de $g(n)$.

Ao enumerar o valor de $g(n)$, adicione o chute $C[n] = n$ à lista. Percorra a lista atrás de chutes errados; isto é, pares $C[i] = j$ tais que $\Phi(w_i, x) > h_j(x)$ para algum x com $|x| = n$. Dentre os chutes errados, escolha i de forma a minimizar $C[i]$. Defina, então, $g(n) = h_{k_i}(n)$ e atualize o chute $C[i]$ para n. Caso não haja chutes errados, apenas defina $g(n) = h_n(n)$.

Caso ϕ_{w_i} esteja em algum $\mathcal{C}_\Phi(h_k)$, assim que redefinirmos o chute para algum valor maior que k, erraremos, no máximo, mais uma quantidade finita de vezes. Caso ϕ_{w_i} não esteja na união das classes de complexidade, erraremos o chute infinitas vezes, e $g(|x|)$ será maior que $\Phi(w_i, x)$ para todos esses erros.

Observe que os novos chutes que adicionamos à lista sempre serão maiores que todos os demais chutes existentes; e sempre redefinimos um chute errado para o maior valor de um chute até agora. Portanto, depois que adicionamos o chute $C[i] = i$ à lista (evento que ocorreu ao computar $g(i)$), ocorrerão, no máximo, $i-1$ chutes diferentes que resultarão na definição de $g(n)$ como $h_k(n)$ para algum $k < i$.

Então, se $\phi_{w_i} \in \mathcal{C}_\Phi(h_k)$, teremos $\Phi(w_i, x) > g(|x|)$ para, no máximo, $k-1$ valores de $|x|$ maiores que k. Concluímos que

$$\phi_{w_i} \in \mathcal{C}_\Phi(g),$$

pois $\Phi(M, x) > g(|x|)$ apenas um número finito de vezes.

```
1  C[1] ← 1
2  Imprima g(1) = h_1(1)
3  for n = 2 to ∞ do
4      C[n] ← n
       /* Valor sentinela:  */
5      chute_errado ← ∞
6      for i ← 1 to n do
           /* acharemos o menor chute errado */
7          if Φ(w_i, x) > h_{C[i]}(|x|) para algum x de tamanho n then
               /* Achamos um chute errado.  */
8              if chute_errado = ∞ or C[i] < C[chute_errado]
               then
9                  chute_errado ← i
10             end
11         end
12     end
13     if chute_errado = ∞ then
           /* Todos os chutes estavam certos.  */
14         Imprima g(n) = h_n(n)
15     end
16     else
17         k ← C[chute_errado]
18         Imprima g(n) = h_k(n)
19         C[chute_errado] ← n
20     end
21 end
```

Algoritmo 3.1: Algoritmo que enumera os valores da função g, cuja existência é afirmada pelo teorema da união.

Isso prova

$$\bigcup_i \mathcal{C}_\Phi(h_i) \subseteq \mathcal{C}_\Phi(g).$$

Do outro lado, pegue ϕ_{w_i} fora da união das classes de complexidade. Qualquer índice w_j para ϕ_{w_i} terá, para todo k, $\Phi(w_j,x) > h_k(|x|)$ para infinitos x. Em todos esses valores de $|x|$, iremos "errar o chute" e definir $g(n)$ para um valor menor que $\Phi(w_j,x)$, para algum x com $|x| = n$.

Isso prova

$$\overline{\bigcup_i \mathcal{C}_\Phi(h_i)} \subseteq \overline{\mathcal{C}_\Phi(g)}.$$

Combinando as duas inclusões, prova-se o teorema. ■

Observe que a hipótese $h_i(n) \leq h_{i+1}(n)$ para todo i e todo n pode ser enfraquecida. Podemos exigir, apenas, que a desigualdade seja verdadeira para todo n maior que algum n_0, independente de i — isto é, eventualmente todos os h_i deixam de se cruzar — ou que, para todo i, exista algum n_i em que $h_i(n) \leq h_j(n)$ para todo $n > n_i$ e todo $j > i$ — isto é, eventualmente, cada h_i deixa de cruzar com todos os outros.

Neste último caso (que abrange o anterior), apenas teríamos que alterar a quantidade máxima de vezes em que podem haver chutes errados. Se $\phi_{w_t} \in \mathcal{C}_\Phi(h_i)$, então existem, no máximo, $i-1$ definições de $g(n)$ que poderiam ser baseados em h_k com $k < j$. No teorema original, esses são os únicos valores que temos de nos preocupar (são esses os $i-1$ "chutes errados" que podem haver após definir $g(i)$). Agora, além desses valores, precisamos levar em conta que todas as definições até n_i podem ser inferiores a h_i (e, possivelmente, abaixo da complexidade de ϕ_{w_t}).

Em outras palavras, é essencial que as funções h_i eventualmente parem de se cruzar.

Contraexemplo 3.13. Usaremos a complexidade de tempo para mostrar que existem uniões de classes de complexidade que não são classes de complexidade. Defina

$$f_1 = \begin{cases} n, & \text{se } n \text{ for par.} \\ n^3 + 2n, & \text{se } n \text{ for ímpar.} \end{cases}$$

$$f_2 = \begin{cases} n, & \text{se } n \text{ for ímpar.} \\ n^3 + 2n, & \text{se } n \text{ for par.} \end{cases}$$

É possível demonstrar que existe uma linguagem L em

$$\mathcal{C}_\mathcal{T}(n^3) \backslash \mathcal{C}_\mathcal{T}(n^2);$$

isto é, L exige mais de n^2 de espaço para ser resolvida (HOPCROFT; ULLMAN, 1979, p. 299). (O símbolo $\backslash$ denota subtração de conjuntos.)

Seja Σ o alfabeto de L, tome \$ um símbolo fora de Σ e construa as linguagens

$$L_1 = L \cup L\$ \cup (\Sigma\Sigma)^*$$
$$L_2 = L \cup L\$ \cup \Sigma(\Sigma\Sigma)^*$$

L_1 contém todas as palavras de tamanho par de Σ^*, todas as palavras de tamanho ímpar de L, e todas as palavras de tamanho par de L concatenadas

com \$. L_1 foi construída de forma que pertencesse a $\mathcal{C}_\mathcal{T}(f_1)$; um algoritmo que resolve L_1 é, primeiro, determinar se a entrada possui tamanho par e aceitando neste caso (gastando n), ou então apagar um possível \$, voltar para o começo (gastando mais n) e executar o algoritmo de L (gastando mais n^3). Note que todo algoritmo que opera em tempo $T(n)$ para L_1 pode ser convertido num algoritmo que opera em tempo $T(n)+2n$ para L; basta verificar a paridade e emendar um \$ no final se necessário. Isso indica que f_1 é a complexidade "ótima" de L_1; isto é, todas as máquinas que resolvem L_1 precisam ter complexidade de tempo eventualmente acima de n^2, para todo n par.

L_2 é análogo, mas para f_2. Observe que L não pertence nem a $\mathcal{C}_\mathcal{T}(f_1)$ nem a $\mathcal{C}_\mathcal{T}(f_2)$.

Suponha que exista uma função g tal que

$$\mathcal{C}_\mathcal{T}(f_1) \cup \mathcal{C}_\mathcal{T}(f_2) = \mathcal{C}_\mathcal{T}(g).$$

Considere L' como uma linguagem que exija tempo n^2 para ser resolvida em quase todas as instâncias. Note que L' não pertence à união das duas classes de complexidade, por exigir tempo n^2 mesmo nos casos em que $f_1(n) = n$ ou $f_2(n) = n$. Mostraremos que $L' \in \mathcal{C}_\mathcal{T}(g)$.

Seja M_1 um algoritmo que aceita L_1 tal que $\mathcal{T}(M_1, x) \leq g(|x|)$ para quase todo x. Essencialmente, após algum n_1, para toda palavra maior que este número precisamos ter $\mathcal{T}(M_1, x) \leq g(|x|)$. Pelo raciocínio acima, $\mathcal{T}(M_1, x)$, para $|x|$ par, não pode ser menor que $|x|^2$ sempre; portanto, $g(n) > n^2$ quase sempre que n for par.

A mesma técnica, se aplicada a L_2, mostra que $g(n) > n^2$ quase sempre que n for ímpar. Combinando ambos, temos $g(n) > n^2$ quase sempre. Mas isso colocaria L' em $\mathcal{C}_\mathcal{T}(g)$ — contradição.

Portanto,

$$\mathcal{C}_\mathcal{T}(f_1) \cup \mathcal{C}_\mathcal{T}(f_2)$$

não é uma classe de complexidade, é apenas um conjunto. □

Observe que a ausência de monotonicidade não foi o "grande vilão" deste contraexemplo. A prova do teorema da união, em nenhum momento, assumiu monotonicidade das funções envolvidas.

3.2 MEDIDAS DE COMPLEXIDADE COMPUTACIONAL

Nesta seção, desceremos da "estratosfera", em que lidávamos com os axiomas de Blum, e trabalharemos com medidas de complexidade concretas.

Definição provisória 3.14. Seja $\mathcal{B}$ o conjunto das funções booleanas (isto é,

as funções cuja imagem é $\{0,1\}$). Então,

$$\begin{aligned}\mathsf{DTIME}(f) &= \mathcal{C}_{\mathcal{T}}(f) \cap \mathcal{B} \\ \mathsf{DSPACE}(f) &= \mathcal{C}_{\mathcal{S}}(f) \cap \mathcal{B} \\ \mathsf{NTIME}(f) &= \mathcal{C}_{\mathcal{NT}}(f) \cap \mathcal{B} \\ \mathsf{NSPACE}(f) &= \mathcal{C}_{\mathcal{NS}}(f) \cap \mathcal{B}\end{aligned}$$

Isto é, $\mathsf{DTIME}(f)$ são todas as funções booleanas de $\mathcal{C}_{\mathcal{T}}(f)$, por exemplo. Observe que existe uma correspondência direta entre uma função booleana $g : \{0,1\}^* \to \{0,1\}$ e a linguagem correspondente $L = \{x \mid g(x) = 1\}$; portanto, usaremos os termos intercambiavelmente.

Estas são as principais medidas de complexidade computacional para máquinas de Turing.

Esta noção de complexidade de espaço é um pouco diferente da definição dada por Hopcroft e Ullman (1979, p. 285); a definição deles exige apenas que, ao computar x, a máquina nunca leia mais do que $f(n)$ células da fita. Note que não há exigência de parada; desta forma, a complexidade de espaço acaba não sendo uma medida de complexidade, pois $\mathcal{S}(M,x)$ pode estar definido mesmo quando $M(x)$ não está. De fato, são justamente estes casos que mereceram atenção especial ao definir $\mathcal{S}$ da forma como definimos e provar que esta função satisfaz ao axioma 1. Mesmo Hopcroft e Ullman (1979, p. 313) notam que é necessário fazer este ajuste.

Outra diferença é o fato de nós permitirmos às máquinas que computam linguagens em (por exemplo) $\mathsf{DSPACE}(f)$ extrapolar o limite de $f(n)$ células para um número finito de entradas. Isto não costuma causar problemas; podemos embutir no controle finito da máquina a "resposta certa" para todas as palavras que violam o limite.

Entretanto, mesmo assim, a máquina lerá ao menos uma célula da fita de trabalho, pois ela precisa realizar ao menos um movimento para escrever a resposta; encontramos problemas caso $f(n)$ seja zero em algum ponto. Por exemplo, defina f por

$$f(n) = \begin{cases} 0, & n < 2 \\ 1, & n \geq 2 \end{cases}$$

Na definição de Hopcroft e Ullman (1979, p. 288), $\mathsf{DTIME}(f)$ é o conjunto vazio, pois toda máquina de Turing é obrigada a ler ao menos a célula inicial;[15] enquanto que, em nossa definição, $\mathsf{DSPACE}(f)$ corresponde ao con-

[15] A única possível exceção é a máquina de Turing cujo estado inicial é igual ao estado final. Embora seja estranho falar de "complexidade de espaço" de uma máquina que aceita a entrada sem sequer olhar um único símbolo, é admissível esta interpretação em que $\mathsf{DSPACE}(f) = \{\Sigma^*\}$.

junto das linguagens regulares.[16]

Este exemplo é admitidamente forçado. Qualquer máquina que queira aceitar alguma linguagem em $\mathsf{DSPACE}(f)$ é obrigada a violar a restrição de ocupar menos espaço do que $f(n)$ para $n < 2$. E, conforme discutido anteriormente, é exatamente nestes exemplos forçados em que as definições divergem. Portanto, adotaremos esta hipótese de Hopcroft e Ullman (1979, p. 287).

Definição 3.15.

$$\mathsf{DSPACE}(f(n)) = \mathcal{C}_{\mathcal{S}}(\max(f(n),1)) \cap \mathcal{B}$$
$$\mathsf{NSPACE}(f(n)) = \mathcal{C}_{\mathcal{NS}}(\max(f(n),1)) \cap \mathcal{B}$$

Para complexidade de tempo, existem hipóteses similares. Hopcroft e Ullman (1979, p. 287) assumem que $f(n) \geq n+1$ para complexidade de tempo. A justificativa é que qualquer máquina de Turing precisa de, ao menos, $n+1$ movimentos para ler o primeiro espaço em branco após uma palavra de tamanho n. Isto é, este é o tempo mínimo necessário para ler toda a entrada. Papadimitriou (1994, p. 33) adota uma suposição similar: a de que $f(n) \geq n$.

Adotaremos a hipótese de Hopcroft e Ullman (1979, p. 287).

Definição 3.16.

$$\mathsf{DTIME}(f(n)) = \mathcal{C}_{\mathcal{T}}(\max(f(n),n+1)) \cap \mathcal{B}$$
$$\mathsf{NTIME}(f(n)) = \mathcal{C}_{\mathcal{NT}}(\max(f(n),n+1)) \cap \mathcal{B}$$

Esta suposição é, entretanto, passível de objeções. Existem máquinas de Turing que aceitam uma entrada sem ter de lê-la por completo. Podemos cumprir exigências como $\mathcal{T}(M,x) \leq 2 + |x|/2$; é uma situação um pouco diferente daquela que tínhamos com complexidade de espaço, em que éramos *obrigados* a violar as restrições de espaço em alguns casos.

Uma ressalva: é possível provar que para qualquer linguagem em $\mathsf{DTIME}(2+n/2)$ (pela definição anterior), existe algum número racional k tal que a pertinência a $L(M)$ é determinada analisando-se apenas as primeiras k letras da entrada. Portanto, as linguagens excluídas por esta hipótese nem eram muito interessantes.

Existe mais uma inconsistência em relação às classes de complexidade: as constantes em frente às funções. Arora e Barak (2009, p. 25) defi-

[16] Para provar isso, observe que, exceto um número finito de entradas, a máquina terá apenas uma célula de memória para usar. Então, ela pode armazenar esta informação no controle finito, funcionando como um autômato finito bidirecional (*two-way deterministic finite automaton*), que é equivalente a um autômato finito determinístico (HOPCROFT; ULLMAN, 1979, p. 40).

nem $\mathsf{DTIME}(f)$ como sendo o conjunto das linguagens para as quais existem máquinas cujo tempo de execução é menor que $cf(n)$, para alguma constante $c > 0$. As definições de NTIME, DSPACE e NSPACE de Arora e Barak (2009, p. 41, p. 78, p. 79) também permitem este fator constante.

Esta definição está de acordo com a prática comum na análise de complexidade de algoritmos de desprezar as constantes. Ao menos no caso da complexidade de espaço, conseguimos provar equivalência entre as definições.

Teorema 3.17. *Para toda constante $c > 0$,*

$$\mathsf{DSPACE}(f) = \mathsf{DSPACE}(cf)$$
$$\mathsf{NSPACE}(f) = \mathsf{NSPACE}(cf)$$

Demonstração. Assuma sem perda de generalidade que $c < 1$. Seja M uma máquina que $L(M) \in \mathsf{DSPACE}(f)$. O truque é representar várias células de M num único símbolo de fita. Mais precisamente, cada símbolo de M' conterá $\lceil 1/c \rceil$ células de M. Como na complexidade de M não são contabilizados o tamanho da fita de entrada, a complexidade de M' é menor do que cf. ■

Corolário 3.18. *Nossas definições de complexidade de espaço são equivalentes à complexidade de espaço de Arora e Barak (2009, p. 78).* □

Para complexidade de tempo, a história não é tão bonita assim. Precisamos separar em dois casos.

Teorema 3.19 (Aceleramento linear[17]). *Se*

$$\lim_{n\to\infty} \frac{n}{f(n)} = 0,$$

então, para toda constante racional $c > 0$,

$$\mathsf{DTIME}(f) = \mathsf{DTIME}(cf)$$
$$\mathsf{NTIME}(f) = \mathsf{NTIME}(cf)$$

Esta demonstração foi retirada de (HOPCROFT; ULLMAN, 1979, p. 290).

Demonstração. Assuma sem perda de generalidade que $c < 1$. Dada M que aceita $L \in \mathsf{DTIME}(f)$, construiremos uma M', necessariamente multifitas, que faz vários movimentos de M de uma só vez.

[17] Do inglês "linear speedup".

Fixe um valor de r agora. A ideia é codificar trechos da fita de M com r células na fita de M', incluindo a posição da cabeça de leitura (se estiver lá), de maneira similar ao que fizemos com a complexidade de espaço.

Para cada movimento, M' irá "carregar na memória cache" as células que estão sob o cabeçote de leitura e as células imediatamente à esquerda e à direita. Isto é, M' armazenará esta informação no controle finito. Esta etapa custa quatro movimentos.

Com $3r$ posições de memória de cada fita e o cabeçote de leitura nas r posições centrais, M' pode calcular todos os movimentos que M faria nesta situação. Observe que, como estes movimentos dependem apenas das células da fita de M que agora estão no controle finito de M', tal cálculo pode ser embutido nas regras de transição de estados de M'. Portanto, esta etapa é gratuita. Como a cabeça de leitura de M estava nas r posições centrais, acabamos de executar, no mínimo, r movimentos de M', sem custo de tempo.

Agora, com mais quatro movimentos, nós "submetemos" as alterações da "memória cache" na fita de M'. Ao final, com 8 movimentos de M', executamos ao menos r movimentos de M. Portanto, após compactarmos a entrada neste formato, alcançaremos um estado de aceitação ou rejeição em, no máximo,

$$\left\lceil \frac{8f(n)}{r} \right\rceil$$

etapas.

O problema é, justamente, fazer esta compactação inicial. Podemos ler a entrada sequencialmente e ir apagando-a, enquanto que a compactamos em outra fita (custo: n). Ao final, reposicione o cabeçote no começo (custo: n/r) e consideramos a fita de entrada como uma fita de trabalho e a fita com a entrada codificada como a fita de entrada. Custo:

$$n + \left\lceil \frac{n}{r} \right\rceil .$$

Observe que assumimos que existem ao menos duas fitas disponíveis.

Custo total:

$$n + \left\lceil \frac{n}{r} \right\rceil + \left\lceil \frac{8f(n)}{r} \right\rceil$$

Como

$$\lim_{n\to\infty} \frac{n}{f(n)} = 0,$$

para $r > 8c$, podemos fazer o custo final ser menor que $cf(n)$ para todo n suficientemente grande. Isso prova o teorema. ■

Teorema 3.20.

$$\mathsf{DTIME}(cn) = \mathsf{DTIME}((1+\varepsilon)n)$$
$$\mathsf{NTIME}(cn) = \mathsf{NTIME}((1+\varepsilon)n)$$

para qualquer $c > 1$ *e* $\varepsilon > 0$.

Demonstração. Escolha $r = \varepsilon/16$ na demonstração do teorema anterior. ■

Corolário 3.21. *Se existe algum* $c > 1$ *tal que*

$$f(n) \geq cn$$

para quase todo n, nossas definições de complexidade de tempo em relação à função f *são equivalentes às de Arora e Barak (2009, p. 25).* □

Papadimitriou (1994, p. 32) dá uma caracterização elegante do aceleramento linear que cobre os dois casos:

Teorema 3.22. *Se L é aceita em tempo* $f(n)$ *por alguma máquina de Turing, então L é aceita em tempo* $cf(n)+n+2$ *por alguma máquina de Turing, para qualquer c.* □

Já provamos que nossa definição é equivalente à de Arora e Barak (2009, p. 25) (exceto no caso extremo em que $f(n)$ fica eventualmente menor que cn para todo $c > 1$). Usando estes dois teoremas, podemos provar a afirmação dada no início do capítulo de que nossas definições e as de Hopcroft e Ullman (1979, p. 288) são equivalentes.

Teorema 3.23. *Se* $L \in \mathsf{DSPACE}(f)$ *(ou* $L \in \mathsf{NSPACE}(f)$*), então L é decidida por uma máquina de Turing determinística (ou, respectivamente, não-determinística) que jamais ocupa mais de* $\max(f(n), 1)$ *posições de fita.*

Observe que, em nossa definição de DSPACE e NSPACE, permitimos à máquina violar a restrição de máximo de $f(n)$ células, num número finito de instâncias. De fato, autorizamos-na a nem parar nestes casos. São justamente esses casos que precisam ser tratados.

Demonstração. Como é um número finito de instâncias, podemos gravar todas elas no controle finito de M', junto do *status* destas palavras quanto à pertinência a $L(M)$.[18]

[18] Observe que, embora saibamos que tais cadeias existem, e que M possui comportamento não ambíguo nestas cadeias, não podemos usar um computador nem para encontrar todas as instâncias nem para determinar o comportamento de M nessas instâncias.

De certa forma, este teorema não é "computável".

M' lerá a entrada e testará se é alguma das cadeias armazenadas. Caso a entrada pertença à lista embutida, a aceite ou rejeite de acordo com o comportamento de M. Senão, retorne à posição inicial e execute M na entrada. Isto pode ser feito usando técnicas para autômatos finitos, resultando num gasto nulo de espaço.

Em qualquer caso, jamais ocuparemos mais de $\max(f(n),1)$ células. Note que M' é determinística se e só se M o for. ■

Complexidade de tempo é um pouco mais delicado.[19]

Teorema 3.24. *Se $L \in \mathsf{DTIME}(f)$ (ou $L \in \mathsf{NTIME}(f)$), então L é decidida por uma máquina de Turing determinística (ou, respectivamente, não-determinística) que jamais executa mais de* $\max(f(n), n+1)$ *movimentos.*

Demonstração. Suponha que k seja o tamanho da maior palavra que viola à restrição de tempo. Manteremos, no controle finito, as k primeiras células da fita de M.

M' começará deslocando-se k células à direita, populando este pedaço interno da fita. Mas serão feitas duas cópias deste pedaço interno. A primeira será populada apenas uma vez, e servirá para checar se a palavra pertence à linguagem, caso atinjamos o primeiro branco antes de realizar os k primeiros movimentos. Neste caso, gastamos $|x|$ movimentos ($|x|-1$ movimentos para deslocar-se entre cada letra e mais 1 para alcançar o caractere em branco), e com mais um movimento decidimos se aceitamos ou não a palavra, em tempo $|x|+1$.

A segunda cópia será tratada por nós como uma extensão da fita original. Ao executar o k-ésimo movimento, M' marcará a entrada com um caractere novo, e jamais mover-se-á à esquerda deste caractere. Sempre que M precisar ir àquela região, M' simulará a atuação de M dentro do próprio controle finito, na segunda cópia das k primeiras posições da fita. Esta alteração será feita apenas na fita de entrada; as demais fitas continuam operando normalmente.

De fato, simularemos M no controle finito de M' desde o início da execução do programa. Quando M' efetuar os k primeiros movimentos, M já terá executado, dentro de si, k movimentos diferentes. E, nas demais fitas, as posições das cabeças de leitura são exatamente as mesmas de M. Portanto, o funcionamento de M' é parecido com o de M — exceto nos primeiros k caracteres da entrada.

Como cada movimento de M corresponde a um movimento de M', o limite de $f(n)$ de tempo é respeitado quando $n > k$. Juntando com a análise

[19] Caso assumíssemos que $f(n) > cn$ para algum $c > 1$, poderíamos agir como no teorema anterior e usar o aceleramento linear; mas queremos abranger casos como, por exemplo, $f(n) = n + 17$.

anterior (do que acontece quando a entrada é mais curta que k), concluímos que M' aceita L e respeita às exigências de tempo para todo x. ■

Estes dois teoremas mostram que toda linguagem em $\mathsf{DSPACE}(f)$, de acordo com nossa definição, também está em $\mathsf{DSPACE}(f)$ de acordo com as definições de Hopcroft e Ullman (1979, p. 288) e Papadimitriou (1994, p. 141) (e afirmações análogas para DTIME, NSPACE e NTIME).

Finalizamos esta seção demostrando que a premissa $f(n) \geq n+1$ para complexidade de tempo apenas descarta linguagens que não são muito interessantes.

Proposição 3.25. *Suponha que $\mathcal{T}(M,x) \leq f(|x|)$ para alguma f, e $f(n_0) \leq n_0$ para algum n_0, então pertinência a $L(M)$ é decidida analisando apenas os n_0 primeiros caracteres da palavra.*

Observe que a hipótese é de que $\mathcal{T}(M,x) \leq f(|x|)$, não $L(M) \in \mathsf{DTIME}(f)$.

Demonstração. Para qualquer palavra x de tamanho n_0, M analisará, no máximo, até seu último caractere. Para chegar lá, M gastará, no mínimo, $|x| - 1 = n_0 - 1$ movimentos; como

$$\mathcal{T}(M,x) \leq f(|x|) = f(n_0) \leq n_0,$$

se M ainda não está num estado final, o próximo movimento terá que ser para um — caso contrário, violaria a equação acima.

Observe que M sequer pode analisar o caractere branco que vem após o x, pois precisaria de ao menos mais um movimento para atingir um estado final. Exatamente por isso, não importa o caractere que vem após o x. Qualquer palavra de tamanho maior que n_0 pode ser quebrada em xy, com $|x| = n_0$. Como M não pode violar a restrição discutida no parágrafo anterior, M deve determinar pertinência de xy à linguagem baseando-se apenas em x.

Como em nossa argumentação exigimos apenas que $|x| = n_0$, isso prova o teorema. ■

Defina o conjunto A como sendo as palavras de tamanho n_0 de $L(M)$, e B as palavras de tamanho menor que n_0. Podemos, então, reformular a conclusão do teorema como

$$L(M) = A\Sigma^* \cup B,$$

em que A e B são conjuntos *finitos*.

O converso também é verdadeiro (qualquer linguagem dessa forma satisfaz às conclusões do teorema anterior). Portanto, são um subconjunto próprio das linguagens regulares.

3.2.1 Relações entre medidas de Complexidade

Por serem medidas de complexidade específicas, podemos impor relações mais fortes entre elas do que as que são fornecidas pelo teorema 3.6.

Proposição 3.26.

$$\mathsf{DTIME}(f) \subseteq \mathsf{NTIME}(f) \tag{3.6}$$
$$\mathsf{DSPACE}(f) \subseteq \mathsf{NSPACE}(f) \tag{3.7}$$
$$\mathsf{DTIME}(f) \subseteq \mathsf{DSPACE}(f) \tag{3.8}$$
$$\mathsf{NTIME}(f) \subseteq \mathsf{DSPACE}(f) \tag{3.9}$$

Demonstração. 3.6 e 3.7 são consequências diretas do fato de que toda máquina determinística é, em particular, não determinística.

Para 3.8, note que, em $f(n)$ movimentos, a máquina pode ler, no máximo, $f(n)$ diferentes células — afinal, no máximo uma célula nova pode ser visitada a cada movimento. Portanto, se a máquina não extrapolar o limite de $f(n)$ movimentos, certamente não extrapolará o limite de $f(n)$ células da fita.

Para 3.9, usaremos uma máquina com múltiplas fitas.

Se M é uma máquina que reconhece $L \in \mathsf{NTIME}(f)$, existe um limite na quantidade de possíveis transições que M pode fazer em cada estado; digamos, t transições diferentes. Cada cadeia sobre $\{0,\ldots,t-1\}$ representa uma possível sequência de transições, que pode levar à parada ou não.

Em uma das fitas, a nova máquina M' enumerará todas as palavras de $\{0,\ldots,t-1\}^*$. Para cada palavra enumerada, M' simulará M na entrada, escolhendo as transições de acordo com a palavra que foi enumerada na outra fita. No evento de alguma transição ser para um estado final, consideraremos tal palavra "fechada".

Se a transição for para um estado de aceitação, aceitamos a entrada; e, se todas as palavras de um mesmo tamanho k forem fechadas sem aceitação, nenhuma palavra codifica uma sequência de transições que leva a um estado de aceitação — todas as menores que k já foram analisadas e todas as maiores que k possuem uma palavra de tamanho k que já foi fechada, fazendo com que a palavra inteira fique fechada. Nesta situação, podemos rejeitar a entrada.

Como $L \in \mathsf{NTIME}(f)$, qualquer sequência de transições leva a algum estado final em, no máximo, $f(n)$ transições. Portanto, sabemos que fecharemos todas as palavras de tamanho $f(n)$,

Como as fitas de trabalho de M não ocupam mais do que $f(n)$ células, e a fita de enumeração de M' nunca precisará enumerar uma palavra mais longa que $f(n)$, a complexidade de espaço de M' é $f(n)$. Concluímos que $L \in \mathsf{DSPACE}(f)$. ■

Teorema 3.27. *Suponha que $f(n) \geq \log n$ para todo n. Se $L \in \mathsf{DSPACE}(f)$, então existe uma constante c tal que $L \in \mathsf{DTIME}(c^f)$.*

Demonstração. Seja M uma máquina que aceita L em espaço f. Conforme observado na equação 3.1, existem constantes a e c tais que, caso M ocupe exatamente k células na fita, existirão, no máximo,

$$ac^k$$

diferentes configurações na fita. Para $k = f(n)$, a equação lê

$$ac^{f(n)}.$$

Como M é um decisor, M encerra sua computação nesta quantidade de passos.

Observe que $c^{f(n)} \geq n$, pois $f(n) \geq \log n$. Portanto, podemos usar o aceleramento linear para nos livrar daquela constante a, provando, assim, o teorema. ■

Teorema 3.28. *Suponha que $f(n) \geq \log n$ para todo n. Se $L \in \mathsf{NTIME}(f)$, então existe uma constante c tal que $L \in \mathsf{DTIME}(c^f)$.*

Demonstração. Pela equação 3.9, $L \in \mathsf{DSPACE}(f)$. Combinando com o teorema anterior, temos $L \in \mathsf{DTIME}(c^f)$ para algum c. ■

Papadimitriou (1994, p. 47) observa que o último resultado pode ser expressado como

$$\mathsf{NTIME}(f) \subseteq \bigcup_{c>0} \mathsf{DTIME}(c^f).$$

Analogamente, o penúltimo é equivalente a

$$\mathsf{DSPACE}(f) \subseteq \bigcup_{c>0} \mathsf{DTIME}(c^f).$$

3.2.2 Teorema de Savitch

As demonstrações dos teoremas da seção 3.2.1 observam que, se uma máquina de Turing respeita certa limitação de complexidade, esta mesma máquina respeitará alguma outra limitação. Por exemplo, no teorema **??**, a mesma máquina que aceita a linguagem em espaço $f(n)$ a aceita em tempo $ac^{f(n)}$. A única exceção é a equação 3.9, na qual fizemos uma máquina determinística simular uma máquina não-determinística sem estourar certo limite de espaço. Mas, observe que, em nenhum ponto daquela demonstração, a

máquina simuladora calcula o valor de f; isto é, os limites de complexidade da máquina simulada estavam implícitos.

Digamos que a máquina simulada M tenha um limite de complexidade f. Caso a máquina que está simulando M possa, de alguma forma, computar f, ela poderá, por exemplo, prever o tempo de execução (ou a quantidade de memória gasta) por M antes de fazer a simulação; esta previsão dá à máquina simuladora flexibilidade adicional, que permite fazer afirmações mais fortes a respeito das relações entre as classes de complexidade.

Uma destas afirmações é o teorema de Savitch. Em essência, o teorema diz que, sob condições apropriadas,

$$\mathsf{NSPACE}(f) \subseteq \mathsf{DSPACE}(f^2).$$

A demonstração deste teorema também envolve uma simulação. Aqui, a máquina simuladora (que é determinística e roda em espaço $f(n)^2$) calculará a complexidade de espaço da máquina simulada (que é não-determinística e roda em espaço $f(n)$) para garantir que a sua própria limitação de espaço não seja violada. Entretanto, a computação de f não pode extrapolar o limite de espaço que a própria máquina simuladora possui; portanto, precisaremos nos restringir a funções f que podem ser "computadas rapidamente". A seguinte definição captura esta noção.

Definição 3.29. Uma função $S : \mathbb{N} \to \mathbb{N}$ é dita ser *espaço-construtível*[20] se existir uma máquina de Turing determinística M tal que, para todo x,

$$M(x) = \langle f(|x|) \rangle$$

e

$$\mathcal{S}(M,x) \leq f(|x|).$$

Em outras palavras, M calcula f sem gastar mais espaço do que o próprio valor produzido. (ARORA; BARAK, 2009, p. 79)

Todas as funções "bem-comportadas" como polinômios, exponenciais, fatorial, e a soma, multiplicação e composição destas funções são espaço-construtíveis; portanto, embora usar apenas funções espaço-construtíveis limite as funções que podemos usar, a maior parte dos algoritmos "encontrados no dia-a-dia" possuem limites de espaço que são construtíveis.

Teorema 3.30 (Teorema de Savitch). *Se f é uma função espaço-construtível e $f(n) \geq \log n$, então*

$$\mathsf{NSPACE}(f) \subseteq \mathsf{DSPACE}(f^2). \qquad \square$$

[20] Do inglês *space-constructible*

Omitiremos a demonstração do teorema de Savitch pois tanto o enunciado preciso quanto a demonstração deste teorema requerem o desenvolvimento mais detalhado da noção de "espaço-construtível". Ela pode ser encontrada, por exemplo, no livro de Hopcroft e Ullman (1979, p. 297, 301) ou no de Arora e Barak (2009, p. 79, 86).

3.2.3 Principais Classes de Complexidade Computacional

Definição 3.31.

$$\mathsf{P} = \bigcup_{k \geq 1} \mathsf{DTIME}(n^k)$$
$$\mathsf{NP} = \bigcup_{k \geq 1} \mathsf{NTIME}(n^k)$$
$$\mathsf{PSPACE} = \bigcup_{k \geq 1} \mathsf{DSPACE}(n^k)$$
$$\mathsf{NPSPACE} = \bigcup_{k \geq 1} \mathsf{NSPACE}(n^k)$$

De acordo com o teorema da união, estes conjuntos são classes de complexidade.

As classes P e NP são as protagonistas do problema mais importante da Ciência da Computação teórica. De um lado, temos a classe P, as linguagens que podem ser resolvidas em tempo polinomial. De outro, temos a classe NP, as linguagens que podem ser verificadas em tempo polinomial.[21]

[21] A ideia é que, se dispomos de uma sequência de transições para uma máquina não-determinística, podemos rapidamente executar esta sequência de transições e verificar que a entrada, de fato, pertence à linguagem; isto é, depois que o problema está resolvido, é fácil verificar se a solução está correta ou não.

Existe uma caracterização alternativa da classe NP baseada nesta ideia. Uma linguagem L *polinomialmente equilibrada* é um conjunto de pares (x,y) de palavras tais que $|y| \leq p(|x|)$ para algum polinômio p (LEWIS; PAPADIMITRIOU, 1998, p. 298) — isto é, y não é muito maior do que x.

Se uma linguagem L está em NP, podemos construir uma linguagem $L' \in \mathsf{P}$, polinomialmente equilibrada, tal que o conjunto $\{x \mid (x,y) \in L \text{ para algum } y\}$ é exatamente o conjunto L. É como se x fosse um problema a ser solucionado e y fosse a sua solução.

O contrário também vale: a partir de $L' \in \mathsf{P}$, podemos construir $L \in \mathsf{NP}$. Portanto, podemos interpretar o problema P vs NP como a diferença entre resolver um problema pela primeira vez e verificar que uma solução já pronta funciona.

De acordo com o teorema de Savitch,

$$\mathsf{NSPACE}(n^k) \subseteq \mathsf{DSPACE}(n^{2k}),$$

portanto temos

$$\mathsf{NPSPACE} = \mathsf{PSPACE}.$$

Este resultado é surpreendente: enquanto que, para a complexidade de tempo, o uso de não-determinismo aparenta fornecer ganhos exponenciais, para a complexidade de espaço, o ganho é meramente quadrático.

Pelas equações 3.6 e 3.9, sabemos que a seguinte cadeia de inclusões é verdadeira:

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \quad (= \mathsf{NPSPACE}).$$

Embora os cientistas da computação em geral acreditem que ambas as inclusões sejam estritas (GASARCH, 2012, p. 54), nós sequer tivemos sucesso em demonstrar que

$$\mathsf{P} \subsetneq \mathsf{PSPACE}.$$

4 ORÁCULOS

O conceito de "oráculos computacional" é um mecanismo bastante poderoso para definir novas classes de complexidade computacional. Com ele, podemos formalizar perguntas como "Que linguagens conseguiríamos reconhecer se desse para resolver SAT gratuitamente?", ou "E se o problema da parada fosse decidível?".

Na seção 4.1 será apresentada a definição formal de "oráculo computacional". Neste primeiro momento, trabalharemos apenas com linguagens como oráculos; na seção 5.5 utilizaremos funções em vez de linguagens.

A seção 4.2 discute a interpretação de utilizar um problema indecidível (como o problema da parada) como oráculo. Ainda no contexto da indecidibilidade, esta seção define o que significa dois oráculos serem equivalentes, e a seção 4.3 constrói uma hierarquia de problemas indecidíveis. A seção 4.4 interpreta o conceito de redução em termos de oráculos. Por fim, a seção 4.5 utiliza o conceito de oráculos para construir a hierarquia polinomial.

4.1 DEFINIÇÃO

Definição 4.1. Seja A uma linguagem qualquer. Uma *máquina de Turing com oráculo A* é uma máquina de Turing M^A que possui uma fita especial e três estados adicionais: $q_?$, q_y e q_n. Transitar para $q_?$ significa consultar o oráculo; ao fazer esta transição, caso a palavra nesta fita pertença à linguagem A, no próximo estado da computação M transitará para q_y (a resposta foi positiva); caso contrário, M transitará para q_n.[1]

A definição de aceitação não é alterada. Chamaremos de $L^A(M)$ o conjunto das palavras aceitas por M^A.

Intuitivamente, o oráculo é um dispositivo computacional acoplado à máquina de Turing M. É como se a máquina delegasse parte da computação a outra máquina de Turing; uma "chamada de função".

Observe que a única influência que A possui em M^A são as transições após M ir para o estado $q_?$. Ou seja, A não faz parte de M; de fato, podemos "acoplar" várias linguagens diferentes numa mesma máquina de Turing M e obter diferentes $L^A(M)$ com isso.[2]

[1] Observe que a cabeça da fita não se mexe durante a consulta. Portanto, a máquina pode escrever seu estado atual na fita antes de transitar para $q_?$ e recuperá-lo depois.

[2] É exatamente por causa disso que, na notação do conjunto das palavras aceitas por M^A, o A sobrescrito está junto de L, não de M.

Como a linguagem A não faz parte da máquina M^A, do ponto de vista formal, M^A continua sendo, essencialmente, uma tabela de transições; portanto, podemos estender de maneira natural a codificação $\langle\cdot\rangle$ para máquinas com oráculos. Da mesma forma, conceitos como $M^A(x)$ e função associada podem ser definidos de maneira análoga.

Assim, podemos definir uma enumeração de Gödel aceitável usando oráculos.

Definição 4.2. Seja A uma linguagem. A enumeração de funções recursivas em A, denotada por ϕ^A, é o mapeamento

$$\langle M^A \rangle \mapsto \phi^A_{\langle M \rangle},$$

em que $\phi^A_{\langle M \rangle}$ é a função definida por

$$\phi^A_{\langle M \rangle}(x) \simeq M^A(x).$$

Nem sempre ϕ^A será uma enumeração de Gödel aceitável. O próximo teorema delimita as situações em que isso não acontece.

Teorema 4.3. *ϕ^A é uma enumeração de Gödel aceitável se, e somente se, A for uma linguagem decidível.*

Demonstração. Suponha que A não seja decidível. Defina a função $f : \{0,1\}^* \to \{0,1\}$ por

$$f(x) = \begin{cases} 1, & \text{se } x \in A; \\ 0, & \text{se } x \notin A. \end{cases}$$

Observe que f não é uma função recursiva (pois A não é decidível), mas ela pode ser computada facilmente usando A como oráculo. Portanto, ϕ^A enumera uma função que não é recursiva, fazendo com que não seja uma enumeração de Gödel.

Agora, suponha que A seja decidível. Primeiro, observe que qualquer máquina de Turing sem oráculos M pode ser transformada numa máquina de Turing com oráculo M^A, que simplesmente ignora seu próprio oráculo. Portanto, todas as funções recursivas parciais são enumeradas por ϕ^A.

Agora, como A é uma linguagem decidível, existe uma máquina de Turing que sempre para que decide pertinência a A. Dada uma máquina M^A, podemos transformá-la numa máquina sem oráculo M equivalente — isto é,

$$M^A(x) \simeq M(x);$$

basta trocar as chamadas ao oráculo A pelo algoritmo que a decide. Portanto,

ϕ^A só enumera funções recursivas parciais.

Concluímos que ϕ^A é uma função sobrejetora cuja imagem é o conjunto das funções recursivas parciais. A construção da máquina universal e do teorema S_{mn} é análoga às máquinas sem oráculos.[3] ■

4.2 PROBLEMAS INDECIDÍVEIS E EQUIVALÊNCIA DE ORÁCULOS

É importante ressaltar que esta visão do oráculo A como outra máquina de Turing acoplada à máquina principal é puramente intuitiva; a linguagem A não precisa ser sequer computável para que ela possa ser usada como oráculo. De fato, Hopcroft e Ullman (1979, p. 210) introduzem este conceito no contexto de decidibilidade. Intuitivamente, se houvesse um algoritmo para o problema da parada, poderíamos resolver o problema da vacuidade para máquinas de Turing, por exemplo. Mas não podemos partir da premissa de que "existe uma máquina de Turing que resolve o problema da parada", pois esta hipótese contradiz o teorema da parada. Oráculos podem ser entendidos como uma formalização deste "e se?".

Mais precisamente, defina

$$L_u^1 = \{\langle M, x\rangle \mid M \text{ aceita } x\}$$
$$S_1 = \{\langle M\rangle \mid L(M) = \emptyset\}$$

L_u^1 é o problema da parada; S_1 é o problema da vacuidade.[4]

A demonstração padrão de que S_1 é indecidível é uma redução do problema da parada para o problema da vacuidade. Em essência, ela diz que, caso houvesse um algoritmo para a vacuidade, poderíamos usar este algoritmo para resolver o problema da parada. Como não há algoritmo para a parada, não pode haver algoritmo para a vacuidade.

Em termos de oráculos, isso significa que podemos decidir L_u^1 utilizando um oráculo para S_1. O oposto também ocorre:

Suponha que dispomos de um oráculo para L_u^1. Na entrada $\langle M\rangle$, construa uma máquina M' que ignorará sua própria entrada e enumerar os pares (i, j). Para cada par enumerado, M' executará M na i-ésima palavra por j movimentos. Caso M aceite, M' aceita a entrada, qualquer que seja. Caso contrário, M' enumera o próximo par, e tente de novo. De posse desta má-

[3] Observe que construir a máquina universal e o teorema S_{mn} pode ser feito mesmo que A não seja decidível; o problema é, justamente, a possibilidade de enumerar funções que não são recursivas.

[4] A notação S_n é usada também por Hopcroft e Ullman (1979, p. 210). A notação L_u^n é derivada da notação para o problema da parada destes mesmos autores (HOPCROFT; ULLMAN, 1979, p. 183).

quina, peça ao oráculo se ela aceita ε (ou qualquer palavra). Se sim, significa que, em algum par (i, j), a M aceitou a entrada i; portanto, $L(M) \neq \emptyset$. Caso contrário, significa que M nunca aceitou palavra alguma; portanto, $L(M) = \emptyset$.

Definição 4.4. Dois oráculos A e B são ditos equivalentes se existem máquinas M e N tais que $L^A(M) = B$ e $L^B(N) = A$.

Isto é, usando uma linguagem como oráculo, conseguimos decidir a outra, e vice-versa. O que nós mostramos no parágrafo anterior é que L_u^1 e S_1 são equivalentes.

A intuição por trás da equivalência é que, ao "programar" uma máquina de Turing com oráculo S_1 (por exemplo), podemos fingir que também podemos usar L_u^1 como oráculo, e fazer uma "chamada" a este oráculo; como eles são equivalentes, há um algoritmo que traduz de um para outro, portanto, podemos usar este algoritmo de tradução para simular uma chamada a L_u^1.

4.3 HIERARQUIA ARITMÉTICA

Muitos outros problemas podem ser demonstrados equivalentes (como oráculos) ao problema da parada, como o problema da correspondência de Post (HOPCROFT; ULLMAN, 1979, p. 214). Entretanto, nem todos os problemas podem ser resolvidos com oráculos para a parada. De fato, a mesma técnica que mostra que a parada é indecidível pode ser usado para gerar um problema indecidível para máquinas que usam o problema da parada como oráculo.

Defina

$$L_u^{n+1} = \{\langle M^{L_u^n}, x\rangle \mid M^{L_u^n} \text{ aceita } x\},$$
$$S_{n+1} = \{\langle M^{S_n}\rangle \mid L^{S_n}(M) = \emptyset\}.$$

L_u^{n+1} é o problema da parada para máquinas de Turing que usam L_u^n como oráculo; S_{n+1} é o problema da vacuidade para máquinas de Turing que usam S_n como oráculo.[5] Podemos demonstrar por indução que S_n é equivalente a L_u^n; e, imitando o teorema da parada, podemos mostrar que L_u^{n+1} é indecidível para máquinas que usam L_u^n como oráculo.

Defina

$$\Sigma_n = \{L \mid L = L^{L_u^n}(M) \text{ para alguma máquina } M^{L_u^n}\}.$$

[5] Observe que, se definirmos $L_u^0 = S_0 = \emptyset$ (ou algum conjunto recursivo qualquer), a fórmula é válida também para $n = 1$.

Ou seja, Σ_n é o conjunto das linguagens que são decidíveis usando L_u^n como oráculo.[6]

O parágrafo anterior mostra que as linguagens L_u^{n+1} e S_{n+1} não pertencem a Σ_n. Como, por definição, ambas as linguagens L_u^{n+1} e S_{n+1} pertencem a Σ_{n+1}, os conjuntos Σ_n e Σ_{n+1} são diferentes. Temos, para todo n,

$$\Sigma_n \subset \Sigma_{n+1}.$$

Desta forma, criamos uma hierarquia de problemas indecidíveis.[7]

Estes conjuntos também podem ser denotados por $\mathsf{RE}^{L_u^n}$; as "linguagens recursivamente enumeráveis que usam L_u^n como oráculo". São as linguagens que são reconhecidas pelo mesmo tipo de máquina usado para definir RE, mas agora equipadas com um oráculo para L_u^n.

Observe que, de posse de um oráculo para S_n (por exemplo), ao "programar" uma máquina de Turing com esse oráculo, podemos "fingir" que podemos usar qualquer linguagem de Σ_n como oráculo — como esta linguagem possui um algoritmo que a resolve usando o nosso oráculo S_n, podemos substituir todas as chamadas àquele oráculo por este algoritmo. Abusando um pouco da notação, podemos escrever

$$\begin{aligned} \Sigma_0 &= \mathsf{RE} \\ \Sigma_{n+1} &= \mathsf{RE}^{\Sigma_n}. \end{aligned} \tag{4.1}$$

A interpretação é que Σ_{n+1} é o conjunto das linguagens reconhecidas pelas mesmas máquinas de R, mas equipadas com um oráculo qualquer de Σ_n. Como $S_n \in \Sigma_n$, esta nova definição inclui a definição anterior; Como todas as linguagens de Σ_n podem ser reconhecidas usando um oráculo para S_n, qualquer linguagem de Σ_{n+1} pode ser reconhecida usando apenas S_n (basta fazer aquela troca de chamadas ao oráculo); portanto as duas definições são equivalentes. Usaremos esta notação novamente na seção 4.5, mas indo direto à esta construção alternativa, sem passar pela definição via S_n primeiro.

Estendendo o exemplo do Σ_n, podemos construir a hierarquia aritmética.[8]

[6] Se usarmos a convenção de que $L_u^0 = \emptyset$, o conjunto Σ_0 é exatamente o conjunto das linguagens recursivamente enumeráveis (geralmente denotado por RE).

[7] Em outras palavras, existem problemas "mais indecidíveis" que outros; é uma situação análoga à existência de infinitos de tamanhos diferentes.

[8] A hierarquia aritmética costuma ser caracterizada através de fórmulas de primeira ordem. Isto é, as linguagens de Σ_n são as que podem ser descritas por

$$x \in L \iff \exists x_1 \forall x_2 \exists x_3 \ldots Q x_n P(x, x_1, \ldots, x_n)$$

para algum predicado recursivo P, em que o quantificador Q depende da paridade de n. Π_n é definido de maneira análoga, mas com o primeiro quantificador sendo um $\forall$.

Definição 4.5 (Hierarquia aritmética).

$$\begin{aligned}\Sigma_0 &= \mathsf{RE}\\ \Sigma_{n+1} &= \mathsf{RE}^{\Sigma_n}\\ \Pi_n &= \mathsf{co}\Sigma_n = \left\{\{0,1\}^* \backslash L \mid L \in \Sigma_n\right\}\\ \Delta_n &= \Sigma_n \cap \Pi_n\end{aligned}$$

A hierarquia aritmética provê uma forma de "medir" o quão "indecidível" um problema é. Intuitivamente, quão mais alto um problema está na hierarquia, "mais indecidível" este problema é.

Por exemplo, os problemas

$$\{\langle M\rangle \mid L(M) = \Sigma^*\} \qquad \text{e} \qquad \{\langle M\rangle \mid L(M) \text{ é finita }\}$$

são equivalentes a S_2 (HOPCROFT; ULLMAN, 1979, p. 211, 213), portanto eles estão em $\Sigma_2 \backslash \Sigma_1$. Em outras palavras, de certa forma, é "mais difícil" decidir se uma máquina de Turing aceita todas as palavras do que decidir se ela aceita alguma.

Podemos mostrar que Σ_n é um subconjunto próprio de Σ_{n+1}, e que o mesmo vale para Π_n e Π_{n+1}. Temos

$$\begin{aligned}\Pi_n \cup \Sigma_n &\subset \Delta_{n+1},\\ \Delta_{n+1} &\subset \Sigma_{n+1},\\ \Delta_{n+1} &\subset \Pi_{n+1}.\end{aligned}$$

Esta estrutura de inclusões está esquematizada na figura 1.

Encontraremos uma estrutura de inclusões similar a esta na seção 4.5.

4.4 REDUÇÕES E ORÁCULOS

Definição 4.6 (NP-completude). Uma linguagem L é NP-completa se

- $L \in \mathsf{NP}$; e
- Para toda linguagem $L' \in \mathsf{NP}$ existe alguma função f, computável em tempo polinomial, tal que

$$x \in L' \iff f(x) \in L.$$

Esta construção alternativa é de Davis, Sigal e Weyuker (1994, p. 234).

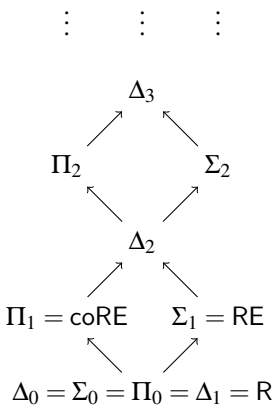

Figura 1 – Estrutura de inclusões da hierarquia aritmética.
As setas denotam inclusões estritas entre dois conjuntos.

Esta é a definição usual de NP-completude, encontrada, por exemplo, nos livros didáticos de Sipser (2006, p. 288) e Arora e Barak (2009, p. 42). Hopcroft e Ullman (1979, p. 324) e Papadimitriou (1994, p. 174) apresentam uma definição um pouco mais restritiva: a função f precisa ser computada em espaço logarítmico (o que implica tempo polinomial).

O termo "NP-completude" tenta formalizar a noção de "ser um problema difícil da classe NP". Com a definição acima, podemos provar que, se acharmos algum algoritmo polinomial para um problema NP-completo, podemos resolver *todos* os problemas da classe NP em tempo polinomial.[9] Isto é, resolver um significa resolver todos; portanto, intuitivamente, estes problemas devem ser os "mais difíceis" da classe NP. Por exemplo, deve ser mais fácil achar um algoritmo polinomial para fatoração de inteiros, que ainda não foi nem provado estar em P nem ser NP-completo (DU; KO, 2014, p. 120), do que achar um algoritmo polinomial para SAT, o primeiro problema natural provado NP-completo (DU; KO, 2014, p. 80).

Podemos interpretar a redução de forma algorítmica, usando o conceito de oráculo. Usaremos como exemplo os problemas SAT e 3SAT.

Sabemos que podemos reduzir SAT para 3SAT. Dada uma instância x de SAT, execute o algoritmo de redução para obter uma instância y de 3SAT.

[9] Na vida real, após provar que certo problema é NP-completo, nós desistimos de achar algoritmos polinomiais para ele e tentamos desenvolver heurísticas (como, por exemplo, limitar o espaço de busca ou programação dinâmica), usar técnicas baseadas em inteligência artificial, mapear para problemas NP-completos mais bem estudados (como SAT e programação linear) ou nos contentar com soluções aproximadas.

x é satisfazível se, e somente se, y o for. Portanto, executar um algoritmo para SAT em x retornará a mesma resposta que executar um algoritmo para 3SAT em y. Ou seja, com um oráculo para 3SAT, podemos resolver SAT em tempo polinomial: o algoritmo de redução opera em tempo polinomial e a chamada ao oráculo gasta apenas uma transição (de $q_?$ para q_y ou q_n). Esta interpretação está sumarizada pelo algoritmo 4.1.

```
1 Leia a entrada x;
2 Execute o algoritmo de redução f em x para obter y;
3 Retorne Oráculo(y);
```

Algoritmo 4.1: Interpretação algorítmica da noção de redução.

Esta interpretação via oráculos mostra que a noção de redução possui uma restrição bastante forte: nós podemos chamar o oráculo apenas uma vez, e esta chamada precisa ser a última coisa que o algoritmo faz — não podemos nem mesmo alterar o valor retornado pelo oráculo.[10]

Embora esta restrição garanta que um algoritmo polinomial para o oráculo permita construir um algoritmo polinomial para a outra linguagem, ela não é necessária. Caso a função `Oráculo`, do algoritmo 4.1, seja implementada em tempo polinomial, podemos fazer quantas chamadas quisermos no algoritmo externo e alterar os valores retornados pelas chamadas à vontade; se o algoritmo externo em si (ignorando as chamadas ao oráculo) for polinomial, o algoritmo resultante também terá complexidade polinomial.[11]

Isto sugere que reduções impõem restrições arbitrárias e podem ser substituídas pelo uso de oráculos.[12]

4.5 HIERARQUIA POLINOMIAL

A hierarquia polinomial é uma generalização da classe NP, baseada no conceito de oráculo. Para defini-la, utilizaremos o abuso de notação mencionado na equação 4.1. Formalmente:

Definição 4.7. Seja $\mathcal{A}$ uma classe computacional. Então,

[10] Esta descrição é semelhante às restrições impostas pela recursão caudal (*tail recursion*).

[11] Este conceito também é válido para as hierarquias indecidíveis. Se A é recursiva em B (isto é, com um oráculo para B, conseguimos decidir A), então, caso B seja decidível, A também o será, mesmo que não tenhamos uma redução de A para B.

[12] Alguns autores (como Arora e Barak (2009, p. 42, p. 65)) distinguem várias noções diferentes de "redução". A redução apresentada na definição de NP-completude é chamada de *redução de Karp*, *redução por mapeamento* ou *redução muitos-para-um* (dependendo do autor). Já o uso de oráculos, por sua vez, constitui noutro tipo de redução: uma *redução de Cook* de A para B é uma máquina M^B que decide A em tempo polinomial (usando B como oráculo).

- $\mathsf{P}^{\mathcal{A}}$ é a classe dos problemas que podem ser resolvidos em tempo polinomial por uma máquina de Turing determinística, usando como oráculo alguma linguagem de $\mathcal{A}$;

- $\mathsf{NP}^{\mathcal{A}}$ é a classe dos problemas que podem ser resolvidos em tempo polinomial por uma máquina de Turing não-determinística, usando como oráculo alguma linguagem de $\mathcal{A}$; e

- $\mathsf{coNP}^{\mathcal{A}}$ é a classe dos complementos dos problemas em $\mathsf{NP}^{\mathcal{A}}$.

(A única diferença entre $\mathsf{P}^{\mathcal{A}}$ e $\mathsf{NP}^{\mathcal{A}}$ é o fato de permitirmos não-determinismo em $\mathsf{NP}^{\mathcal{A}}$.)

Intuitivamente, $\mathsf{P}^{\mathcal{A}}$ é a classe de problemas que podem ser resolvidos se pegarmos as máquinas que resolvem os problemas em P e darmos oráculos em $\mathcal{A}$ para elas.[13]

É importante ressaltar que esta notação não está associada semanticamente ao conceito de potência; de fato, existem oráculos A e B para os quais $\mathsf{P}^A = \mathsf{NP}^A$ e $\mathsf{P}^B \neq \mathsf{NP}^B$ (HOPCROFT; ULLMAN, 1979, p. 362),[14] independente de P ser igual a NP ou não.

Valendo-se desta notação, podemos definir a hierarquia polinomial de maneira muito enxuta.

Definição 4.8 (Hierarquia polinomial[15]).

$$\begin{aligned}
\Delta_0^p = \Sigma_0^p = \Pi_0^p &= \mathsf{P} \\
\Delta_{n+1}^p &= \mathsf{P}^{\Sigma_n^p} \\
\Sigma_{n+1}^p &= \mathsf{NP}^{\Sigma_n^p} \\
\Pi_{n+1}^p &= \mathsf{coNP}^{\Sigma_n^p} \\
\mathsf{PH} &= \bigcup_{n>=0} \Sigma_n^p
\end{aligned}$$

[13] Note que, caso $\mathcal{A}$ possua um problema polinomialmente completo B (isto é, todos os problemas de $\mathcal{A}$ se reduzem em tempo polinomial a algum problema $B \in \mathcal{A}$ em questão), a classe $\mathsf{P}^{\mathcal{A}}$ é a mesma classe que P^B, pois qualquer problema de $\mathcal{A}$ pode ser resolvido em tempo polinomial usando B como oráculo; basta substituir a chamada ao oráculo de $\mathcal{A}$ pelo algoritmo que o resolve usando B. Embora esta troca possa aumentar o tempo de computação, a máquina ainda termina de executar em tempo polinomial. Isso justifica a notação $\mathsf{P}^{\mathcal{A}}$; é como se tivéssemos todos os problemas de $\mathcal{A}$ à nossa disposição.

[14] De fato, Hopcroft e Ullman (1979, p. 362) argumentam que este é um dos motivos pelos quais a questão $\mathsf{P} = \mathsf{NP}$ é tão difícil de ser resolvida — os métodos que conhecemos (como, por exemplo, diagonalização) são facilmente traduzíveis para máquinas com oráculos. Portanto, uma prova de que $\mathsf{P} \neq \mathsf{NP}$ (por exemplo) precisaria empregar um método que deixaria de funcionar ao equipar as máquinas com o oráculo A citado acima.

Os primeiros elementos da hierarquia polinomial podem ser expressados diretamente.

Por exemplo, $\Sigma_1^p = \mathsf{NP}^\mathsf{P}$. Esta classe corresponde às máquinas de Turing não-determinísticas com acesso a oráculos em P que operam em tempo polinomial. Mas o próprio oráculo pode ser computado em tempo polinomial; como há uma quantidade polinomial de chamadas a este oráculo, ao substituí-lo por um algoritmo "de verdade", a máquina resultante ainda terá complexidade polinomial. Portanto, todo problema de Σ_1^p está em NP. E vice-versa: qualquer problema de NP está em NP^P — basta a máquina não usar seu oráculo. Portanto, $\Sigma_1^p = \mathsf{NP}$.

O mesmo raciocínio mostra que $\Delta_1^p = \mathsf{P}$ e que $\Pi_1^p = \mathsf{coNP}$.[16]

Já Δ_2^p representa a ideia de generalizar reduções.[17] Ao contrário de se comportar da maneira limitada, como no algoritmo 4.1, as máquinas polinomiais de Δ_2^p podem chamar seu oráculo uma quantidade polinomial de vezes, alterando o valor retornado arbitrariamente. Com isso, Δ_2^p engloba tanto NP quanto coNP — acredita-se que a inclusão seja própria.

As inclusões dentro da hierarquia polinomial podem ser sumarizadas por

$$\begin{aligned} \Pi_n^p \cup \Sigma_n^p &\subseteq \Delta_{n+1}^p, \\ \Delta_{n+1}^p &\subseteq \Sigma_{n+1}^p, \\ \Delta_{n+1}^p &\subseteq \Pi_{n+1}^p. \end{aligned}$$

Observe que a estrutura de inclusões da hierarquia polinomial é muito parecida com a estrutura de inclusões da hierarquia aritmética (o que justifica o uso dos mesmos símbolos para ambas as hierarquias), mas, no caso da hierarquia polinomial, não conseguimos demonstrar que as inclusões são estritas. (De fato, uma delas, $\Delta_1^p \subseteq \Sigma_1^p$, é exatamente o problema P vs NP.) Esta estrutura está esquematizada na figura 2.

---

[15] Através de problemas completos para os níveis anteriores, Arora e Barak (2009, p. 102) mostram uma caracterização da hierarquia polinomial usando apenas um oráculo por nível. Por exemplo, Σ_n^p é equivalente a $\mathsf{NP}^{\Sigma_{n-1}\mathsf{SAT}}$, em que $\Sigma_{n-1}\mathsf{SAT}$ é um problema completo para Σ_{n-1}^p.

[16] Poderíamos ter escolhido qualquer subconjunto de P como início da hierarquia polinomial, que o restante seria o mesmo. De fato, Meyer e Stockmeyer (1972, p. 128) introduziram a hierarquia polinomial ao mundo escolhendo $\Delta_0^p = \Sigma_0^p = \Pi_0^p = \emptyset$.

[17] Esta é uma interpretação minha da hierarquia polinomial. Os criadores originais da hierarquia, Meyer e Stockmeyer (1972, p. 128), tinham por objetivo achar problemas "naturais" que fossem "difíceis". As separações entre as classes de complexidade, como, por exemplo, as demonstrações por Hopcroft e Ullman (1979, p. 299) são baseadas em diagonalização, e resultam em linguagens artificiais. O que Meyer e Stockmeyer (1972, p. 129) queriam era achar problemas que surgem "naturalmente" ao analisar problemas computacionais, que possuíssem as mesmas características destes resultantes de diagonalizações (isto é, que separassem classes de complexidade — fossem "difíceis").

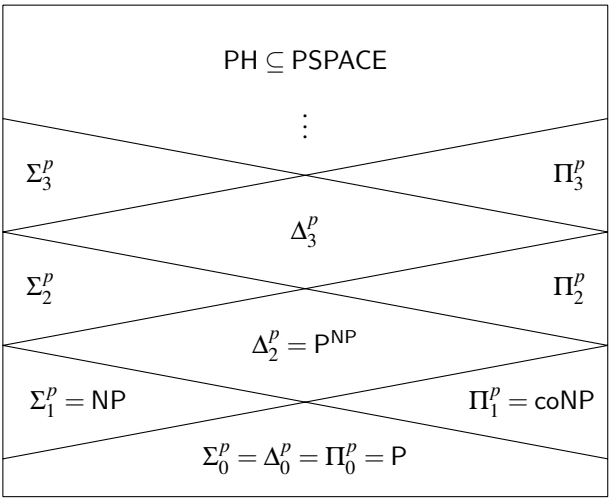

Figura 2 – Estrutura de inclusões da hierarquia polinomial.
Um conjunto estar listado abaixo do outro (mesmo que na diagonal) denota que o conjunto de baixo é um subconjunto do conjunto de cima.

5 FUNÇÕES NÃO-DETERMINÍSTICAS

Na seção 3.1, definimos dois axiomas que compreendem toda a informação necessária para definir uma medida de complexidade abstrata. As máquinas de Turing foram interpretadas como representações de funções de inteiros (isto é, funções cujo domínio e contradomínio são os números inteiros); neste contexto, as medidas de complexidade se referiam a funções, não a linguagens.

As duas medidas mais importantes, complexidade de tempo e espaço determinísticas, foram definidas nos exemplos 3.2 e 3.3, respectivamente. O exemplo 3.4, porém, ficou incompleto; o que exatamente é uma "função não-determinística"? O objetivo deste capítulo é prover uma definição para este termo e interpretá-lo em termos da hierarquia polinomial.

5.1 MOTIVAÇÃO

No contexto de problemas de decisão, embora não-determinismo não acrescente poder computacional — isto é, todo problema que uma máquina de Turing não-determinística resolve também pode ser resolvido usando máquinas determinísticas — não-determinismo aparenta reduzir a complexidade de um problema. Em particular, máquinas determinísticas aparentam ser exponencialmente mais lentas do que máquinas não-determinísticas para problemas como SAT.

Entretanto, problemas de decisão apenas retornam uma resposta booleana: descobrimos se a instância em questão possui solução ou não. Em problemas do mundo real, geralmente queremos *a solução*, não apenas saber se ela existe. Por exemplo, em vez de apenas descobrir que certa fórmula lógica não é uma tautologia, queremos saber exatamente qual atribuição de valores-verdade torna a fórmula falsa — por exemplo, esta fórmula pode ser gerada por um verificador formal de software, e a atribuição de valores-verdade que invalida a fórmula codifica uma entrada para o programa em que ele não faz o que deveria. Conhecer esta instância é importante para corrigir o software sendo verificado.

Nós já medimos a complexidade de uma função calculada por uma máquina de Turing determinística. Entretanto, para que possamos aferir a complexidade de uma função não-determinística, precisamos especificar como uma máquina não-determinística pode computar uma função.

5.2 DEFINIÇÃO

Se M é uma máquina de Turing determinística que computa a função f, para descobrir o valor de $f(x)$, simplesmente rodamos M em x. Existe apenas uma sequência de passos possível antes de M parar; o valor que M deixar na fita é $f(x)$.

Para máquinas não-determinísticas, ao rodar M numa entrada x arbitrária, podemos ver vários ramos de computação, e cada ramo pode encerrar com um valor diferente na fita. Então, em vez de termos um único valor para $M(x)$, existe um conjunto de valores possíveis. Não queremos um conjunto, e sim, um único valor. Adotaremos uma regra arbitrária para extrair um valor deste conjunto, que será justificada posteriormente.

Definição 5.1 (Função não-determinística[1]). Seja M uma máquina de Turing não-determinística e x uma palavra. Se a árvore de computação que M gera ao processar x for finita (isto é, nenhuma sequência de transições de M leva a um loop infinito), definiremos $M(x)$ como sendo o maior valor que a fita atinge nas folhas da árvore. Se a árvore não for finita, deixaremos $M(x)$ indefinido.

Caso a máquina gere palavras em vez de números, consideraremos o ordenamento lexicográfico, tratando sempre palavras menores como precedendo palavras maiores. Desta forma, a palavra vazia ε é lexicograficamente menor do que todas as outras. Esta suposição facilitará o projeto de linguagens adiante.

A ideia de pegar o máximo do conjunto vem da analogia com funções booleanas. Podemos interpretar um decisor para uma linguagem $L \subseteq \Sigma^*$ como uma máquina que computa a função característica de L. Isto é, se M decide a linguagem L, é como se M computasse a função $f : \Sigma^* \to \{0,1\}$ definida por

$$f(x) = \begin{cases} 1, & \text{se } x \in L; \\ 0, & \text{se } x \notin L. \end{cases}$$

De fato, podemos reescrever a máquina M para que escreva 1 na fita antes de aceitar a entrada e 0 antes de rejeitá-la.

Uma máquina não-determinística para SAT, por exemplo, pode chutar uma atribuição de valores-verdade e escrever 1 na fita se aquela atribuição satisfaz à fórmula e 0 caso contrário. Intuitivamente, os ramos que escreveram

[1] O termo "função não-determinística" está tecnicamente incorreto. Funções não são determinísticas ou não-determinísticas; o que é determinístico ou não são os dispositivos computacionais que as calculam. Toleraremos este abuso de nomenclatura para encurtar o texto.

1 são aqueles em que a máquina obteve sucesso em provar que a instância pertence à linguagem, e os ramos que escreveram 0 são aqueles em que a máquina fracassou.

Uma instância insatisfazível resultaria no conjunto $\{0\}$; uma instância tautológica retornaria o conjunto $\{1\}$, e as instâncias satisfazíveis, mas não tautológicas ficam no meio-termo: $\{0,1\}$. A função característica de SAT retorna 1 nos dois últimos casos e 0 no primeiro; corresponde, exatamente, ao maior valor destes conjuntos.

A exigência de a árvore de computação ter tamanho finito é importante. Caso apenas exigíssemos que o conjunto de folhas fosse finito, por exemplo, poderíamos construir uma máquina não-determinística que compute a função característica do problema da parada:

Basta que a máquina assuma dois ramos de computação. No o primeiro ramo, a máquina sempre escreve 0 na fita e para. No segundo ramo, a máquina age como uma máquina universal e simula a entrada. Se a máquina simulada parar, escreva 1 na fita e pare também.

Quando alimentada com o par $\langle M, x\rangle$ na entrada, caso M pare ao computar x, o "conjunto retornado" por nossa máquina não-determinística é $\{0,1\}$; 0 por causa do primeiro ramo (que sempre retorna 0) e 1 por causa do segundo ramo. Caso M nunca pare, apenas o primeiro ramo retornará, resultando no conjunto $\{0\}$. Como o resultado da função é o máximo destes conjuntos, temos, exatamente, a função característica do problema da parada.

Entretanto, como exigimos que a árvore de computação seja finita, podemos demonstrar equivalência entre os dois modelos, assim como pudemos mostrar equivalência entre determinismo e não-determinismo para decisores.[2]

Teorema 5.2. *Toda função calculada por uma máquina não-determinística é calculada por uma máquina determinística e vice-versa.*

Demonstração. Suponha que M é não-determinística e calcula certa função. Para calcular a mesma função usando uma máquina determinística, enumere todas as sequências finitas de transições de M, ordenando por tamanho; isto é, primeiro todas as sequências de tamanho 0, depois todas as de tamanho 1, depois todas as de tamanho 2, e assim por diante.

Para cada sequência de transições, rode M na entrada de acordo com a sequência. Se M parar, compare o que aquele ramo retornou com o maior valor obtido até agora — caso seja o primeiro valor retornado, será este o maior valor — e mantenha apenas o maior dentre estes valores.

[2] O termo "equivalência" é utilizado aqui no sentido de poder computacional, não eficiência; isto é, a classe de funções computadas por estes dispositivos (máquinas determinísticas e não-determinísticas) é o mesmo, embora as máquinas não-determinísticas sejam (aparentemente) muito mais rápidas.

Quando todas as sequências de transições de um mesmo tamanho retornarem, teremos chegado ao fim da árvore; limpe a fita, mantendo apenas o maior valor encontrado no percurso. Este valor corresponde ao maior valor que qualquer ramo da árvore retorna, o que corresponde ao valor da função computada por M, caso esteja definido.

Se a função não está definida na entrada atual, há dois casos. Se a árvore é infinita, nossa máquina determinística jamais parará de enumerar transições, pois existe ao menos uma sequência de transições que nunca para. Se a árvore é finita, mas nenhum ramo finaliza a computação (por exemplo, faz uma transição inválida), o "maior valor que qualquer ramo retorna" estará indefinido assim que chegarmos ao fim da árvore. Basta fazer uma transição inválida também neste caso.

A volta é trivial. ■

Usando as mesmas técnicas usadas para máquinas determinísticas, podemos construir uma máquina não-determinística universal e o teorema S_{mn}; portanto, acabamos de provar o seguinte teorema (e, efetivamente, concluir o exemplo 3.4).

Corolário 5.3. *Seja ϕ a função que associa a cada $w = \langle M \rangle$ (em que M é não-determinística) a função ϕ_w dada por*

$$\phi_w(x) \simeq M(x).$$

Então, ϕ é uma enumeração de Gödel aceitável. □

5.3 COMPLEXIDADE FUNCIONAL

De posse do formalismo de funções não-determinísticas, finalmente, concluímos a definição $\mathcal{NT}$ e $\mathcal{NS}$ para funções. FDTIME, FDSPACE, FNTIME e FNSPACE são definidos para funções de maneira análoga às definições para decisores. Por simetria, também definiremos coFNTIME e coFNSPACE; aqui, ao invés de tomarmos o máximo dentre os ramos, pegaremos o mínimo para ser o valor da função. (São as funções "co-não-determinísticas".) Nos concentraremos nas classes de complexidade de tempo, onde o uso de não-determinismo aparenta ter mais impacto.

Definição 5.4.

$$\mathsf{FP} = \bigcup_{k>0} \mathsf{FDTIME}(n^k)$$
$$\mathsf{FNP} = \bigcup_{k>0} \mathsf{FNTIME}(n^k)$$
$$\mathsf{coFNP} = \bigcup_{k>0} \mathsf{coFNTIME}(n^k)$$

FP são as funções que podem ser calculadas em tempo polinomial por máquinas determinísticas, e FNP, por máquinas não-determinísticas.[3] coFNP é o análogo a coNP.

Exemplo 5.5. Todos os algoritmos polinomiais discutidos em cursos de projeto e análise de algoritmos correspondem a funções em FP. Por exemplo, temos cálculo de determinantes, programação linear[4] e parsing de linguagens livres de contexto. □

Proposição 5.6. *Todas as funções características de problemas em* NP *estão em* FNP*, e todas as funções características de problemas em* coNP *estão em* coFNP. □

Exemplo 5.7. Os problemas em NP podem ser facilmente generalizados para funções em FNP. Por exemplo, podemos pegar uma máquina não-determinística para SAT, e modificá-la para que, após achar uma atribuição de valores-verdade, a escreva na fita. Caso contrário, escreva a palavra vazia. A função não-determinística computada por esta máquina é a maior atribuição de valores-verdade, lexicograficamente. Esta função está em FNP.

Similarmente, podemos encontrar o maior clique num grafo, achar fatores de um número, e encontrar um isomorfismo entre dois grafos usando o poder computacional de FNP.

Já problemas de minimização são mais facilmente interpretados como funções de coFNP. Por exemplo, encontrar o caminho hamiltoniano de menor custo, e calcular o número cromático. □

[3] Papadimitriou (1994, p. 229) também define uma classe de problemas para o símbolo FNP, mas a definição dele é substancialmente diferente da nossa. Veja a seção 6.1 para uma comparação.

[4] Programação linear é resolvível em tempo polinomial quando o domínio do problema são os números racionais. Se restringirmos o domínio da solução a números inteiros, o problema é NP-completo (PAPADIMITRIOU, 1994, p. 202).

Programação linear nos inteiros geralmente é chamada de *programação inteira* (do inglês *integer programming*).

O contrário também é possível; isto é, podemos transformar problemas de FNP em problemas de NP.

Teorema 5.8. *Se f é uma função de* FNP*, então a linguagem*

$$L_f = \{\langle x, y\rangle \mid f(x) \geq y\},$$

em que $\geq$ significa comparação lexicográfica, pertence a NP.

Demonstração. A ideia é fazer com que a máquina não-determinística M, que reconhecerá L_f, compute f usando seu próprio não-determinismo, e aceitará a entrada caso ache algum ramo de computação que produza um valor maior ou igual a y.

Mais formalmente, seja N uma máquina não-determinística que computa f em tempo polinomial. Considere que a entrada é $\langle x, y\rangle$. A primeira coisa que M fará é executar N em x. Sabemos que isso é possível pois ambas as máquinas são não-determinísticas.

Após N parar (o que ocorrerá em tempo polinomial), N deixará na fita um "candidato" a $f(x)$. De fato, todas as folhas da computação de N estarão disponíveis nos ramos de M. Como N computa f, apenas o maior destes valores será $f(x)$; portanto, se algum ramo encontrar algum candidato que seja maior ou igual a y, a entrada $\langle x, y\rangle$ pode ser aceita.

O contrário também ocorre. Se todos os ramos de computação gerarem candidatos a $f(x)$ que são menores que y, o próprio valor de $f(x)$ será menor que y, portanto a entrada pode ser rejeitada.

Em outras palavras, a máquina M deve aceitar exatamente quando encontrar alguma folha da computação de $N(x)$ que é maior que y. ■

De certa forma, este teorema nos permite fazer uma "busca binária" atrás do valor de $f(x)$ usando apenas algum dispositivo que reconheça L_f. Voltaremos a esta ideia na seção 5.5.

5.4 FUNÇÕES FNP-COMPLETAS

Da mesma forma como definimos problemas NP-completos, queremos, de alguma forma, representar completude dentro da classe FNP. No caso de NP-completude, a redução de L para L' (por exemplo) é feita por uma função f de forma que as "respostas" às questões "$x \in L$?" e "$f(x) \in L'$?" são as mesmas. Há uma barreira trivial ao definir "FNP-completude" de maneira análoga, que é a possibilidade de os contradomínios diferirem: para toda função $f : \Sigma^* \to \Sigma^*$ em FNP, podemos construir uma função g tal que $g(x) = \alpha$ para todo x, para algum $\alpha \notin \Sigma$. Isto é, para qualquer x, g retorna a palavra

com o único símbolo α. Como $\alpha \notin \Sigma^*$, f é inerentemente incapaz de produzir α na saída; portanto, qualquer que seja a função de redução h, $g(x)$ e $f(h(x))$ serão sempre diferentes.

Entretanto, intuitivamente, lidar com Σ diretamente não é estritamente necessário, pois sempre podemos codificar os caracteres de Σ usando binário. Portanto, para possibilitar uma definição direta de completude de funções, assumiremos que o domínio e o contradomínio das funções de FNP é $\{0,1\}^*$.

Definição 5.9. Uma função f é FNP-*completa* se $f \in$ FNP e, para toda função $g \in$ FNP, existir alguma função $h \in$ FP tal que

$$g(x) = f(h(x))$$

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

Na definição de NP-completude, a função h que faz a redução é obrigada a ter a seguinte propriedade: $x \in L$ se e somente se $h(x) \in L'$. Ou seja, os valores booleanos das proposições "$x \in L$" e "$h(x) \in L'$" são os mesmos. Se g e f são, respectivamente, as funções características de L e L', podemos expressar esta restrição como $g(x) = f(h(x))$, que é exatamente a expressão usada na definição de FNP-completude. Portanto, de certa forma, nossa definição de FNP-completude é uma generalização da noção de NP-completude.

Podemos fabricar um problema FNP-completo usando o problema da parada, mas impondo uma limitação no tempo de execução.

Dada uma máquina não-determinística M, uma entrada x e uma limitação de tempo n, defina $T(M,x,n)$ como sendo o conjunto de todas as palavras deixadas na fita, após exatamente n passos de computações não-determinísticas de M em x. (Inclua também os resultados de ramos que pararam antes de n passos.) Caso todos os ramos de computação de M encerrem em menos de n etapas, então $T(M,x,n)$ conterá todas as palavras das folhas da árvore de computação; neste caso, a maior palavra (lexicograficamente) de $T(M,x,n)$ é $M(x)$. Se existirem ramos de computação que não se encerram em n etapas, $T(M,x,n)$ conterá computações parciais, que podem ser lexicograficamente maiores que $M(x)$; portanto, neste caso, nada podemos afirmar sobre o máximo de $T(M,x,n)$.

Usaremos esta função T para definir nossa função FNP-completa.

Teorema 5.10. *Defina a função* HaltFNP : $\{0,1\}^* \to \{0,1\}^*$ *por*

$$\mathsf{HaltFNP}(\langle M,x,n\rangle) = \max(T(M,x,n)),$$

em que T é o conjunto definido acima e max *é o máximo lexicográfico dentre as palavras do conjunto. Então* HaltFNP *é* FNP*-completa.*

Demonstração. Seja g uma função de FNP. Pela definição de FNP, existe alguma máquina M e um polinômio p tal que, ao rodar M em x, todos os ramos de computação são mais curtos que $p(|x|)$, e $M(x) = g(x)$.

Defina a função $h : \{0,1\}^* \to \{0,1\}^*$ por

$$h(x) = \langle M, x, 1^{p(|x|)} \rangle.$$

Como M roda em tempo menor que $p(|x|)$, o maior valor (lexicograficamente) de $T(M,x,p(|x|))$ é $M(x)$, que é igual a $g(x)$ pois M computa g. Portanto,

$$\mathsf{HaltFNP}(h(x)) = M(x) = g(x).$$

O polinômio p pode ser computado em tempo $O(p)$, e os outros dois termos da tripla $\langle M, x, 1^{p(|x|)} \rangle$ só precisam ser copiados. Portanto, $h \in \mathsf{FP}$.

Isso mostra que toda função de FNP é redutível a HaltFNP; agora, falta mostrar que $\mathsf{HaltFNP} \in \mathsf{FNP}$.

Para isso, simplesmente rode M em x, usando o próprio não-determinismo para simular o não-determinismo de M e um contador para que o tempo de simulação não extrapole n. Então, apague todo o conteúdo da fita, deixando apenas o resultado da simulação de M em x; depois encerre a computação.

Desta forma, todas as folhas desta árvore de computação corresponderão exatamente aos valores de $T(M,x,n)$; portanto, o valor da função calculado por esta máquina não-determinística é $\mathsf{HaltFNP}(\langle M, x, n \rangle)$, o que prova que $\mathsf{HaltFNP} \in \mathsf{FNP}$. ■

Temos nossa primeira função FNP-completa. Entretanto, esta função é um tanto artificial. Idealmente, gostaríamos que alguma função mais natural (como, por exemplo, alguma generalização de SAT) fosse FNP-completa.

Podemos tentar seguir justamente este caminho (generalizar SAT), tentando reduzir da função HaltFNP definida acima. A generalização natural de SAT para FNP é a função que retorna alguma atribuição que satisfaz a fórmula (a maior atribuição, no caso), ou ε caso a fórmula não seja satisfazível. Sabemos da demonstração de NP-completude de SAT que é possível codificar estados de computação em fórmulas lógicas (COOK, 1971, p. 152–153); como esta codificação impõe, na fórmula lógica, o estado final da fita, podemos recuperar qual é a palavra deixada na fita pela computação codificada na fórmula, usando uma atribuição que satisfaz aquela fórmula.

Entretanto, não podemos simplesmente retornar uma atribuição para a fórmula gerada, pois a atribuição contém muito mais informação do que precisamos. Ela contém todas as etapas da computação, incluindo todos os estados intermediários e todas as escolhas não-determinísticas feitas pela má-

quina sendo reduzida. Nós só precisamos da fita deixada pela máquina, que corresponde a alguns bits da atribuição.

Podemos tentar a seguinte estratégia: passamos à máquina que calcula nossa generalização de SAT não apenas a fórmula lógica, mas também um número n, e a máquina retornará apenas os últimos n bits das atribuições que satisfazem a fórmula. Entretanto, isto assume que todos os tamanhos de fita serão iguais, o que pode não ser verdade. Portanto, precisamos, de alguma forma, codificar na própria fórmula lógica um meio de computar este n.

Ignoremos por um momento a interpretação de uma palavra sobre $\{0,1\}^*$ como uma atribuição de valores-verdade. Queremos codificar, de maneira variável e não ambígua, uma instrução de como extrair uma subpalavra de uma palavra dada. Para codificar a subpalavra w, coloque, no início da palavra que conterá w, a cadeia $0^{|w|}1$, seguida de w. Por exemplo, a palavra

$$0000\ 1\ 0110\ 10101010$$

"esconde" a subpalavra

$$0110$$

pois a palavra original começa com o prefixo $0^4 1$, indicando que os quatro caracteres seguintes codificam a palavra a ser extraída. (Poderíamos interpretar que a subpalavra também "esconde" a "subsubpalavra" 1, mas não precisamos ir tão longe.)

Definiremos uma função $d : \{0,1\}^* \to \{0,1\}^*$ que fará a decodificação. $d(0^n 1\omega)$ será definido como sendo os n primeiros dígitos de ω. Por completude, caso $|\omega| < n$, defina $d(0^n 1\omega) = \omega$, e $d(x) = \varepsilon$ se x não contiver um caractere 1.

Exemplo 5.11.

$$\begin{aligned} d(0000\ 1\ 0110\ 10101010) &= 0110 \\ d(0110) &= 1 \\ d(1) &= \varepsilon \\ d(11010) &= \varepsilon \\ d(0000) &= \varepsilon \\ d(0001\ 10) &= 10 \end{aligned}$$ □

Agora podemos construir uma função FNP-completa um pouco mais natural.

Definição 5.12. A função $\mathsf{F}d\mathsf{SAT} : \{0,1\}^* \to \{0,1\}^*$ é definida por

$$\mathsf{F}d\mathsf{SAT}(\varphi) = \max\{d(w) \mid w \text{ codifica uma atribuição que satisfaz } \varphi\}$$

Por completude, se x não for uma fórmula lógica (ou não for satisfazível), defina $\mathsf{F}d\mathsf{SAT}(x) = \varepsilon$.

Teorema 5.13. *A função* $\mathsf{F}d\mathsf{SAT}$ *definida acima é* FNP*-completa.*

Demonstração. Basta seguir as ideias anunciadas acima.

Reduza da função $\mathsf{HaltFNP}$ do teorema 5.10. Para uma entrada $\langle M,x,n\rangle$, utilize a mesma técnica usada na demonstração da NP-completude de SAT para construir um circuito lógico que codifique os n primeiros movimentos de M em x. Então, codifique um circuito lógico que "prepara" a atribuição que satisfaz a fórmula para "passar" pela função d e retornar apenas a fita.

Este circuito tem como entrada todas as variáveis que codificam a fita de M (que podem ter espaços em branco tanto à esquerda quanto à direita), num total de $n+|x|$ variáveis de entrada (pois este é o tamanho máximo da fita resultante), e tem $2(n+|x|)+1$ variáveis de saída, codificando uma cadeia w tal que $d(w)$ seja a fita que M produz após n rodadas computando sobre x.

O circuito pode ser projetado como tendo $n+|x|$ camadas para deslocar a fita para a esquerda, depois uma camada para transformar a fita no formato $0^{|w|}1w$ (com o $0^{|w|}1$ sendo introduzido à esquerda do w), e depois mais $n+|x|$ camadas para deslocar esta cadeia resultante novamente para a esquerda. Preencha o resto com zeros, é irrelevante.

Reordenando as variáveis da fórmula proposicional de forma que as saídas deste circuito combinatorial fiquem no começo, temos nossa transformação h, que pode ser efetuada em tempo polinomial, pois tanto a codificação das etapas de computação quanto a codificação do circuito podem ser feitas em tempo polinomial.

Agora, mostraremos que $\mathsf{F}d\mathsf{SAT}(h(y)) = \mathsf{HaltFNP}(y)$, para $y = \langle M,x,n\rangle$. Pela construção da função h acima, se ω codifica uma atribuição de valores-verdade para $h(y)$, então $d(\omega)$ é a fita que M deixa após n rodadas sobre x. Além disso, para toda computação de M em x por n rodadas, existe alguma atribuição ω' que satisfaz a fórmula lógica $h(y)$. Portanto, o conjunto

$$\{d(\omega) \mid w \text{ codifica uma atribuição que satisfaz } \varphi\}$$

é exatamente $T(M,x,n)$. Ou seja, o máximo deste conjunto é $\mathsf{HaltFNP}(y)$, concluindo que $\mathsf{F}d\mathsf{SAT}(h(y)) = \mathsf{HaltFNP}(y)$.

Portanto, reduzimos $\mathsf{HaltFNP}$ a $\mathsf{F}d\mathsf{SAT}$ e, portanto, todas as funções de FNP para $\mathsf{F}d\mathsf{SAT}$. Mostrar que $\mathsf{F}d\mathsf{SAT} \in \mathsf{FNP}$ é trivial. ■

Observe que, de certa forma, a cadeia inicial 0^n1 "explica" para d como esta função deve extrair a resposta do restante da palavra. Podemos ir além e, em vez de apenas prover um parâmetro para uma função fixa d, prover

uma *função* que extrairá a "parte que interessa" da cadeia. Representaremos a função por uma máquina de Turing, associada a um limite de tempo.

Definição 5.14. Se M for uma máquina de Turing determinística, denote por $t(M,x,n)$ o único elemento de $T(M,x,n)$.

Defina a função $\mathsf{CallbackSAT} : \{0,1\}^* \to \{0,1\}^*$ por

$$\mathsf{CallbackSAT}(\langle \varphi, M, n \rangle) = \max\{t(M,\omega,n) \mid \ \omega \text{ codifica uma atribuição que satisfaz } \varphi.\}$$

Por completude, se φ não for uma fórmula lógica (ou não for satisfazível), defina $\mathsf{CallbackSAT}(\langle \varphi, M, n \rangle) = \varepsilon$.

Em outras palavras, as folhas da árvore de computação de CallbackSAT serão as atribuições que satisfazem φ, mas depois de serem pós-processadas por M por n rodadas.[5]

Teorema 5.15. CallbackSAT *é* FNP*-completa.*

Demonstração. Basta reduzir de FdSAT. Escolha M como sendo uma máquina que computa d em tempo polinomial. Se p é um polinômio que limita o tempo de computação de M, escolha $n = p(k)$ quando a fórmula de entrada possuir k variáveis, e passe a fórmula φ intocada para CallbackSAT.

Como n é grande o bastante para que a máquina M sempre encerre a execução, $t(M,x,n) = d(x)$. Portanto, o conjunto que $\mathsf{CallbackSAT}(\langle \varphi, M, n \rangle)$ maximiza é

$$\{d(\omega) \mid \ \omega \text{ codifica uma atribuição que satisfaz } \varphi.\},$$

que é exatamente o conjunto maximizado por $\mathsf{F}d\mathsf{SAT}(\varphi)$. Portanto,

$$\mathsf{F}d\mathsf{SAT}(\varphi) = \mathsf{CallbackSAT}(\langle \varphi, M, n \rangle).$$

E, para executar CallbackSAT usando recursos de FNP, basta chutar uma atribuição de valores-verdade para φ e simular M por n rodadas nas atribuições que satisfizerem a fórmula. ■

5.5 ORÁCULOS FUNCIONAIS

Na seção anterior, definimos a função CallbackSAT, que incorpora na definição da função o desejo de fazer algum tipo de processamento adicional com a saída. Agora, definiremos a noção de *oráculo funcional*, que expande bastante esta capacidade de pós-processamento.

[5] Este mecanismo é semelhante à noção de *callback* em linguagens de programação.

Definição 5.16 (Oráculo funcional). Seja $f : \Sigma^* \to \Sigma^*$. Uma máquina que usa f como oráculo possui uma fita especial para se "comunicar" com f. A máquina escreve uma palavra nesta fita, digamos x, e consulta o oráculo (isto é, transita para $q_?$). Caso $f(x)$ exista, a máquina transitará para q_y e o conteúdo da "fita de comunicação" é alterado para $f(x)$. Caso $f(x)$ não exista, a máquina transita para q_n, com a fita inalterada.

Os estados q_y e q_n são apenas um preciosismo para que a definição permita qualquer função como oráculo, mesmo que não seja computável. No caso de máquinas de Turing, as funções computadas só não estarão definidas nas entradas em que a máquina não parar. Como estamos lidando com máquinas que possuem limites de tempo, a máquina sempre para (portanto a função sempre está definida), tornando o estado q_n redundante.

Definição 5.17. Seja $\mathcal{A}$ um conjunto de funções. $\mathsf{FP}^{\mathcal{A}}$ é o conjunto das funções que podem ser calculadas em tempo polinomial por alguma máquinas de Turing que usa alguma função de $\mathcal{A}$ como oráculo. Caso $\mathcal{A}$ seja um conjunto de linguagens, interpretaremos a máquina como usando a função característica correspondente. $\mathsf{FNP}^{\mathcal{A}}$ e $\mathsf{coFNP}^{\mathcal{A}}$ são definidas de maneira análoga.

$\mathsf{FP}^{\mathcal{A}}$, $\mathsf{FNP}^{\mathcal{A}}$ e $\mathsf{coFNP}^{\mathcal{A}}$ são os análogos funcionais de $\mathsf{P}^{\mathcal{A}}$, $\mathsf{NP}^{\mathcal{A}}$ e $\mathsf{coNP}^{\mathcal{A}}$, respectivamente.

Agora, remetendo ao teorema 5.8, podemos mostrar que, de certa forma, não ganhamos muito poder computacional ao utilizar a noção de "função não-determinística" em oráculos.

Teorema 5.18.

$$\mathsf{FP}^{\mathsf{NP}} = \mathsf{FP}^{\mathsf{FNP}}$$

Demonstração. Como as funções características de NP estão todas em FNP, a inclusão $\mathsf{FP}^{\mathsf{NP}} \subseteq \mathsf{FP}^{\mathsf{FNP}}$ segue direto da definição.

Seja $f \in \mathsf{FP}^{\mathsf{FNP}}$; existe uma máquina polinomial determinística M que computa f fazendo chamadas a um certo oráculo g. Usaremos a linguagem L_g, do teorema 5.8, para fazer uma busca binária atrás do valor de g.

Seja x a entrada de M. Substitua cada chamada a g pelo seguinte procedimento:

Primeiro, calcule qual o menor n tal que $\langle x, 0^n \rangle \in L_g$; como a ordenação lexicográfica que impomos considera primeiro o tamanho, isto nos dará qual é o tamanho de $g(x)$. Depois, faremos uma busca binária. Avance da esquerda para a direita, colocando um bit de 0^n para 1 de cada vez. Os bits

que fazem o par $\langle x,y\rangle$ permanecerem em L_g são mantidos, os demais são colocados de volta para 0. Isto calculará o valor de $g(x)$, bit a bit.

Este procedimento opera em tempo polinomial em n. Como $n = |g(x)|$ e $g \in \mathsf{FNP}$, n precisa ser polinomial em $|x|$; portanto, este procedimento opera em tempo polinomial em $|x|$. Já que M faz uma quantidade polinomial de chamadas a g, a máquina resultante também terá tempo de execução polinomial.

Cada chamada ao oráculo g foi substituída por um procedimento que devolve sempre a mesma resposta que g; portanto, a resposta final desta máquina modificada será igual à resposta de M, o que prova que $f \in \mathsf{FP}^{\mathsf{NP}}$. ■

Podemos definir e construir linguagens completas para os três conjuntos $\mathsf{FP}^{\mathcal{A}}$, $\mathsf{FNP}^{\mathcal{A}}$ e $\mathsf{coFNP}^{\mathcal{A}}$, usando as mesmas técnicas usadas para definir e construir problemas FNP-completos.

Definição 5.19. Seja $\mathcal{A}$ um conjunto de funções. Uma função f é $\mathsf{FP}^{\mathcal{A}}$-*completa* se $f \in \mathsf{FP}^{\mathcal{A}}$ e, para toda função $g \in \mathsf{FP}^{\mathcal{A}}$, existir alguma função $h \in \mathsf{FP}$ tal que

$$g(x) = f(h(x))$$

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

$\mathsf{FNP}^{\mathcal{A}}$ e $\mathsf{coFNP}^{\mathcal{A}}$-completude são definidos analogamente.

Esta definição é idêntica à definição de FNP-completude, mas equipando tudo com oráculos — com exceção da função h, que ainda precisa operar em tempo determinístico polinomial sem oráculos.

Usando problemas completos para $\mathcal{A}$, podemos construir facilmente problemas completos para $\mathsf{FP}^{\mathcal{A}}$, $\mathsf{FNP}^{\mathcal{A}}$ e $\mathsf{coFNP}^{\mathcal{A}}$.

Teorema 5.20. *Se $\mathcal{A}$ possuir um problema polinomialmente completo, então $\mathsf{FP}^{\mathcal{A}}$, $\mathsf{FNP}^{\mathcal{A}}$ e $\mathsf{coFNP}^{\mathcal{A}}$ possuem problemas polinomialmente completos.*

Demonstração. Usaremos essencialmente o mesmo truque usado no teorema 5.10 para mostrar que HaltFNP é FNP-completa.

Para o caso $\mathsf{FNP}^{\mathcal{A}}$, defina

$$\mathsf{HaltFNP}^{\mathcal{A}}(\langle M,x,n\rangle) = \max(T(M^A,x,n)),$$

em que A é um problema polinomialmente completo para $\mathcal{A}$. Mostraremos que $\mathsf{HaltFNP}^{\mathcal{A}}$ é $\mathsf{FNP}^{\mathcal{A}}$-completa.

Primeiro, mostraremos que esta função está em $\mathsf{FNP}^{\mathcal{A}}$. Para isso, a máquina que computa esta função usará A como oráculo. Basta simular M^A em x por n rodadas, e usar o próprio oráculo para resolver as chamadas de M. Como $A \in \mathcal{A}$ e estas simulações podem ser executadas em tempo polinomial, temos $\mathsf{HaltFNP}^{\mathcal{A}} \in \mathsf{FNP}^{\mathcal{A}}$.

Agora, à completude. Seja $f \in \mathsf{FNP}^{\mathcal{A}}$. Primeiro, mostraremos como computar f usando A como oráculo (e não usando um oráculo arbitrário de $\mathcal{A}$) e depois como usar a máquina que faz esta computação para reduzir f a $\mathsf{HaltFNP}^{\mathcal{A}}$.

Seja M uma máquina que computa f usando $B \in \mathcal{A}$ como oráculo. Como A é $\mathcal{A}$-completa, podemos substituir todas as chamadas a B por chamadas a A; basta usar a função que reduz B para A. Como esta função de redução opera em tempo polinomial, a máquina resultante também opera em tempo polinomial, e computa f usando apenas A como recurso.

Agora reduziremos f a $\mathsf{HaltFNP}^{\mathcal{A}}$. Seja p o polinômio que limita o tempo de computação da nova máquina M'. Na entrada x, basta escrever $\langle M', x, p(|x|)\rangle$ na fita. Tempo $p(|x|)$ é suficiente para que M' responda em todos os ramos; portanto,

$$\mathsf{HaltFNP}^{\mathcal{A}}(\langle M', x, p(|x|)\rangle) = M'(x) = f(x),$$

o que prova que $\mathsf{HaltFNP}^{\mathcal{A}}$ é $\mathsf{FNP}^{\mathcal{A}}$-completa.

Para $\mathsf{FP}^{\mathcal{A}}$ e $\mathsf{coFNP}^{\mathcal{A}}$ a demonstração é análoga. ■

5.6 HIERARQUIA POLINOMIAL

Munidos da definição de oráculo funcional, da seção 5.5, podemos construir um análogo à hierarquia polinomial, mas usando funções.

Definição 5.21 (Hierarquia Polinomial Funcional).

$$\begin{aligned}
\Sigma_0^f = \Delta_0^f = \Pi_0^f &= \mathsf{FP},\\
\Sigma_{n+1}^f &= \mathsf{FNP}^{\Sigma_n^f},\\
\Delta_{n+1}^f &= \mathsf{FP}^{\Sigma_n^f},\\
\Pi_{n+1}^f &= \mathsf{coFNP}^{\Sigma_n^f}.
\end{aligned}$$

Σ_n^f é a generalização funcional de Σ_n^p, Δ_n^f é a de Δ_n^p e Π_n^f a de Π_n^p. Alguns casos particulares desta hierarquia já foram discutidos; por exemplo,

$$\begin{aligned}
\Sigma_1^f &= \mathsf{FNP},\\
\Delta_1^f &= \mathsf{FP},\\
\Pi_1^f &= \mathsf{coFNP},\\
\Delta_2^f &= \mathsf{FP}^{\mathsf{FNP}}.
\end{aligned}$$

Assim como na hierarquia polinomial, Δ_n^f está contido tanto em Σ_n^f quanto em Π_n^f. O fato de Σ_n^f estar contido em Δ_{n+1}^f é simples de ser verificado, pois Δ_{n+1}^f possui acesso a oráculos que resolvem problemas de Σ_n^f. Para provar que Δ_{n+1}^f também inclui Π_n^f, precisaremos, de alguma forma, inverter o ordenamento das cadeias de $\{0,1\}^*$.

Proposição 5.22. *Para todo n,*

$$\Pi_n^f \subseteq \Delta_{n+1}^f.$$

Demonstração. Uma função $f \in \Pi_n^f$ é computada por uma máquina de Turing não-determinística M com acesso a um oráculo de Σ_{n-1}^f, tomando-se o mínimo de seus ramos. Construiremos uma máquina não-determinística M', usando o mesmo oráculo, de forma que a função em Σ_n^f que M' computa precisará apenas de um pós-processamento para resultar no valor de f. Este pós-processamento será feito por uma máquina determinística, que usará M' como oráculo; esta máquina determinística, portanto, computará f usando os recursos de Δ_{n+1}^f. (A ideia é que M' inverta o ordenamento das cadeias produzidas por M. No pós-processamento, iremos "desfazer" a inversão.)

Usaremos a função d de decodificação definida anteriormente. Na entrada x, M' começará executando exatamente como M até M parar, deixando (digamos) y na fita neste ramo de computação. (Podemos fazer esta computação pois o oráculo de M está em Σ_{n-1}^f, portanto também podemos usá-lo como oráculo.) Como M opera em tempo polinomial, $|y|$ está limitado superiormente por $p(|x|)$ para algum polinômio p. Então, antes de encerrar a computação, M' substituirá o valor y por

$$0^{|y|}1y0^{2*p(|x|)-2*|y|}.$$

A primeira cadeia de zeros garante que o valor de $d(M'(x))$ corresponda a algum dos ramos da computação de M. A última garante que todos os ramos de computação de M' resultem numa palavra de tamanho $2*p(|x|)+1$.

Chame de $e(y)$ a cadeia resultante da substituição acima, de forma que $d(e(y)) = y$ para todo y de tamanho menor que $p(|x|)$. Toda esta demonstração se baseia na observação de que, se $y_1 < y_2$, então $e(y_1) > e(y_2)$, para $|y_1|, |y_2| \leq p(|x|)$. Portanto, ao aplicar a transformação e, efetivamente invertemos localmente a ordenação das cadeias de $\{0,1\}^*$.

Dessa forma, se tanto M quanto M' tomam menos de k passos de com-

putação ao processar x,

$$\begin{aligned} M'(x) &= \max T(M',x,k) \\ &= \max\{e(y) \mid y \in T(M,x,k)\} \\ &= e\left(\min\{y \mid y \in T(M,x,k)\}\right) \\ &= e\left(\min T(M,x,k)\right) \\ &= e(f(x)). \end{aligned}$$

Como consequência, acabamos de demonstrar que a função $e \circ f$ pertence a Σ_n^f. Agora, basta usar uma terceira máquina M'', determinística, que usa $e \circ f$ como oráculo, e calcular d na resposta para obter o valor original de f.

Portanto, $f \in \Delta_{n+1}^f$. ■

O teorema 5.18 pode ser refraseado como

$$\Delta_2^f = \mathsf{FP}^{\Sigma_1^p}.$$

Observe que o oráculo utilizado é Σ_1^p, não Σ_1^f. Podemos generalizar esta relação para todos os níveis de nossa hierarquia polinomial funcional.

Teorema 5.23. *Para todo n,*

$$\begin{aligned} \Delta_{n+1}^f &= \mathsf{FP}^{\Sigma_n^p}, \\ \Sigma_{n+1}^f &= \mathsf{FNP}^{\Sigma_n^p}, \\ \Pi_{n+1}^f &= \mathsf{coFNP}^{\Sigma_n^p}. \end{aligned}$$

Demonstração. Faremos por indução. Observe que, como Π_n^p contém os complementos dos conjuntos de Σ_n^p,

$$\mathsf{coFNP}^{\Sigma_n^p} = \mathsf{coFNP}^{\Pi_n^p};$$

esta observação ajudará a resolver o caso de Π_{n+1}^f.

Para $n = 0$, é simples, pois $\Sigma_0^p = \mathsf{P}$.

$$\begin{aligned} \Delta_1^f &= \mathsf{FP}^{\mathsf{FP}} = \mathsf{FP} = \mathsf{FP}^{\mathsf{P}} = \mathsf{FP}^{\Sigma_0^p}; \\ \Sigma_1^f &= \mathsf{FNP}^{\mathsf{FP}} = \mathsf{FNP} = \mathsf{FNP}^{\mathsf{P}} = \mathsf{FNP}^{\Sigma_0^p}; \\ \Pi_1^f &= \mathsf{coFNP}^{\mathsf{FP}} = \mathsf{coFNP} = \mathsf{coFNP}^{\mathsf{P}} = \mathsf{coFNP}^{\Sigma_0^p}. \end{aligned}$$

Para $n \geq 1$, usaremos a mesma técnica usada para demonstrar o teo-

rema 5.18. Em essência, demonstraremos que uma chamada a um oráculos de Σ_n^f podem ser substituídas por várias chamadas a um oráculo de Σ_n^p (e algum processamento intermediário). Como este número de chamadas se manterá polinomial, toda a computação em si continuará sendo em tempo polinomial. Além disso, tomaremos cuidado para que a sequência de passos de computação que simulará a chamada a Σ_n^f seja feita de maneira determinística; assim, a mesma demonstração pode ser repetida para FP, FNP e coFNP.

Fixe um oráculo $f \in \Sigma_n^f$. Construiremos um oráculo $L \in \Sigma_n^p$ e mostrar como substituir uma única chamada a f por uma sequência de chamadas a L. Defina

$$L = \{\langle x, y\rangle \mid f(x) \geq y\}.$$

(Observe que L é L_f, nos termos do teorema 5.8.) Primeiro, mostraremos que $L \in \Sigma_n^p$.

O ponto crucial desta "subdemonstração" é perceber que, como f pertence a Σ_n^f, f pertence a $\mathsf{FNP}^{\Sigma_{n-1}^p}$, pela hipótese indutiva. De resto, procederemos como no teorema 5.8.

Seja M uma máquina que calcula f, usando $L' \in \Sigma_{n-1}^p$ como oráculo — a hipótese indutiva garante que tais M e L' existem. Construa uma máquina M' que, na entrada $\langle x, y\rangle$, agirá como M em x e calculará um dos ramos da computação de M em x. Se o resultado na fita for maior ou igual a y, aceite; caso contrário, rejeite.

Observe que M' reconhece uma linguagem de Σ_n^p. Argumentaremos que $L(M') = L$. De fato, se $f(x) > y$, algum dos ramos de computação de M produzirá um valor maior ou igual a y, portanto M' aceitará a entrada. Caso contrário, todos os ramos geram valores menores que y, portanto M' rejeitará a entrada. Assim, $L(M') = L$, o que prova que $L \in \Sigma_n^p$.

Agora, mostraremos como calcular $f(x)$ usando apenas chamadas a L: basta fazer busca binária assim como na demonstração do teorema 5.18. Primeiro, encontre o menor n tal que $f(x) < 0^n$, exclua um 0, e vá fazendo chamadas sucessivas a L para descobrir os demais bits de $f(x)$. Desta forma, com $2n+1$ chamadas a L, calculamos $f(x)$.

Em outras palavras,

$$f \in \mathsf{FP}^{\Sigma_n^p}.$$

Portanto, se $g \in \mathsf{FP}^f$, por exemplo, temos

$$\begin{aligned} g &\in \mathsf{FP}^f \\ &\subseteq \mathsf{FP}^{\mathsf{FP}^{\Sigma_n^p}} \\ &= \mathsf{FP}^{\Sigma_n^p}. \end{aligned}$$

Como esse processo pode ser feito para qualquer $f \in \Sigma_n^f$, acabamos de mostrar que

$$\Delta_{n+1}^f \subseteq \mathsf{FP}^{\Sigma_n^p}.$$

Analogamente,

$$\Sigma_{n+1}^f \subseteq \mathsf{FNP}^{\Sigma_n^p},$$
$$\Pi_{n+1}^f \subseteq \mathsf{coFNP}^{\Sigma_n^p}.$$

As inclusões contrárias são triviais. ■

5.7 COMPOSIÇÃO DE FUNÇÕES

Trabalhar com funções computadas por máquinas de Turing determinísticas nos permite implementar a composição destas funções de maneira muito simples: basta "concatenar" as duas máquinas. Assim que a primeira máquina parar, o cabeçote de leitura é movido para o começo da fita e a segunda máquina assume o controle. Nesta seção investigaremos a composição de funções não-determinísticas.

Definição 5.24 (Fecho sob composição). Dado um conjunto $\mathcal{F}$ de funções de $\{0,1\}^*$ em $\{0,1\}^*$, o *fecho compositivo* de $\mathcal{F}$, denotado por $\mathcal{F}^\circ$, é o conjunto de todas as funções que podem ser obtidas através da composição de um número finito de funções de $\mathcal{F}$. Isto é,

$$\mathcal{F}^\circ = \{g \mid \text{Existem funções } f_1, f_2, \ldots, f_n \text{ em } \mathcal{F} \text{ tais que } g = f_1 \circ f_2 \circ \cdots \circ f_n\}.$$

Se entendermos que a composição de zero funções resulta na função identidade, podemos construir esta operação da mesma forma como construímos o fecho de Kleene:

Defina $\mathcal{F}^{\circ 0} = \{I\}$, em que I é a função de identidade. Defina

$$\mathcal{F}^{\circ n+1} = \{f \circ g \mid f \in \mathcal{F}, g \in \mathcal{F}^{\circ n}\}.$$

Assim, temos

$$\mathcal{F}^\circ = \bigcup_{n \in \mathbb{N}} \mathcal{F}^{\circ n}.$$

Esta operação também possui propriedades análogas às do fecho de Kleene.

Proposição 5.25. *Se $\mathcal{F}$ e $\mathcal{G}$ são conjuntos de funções, então*

$$(\mathcal{F}^\circ)^\circ = \mathcal{F}^\circ \tag{5.1}$$

$$\mathcal{F} \subseteq \mathcal{F}^\circ \tag{5.2}$$

$$(\mathcal{F}^\circ \cup \mathcal{G}^\circ)^\circ = (\mathcal{F} \cup \mathcal{G})^\circ. \tag{5.3}$$

Além disso, se $\mathcal{F} \subseteq \mathcal{G}$, então

$$\mathcal{F}^\circ \subseteq \mathcal{G}^\circ. \tag{5.4}$$

Demonstração. Todas as funções de $(\mathcal{F}^\circ)^\circ$ são composições de funções de $\mathcal{F}^\circ$, que, por sua vez, são composições de funções de $\mathcal{F}$; portanto, as funções de $(\mathcal{F}^\circ)^\circ$ são composições de funções de $\mathcal{F}$, o que prova a fórmula 5.1.

Para a fórmula 5.2, basta escolher $n = 1$ na definição de fecho compositivo.

Para a 5.3, observe que as funções de $(\mathcal{F}^\circ \cup \mathcal{G}^\circ)^\circ$ são composições de funções de $\mathcal{F}^\circ$ e $\mathcal{G}^\circ$, possivelmente intercaladas; portanto, usando o mesmo raciocínio do primeiro parágrafo, concluímos que elas são composições de funções de $\mathcal{F} \cup \mathcal{G}$.

E, para a 5.4, note que todas as funções de $\mathcal{F}^\circ$ são composições de funções de $\mathcal{F}$, portanto também de $\mathcal{G}$; assim todas as funções de $\mathcal{F}^\circ$ também estão em $\mathcal{G}^\circ$. ■

Ao compor duas funções calculadas por máquinas de Turing determinísticas, o tempo de execução da máquina resultante será proporcional à soma dos tempos de execução das máquinas originais. Portanto, caso ambas as funções estejam em FP, o tempo de computação resultante também será limitado por um polinômio, resultando em outra função de FP. Portanto, a classe FP é fechada sob composição ($\mathsf{FP}^\circ = \mathsf{FP}$).

Ao equipar as máquinas com oráculos, encontramos um pequeno problema na hora de concatenar a máquina. Caso os oráculos sejam iguais, a concatenação descrita anteriormente funciona, pois a máquina resultante usará o mesmo oráculo das máquinas originais. Caso os oráculos sejam diferentes, não podemos simplesmente emendar as duas máquinas pois esta nova máquina precisaria de dois oráculos; mas podemos fazer a coisa funcionar sob as condições certas.

Teorema 5.26. *Se $\mathcal{A}$ possui um problema polinomialmente completo, então $\mathsf{FP}^{\mathcal{A}}$ é fechado sob composição.*

Demonstração. Basta unificar os oráculos usando o problema completo de $\mathcal{A}$. Seja A o problema completo de $\mathcal{A}$, e suponha que g é a composição

$$f_1 \circ f_2 \circ \cdots \circ f_n,$$

para $f_i \in \mathsf{FP}^{\mathcal{A}}$.

Tome máquinas $M_1, \ldots, M_n$ que computam, respectivamente, $f_1, \ldots, f_n$, e substitua seus oráculos pelo problema A; sabemos que isso é possível pois A é polinomialmente completo para $\mathcal{A}$. Agora, basta concatenar as máquinas. A substituição do oráculo manteve todas as máquinas M_i operando em tempo polinomial, portanto a máquina resultante também operará em tempo polinomial; isso mostra que $g \in \mathsf{FP}^{\mathcal{A}}$, o que prova que $\mathsf{FP}^{\mathcal{A}}$ é fechado sob composição. ■

Corolário 5.27. *Δ_n^f é fechado sob composição.* □

Para funções calculadas por máquinas não-determinísticas, não podemos meramente emendar as duas máquinas pois, em alguns ramos de computação, a segunda máquina pode estar operando com algum valor menor do que máximo — que é o valor real da função.

Para FNP, por exemplo, podemos calcular as funções de FNP° usando o poder computacional de $\mathsf{FP}^{\mathsf{FNP}}$: como FNP possui problemas completos, nós podemos emendar máquinas determinísticas que somente chamam seus oráculos de FNP para calcular a função composta. Portanto,

$$\mathsf{FNP}^{\circ} \subseteq \mathsf{FP}^{\mathsf{FNP}}.$$

De fato, podemos fortalecer este "$\subseteq$" para um "$=$".

Teorema 5.28. *Se $\mathcal{A}$ possui problemas polinomialmente completos, então*

$$(\mathsf{FNP}^{\mathcal{A}})^{\circ} = \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$$

Demonstração. Provar que o primeiro conjunto está contido no segundo é fácil. Pelo teorema 5.20, o conjunto $\mathsf{FNP}^{\mathcal{A}}$ possui problemas polinomialmente completos; portanto, pelo teorema 5.26, $\mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$ é fechado sob composição. Agora, como

$$\mathsf{FNP}^{\mathcal{A}} \subseteq \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}},$$

pela propriedade 5.4 da proposição 5.25,

$$(\mathsf{FNP}^{\mathcal{A}})^{\circ} \subseteq \left(\mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}\right)^{\circ} = \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}.$$

Para a inclusão contrária, usaremos um caminho indireto. Demonstraremos primeiro que algum problema completo de $\mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$ pode ser escrito como a composição de dois problemas de $\mathsf{FNP}^{\mathcal{A}}$. Chame de f este problema. Como f é polinomialmente completo para $\mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$, para toda função $g \in \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$ existe uma função $h \in FP$ tal que $g = f \circ h$. Como $h \in \mathsf{FNP}^{\mathcal{A}}$ e $f \in \left(\mathsf{FNP}^{\mathcal{A}}\right)^{\circ}$, a composição $f \circ h = g$ também estará em $\left(\mathsf{FNP}^{\mathcal{A}}\right)^{\circ}$.

Tiraremos o problema f da demonstração do teorema 5.20. Escolha

$$f = \mathsf{HaltFP}^{\mathsf{FNP}^{\mathcal{A}}},$$

em que $\mathsf{HaltFP}^{\mathsf{FNP}^{\mathcal{A}}}$ é definida por

$$\mathsf{HaltFP}^{\mathsf{FNP}^{\mathcal{A}}}(\langle M,x,n\rangle) = t(M^A,x,n),$$

em que A é um problema completo para $\mathsf{FNP}^{\mathcal{A}}$. Observe que f é, nos termos do teorema 5.20, $\mathsf{HaltFP}^{\mathcal{B}}$ para $\mathcal{B} = \mathsf{FNP}^{\mathcal{A}}$. A é um problema polinomialmente completo para $\mathcal{B}$, portanto, pelo mesmo teorema, f é polinomialmente completo para $\mathsf{FP}^{\mathcal{B}} = \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$.

De posse do problema completo f, o próximo passo é escrevê-lo como uma composição de duas funções de $\mathsf{FNP}^{\mathcal{A}}$. Estas duas funções serão computadas por máquinas que usam o problema A, que é completo para $\mathsf{FNP}^{\mathcal{A}}$.

Dada a entrada $\langle M,x,n\rangle$, o problema f consiste em simular a máquina M^A por n estados, na entrada x. M é determinística, portanto a mera simulação é fácil de ser feita, usando recursos de $\mathsf{FNP}^{\mathcal{A}}$. O problema são as chamadas ao oráculo A, que é polinomialmente completo para $\mathsf{FNP}^{\mathcal{A}}$. Como este oráculo representa uma função não-determinística, não podemos simplesmente computar A e prosseguir a simulação com o que quer que tenhamos obtido, pois o valor obtido neste ramo pode não ser o valor correto; usando apenas recursos de $\mathsf{FNP}^{\mathcal{A}}$, só temos acesso à maximização global no final de todas as etapas de computação. Então, precisaremos usar esta "pós-maximização" para obter esta maximização intermediária.

Seja p um polinômio que limita o tamanho de $A(x)$; isto é, $|A(x)| \leq p(|x|)$ para todo x. Sabemos que tal polinômio existe pois $A \in \mathsf{FNP}^{\mathcal{A}}$, e uma máquina de $\mathsf{FNP}^{\mathcal{A}}$ só tem "tempo" de escrever uma quantidade polinomial de bits na saída.

Se a máquina M faz k invocações ao oráculo A na entrada x nos primeiros n movimentos, enumere os valores chamados por $y_1, y_2, \ldots, y_k$. Defina

$$Y(M,x,i,n) = 0^{|A(y_i)|}1A(y_i)\ 0^{2*p(n)+1-|A(y_i)|};$$

ou seja, $Y(M,x,i,n)$ é a resposta da i-ésima invocação ao oráculo $(A(y_i))$, mas

codificada num formato especial: a resposta $A(y_i)$ é sucedida por uma cadeia de zeros de tamanho variável, fazendo com que todos os $Y(M,x,i,n)$ sejam de tamanho $2*p(n)+1$; além disso, a resposta está precedida de $0^{|A(y_i)|}1$, de forma que podemos recuperar $A(y_i)$ usando a função d de decodificação. (Observe que todas as perguntas feitas ao oráculo terão tamanho menor ou igual a n, portanto todas as respostas terão tamanho menor ou igual a $p(n)$. O $p(n)+1$ adicional é para englobar o prefixo inicial para a função d.) Defina $Y(M,x,i,n) = 0^{2*p(|x|)+1}$ caso $i > k$.

É importante notar que esta codificação preserva a ordem lexicográfica de todas as palavras de tamanho menor ou igual a $p(n)$.

Nossa primeira função F, dentre as duas que serão compostas para formar f, é definida por

$$F(\langle M,x,n\rangle) = \langle M,x,n\rangle Y(M,x,1,n)Y(M,x,2,n)\cdots Y(M,x,n,n).$$

Isto é, F retorna as respostas a todas as perguntas feitas pela máquina M ao oráculo A ao rodar n passos sobre x — além da entrada inicial $\langle M,x,n\rangle$, é claro. Mostraremos que $F \in \mathsf{FNP}^{\mathcal{A}}$. (A entrada inicial $\langle M,x,n\rangle$ é incluída pois precisaremos propagar esta informação para a segunda função que comporemos com F, mas, por ora, ignoraremos a existência deste "pedacinho" de F na argumentação.)

A máquina não-determinística que calcula esta função usará exatamente a "pós-maximização" global, que ela possui por calcular uma função não-determinística, para conseguir maximizações locais nas perguntas ao oráculo A. Entretanto, como F e A estão no mesmo "nível de complexidade" (isto é, ambas as funções estão em $\mathsf{FNP}^{\mathcal{A}}$), não podemos usar A como oráculo; aí entrará a notação de Y.

Chame de M_A a máquina não-determinística que calcula A, e M_F a máquina que calcula F (construiremos M_F agora). M_F simplesmente simula a execução de M em x. Quando M faz uma pergunta ao oráculo, M_F age como M_A, gerando vários ramos de computação. (Alguns levarão à resposta correta $A(y_1)$, mas outros não.) Independente de M_F ter acertado o valor de M_A ou não, M_F escreverá a resposta que acabou de achar na fita de saída, deixando-a na formatação dos $Y(M,x,i,n)$. Então, M_F retorna o controle a M, imaginando que a resposta ao oráculo está no ramo atual da fita, e prossegue a computação. Quando a quantidade de passos passar de n, basta preencher os Y faltantes com zeros.

Em outras palavras, cada ramo de computação de M_F retornará uma lista de palavras $w_1w_2\ldots w_n$, todas elas com tamanho $2*p(n)+1$. $d(w_1)$ é um candidato a $A(y_i)$; De fato, $d(w_1)$ é uma das folhas da computação de M_A em y_1. Caso $d(w_1)$ seja a resposta certa, a simulação feita por M_F coincidirá

com a computação de M em x, e a próxima pergunta de M ao oráculo será y_2. Então, $d(w_2)$ será um candidato a $A(y_2)$, novamente aparecendo numa das folhas da computação de M_A em y_2, e assim por diante.

Entretanto, caso w_1 não seja a resposta certa, M_F prosseguirá a simulação com a resposta errada, e a próxima pergunta feita por M ao oráculo (neste ramo de computação) pode não ser y_2; neste caso, a simulação é "inválida", no sentido de não corresponder exatamente à computação de M em x. Argumentaremos que, no máximo lexicográfico destas listas de palavras, todos os w_i correspondem às respostas certas do oráculo A; isto é, estes erros não ocorrem e M_F corretamente computa F.

Primeiro, note que todos os ramos de computação retornarão palavras de mesmo tamanho. Dessa forma, dadas duas listas, $w_1 w_2 \dots w_n$ e $v_1 v_2 \dots v_n$, se $w_1 > v_1$ lexicograficamente, a primeira lista tem precedência sobre a segunda. Portanto, o máximo dentre os ramos de computação também terá de maximizar, localmente, o candidato à primeira resposta ao oráculo. Isto é, se o máximo lexicográfico é

$$\omega_1 \omega_2 \dots \omega_n,$$

sabemos com certeza que $d(\omega_1) = A(y_1)$, pois algum ramo de computação gerou $A(y_1)$ como candidato à primeira resposta, e nenhum outro ramo pode ter gerado um valor maior.

Agora, como acertamos a primeira resposta do oráculo, a simulação de M_F corresponderá à computação de M em x, e a segunda pergunta feita na simulação será y_2. Usando um raciocínio similar, concluímos que $d(\omega_2) = A(y_2)$, e assim por diante, até concluir que $d(\omega_i) = A(y_i)$ para todo i.

Isso prova que $F \in \mathsf{FNP}^{\mathcal{A}}$. Agora, construiremos a segunda função, G, que será composta com F para gerar f. (A função G usará aquele $\langle M,x,n\rangle$ que da definição da função F que ignoramos até agora — note que a argumentação continua válido pois este valor age como uma constante presente em todos os ramos de computação.)

A máquina M_G que calculará G pegará a saída de M_F e descobrirá o valor de $t(M,x,n)$. Para isso, basta simular M em x por n rodadas; as respostas à todas as consultas a A já foram descobertas por M_F. Portanto, M_G pode ser determinística sem oráculos.

Assim, por construção, $G(F(\langle M,x,n\rangle)) = t(M,x,n)$; portanto,

$$G \circ F = \mathsf{HaltFP}^{\mathsf{FNP}^{\mathcal{A}}} = f.$$

Como G e F pertencem a $\mathsf{FNP}^{\mathcal{A}}$, mostramos, finalmente, que

$$f \in \left(\mathsf{FNP}^{\mathcal{A}}\right)^{\circ}.$$

Agora, podemos repetir a argumentação dada no início da demonstração.

Se $g \in \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$, então existe alguma função $h \in \mathsf{FP}$ tal que $f \circ h = g$ (pois f é completa para $\mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$). Portanto, como $f \in \left(\mathsf{FNP}^{\mathcal{A}}\right)^{\circ}$ e $\mathsf{FP} \subseteq \mathsf{FNP}^{\mathcal{A}}$, temos $g \in \left(\mathsf{FNP}^{\mathcal{A}}\right)^{\circ}$, o que prova que

$$\mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}} \subseteq \left(\mathsf{FNP}^{\mathcal{A}}\right)^{\circ}. \qquad \blacksquare$$

Corolário 5.29.

$$(\Sigma_n^f)^{\circ} = \Delta_{n+1}^f. \qquad \square$$

Com este corolário, atingimos um dos objetivos deste trabalho; nós encontramos uma relação na hierarquia polinomial através da composição de funções não-determinísticas.

Analisando mais cuidadosamente a demonstração, podemos fortalecer levemente o teorema. A função G é calculada por uma máquina determinística sem oráculos; portanto, na verdade, temos $G \in \mathsf{FP}$. Portanto, toda função $f \in \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$ pode ser escrita como

$$G \circ F \circ h,$$

em que G e h pertencem a FP e F pertence a $\mathsf{FNP}^{\mathcal{A}}$.

Mas, como h é uma função determinística que opera em tempo polinomial, podemos "emendar" a máquina que computa h no início da máquina que computa F sem prejuízos. Ou seja, a função $H = F \circ h$ pertence a $\mathsf{FNP}^{\mathcal{A}}$; provamos, portanto, o seguinte teorema.

Corolário 5.30. *Se $\mathcal{A}$ possui problemas polinomialmente completos, então todo problema $f \in \mathsf{FP}^{\mathsf{FNP}^{\mathcal{A}}}$ pode ser escrito como*

$$G \circ H, \qquad \square$$

em que $G \in \mathsf{FP}$ e $H \in \mathsf{FNP}^{\mathcal{A}}$. Além disso, G não depende de f.

Em outras palavras, compondo apenas duas funções de Σ_n^f, obtemos todo o poder computacional das repetidas chamadas a oráculos de Δ_{n+1}^f. Isso sugere que talvez tenhamos "exagerado na dose" ao definir funções não-determinísticas; isto é, ao definir o valor da função como sendo o máximo global dentre todos os ramos de computação, estamos dando à máquina quase tanto poder computacional quanto fornecer um oráculo para a sua própria classe de complexidade.

6 COMPARAÇÃO COM OUTROS TRABALHOS

Este capítulo discute brevemente outros trabalhos que, de certa forma, tentam capturar a noção de "função não-determinística".

6.1 PROBLEMAS DE BUSCA VS PROBLEMAS DE DECISÃO (PAPADIMITRIOU)

Papadimitriou (1994, p. 229) define FNP diferente de nossa definição (na seção 5.3).

Definição 6.1. Uma linguagem L é *polinomialmente equilibrada* se existir algum polinômio p tal que todos os elementos de L são pares ordenados da forma (x,y) em que $|y| \leq p(|x|)$.

Isto é, L é constituída de pares, de forma que o segundo elemento do par não é muito maior que o primeiro elemento.

Linguagens de NP e linguagens polinomialmente equilibradas estão relacionadas por certificados de pertinência. Por exemplo, para a linguagem SAT, podemos provar eficientemente (isto é, em tempo polinomial) que determinada instância φ é satisfazível se fornecermos uma atribuição de valores-verdade que satisfaz a instância; esta atribuição é uma *testemunha* ou *certificado* para a pertinência de φ a SAT.

Para cada linguagem $L \in \mathsf{NP}$, podemos sistematicamente prover certificados de pertinência para L: como sabemos que existe uma máquina de Turing não-determinística que decide L, podemos fornecer a sequência de transições não-determinísticas como um certificado de pertinência. Como esta máquina opera em tempo polinomial, para cada $x \in L$, o par

$$(x,y),$$

em que y é esta sequência de transições, satisfará $|y| \leq p(|x|)$, para algum polinômio p — no caso, p é o próprio limite de tempo da máquina não-determinística que reconhece L.

Dessa forma, podemos, sistematicamente, associar uma linguagem $L \in \mathsf{NP}$ a uma linguagem polinomialmente equilibrada $R_L \in \mathsf{P}$. Assim, construiremos o conjunto FNP definido por Papadimitriou (1994, p. 229).

Definição 6.2. Se $L \in \mathsf{NP}$, chame de R_L uma linguagem polinomialmente

equilibrada associada com L. Então defina FNP por

$$\mathsf{FNP} = \{R_L \mid L \in \mathsf{NP}\}$$

(PAPADIMITRIOU, 1994, p. 229)

Estes problemas são por vezes chamados de *problemas de busca* (do inglês *search problem*), em oposição a *problemas de decisão*.

Do ponto de vista computacional, a definição de Papadimitriou captura a ideia de encontrar alguma solução para o problema L, potencialmente descartando uma quantidade exponencial de outras soluções.

Esta definição contorna o problema de definir "função não-determinística" trabalhando diretamente com os certificados de pertinência. Entretanto, a ausência do conceito de função dificulta a formalização do conceito de oráculo (pois não há mais uma única resposta "certa", mas sim várias) e impossibilita a análise do efeito da composição de funções, como a feita na seção 5.7.

6.2 CONJUNTO POTÊNCIA COMO CONTRADOMÍNIO (ANDREEV)

Andreev (1994, p. 3) apresenta uma abordagem diferente. Em vez de tentar amarrar algum dos ramos de computação, a função retornará todos eles.

Definição 6.3. Sejam $\mathcal{A}$ e $\mathcal{B}$ conjuntos finitos, e $2^{\mathcal{B}}$ o conjunto potência de $\mathcal{B}$. Uma *função não-determinística de $\mathcal{A}$ em $\mathcal{B}$* é uma função da forma

$$f : \mathcal{A} \to 2^{\mathcal{B}}.$$

(ANDREEV, 1994, p. 3)

Os conjuntos $\mathcal{A}$ e $\mathcal{B}$ usados por Andreev (1994, p. 4) são da forma $\{0,1\}^k$. A ideia é que $\mathcal{A}$ represente as possíveis entradas para um circuito booleano, e que $\mathcal{B}$ represente as saídas desse circuito. Dado um circuito computacional S, com k entradas e l saídas, dizemos que S computa a função não-determinística F de $\{0,1\}^k$ em $\{0,1\}^l$ se, para todo $a \in \{0,1\}^k$, $S(a) \in F(a)$. Isto é, todas as saídas possíveis de S estão previstas em F. Esta definição é similar à definição de Papadimitriou (1994, p. 229), no sentido de que basta retornar um valor de $F(a)$. Andreev (1994, p. 4) define, então, a *complexidade* de uma função não-determinística F como sendo o tamanho do menor circuito que computa F.

O fato de Andreev se restringir a domínio e contradomínio finitos mostra que esta definição não "combina" corretamente a noção de função deter-

minística com máquinas não-determinísticas, pois as funções operam sobre o domínio $\{0,1\}^*$.

Além disso, como o domínio e o contradomínio das funções não-determinísticas de Andreev são diferentes, seria necessário um malabarismo adicional para desenvolver a noção de composição de funções, assim como na definição de Papadimitriou.

6.3 DEFINIÇÃO DE HOPCROFT E ULLMAN

A definição de Hopcroft e Ullman (1979, p. 313), mencionada na introdução, é a que motivou este trabalho.

Definição 6.4. Seja M uma máquina de Turing não-determinística. Dizemos que $M(x) = y$ se algum dos ramos de computação de M em x produz y, e nenhum outro ramo de computação que se encerre produz um valor diferente de y. (HOPCROFT; ULLMAN, 1979, p. 313)

O problema é que esta definição não enumera corretamente as funções recursivas. O grande vilão está na permissão que damos à máquina M não parar em todos os ramos de computação.

Por exemplo, construa uma máquina N que, na entrada $\langle M,x\rangle$, assuma dois ramos de computação: no primeiro, N incondicionalmente escreve 1 na fita de saída e para. No outro, N simula a máquina determinística M em x e, caso M pare ao computar x, N escreve 0 na fita.

Se M não para em x, então N possui um único ramo de computação que para, e este ramo produz 1 na fita; portanto, $N(\langle M,x\rangle) = 1$. Caso M pare ao computar x, $N(\langle M,x\rangle)$ estará indefinido, pois dois ramos de computação de N escrevem coisas diferentes na fita.

Ou seja, esta máquina computa

$$f(\langle M,x\rangle) = 1, \text{ se } M \text{ não parar ao computar } x,$$

que é exatamente o complemento do problema da parada. Portanto, pela definição de Hopcroft e Ullman, conseguimos enumerar funções não-recursivas.

Mas, se impormos à definição de Hopcroft e Ullman a restrição adicional de que todos os ramos de M devem parar para que $M(x)$ exista, então não há a necessidade de haver vários ramos de computação, pois todos eles retornam o mesmo resultado. Basta fixar uma das possíveis transições não-determinísticas, tornando a máquina *determinística*. Neste caso, perdemos o aparente ganho de tempo exponencial ao usar não-determinismo.

6.4 PROBLEMAS DE OTIMIZAÇÃO (KRENTEL)

Dentre os trabalhos avaliados, o artigo de Krentel (1988, p. 493) é o que mais se aproxima do trabalho desenvolvido neste TCC.

Krentel define a classe OptP, os problemas de otimização que rodam em tempo polinomial, como sendo, na terminologia da seção 5.3,

$$\mathsf{OptP} = \mathsf{FNP} \cup \mathsf{coFNP}.$$

Entretanto, a principal diferença é na forma como é definida completude.

Definição 6.5. Uma função $f \in \mathsf{OptP}$ é completa para OptP se, para toda função $g \in \mathsf{OptP}$, existir um par de funções $T_1, T_2 \in \mathsf{P}$ tais que

$$g(x) = T_2(x, f(T_1(x))).$$

Nossa definição de FNP-completo buscava generalizar diretamente a noção de NP-completude. Conforme observado na demonstração do teorema 5.22, podemos, em tempo polinomial determinístico, inverter a ordenação das palavras, se elas tiverem um tamanho fixo. Portanto, usando este pós-processamento, podemos tornar uma função que maximiza seus "valores não-determinísticos" (isto é, sua árvore de computação não-determinística) numa função que os minimiza. Como Krentel já embutiu os problemas de maximização e minimização em OptP, não há problema em permitir o pós-processamento feito por T_2.

Entretanto, graças a T_2, a classe OptP acaba "funcionando" como nossa classe Δ_n^f, pois, como observado no teorema 5.30, podemos resolver todos os problemas de Δ_n^f apenas pós-processando um resultado de FNP. Portanto não é possível construir uma "hierarquia polinomial funcional" com base em OptP.

7 CONSIDERAÇÕES FINAIS

Neste trabalho, desenvolvemos uma teoria de funções não-determinísticas, guiados pelo objetivo de poder definir sua complexidade em termos dos axiomas de Blum. Conseguimos generalizar a classe NP para problemas funcionais (a classe FNP) da mesma forma como a classe FP generaliza a classe P; problemas de NP, como SAT, possuem uma generalização natural em nossa classe FNP, como encontrar uma atribuição satisfazível, no caso de SAT.

O objetivo de manter compatibilidade com os axiomas de Blum foi atingido. É interessante notar que, dos trabalhos analisados neste capítulo, apenas Hopcroft se preocupa com os axiomas de Blum. Além disso, nenhum dos autores tenta interpretar as funções não-determinísticas como uma enumeração de Gödel, que é um passo necessário para que uma medida de complexidade satisfaça aos axiomas de Blum.

Também pudemos construir uma "hierarquia polinomial funcional", simplesmente adaptando as definições da hierarquia polinomial "tradicional". O fato de estarmos trabalhando com funções cujo domínio e contradomínio coincidem nos permitiu compô-las, possibilitando resultados como o teorema 5.28, que nos deu uma interpretação de Δ_n^f em termos da composição de funções de Σ_n^f.

Dentre os trabalhos analisados, o único que contém definições parecidas com o que foi desenvolvido neste TCC é o de (KRENTEL, 1988, p. 3), que foi encontrado apenas no final do desenvolvimento. Com isso, as referências bibliográficas acabaram todas "presas" no século passado. Seria interessante refazer a pesquisa bibliográfica, mas direcionando para trabalhos que lidem com problemas de otimização (como a classe OptP, de Krentel).

Os dois principais problemas que ficaram em aberto estão relacionados à hierarquia polinomial funcional desenvolvida. A demonstração do teorema 5.28 utilizou o problema completo

$$\mathsf{HaltFNP}^{\mathsf{FNP}^{\mathcal{A}}},$$

retirado do teorema 5.20. A formulação deste problema é baseada no problema da parada. Como observado por Papadimitriou (1994, p. 255), esse tipo de classe de complexidade[1] sempre possuirá um problema completo da

[1] As *classes de complexidade sintáticas* são aquelas em que a pertinência à classe depende apenas do modelo de máquina utilizado (PAPADIMITRIOU, 1994, p. 255). (Note que esta não é uma definição formal.) Por exemplo, classes como P, NP, FNP etc. são classes de complexidade sintáticas; enquanto que a classe dos problemas NP-completos não é — esta é uma classe

forma como é HaltFNP. Fica em aberto determinar se existem problemas mais naturais que sejam completos para os demais níveis da hierarquia polinomial funcional.

Outro problema é o mencionado no final do capítulo 5, que é a possibilidade de termos "exagerado na dose" ao definir o valor de $M(x)$ como sendo o máximo dos ramos de computação. Dessa forma, com um simples pós-processamento (conforme o teorema 5.30), podemos simular várias chamadas a um mesmo oráculo. Seria interessante tentar "enfraquecer" esta definição, não permitindo mais que uma única chamada dê informação equivalente a várias, criando uma hierarquia mais densa de problemas.

semântica.

REFERÊNCIAS

ANDREEV, A. E. Complexity of nondeterministic functions. **BRICS Report Series**, v. 47, 1994. Disponível em: <http://www.brics.dk/RS/94/2/BRICS-RS-94-2.pdf>.

ARORA, S.; BARAK, B. **Computational Complexity - A Modern Approach**. Cambridge University Press, 2009. ISBN 978-0-521-42426-4. Disponível em: <http://theory.cs.princeton.edu/complexity/book.pdf>. Acesso em: 13 mar. 2015.

BLUM, M. A machine-independent theory of the complexity of recursive functions. **J. ACM**, v. 14, n. 2, p. 322–336, 1967. Disponível em: <http://port70.net/~nsz/articles/classic/blum_complexity_1976.pdf>. Acesso em: 8 mar. 2015.

COOK, S. A. The complexity of theorem-proving procedures. In: **Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA**. [S.l.: s.n.], 1971. p. 151–158.

DAVIS, M. D.; SIGAL, R.; WEYUKER, E. J. **Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science**. second. San Diego, CA, USA: Academic Press Professional, Inc., 1994. ISBN 0-12-206382-1.

DU, D.-Z.; KO, K.-I. **Theory of Computational Complexity**. [S.l.]: Wiley, 2014. (Wiley Series in Discrete Mathematics and Optimization). ISBN 978-1118306086.

EPSTEIN, R. L.; CARNIELLI, W. A. **Computability: Computable Functions, Logic, and the Foundations of Mathematics**. Socorro, NM, USA: Advanced Reasoning Forum, 2008. 384 p. ISBN 9780981550732.

GAREY, M. R.; JOHNSON, D. S. **Computers and Intractability: A Guide to the Theory of NP-Completeness**. New York, NY, USA: W. H. Freeman & Co., 1979. ISBN 0716710447.

GASARCH, W. I. Guest column: the second P =? NP poll. **SIGACT News**, v. 43, n. 2, p. 53–77, 2012. Disponível em: <http://www.cs.umd.edu/~gasarch/papers/poll2012.pdf>. Acesso em: 8 mar. 2015.

HOPCROFT, J. E.; MOTWANI, R.; ULLMAN, J. D. **Introduction to automata theory, languages, and computation**. second edition. [S.l.]: Addison-Wesley-Longman, 2001. (Addison-Wesley series in computer science). ISBN 978-0-201-44124-6.

HOPCROFT, J. E.; ULLMAN, J. D. **Introduction to Automata Theory, Languages and Computation**. [S.l.]: Addison-Wesley, 1979. ISBN 0-201-02988-X.

KOZEN, D. **Theory of Computation**. [S.l.]: Springer, 2006. (Texts in Computer Science). ISBN 978-1-84628-297-3.

KRENTEL, M. W. The complexity of optimization problems. **Journal of Computer and System Sciences**, v. 36, n. 3, p. 490 – 509, 1988. ISSN 0022-0000.

LEWIS, H. R.; PAPADIMITRIOU, C. H. **Elements of the theory of computation**. [S.l.]: Prentice-Hall, 1998. ISBN 0-13-262478-8.

MCCREIGHT, E. M.; MEYER, A. R. da S. Classes of computable functions defined by bounds on computation: Preliminary report. In: **Proceedings of the First Annual ACM Symposium on Theory of Computing**. New York, NY, USA: ACM, 1969. (STOC '69), p. 79–88.

MEYER, A. R. da S.; STOCKMEYER, L. J. The equivalence problem for regular expressions with squaring requires exponential space. In: **Proceedings of the 13th Annual Symposium on Switching and Automata Theory (Swat 1972)**. Washington, DC, USA: IEEE Computer Society, 1972. (SWAT '72), p. 125–129. Disponível em: <http://people.csail.mit.edu/meyer/rsq.pdf>.

PAPADIMITRIOU, C. H. **Computational complexity**. [S.l.]: Addison-Wesley, 1994. ISBN 978-0-201-53082-7.

ROGERS JR., H. **Theory of recursive functions and effective computability (Reprint from 1967)**. Cambridge, MA, USA: MIT Press, 1987. ISBN 978-0-262-68052-3.

SIPSER, M. **Introduction to the Theory of Computation**. second edition. [S.l.]: Cenage Learning, 2006. ISBN 978-0-534-95097-3.

APÊNDICE A – MÁQUINAS DE TURING

Este apêndice contêm uma formalização de máquinas de Turing e seu uso como decisores e computadores de funções recursivas. O objetivo deste apêndice é prover uma fundamentação matemática para algumas afirmações feitas no texto, como, por exemplo, como uma máquina de Turing pode simular outra, e o porquê de as codificações identificarem unicamente uma função recursiva.

A.1 DEFINIÇÃO MATEMÁTICA

Definição A.1. Seja $D = \{\leftarrow, \rightarrow, \cdot\}$. Uma *máquina de Turing M* é uma quádrupla

$$M = (Q, \Gamma, \delta, q_0),$$

em que

- Q é um conjunto finito de estados.
- Γ é um conjunto finito de símbolos de fita, tais que $0, 1, \sqcup \in \Gamma$.
- $q_0 \in Q$ é o estado inicial.
- δ é uma função parcial definida sobre

  $$\delta : Q \times \{0, 1, \sqcup\} \times \Gamma \rightarrow Q \times D \times D \times \Gamma \times \{0, 1, \varepsilon\}.$$

  Uma tupla (q, a, b) no domínio representa os símbolos que a máquina está lendo das fitas, em que

  - $a \in \{0, 1, \sqcup\}$ é lido da fita somente-leitura de entrada.
  - $b \in \Gamma$ é lido da fita de trabalho.
  - e q é o estado atual.

  Uma tupla (q', d_1, d_2, e, s) no domínio representa a ação da máquina.

  - q' é o estado para o qual a máquina transitará.
  - $d_1 \in D$ é a direção do movimento do cabeçote de leitura.
  - $d_2 \in D$ é a direção do movimento do cabeçote de trabalho.
  - $e \in \Gamma$ é o que a máquina escreveu na fita de trabalho.
  - $s \in \{0, 1, \varepsilon\}$ é o que a máquina escreveu na fita de saída.

0 e 1 são os símbolo do alfabeto binário $\{0, 1\}$, que é o alfabeto sobre o qual são definidas as funções recursivas. $\sqcup$ é o caractere branco.

Esta definição interpreta máquinas como calculadores de funções. Em relação às definições usuais de "máquina de Turing" (encontrada, por exemplo, em (HOPCROFT; ULLMAN, 1979, p. 148), (SIPSER, 2006, p. 140), e (KOZEN, 2006, p. 6)), nossa definição (mais próxima da definição de Arora e Barak (2009, p. 12)) apresenta alguns símbolos "faltando".

- Omitimos Σ, que representa o alfabeto de entrada, pois usaremos sempre $\Sigma = \{0,1\}$.
- Os estados de aceitação e rejeição são desnecessários, pois a máquina não decide uma palavra, mas sim, calcula uma função.
- O símbolo $\sqcup$, o caractere branco, foi "embutido" diretamente em Γ.
- Não há marcadores de início/fim de fita.

Definição A.2. Uma *máquina de Turing não-determinística M* é uma quádrupla

$$M = (Q, \Gamma, \delta, q_0),$$

em que Q, Γ e q_0 possuem a mesma semântica das máquinas determinísticas, mas a imagem de δ, em vez de ser $Q \times D \times D \times \Gamma \times \{0,1,\varepsilon\}$, é o conjunto potência deste conjunto. Isto é, δ é definida sobre

$$\delta : \{0,1,\sqcup\} \times \Gamma \to 2^{Q \times D \times D \times \Gamma \times \{0,1,\varepsilon\}}.$$

A.1.1 Estados de computação

Os estados de computação de uma máquina de Turing representam a "descrição instantânea" de uma etapa do processo de computação. Eles agregam informações como o estado da máquina e das fitas. Nossa definição de "computação" será um pouco diferente da definição de Hopcroft e Ullman (1979, p. 149), visando simplificar a definição da função T da seção 5.4.

Definição A.3. Seja $M = (Q, \Gamma, \delta, q_0)$. Um *estado de computação E* de *M* é uma tupla

$$E = (x, y, z, q, m, n),$$

em que

- $x \in \{0,1\}^*$ é a fita de entrada.
- $y \in \Gamma^*$ é a fita de trabalho.
- $z \in \{0,1\}^*$ é fita de saída.

- $q \in Q$ é o estado atual da máquina.
- $m \in \mathbb{N}$ é a posição do cabeçote de leitura.
- $n \in \mathbb{N}$ é a posição do cabeçote de trabalho.

Construiremos agora o formalismo de transição de estados.

Por conveniência, denote por $w[i]$ a i-ésima letra da palavra w; indexaremos w em 1, portanto $w[1]$ é o primeiro símbolo de w e $w[|w|]$ é o último. Também identifique o conjunto $D = \{\leftarrow, \rightarrow, \cdot\}$ com o conjunto $\{-1, 1, 0\}$; isto é, $\leftarrow = -1$, $\rightarrow = 1$ e $\cdot = 0$. Assim, se $d \in D$ é um deslocamento e $n \in \mathbb{N}$ é uma posição do cabeçote, $n + d$ é o efeito de deslocar n por d.

A partir de um estado de computação $E = (x, y, z, q, m, n)$, para descobrir qual é a próxima ação da máquina $M = (Q, \Gamma, \delta, q_0)$, precisamos consultar a função δ. Precisamos passar a δ uma tupla (q, a, b), em que q é o próprio estado q de E, a é o que está debaixo do cabeçote da primeira fita e b é o que está debaixo do cabeçote da segunda fita (lembre-se de que não autorizaremos M a consultar a fita de saída). Se $1 \leq m \leq |x|$, então $a = x[m]$. Caso contrário, o cabeçote de leitura da máquina estará fora da cadeia de entrada. Interpretaremos a fita como sendo preenchida com brancos ($\sqcup$) onde não há outros caracteres; portanto, neste caso, $a = \sqcup$. Similarmente, se $1 \leq n \leq |y|$, $b = y[n]$; caso contrário, $b = \sqcup$. Portanto, a ação da máquina M no estado E será $\delta(q, a, b)$.

*Notação.*Denotaremos a ação da máquina $M = (Q, \Gamma, \delta, q_0)$ no estado E por $\delta(E)$.[1]

Note que, caso M for uma máquina não-determinística, $\delta(q, a, b)$ retornará um conjunto; portanto, $\delta(E)$ será uma lista de ações, não apenas uma.

Seja $E = (x, y, z, q, m, n)$ um estado de computação e $A = (q', d_1, d_2, e, s)$ uma ação da máquina de Turing (pode ser um valor retornado por δ se a máquina for determinística, ou um dos elementos do conjunto retornado por δ se a máquina for não-determinística). O *efeito de A em E* é a tupla (x', y', z', q', m', n'), dada por

- $x' = x$ (isto é, nunca alteraremos x).
- Se $n = 0$, $y' = y$ (a máquina não pode escrever no primeiro branco da fita de trabalho); se $n = |y| + 1$, $y' = ye$ (se a máquina estiver imediatamente à direita dos elementos da fita, estendemos a representação da fita com o caractere que a máquina escreveu); caso contrário, y' é y com o n-ésimo símbolo trocado por e — isto é, $y'[i] = y[i]$ se $i \neq n$, e $y'[n] = e$.

[1] Observe que estamos abusando da notação de chamada de função aqui.

- $z' = zs$. Se s for 0 ou 1, isso corresponde a escrever um destes caracteres na fita de saída. Se s for ε, isso corresponde a manter o cabeçote na fita de saída intacto.

- q' é o próprio q' de A.

- Se $m = 0$ e $d_1 = -1$ (quer dizer, $d_1 = \leftarrow$), então $m' = 0$; caso contrário, $m' = m + d_1$. Não deixaremos o cabeçote de leitura ler mais de um branco à esquerda do início, mas permitiremos que ele "navegue" livremente à direita.

- Se $n = 0$ e $d_2 = -1$, então $n' = 0$; senão, $n' = n + d_2$. Também não permitiremos o cabeçote de trabalho "navegar" demais à esquerda, portanto, efetivamente, a fita é apenas infinita à direita.

*Notação.*O efeito de A em E será denotado por $A(E)$.[2]

Usando estas duas noções, podemos definir como uma máquina de Turing computa sobre uma palavra.

Definição A.4. Seja $M = (Q, \Gamma, \delta, q_0)$ uma máquina de Turing determinística e $x \in \{0,1\}^*$ uma palavra. O k-ésimo estado de computação de M em x, denotado por $\mathbb{E}(M,x,k)$, é definido recursivamente por

$$\begin{aligned} \mathbb{E}(M,x,0) &= (x, \varepsilon, \varepsilon, q_0, 1, 1); \\ \mathbb{E}(M,x,k+1) &= A(\mathbb{E}(M,x,n)), \qquad \text{se } A = \delta(\mathbb{E}(M,x,k)). \end{aligned}$$

Se, para algum n, $\delta(\mathbb{E}(M,x,k))$ não existir — que é quando a tripla (q,a,b) não pertencer ao domínio de δ, na terminologia do parágrafo que define $\delta(E)$ — então defina $\mathbb{E}(M,x,k+1) = \mathbb{E}(M,x,k)$.

De acordo com a definição, quando a máquina para, todos os estados futuros serão os mesmos. De certa forma, o contrário também acontece; para que dois estados sejam iguais, nenhum dos cabeçotes da máquina podem se mexer (nem mesmo o de saída), e a máquina também não pode escrever na sua fita de trabalho um símbolo diferente do qual ela está lendo. Podemos sistematicamente remover todas as transições que causam este "travamento" de uma máquina de Turing, portanto não perdemos generalidade ao assumir que a máquina parou nestes casos. Então, podemos, finalmente, definir formalmente $M(x)$ e $t(M,x,k)$.

[2] Outro abuso da notação de chamada de função.

Definição A.5. Se $\mathbb{E}(M,x,k) = (x,y,z,q,m,n)$, defina $t(M,x,k) = z$. Se existir algum k_0 tal que

$$\mathbb{E}(M,x,k) = \mathbb{E}(M,x,k+1)$$

para todo $k \geq k_0$, defina $M(x) = t(M,x,k_0)$.

Para máquinas de Turing não-determinísticas, não há apenas uma possível sequência de computação, portanto, $\mathbb{E}$ deve ser um conjunto. Entretanto, isto cria um problema; pode ser que hajam dois estados E e F, no conjunto $\mathbb{E}(M,x,k)$, que se sucedem mutuamente; isto é, $A(E) = F$ para algum $A \in \delta(E)$, e $B(F) = E$ para algum $B \in \delta(F)$. Se E e F forem os únicos elementos de $\mathbb{E}(M,x,k)$ e A e B forem as únicas ações possíveis de M, então $\mathbb{E}(M,x,k) = \mathbb{E}(M,x,k+1)$, mesmo que M não tenha *parado* de computar em x.

Precisamos, portanto, ser mais cuidadosos ao definir transição e parada.

Definição A.6. Seja $M = (Q,\Gamma,\delta,q_0)$ uma máquina de Turing não-determinística e $x \in \{0,1\}^*$ uma palavra. $\mathbb{A}(M,x,k)$ é o k-ésimo conjunto de estados *ativos* de computação de M em x, e $\mathbb{T}(M,x,k)$ é o de estados *terminais* de computação. Estes dois conjuntos são mutuamente definidos recursivamente por

$$\begin{aligned}
\mathbb{A}(M,x,0) &= \{(x,\varepsilon,\varepsilon,q_0,1,1)\};\\
\mathbb{T}(M,x,0) &= \emptyset;\\
\mathbb{A}(M,x,k+1) &= \{A(E) \mid E \in \mathbb{A}(M,x,k) \wedge A \in \delta(E)\}\\
\mathbb{T}(M,x,k+1) &= \mathbb{T}(M,x,k) \cup \{E \in \mathbb{A}(M,x,k) \mid \delta(E) \text{ não está definido}\}.
\end{aligned}$$

Observe que $\mathbb{T}(M,x,k+1)$ necessariamente inclui $\mathbb{T}(M,x,k)$; portanto, $\mathbb{T}(M,x,k)$ conterá todos os ramos de computação que já se encerraram. Desta forma, a união

$$\mathbb{T}(M,x,k) \cup \mathbb{A}(M,x,k)$$

corresponde a $\mathbb{E}(M,x,k)$, do caso determinístico. Podemos, portanto, definir o símbolo $T(M,x,n)$ usando este conjunto.

Definição A.7.

$$T(M,x,n) = \{z \mid (x,y,z,q,m,n) \in \mathbb{A}(M,x,n) \cup \mathbb{T}(M,x,n)\}$$

Tecnicamente, o conjunto $\mathbb{A}(M,x,k)$ pode conter estados terminais de computação; para um estado terminal E, $\delta(E)$ não está definido, então $\mathbb{A}(M,x,k+1)$ não conterá estados da forma $A(E)$. Além disso, este estado

E estará em $\mathbb{T}(M,x,k+1)$; portanto, tecnicamente, $\mathbb{A}(M,x,k)$ contém os estados que estavam ativos no passo anterior, mas que não necessariamente estarão ativos agora.

Caso todos os ramos de computação de M em x eventualmente se encerrem, $\mathbb{A}(M,x,k)$ eventualmente ficará vazio; diremos, portanto, que M para ao computar x se $\mathbb{A}(M,x,k_0) = \emptyset$ para algum k_0.

Definição A.8. Seja M uma máquina de Turing não-determinística. Se $\mathbb{A}(M,x,k_0) = \emptyset$, defina $M(x)$ como sendo o maior valor (lexicograficamente) de $T(M,x,k_0)$.

(A ordenação lexicográfica usada neste texto considera que palavras menores precedem palavras maiores, independente de quais símbolos estejam envolvidos; por exemplo, $1 < 000$.)

Esta definição é a formalização da definição 5.1.

A.2 CODIFICAÇÃO DE MÁQUINAS DE TURING

O objetivo de definir uma codificação em binário para máquinas de Turing é poder enumerar as funções recursivas que elas computam e, de alguma forma, manipulá-las. Portanto, representaremos em nossa codificação apenas os aspectos "mais importantes" das máquinas de Turing.

Exclusivamente do ponto de vista de *funções*, a única coisa que importa é o resultado final: $M(x)$. Entretanto, existem várias maneiras de se computar uma função, e nós estamos usando medidas de complexidade para distinguir estas maneiras. Portanto, precisaremos ser um pouco mais conservadores antes de descartar alguma informação. Em particular, estaremos interessados em, de alguma forma, manter os estados de computação das máquinas.

Portanto, a quantidade de estados de uma máquina M (isto é, $|Q|$) é uma informação relevante; entretanto, *quais* estados são utilizados não é. Poderíamos assumir que Q sempre é um conjunto da forma

$$Q = \{1,2,\ldots,n\},$$

e que $q_0 = 1$; existe um mapeamento direto entre estados de computação para esta máquina e estados de computação da máquina original, de forma que a estrutura dos $\mathbb{E}$, $\mathbb{A}$ e $\mathbb{T}$ se mantenha.

Da mesma forma, exatamente quais símbolos estão em Γ não é relevante. Ficamos apenas com δ, que é a única informação essencial da quádrupla (Q,Γ,δ,q_0). É exatamente esta informação que codificaremos em $\langle M \rangle$. Nossa construção é similar à de Hopcroft e Ullman (1979, p. 182).

Definição A.9. Seja $M = (Q, \Gamma, \delta, q_0)$ uma máquina de Turing. Identificaremos os elementos do conjunto Q com os números $\{1, 2, \ldots, |Q|\}$, de forma que q_0 seja identificado com 1. Da mesma forma, identificaremos os elementos de Γ com o conjunto $\{1, 2, \ldots, |\Gamma|\}$ de forma que o símbolo 0 corresponda ao número 1, o símbolo 1 corresponda ao número 2, e o símbolo $\sqcup$ corresponda ao número 3. Por fim, identifique D e $\{0, 1, \varepsilon\}$ com $\{1, 2, 3\}$. Codifique a transição

$$\delta(q, a, b) = (q', d_1, d_2, e, s)$$

por

$$0^q 1 0^a 1 0^b 1 0^{q'} 1 0^{d_1} 0^{d_2} 1 0^e 1 0^s.$$

(Nós identificamos os conjuntos Q, Γ etc. com subconjuntos de $\mathbb{N}$ para que a notação 0^q faça sentido. Além disso, escolhemos os valores como começando em 1 para que sempre haja ao menos um dígito 0 entre cada par de 1's.)

Agora, defina

$$\langle M \rangle = 11 c_1 11 c_2 11 \ldots 11 c_n 11,$$

em que cada c_i é uma codificação para uma transição, e todas as transições são codificadas em algum c_i. Se a máquina de Turing for não-determinística, basta usar vários c_i com o mesmo (q, a, b).

Rigorosamente falando, usar o símbolo de igualdade na definição de $\langle M \rangle$ é um abuso de notação, pois o lado direito pode se apresentar em várias ordens. Entretanto, qualquer que seja a ordem, a máquina representada é a mesma, portanto este abuso de notação (e nomenclatura) é tolerável.

Dessa forma, definimos parte do mapeamento $w \mapsto f_w$, pois $f_{\langle M \rangle}$ sempre representa a função que M computa (para as várias representações $\langle M \rangle$ de M). Entretanto, existem cadeias que não codificam máquina alguma, de acordo com a definição acima; para que o mapeamento fique completo, para estas cadeias w, defina f_w como a função que não está definida para w algum.[3]

A.2.1 Simulação eficiente de máquinas de Turing

Em vários pontos do texto, é usado o fato de que uma máquina de Turing pode simular outra com apenas uma perda polinomial no tempo de execução. Como estamos ignorando as diferenças polinomiais entre os algoritmos, esta perda de desempenho é aceitável, e "merece" o título de "eficiente".

Para fazer esta simulação eficiente, a máquina que está simulando terá

[3] Observe que fixar f_w, nestes casos, para alguma função recursiva arbitrária também funcionaria.

de manter, na fita de trabalho, uma cópia da fita de entrada da máquina simulada, a fita de trabalho da máquina simulada, e um espaço adicional para gravar o estado atual e o que está por baixo dos cabeçotes da máquina simulada.

Se organizado da forma

$$q \llcorner a \llcorner b \llcorner x \llcorner y,$$

com os três primeiros espaços tendo o tamanho máximo de um estado de Q, de um símbolo de $\{0,1\}$, e de um símbolo de Γ, respectivamente, a estrutura da fita fica fixa, permitindo que o y cresça arbitrariamente.

Observe que esta informação pode ter vindo de uma codificação $\langle M \rangle$; portanto, a máquina simulada pode ter mais símbolos de fita do que a máquina simuladora, então a máquina simuladora precisará usar codificações. Os símbolos q e b podem ter tamanho variado, mas existe um valor máximo possível; portanto, basta "alocar" esse espaço máximo no início da simulação. $x \in \{0,1\}^*$, portanto é fácil de lidar com ele — apenas coloque alguma marcação para saber onde está o cabeçote de leitura da máquina simulada. y volta a causar problemas; ele pode ser da forma

$$0^{i_1}1^{n-i_1}2 \;\; 0^{i_2}1^{n-i_1}2 \cdots 2 \;\; 0^{i_k}1^{n-i_k},$$

em que n é a quantidade de símbolos de fita. Assim, todas as subcadeias $0^{i_j}1^{n-i_j}$ possuem tamanho fixo. Precisaremos marcar aqui também qual é a posição do cabeçote de trabalho.

Para avançar o estado de computação, a máquina inicialmente descobre qual transição será usada, atualizando a e b e varrendo $\langle M \rangle$. Isso pode ser feito em tempo proporcional a $n * m$, em que m é o tamanho dos símbolos codificados e m é o tamanho da fita no momento. Depois, em mais um passo de custo proporcional a $n * m$, podemos atualizar as posições dos cabeçotes, atualizar o estado atual, e substituir a célula da fita sob o cabeçote de trabalho. Se, neste passo, M decidir escrever algo na fita de saída, basta utilizar a própria fita de saída da máquina simuladora. Portanto, podemos efetuar um passo de computação em tempo $O(n * m)$.

(Em alguns casos, precisaremos utilizar o valor de saída de M, para pós-processamento, ou usar como entrada em outra máquina. Neste caso, adicione mais uma trilha à fita de trabalho da máquina simuladora; quer dizer, o Γ da máquina simuladora terá pares de símbolos. O símbolo "de cima" é a fita que estávamos usando até agora, e o símbolo "de baixo" armazenará a saída de M. Observe que isso não aumenta a complexidade assintótica do algoritmo.)

Se a máquina M realizar k passos de computação antes de parar, o

valor de m será menor ou igual a k. Portanto, podemos simular M em x em tempo $O(k^3)$, sendo que a constante "oculta" pela notação O depende apenas da máquina simuladora, não da máquina simulada.

Hopcroft e Ullman (1979, p. 292) mostram que, se pudermos usar duas fitas para fazer a simulação, ela pode ser feita de forma consideravelmente mais eficiente: $O(k \ln k)$, mesmo que a máquina simulada use mais fitas e mais símbolos de fita do que a máquina simuladora. Entretanto, este desaceleramento cúbico será suficiente para nossos propósitos.