Complexidade Computacional e o Problema P vs NP

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Igor Carboni Oliveira e aprovada pela Banca Examinadora.

Campinas, 10 de agosto de 2010.

Prof. Dr. Arnaldo Vieira Moura (Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Bibliotecária: Silvania Renata de Jesus Ribeiro Cirilo – CRB8 / 6592

Oliveira, Igor Carboni

Ol4c Complexidade computacional e o problema P vs NP/Igor Carboni Oliveira-- Campinas, [S.P. : s.n.], 2010.

Orientador : Arnaldo Vieira Moura

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Computação Científica.

1. Complexidade computacional. 2. Diagonalização. 3. Algoritmos.. I. Moura, Arnaldo Vieira. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Título em inglês: Computational complexity and the P vs NP problem.

Palavras-chave em inglês (Keywords): 1. Computational complexity. 2. Diagonalization. 3. Algorithms.

Área de concentração: Teoria da Computação.

Titulação: Mestre em Ciência da Computação.

Banca examinadora: Prof. Dr. Arnaldo Vieira Moura – (IC-UNICAMP)
Prof. Dr. Flávio Keidi Miyazawa – (IC-UNICAMP)
Prof. Dr. José Coelho de Pina – (IME-USP)
Prof. Dr. Julio César Lopez Hernandéz – (IC-UNICAMP)
Profa. Dra. Ana Cristina Vieira de Melo - (IME-USP)

Data da defesa: 02/08/2010

Programa de Pós-Graduação: Mestrado em Ciência da Computação.

TERMO DE APROVAÇÃO

Dissertação Defendida e Aprovada em 02 de agosto de 2010, pela Banca examinadora composta pelos Professores Doutores:

Prof. Dr. José Coelho de Pina Júnior
IME / USP

Prof. Dr. Flávio Keidi Miyazawa
IC / UNICAMP

Prof. Dr. Arnaldo Vieira Moura
IC / UNICAMP

Instituto de Computação
Universidade Estadual de Campinas

Complexidade Computacional e o Problema P vs NP

Igor Carboni Oliveira[1]

Agosto de 2010

Banca Examinadora:

- Prof. Dr. Arnaldo Vieira Moura (Orientador)
- Prof. Dr. Flávio K. Miyazawa
  Instituto de Computação - UNICAMP
- Prof. Dr. José Coelho de Pina
  Instituto de Matemática e Estatística - USP
- Prof. Dr. Julio C. López Hernández (Suplente)
  Instituto de Computação - UNICAMP
- Profa. Dra. Ana Cristina Vieira de Melo (Suplente)
  Instituto de Matemática e Estatística - USP

[1]Suporte financeiro da Fundação de Amparo à Pesquisa do Estado de São Paulo (2008/07040-0).

Resumo

A teoria de complexidade computacional procura estabelecer limites para a eficiência dos algoritmos, investigando a dificuldade inerente dos problemas computacionais. O problema P vs NP é uma questão central em complexidade computacional. Informalmente, ele procura determinar se, para uma classe importante de problemas computacionais, a busca exaustiva por soluções é essencialmente a melhor alternativa algorítmica possível.

Esta dissertação oferece tanto uma introdução clássica ao tema, quanto uma exposição a diversos teoremas mais avançados, resultados recentes e problemas em aberto. Em particular, o método da diagonalização é discutido em profundidade.

Os principais resultados obtidos por diagonalização são os teoremas de hierarquia de tempo e de espaço (Hartmanis e Stearns [54, 104]). Apresentamos uma generalização desses resultados, obtendo como corolários os teoremas clássicos provados por Hartmanis e Stearns. Essa é a primeira vez que uma prova unificada desses resultados aparece na literatura.

Abstract

Computational complexity theory is the field of theoretical computer science that aims to establish limits on the efficiency of algorithms. The main open question in computational complexity is the P vs NP problem. Intuitively, it states that, for several important computational problems, there is no algorithm that performs better than a trivial exhaustive search.

We present here an introduction to the subject, followed by more recent and advanced results. In particular, the diagonalization method is discussed in detail. Although it is a classical technique in computational complexity, it is the only method that was able to separate strong complexity classes so far.

Some of the most important results in computational complexity theory have been proven by diagonalization. In particular, Hartmanis and Stearns [54, 104] proved that, given more resources, one can solve more computational problems. These results are known as hierarchy theorems. We present a generalization of the deterministic hierarchy theorems, recovering the classical results proved by Hartmanis and Stearns as corollaries. This is the first time that such unified treatment is presented in the literature.

Prefácio

A complexidade computacional é uma disciplina fundamental para a ciência da computação. Seus resultados são elegantes, profundos, e muitas vezes imprevisíveis. Espero, sinceramente, que parte do meu fascínio e interesse por essa disciplina seja transmitido ao leitor.

Esta dissertação de mestrado foi escrita para ser usada por estudantes interessados em aprender complexidade computacional. Existem excelentes livros [7, 47, 88, 103] sobre o tema na literatura. Meu objetivo foi escrever um texto em português para ser usado de forma complementar a essas obras. Por isso, certos tópicos relevantes que são muito bem abordados nesses livros foram omitidos. Procurei destacar os métodos mais importantes e apresentar alguns resultados avançados que não são discutidos em profundidade nesses textos.

O único pré-requisito para leitura da dissertação é um pouco de maturidade matemática, embora uma exposição anterior a um curso de projeto e análise de algoritmos seja útil. Para facilitar a leitura, a maioria das demonstrações são feitas em detalhes.

Segue uma breve descrição de cada capítulo da dissertação.

Capítulo 1: Introdução. Neste capítulo introduzimos os principais conceitos utilizados em complexidade computacional. Após uma breve discussão sobre os objetivos dessa disciplina, definimos o modelo computacional das Máquinas de Turing. Essas máquinas formalizam a noção de algoritmo utilizada em complexidade computacional. A seguir, discutimos como medir a complexidade computacional de um algoritmo. Finalmente, mostramos como provar um limitante inferior envolvendo um tipo de máquina de Turing um pouco menos eficiente.

Capítulo 2: Introdução ao Problema P vs NP. Neste capítulo abordamos o principal problema em aberto da teoria de complexidade computacional: a relação entre as classes de complexidade P e NP. Informalmente, o problema P vs NP procura determinar se, para uma classe importante de problemas computacionais, a busca exaustiva por soluções é essencialmente a melhor alternativa algorítmica possível. Apresentamos diversas

formulações equivalentes para esse problema, além de discutirmos sua importância e os principais métodos empregados na tentativa de resolvê-lo.

Capítulo 3: Simulação e Diagonalização. Intuitivamente, esperamos que com mais recursos computacionais seja possível resolver mais problemas. Os teoremas de hierarquia de tempo e de espaço, alguns dos resultados mais importantes provados em complexidade computacional, estabelecem exatamente isso. O argumento utilizado na prova desses teoremas é conhecido como método da diagonalização. Neste capítulo vamos estudar essa técnica em profundidade. Veremos também como generalizar e unificar a demonstração dos teoremas de hierarquia e de outros resultados importantes em complexidade computacional.

Capítulo 4: O Problema P vs NP em Profundidade. Neste capítulo vamos discutir alguns tópicos mais avançados relacionados com o problema P vs NP. Inicialmente, veremos como a hierarquia polinomial generaliza a definição das classes de complexidade P e NP. Vamos mostrar também que algoritmos muito eficientes em tempo e espaço não são capazes de decidir a linguagem SAT. Discutiremos em seguida algumas propriedades estruturais das linguagens em NP. Por último, demonstraremos que existem algoritmos assintoticamente ótimos para todos os problemas da classe NP.

Capítulo 5: Os Limites da Diagonalização. Veremos neste capítulo que alguns métodos discutidos nesta dissertação não são capazes de resolver o problema P vs NP. Discutiremos em seguida como essa limitação se relaciona com resultados de independência formal em matemática. Além disso, vamos estudar o comportamento de diversos problemas em aberto da complexidade computacional em universos computacionais alternativos. Finalmente, discutiremos como métodos mais modernos superam a limitação enfrentada por algumas das técnicas estudadas anteriormente.

Ressalvo que muitos resultados importantes envolvendo o problema P vs NP não estão presentes. A complexidade computacional é uma área extremamente ativa em teoria da computação. Dado o meu conhecimento atual sobre o tema e o tempo disponível, seria impossível abordar com profundidade todos os desenvolvimentos recentes.

Finalmente, qualquer crítica ou sugestão será muito bem-vinda. Torço para que no futuro mais estudantes e pesquisadores brasileiros se interessem pelos problemas e desafios da teoria de complexidade computacional. Além disso, espero que o meu esforço tenha sido suficiente para que os resultados apresentados nesta dissertação possam ser entendidos em tempo polinomial no tamanho de cada demonstração.

Agradecimentos

Aos meus pais, Carlos Magno de Oliveira e Helena Maria Carboni Oliveira, à minha irmã, Iana Carboni Oliveira, e ao meu amor, Nayara Fonseca de Sá, por tudo de especial que representam na minha vida.

Ao meu orientador, professor Arnaldo Veira Moura, por ter me dado a oportunidade e liberdade para estudar diversos tópicos do meu interesse, por tantos conselhos sábios oferecidos durante o mestrado, e por ter insistido constantemente para que eu melhorasse o texto final desta dissertação.

Aos professores Walter Carnielli e Orlando Lee, meus orientadores durante a graduação, por toda ajuda que me ofereceram naquele período.

Aos meus amigos dos primeiros anos de graduação, agora físicos e matemáticos, por terem despertado meu interesse pelos aspectos mais teóricos e fundamentais da ciência.

Ao professor Cid Carvalho de Souza, por ter despertado meu interesse por complexidade computacional através de suas aulas de projeto e análise de algoritmos.

Ao Anderson de Araújo, por ter revisado os primeiros capítulos da dissertação.

Aos funcionários do Instituto de Computação, por toda assistência prestada durante os meus anos na Unicamp.

À FAPESP, pelo suporte financeiro oferecido durante o mestrado.

Sumário

Resumo **v**

Abstract **vi**

Prefácio **vii**

Agradecimentos **ix**

1 Introdução **1**
1.1 Motivação 1
1.2 Computabilidade vs Complexidade 3
1.3 Complexidade Computacional 4
1.4 O Problema P vs NP 4
1.5 Terminologia Básica e Notação 5
1.6 Máquinas de Turing 6
1.7 Eficiência dos Algoritmos 10
1.8 Um Exemplo de Limitante Inferior 11
1.9 Referências Adicionais 14

2 Introdução ao Problema P vs NP **17**
2.1 Introdução 17
2.2 Definição 20
2.3 Decisão vs Busca 24
2.4 Uma Formulação Alternativa 25
2.5 Reduções e Problemas Completos 27
2.6 Importância Matemática 31
2.7 Principais Abordagens 32
2.8 Resultados Básicos 35
2.9 Referências Adicionais 37

3 Simulação e Diagonalização **39**
3.1 Introdução . . . 39
3.2 Um Teorema Geral de Hierarquia . . . 44
3.3 Consequências do Resultado Anterior . . . 49
3.4 O Teorema da Lacuna . . . 51
3.5 A Hierarquia Não-Determinística . . . 53
3.6 O Método da Completude . . . 55
3.7 Referências Adicionais . . . 57

4 O Problema P vs NP em Profundidade **59**
4.1 A Hierarquia Polinomial . . . 59
4.2 Alternância . . . 66
4.3 Cotas Inferiores para SAT . . . 70
4.4 Reduções e NP $\cap$ coNP . . . 74
4.5 O Teorema de Ladner . . . 77
4.6 NP-Completude, Isomorfismo e Densidade . . . 81
4.7 Algoritmos Ótimos para NP . . . 86
4.8 Referências Adicionais . . . 88

5 Os Limites da Diagonalização **91**
5.1 Introdução . . . 91
5.2 Uma Barreira Fundamental . . . 94
5.3 P vs NP e Resultados de Independência . . . 96
5.4 Resultados Adicionais . . . 99
5.5 Oráculos Aleatórios . . . 100
5.6 Relativização Positiva . . . 102
5.7 O Argumento Contrário de Kozen . . . 103
5.8 É Possível Superar a Relativização? . . . 104
5.9 Referências Adicionais . . . 106

6 Conclusão **107**

Bibliografia **108**

Capítulo 1

Introdução

"The most beautiful thing we can experience is the mysterious.
It is the source of all true art and science."
Albert Einstein.

Neste capítulo introduzimos os principais conceitos utilizados em complexidade computacional. Após uma breve discussão sobre os objetivos dessa disciplina, definimos o modelo computacional das Máquinas de Turing. Essas máquinas formalizam a noção de algoritmo utilizada em complexidade computacional. A seguir, discutimos como medir a complexidade computacional de um algoritmo. Finalmente, mostramos como provar um limitante inferior envolvendo um tipo de máquina de Turing um pouco menos eficiente.

1.1 Motivação

Esta dissertação de mestrado versa sobre uma das questões mais profundas investigadas pela ciência moderna. Antes de discutirmos a importância do problema que vamos estudar, convidamos o leitor a refletir sobre a seguinte questão:

Qual é o objetivo da ciência da computação?

De maneira geral, pode-se dizer que a ciência da computação procura entender as limitações e oportunidades oferecidas pelo nosso universo em relação a nossa capacidade de processar, armazenar e transmitir informações. Ainda que possa parecer ampla demais, essa definição ilustra razoavelmente bem a direção das pesquisas em ciência da compu-

tação ao longo das décadas. Desde o advento dos primeiros computadores até a época das pesquisas em computação quântica, tentamos entender e tirar vantagem do mundo computacional ao nosso redor.

Embora a visão quântica do mundo traga consigo uma promessa de novas possiblidades computacionais, diversos problemas fundamentais persistem no universo da computação clássica, sendo que o mais importante deles é o problema P vs NP.

De modo fascinante, durante o século passado diversos pesquisadores mostraram que existem barreiras fundamentais limitando a nossa capacidade de processar informações. Por exemplo, demonstrou-se que existem problemas computacionais extremamente difíceis. Em outras palavras, mesmo que pudéssemos utilizar em conjunto os melhores computadores atuais durante todo o tempo de vida do universo, não seríamos capazes de encontrar soluções para muitos problemas, embora elas existam.

A teoria de complexidade computacional procura classificar os problemas computacionais de acordo com o seu grau de dificuldade. É uma disciplina relativamente recente, com muitos problemas em aberto e alguns resultados extraordinários. O principal desafio para os pesquisadores dessa disciplina é o desenvolvimento de métodos matemáticos que possam ser usados para provar que certos problemas computacionais são inerentemente difíceis.

O problema P vs NP é uma questão central em complexidade computacional. Informalmente, ele procura determinar se, para uma classe importante de problemas computacionais, a busca exaustiva por soluções é essencialmente a melhor alternativa algorítmica possível. Infelizmente, essa não é uma solução viável para milhares de problemas presentes na física, biologica, matemática e outras disciplinas, mas é o melhor que sabemos fazer. Apesar de ter sido intensamente estudado por décadas, o problema permanece em aberto. Aparentemente, os métodos atuais utilizados em matemática e ciência da computação não são suficientes para resolver o problema.

Sem dúvida alguma, a solução do problema P vs NP será um grande passo para a ciência da computação. É razoável supor que toda civilização avançada questionou em algum momento a sua habilidade de computar, de transformar informação em conhecimento. É fascinante presenciar o momento em que nós adquirimos conhecimento suficiente para formular uma questão tão fundamental como essa. Encontrar a resposta para o problema P vs NP é um desafio a ser vencido pela ciência moderna, fruto da razão e inteligência humanas. Independentemente da resposta, sua solução terá necessariamente que revolucionar a matemática e a ciência da computação dos dias de hoje.

1.2 Computabilidade vs Complexidade

Apesar da noção de algoritmo ser utilizada há pelo menos dois mil anos, apenas no século vinte o conceito de computação foi formalmente definido e estudado. Isso ocorreu principalmente devido a suspeita dos pesquisadores da época de que certos problemas computacionais poderiam ser indecidíveis por meio de computação.

Para provar que um problema pode ser resolvido por um algoritmo, basta dar uma descrição detalhada de um procedimento finito capaz de resolver o problema computacional em consideração. Porém, para argumentar que determinada tarefa não pode ser resolvida por meio de computação, torna-se necessária a definição formal do conceito de algoritmo. Somente assim podemos provar que nenhum procedimento satisfazendo a definição de algoritmo é capaz de realizar a tarefa desejada.

Portanto, a primeira grande questão enfrentada pelos pesquisadores foi encontrar uma definição satisfatória para esse conceito. De modo fascinante, verificou-se que todas as definições propostas para a noção de computação eram equivalentes, ou seja, os diversos modelos computacionais sugeridos resolviam o mesmo conjunto de problemas. Devido a esse fato, a noção de computação se tornou um conceito científico robusto e fundamental.

A partir da definição formal de algoritmo, amplamente aceita, os pesquisadores obtiveram sucesso em provar que existem problemas bem definidos que não podem ser resolvidas por meios algorítmicos. A única maneira de resolver esses problemas é abandonando algum requisito fundamental e indiscutível da definição de algoritmo, como o fato de um programa de computador sempre terminar a sua computação após um número finito de passos, ou possuir uma descrição finita. A teoria de computabilidade tem como objetivo principal determinar quais problemas são computáveis, ou seja, podem ser resolvidos através de algoritmos.

A existência de problemas indecidíveis trouxe consequências extraordinárias. Por exemplo, a partir da indecidibilidade do problema da parada é possível demonstrar que a matemática é uma ciência necessariamente incompleta. Isso significa que em qualquer axiomatização da matemática é possível encontrar sentenças verdadeiras que não podem ser demonstradas a partir das regras e axiomas do sistema formal. Veja a seção de referências desta seção para mais detalhes.

A teoria de computabilidade fornece métodos para classificarmos quais problemas são solucionáveis por algoritmos. Por outro lado, a teoria de complexidade pode ser vista como uma continuação dessa disciplina, uma vez que ela particiona os problemas que podem ser resolvidos por algoritmos em diversas classes, de acordo com a quantidade de recursos computacionais necessários e suficientes para resolver cada problema. Embora a teoria de computabilidade seja uma disciplina fascinante, ela não será discutida nesta dissertação. Recomendamos que o leitor interessado em computabilidade procure alguma

referência indicada na seção final deste capítulo.

1.3 Complexidade Computacional

A complexidade computacional é a área da ciência da computação que procura determinar por quais motivos certos problemas decidíveis são tão difíceis de serem resolvidos por computadores. Essa disciplina, praticamente inexistente há quarenta anos, expandiu-se tremendamente e atualmente é responsável por boa parte das atividades de pesquisa em teoria da computação.

Desde a década de sessenta, quando o uso de computadores deixou de ser restrito a poucas instituições científicas, os programadores perceberam que a existência de um algoritmo (programa) para uma tarefa não era suficiente para que ela pudesse ser resolvida por um computador. Foram descobertos muitos problemas práticos para os quais os melhores algoritmos conhecidos demoravam tanto tempo para executar que inviabilizava completamente a busca de respostas para o problema através de meios computacionas. A grande questão era se essas observações eram uma consequência da nossa incapacidade de encontrar um algoritmo mais eficiente para o problema ou se resultavam de dificuldade inerente do problema em consideração.

Essa situação levou à idéia de se medir a dificuldade de determinadas tarefas com respeito à quantidade de recursos computacionais necessários e suficientes para computá-las. Isso, por sua vez, levou à classificação dos problemas que podem ser resolvidos algoritmicamente de acordo com o seu grau de dificuldade.

Os principais objetivos da teoria que estudaremos são, portanto: estimar os recursos computacionais necessários e suficientes para solucionar problemas algorítmicos concretos; identificar e definir a noção de "eficiência" ou "tratabilidade" dos problemas computacionais; desenvolver métodos para classificá-los dentro de diversas classes de complexidade; comparar a eficiência de diversos modelos computacionais distintos.

1.4 O Problema P vs NP

A pesquisa em complexidade computacional pode ser dividida em dois grandes grupos. Pode-se estudar a dificuldade de um problema computacional específico, ou investigar como certos recursos computacionais e classes de problemas estão relacionados. Apesar dessa divisão, o estudo de classes de problemas pode muitas vezes ser reduzido ao estudo de problemas individuais (completos) que capturam propriedades essenciais da classe.

O problema P vs NP insere-se exatamente nesse contexto. Denota-se por P o conjunto de problemas computacionais com soluções que podem ser encontradas de forma eficiente.

Por sua vez, a classe NP contém os problemas computacionais com soluções que podem ser verificadas de forma eficiente. Intuitivamente, o problema P vs NP pergunta como essas duas classes estão relacionadas, ou seja, se para todo problema cujas soluções são eficientemente checáveis também existe um algoritmo eficiente capaz de encontrar tais soluções. Apesar de ser uma questão envolvendo classes de problemas, a existência de problemas completos para a classe NP torna interessante o estudo de alguns problemas computacionais específicos.

Dado um problema computacional pertencente à classe NP, existe um algoritmo muito simples capaz de resolvê-lo. Ele procede da seguinte forma: gera todas as respostas possíveis para uma dada instância do problema e então aplica um algoritmo capaz de checar essas respostas (a existência desse último é garantida pela definição de NP). Uma desvantagem dessa abordagem é que o algoritmo final não é eficiente. Enunciado de outra forma, o problema P vs NP procura responder se essa busca exaustiva por soluções pode ser sempre evitada. Embora a maioria dos pesquisadores acredite que a busca exaustiva seja essencial, ninguém até agora foi capaz de exibir uma prova matemática desse fato.

O problema P vs NP é de fundamental importância na criptografia moderna. A segurança da Internet e da maioria das transações financeiras depende de algumas hipóteses, tais como a dificuldade inerente de se fatorar números inteiros muito grandes. Se P = NP, essas hipóteses são falsas e a segurança de tais transações estaria comprometida.

A existência de um algoritmo eficiente para algum problema NP-completo (veja a seção 2.5) teria consequências fantásticas não só para a ciência da computação, mas também para muitas outras áreas de natureza distinta. Por exemplo, isso transformaria a matemática contemporânea, ao permitir que um computador encontrasse provas formais de diversos teoremas que possuam uma demonstração de tamanho razoável (veja a seção 2.6). Essas considerações valem também para diversos outros trabalhos criativos que atualmente listamos como inerentemente humanos. Esse é um problema fundamental na inteligência artificial, cuja solução poderia ser facilitada se tivéssemos algoritmos práticos para problemas NP-completos.

1.5 Terminologia Básica e Notação

A notação e os conceitos básicos utilizados no texto são usuais em teoria da computação e podem ser encontrados nos livros de Arora e Barak [7] e Sipser [103], por exemplo. Por conveniência, apresentamos nesta seção as principais convenções adotadas.

O símbolo $\mathbb{N}$ denota o conjunto dos números naturais. Se Σ é um conjunto finito, então uma *palavra* sobre o *alfabeto* Σ é uma sequência finita de elementos de Σ. Denotamos por $\Sigma^\star$ o conjunto de todas as palavras sobre o alfabeto Σ. O tamanho de uma palavra w será representado por $|w|$. Uma palavra w de tamanho k é um palíndromo se e somente se

$w = w_1 w_2 \ldots w_k = w_k w_{k-1} \ldots w_1$. O símbolo ϵ detona a palavra vazia. A concatenação de duas palavras w_1 e w_2 será denotada por $w_1 w_2$. Quando o alfabeto não for especificado, assuma que $\Sigma = \{0, 1\}$.

Dado um objeto x, utilizaremos a notação $\langle x \rangle$ para representar esse objeto como uma palavra. Essa notação será usada para codificar inteiros, matrizes, vetores, etc. Em particular, utilizaremos a notação $\langle x, y \rangle$ para a palavra que representa o par formado pelos objetos x e y, e similarmente para uma quantidade maior de objetos. É fácil perceber que existem representações convenientes para todas as estruturas discutidas no texto através do uso de palavras binárias. Se z é um vetor de elementos, seu i-ésimo elemento será representado por z_i.

Através da representação de objetos arbitrários por palavras poderemos identificar qualquer função $f : A \to B$ com domínio ou imagem que não sejam palavras pela função correspondente que utiliza a representação por palavras dos objetos em A e B. A maior parte das funções discutidas neste texto terão como imagem 0 ou 1.

Quando a imagem de uma função f for um único bit, vamos identificá-la pelo conjunto $L_f = \{x : f(x) = 1\} \subseteq \{0, 1\}^\star$. Todo subconjunto de $\Sigma^\star$ é chamado de *linguagem* e identificamos o problema computacional de computar f como sendo o problema de decidir a linguagem L_f (dado x, verificar se $x \in L_f$). Se L é uma linguagem sobre o alfabeto Σ, então $\overline{L} = \Sigma^\star \backslash L$, ou seja, o complemento de L. Uma linguagem L é dita não-trivial quando $L \neq \emptyset$ e $L \neq \Sigma^\star$.

Denotaremos por $\log n$ o logaritmo do número inteiro n na base 2. Dizemos que uma propriedade $P(n)$ definida sobre o conjunto dos números naturais é válida para n suficientemente grande se existe algum número inteiro N tal que $P(n)$ vale para todo $n \geq N$. Se $f, g : \mathbb{N} \to \mathbb{N}$ são funções, então dizemos que $f = O(g)$ se existe uma constante c tal que $f(n) \leq cg(n)$ para n suficientemente grande. Definimos que $f = \Omega(g)$ se $g = O(f)$, e escrevemos $f = \Theta(g)$ quando temos $f = O(g)$ e $g = O(f)$. Finalmente, dizemos que $f = o(g)$ se para todo $\varepsilon > 0$, $f(n) \leq \varepsilon g(n)$ para n suficientemente grande, e escrevemos $f = \omega(g)$ quando $g = o(f)$.

Por último, o nome de algumas classes de complexidade e de alguns conceitos importantes foram mantidos em inglês.

1.6 Máquinas de Turing

É um fato surpreendente a existência de um modelo computacional aparentemente capaz de simular todos os modelos computacionais fisicamente implementáveis. Nesta seção apresentamos uma definição informal das máquinas de Turing, um modelo amplamente utilizado em complexidade computacional. Para uma apresentação formal e uma discussão abrangente sobre o papel das máquinas de Turing em computação, recomendamos a

leitura de Sipser [103].

Um algoritmo é uma sequência finita de passos elementares capaz de realizar uma determinada tarefa computacional. Dizemos que uma função $f : \Sigma^\star \to \Sigma^\star$ é computada por um algoritmo A se, para todo $x \in \Sigma^\star$, A produz após o término de sua execução o resultado $f(x)$. Nesse caso, dizemos que f é uma função computável. Para qualquer algoritmo, cada passo de sua computação é proveniente de um conjunto finito de regras elementares. Cada regra pode ser aplicada um número arbitrário de vezes. Além da entrada x, o algoritmo também tem acesso a uma memória arbitrariamente grande onde pode desenvolver a sua computação. Por ser uma abstração conveniente, dizemos que A tem acesso a uma memória infinita. Essa memória é dividida em células e cada célula pode armazenar um símbolo de um conjunto finito de símbolos. A computação é realizada passo a passo até o seu término de acordo com a definição do algoritmo A, ou seja, o seu conjunto de instruções.

A definição de A é formada a partir de um conjunto de regras ou instruções bastante simples. A cada passo, o algoritmo A pode realizar uma das seguintes operações: ler um símbolo da célula de memória atual, escrever um símbolo sobre a célula atual, mover a sua posição de leitura para a célula da direita ou da esquerda, e atualizar o seu estado interno de execução. Portanto, a operação de um algoritmo é algo estritamente local. Apesar disso, uma combinação adequada de regras pode levar a um comportamento global extremamente complexo e imprevisível.

O tempo de execução do algoritmo é o número de passos básicos realizados durante a computação. Embora a sua descrição seja finita, o uso da memória e a alternância entre os estados internos do algoritmo pode levar a um tempo de execução arbitrariamente grande. Além disso, como um algoritmo é um conjunto finito de regras bem definidas, ele pode ser representado como uma palavra e servir de entrada para outros algoritmos.

A partir desses fatos básicos sobre algoritmos, vamos introduzir a formalização oferecida pelas máquinas de Turing (MT). Em uma máquina de Turing M, a memória do algoritmo é representada por meio de uma fita infinita para a direita. A máquina M pode possuir diversas fitas de memória. Mais precisamente, uma *fita* é uma sequência infinita de células, cada qual podendo conter um símbolo proveniente de um conjunto finito Γ chamado de *alfabeto* da máquina M. Cada fita é equipada com uma *cabeça de leitura* que pode ler e escrever símbolos na fita uma célula por vez. A primeira fita de M é chamada de *fita de entrada* e é onde se encontra a palavra de entrada no início da computação. A última fita é chamada de *fita de saída* e armazena o resultado da computação após o seu término. As demais fitas são chamadas *fitas auxiliares* e armazenam informações utilizadas pela máquina M durante a sua computação.

A máquina de Turing M possui um conjunto finito de *estados* que vamos denotar por Q. A execução da máquina de Turing é dividida em passos discretos, sendo que a

cada passo a máquina se encontra em um estado $q \in Q$ específico. Existem dois estados especiais: q_0 e q_f. No início de sua execução, a máquina se encontra no estado inicial q_0. O estado atual determina qual será o próximo passo de M, que consiste em: (1) ler os símbolos presentes nas células atuais de todas as fitas; (2) sobrescrever os símbolos atuais com novos símbolos; (3) alterar o seu estado atual para um novo estado de Q; (4) mover a cabeça de leitura de cada fita para a esquerda ou para direita. A computação prossegue através da alternância entre o estado interno de M e o conteúdo das fitas. Imediatamente após entrar no estado final q_f, a máquina termina a sua execução e a sua saída é dada pela palavra presente na fita de saída. Uma definição um pouco mais formal das ideias anteriores é apresentada a seguir.

Definição 1.1. [Máquina de Turing]. *Uma máquina de Turing M é descrita por uma tupla (Γ, Q, δ) tal que:*

(i) Γ é o conjunto finito de símbolos (alfabeto) que as fitas de M podem conter. Assumimos que Γ contém alguns símbolos especiais: um símbolo em branco $\square$; um símbolo inicial $\triangleright$; e os símbolos 0 e 1.

(ii) Q é o conjunto finito de estados da máquina M. Assumimos que Q contém os estados especiais q_0 e q_f descritos anteriormente.

(iii) Uma função de transição $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{E, D\}^k$, onde k é o número de fitas da máquina de Turing M.

Se a máquina está no estado $q \in Q$, $(\sigma_1, \ldots, \sigma_k)$ são os símbolos nas células atuais das fitas de M e $\delta(q, (\sigma_1, \ldots, \sigma_k)) = (q', (\sigma'_1, \ldots, \sigma'_k), z)$, onde $z \in \{E, D\}^k$, então no próximo passo os símbolos σ_i serão alterados para σ'_i nas k fitas de M, a máquina passará para o estado q' e a i-ésima cabeça de leitura vai se mover para a esquerda ou para a direita de acordo com o valor de z_i. Se uma cabeça de leitura que estiver na célula mais extrema à esquerda de sua fita tentar se mover para a esquerda, ela permanecerá na mesma posição da fita.

Todas as fitas com exceção da primeira possuem a célula mais a esquerda inicializada com o símbolo $\triangleright$ e todas as outras células com o símbolo em branco $\square$. A fita de entrada possui inicialmente o símbolo $\triangleright$, seguido da palavra finita de entrada x e $\square$ em todas as outras células. Todas as cabeças de leitura se posicionam inicialmente na primeira célula de cada fita. O estado inicial da máquina é q_0. Cada passo da computação é realizado através da função de transição δ, como descrito anteriormente. Se a máquina atingir o estado q_f, nenhum passo adicional é executado. Indicaremos por $M(x)$ a saída da máquina de Turing M com entrada x, ou seja, M(x) é o conteúdo da fita de saída após M

atingir o estado q_f (desprezamos os infinitos símbolos $\square$ à direita da palavra de saída). Em complexidade computacional estamos interessados apenas em máquinas que terminam a sua computação em todas as entradas possíveis.

Se M é uma máquina de Turing que produz apenas saídas binárias, então denotaremos a linguagem que M decide por $L(M) = \{x \in \Sigma^\star : M(x) = 1\}$.

Embora o formalismo matemático seja fundamental para expressarmos provas de impossibilidade em complexidade computacional, na maioria das vezes não precisaremos nos concentrar em detalhes técnicos do funcionamento das máquinas de Turing. O fato de que muitos resultados são independentes do modelo computacional em consideração é um importante aspecto da teoria de complexidade computacional.

Apresentamos a seguir um exemplo de máquina de Turing. Diversos outros exemplos podem ser encontrados em Sipser [103].

Exemplo 1.2. [Problema dos Palíndromos]. *Considere a linguagem* PALIN $= \{w \in \{0,1\}^\star : w$ *é um palíndromo* $\}$. *Vamos construir uma máquina de Turing M_P tal que $L(M_P) =$* PALIN. *A máquina M_P possuirá uma fita de entrada, uma fita auxiliar e uma fita de saída. Seu alfabeto será composto por $\{\square, \triangleright, 0, 1\}$. Ela opera da seguinte maneira:*

1) *Copia a entrada para a fita auxiliar. A cabeça de leitura da fita auxiliar permanece a direita da palavra copiada.*
2) *Move a cabeça de leitura da fita de entrada para o início da fita.*
3) *Move a cabeça de leitura da fita de entrada para a direita enquanto move a cabeça de leitura da fita auxiliar para a esquerda. Se em algum momento os valores nas duas fitas são diferentes, M_P termina sua computação e escreve* 0 *na fita de saída.*
4) *Quando M_P esgota a fita de entrada (lendo $\square$ nesta fita), M_P escreve* 1 *na fita de saída e termina sua computação.*

É imediato a partir da construção de M_P que $x \in L(M_P)$ se e somente se x é um palíndromo, ou seja, $L(M_P) =$ PALIN.

Como discutido anterioremente, um aspecto importante dos algoritmos e em particular das máquinas de Turing é a possibilidade de representá-los por meio de palavras. Como a função de transição de uma máquina de Turing caracteriza todo o seu comportamento, ela será utilizada para a codificação da máquina. É fácil desenvolver uma representação que satisfaça os seguintes requisitos:

(i) Toda palavra em $\{0,1\}^\star$ representa alguma máquina de Turing.

(ii) Toda máquina de Turing (vista como um objeto matemático) é representada por infinitas palavras.

Finalmente, observamos que os programas escritos em linguagens de programação modernas podem ser eficientemente simulados por máquinas de Turing (veja a seção de referências adicionais no fim do capítulo). Por isso, em termos de poder computacional, temos a equivalência "Algoritmos = Máquinas de Turing = Programas de Computador".

1.7 Eficiência dos Algoritmos

Para estudar a dificuldade dos problemas computacionais é preciso quantificar a noção de eficiência de um algoritmo. Diversas medidas de complexidade podem ser utilizados para isso, como o número de passos básicos realizados pelo algoritmo ou a memória total utilizada durante a computação.

Fixada uma medida de complexidade conveniente, para comparar o desempenho de dois algoritmos é interessante desprezar detalhes técnicos irrelevantes e dar maior importância para a idéia fundamental por trás de cada algoritmo. Por isso, o desempenho de dois algoritmos é comparado através do crescimento da função de complexidade quando instâncias cada vez maiores são utilizadas na entrada.

Pode-se também considerar uma distribuição de probabilidade no conjunto de instâncias de cada tamanho para o cálculo da complexidade final do algoritmo. Por ser mais fácil trabalhar sem esse parâmetro adicional, geralmente a análise de pior caso é adotada. Isso significa que a complexidade do algoritmo para instâncias de tamanho n é definida como sendo o maior valor adquirido pela medida de complexidade adotada em instâncias de tamanho n.

Nesta dissertação utilizaremos como principal medida de complexidade o número de passos básicos utilizados por um algoritmo. Além disso, o problema P vs NP e a maioria dos resultados importantes em complexidade computacional são baseados na análise de pior caso. Isso torna o estudo teórico de diversos problemas viável e conveniente.

Definição 1.3. [Função Computada por uma Máquina de Turing]. *Sejam $f : \{0,1\}^\star \to \{0,1\}^\star$ uma função e M uma máquina de Turing. Dizemos que M computa a função f se para todo $x \in \{0,1\}^\star$, sempre que M é inicializada com entrada x temos $M(x) = f(x)$.*

Definição 1.4. [Complexidade de Tempo]. *A complexidade de tempo exata de uma máquina de Turing M é dada pela função $t_M : \{0,1\}^\star \to \mathbb{N}$, de forma que quando inicializada com x em sua fita de entrada, M termina sua computação em exatamente $t_M(x)$ passos. A complexidade de tempo de M é dada pela função $T_M : \mathbb{N} \to \mathbb{N}$ tal que $T_M(n) = \max\{t_M(x) : x \in \{0,1\}^n\}$.*

É fácil perceber que a máquina de Turing apresentada na seção anterior computa PALIN em tempo $O(n)$, ou seja, $T_{M_P}(n)$ é $O(n)$.

De forma análoga, podemos definir a complexidade de espaço de uma máquina de Turing.

Definição 1.5. [Complexidade de Espaço]. *A complexidade de espaço exata de uma máquina de Turing M é dada pela função $s_M : \{0,1\}^\star \to \mathbb{N}$, de forma que quando inicializada com x em sua fita de entrada, M acessa exatamente $s_M(x)$ células distintas entre todas as suas fitas. A complexidade de espaço de M é dada pela função $S_M : \mathbb{N} \to \mathbb{N}$ tal que $S_M(n) = \max\{s_M(x) : x \in \{0,1\}^n\}$.*

1.8 Um Exemplo de Limitante Inferior

Os pesquisadores que estudam complexidade estão interessados em determinar a complexidade computacional de cada problema. Embora essa seja uma tarefa bastante difícil para a maioria dos problemas computacionais, é possível caracterizar precisamente a complexidade de algumas tarefas simples.

Nesta seção vamos considerar apenas máquinas de Turing com uma única fita. Como vamos trabalhar apenas com linguagens, não há necessidade de ter uma fita separada para saída. Por conveniência, vamos assumir que existem dois estados especiais $q_{aceitar}$ e $q_{rejeitar}$ no conjunto de estados Q de cada máquina de Turing. Uma máquina M aceita a palavra x se em algum passo de sua computação ela entra no estado $q_{aceitar}$. Analogamente, M rejeita x caso seja colocada no estado $q_{rejeitar}$ durante sua computação. Uma vez nesses estados, M termina sua computação imediatamente.

Considere novamente a linguagem PALIN $= \{x \in \{0,1\}^\star : x$ é um palíndromo $\}$. É fácil desenvolver uma máquina de Turing M de fita única capaz de decidí-la em tempo $O(n^2)$. Basicamente, M cruza a fita por no máximo n vezes enquanto checa se o símbolo mais a esquerda ainda não verificado é o mesmo que o símbolo mais a direita ainda não verificado. Como em cada cruzamento a máquina percorre no máximo n células, a complexidade de tempo total é $O(n^2)$.

Como provar que esse zig-zag é essencialmente a melhor maneira de resolver esse problema utilizando máquinas de Turing com uma única fita? Pode parecer intuitivo que cerca de n^2 passos sejam necessários, mas precisamos encontrar uma prova formal para esse fato. Para isso, utilizaremos um conceito importante da teoria de informação chamado complexidade de Kolmogorov. Intuitivamente, a complexidade de Kolmogorov de uma sequência de bits é o tamanho do menor programa capaz de imprimir tal sequência. Mostraremos a seguir que qualquer máquina de Turing de fita única capaz de decidir PALIN precisa de $\Omega(n^2)$ passos para completar a sua computação.

Definição 1.6. [Complexidade de Kolmogorov]. *Seja $\mu = (M_1, M_2, \ldots)$ uma codificação das máquinas de Turing no alfabeto $\{0,1\}$ a partir das respectivas funções de transição. Para cada palavra $x \in \{0,1\}$, a sua complexidade de Kolmogorov em relação a μ é:*

$$K_\mu(x) = min\{|M_j| : M_j(\epsilon) = x\}.$$

Lema 1.7. [Palavras Incompressíveis]. *Para todo $n \in \mathbb{N}$ existe uma palavra x de tamanho n tal que $K_\mu(x) \geq n$.*

Demonstração. Existem 2^n palavras de tamanho n e apenas $2^n - 1$ palavras de tamanho menor ou igual a $n-1$. Como cada máquina de Turing representa no máximo uma palavra, deve existir alguma palavra de tamanho n que não pode ser representada no sentido da complexidade de Kolmogorov por máquinas de Turing de tamanho menor que n. □

O lema anterior prova que existem palavras incompressíveis. Vamos agora descrever a idéia utilizada na prova do limitante inferior $\Omega(n^2)$ para PALIN para máquinas de Turing com uma única fita. Em primeiro lugar, vamos mostrar que o tempo de execução de qualquer algoritmo decidindo PALIN pode ser usado para provar um limitante superior na complexidade de Kolmogorov das palavras do alfabeto $\{0,1\}^\star$. Depois disso mostraremos que, caso o algoritmo seja rápido demais (utilize $o(n^2)$ passos), então teremos uma violação do fato de que existem palavras incompressíveis. Esse argumento é conhecido como *método da incompressibilidade*. A seguir apresentamos a prova formal desse resultado.

Teorema 1.8. *Seja M uma máquina de Turing de fita única que decide* PALIN. *Então a complexidade de tempo de M é $\Omega(n^2)$.*

Demonstração. Considere a computação de M com uma entrada w. Dizemos que em um determinado passo da computação de M ocorre um *cruzamento* entre as células i e $i+1$ da fita com o estado q se: (i) antes desse passo a cabeça de leitura de M se encontra na célula i ou na célula $i+1$ da fita; (ii) após esse passo a cabeça de leitura de M também se encontra na célula i ou na célula $i+1$ da fita; (iii) o estado de M após esse passo é q. A *sequência de cruzamentos* de M com entrada w entre as células i e $i+1$ da fita será denotada por $S_i(M,w) = (q_1, \ldots, q_m)$, de forma que todos os cruzamentos existentes aparecem em $S_i(M,w)$ e os mesmos estão ordenados de acordo com o número do passo da computação no momento do cruzamento. Observe que, dada uma sequência de cruzamentos, cruzamentos de número ímpar são oriundos de passos da computação de M que entram na posição $i+1$ da fita a partir da posição i, e cruzamentos de número par são oriundos de passos da computação de M que entram na posição i da fita a partir da posição $i+1$.

Do ponto de vista das primeiras i posições da fita, tudo o que importa é o processamento que ocorre dentro dessas posições e a sequência de cruzamentos de número par

em $S_i(M, w)$. Isso significa que, independentemente do que aconteça nas outras posições localizadas à direita da célula i, desde que a sequência de cruzamentos de número par seja a mesma teremos os mesmos símbolos escritos nas primeiras i posições de memória ao término da computação. O mesmo ocorre com as posições da fita localizadas à direita da célula i em relação aos cruzamentos de número ímpar em $S_i(M, w)$. Esse resultado ilustra o fato de que a computação é um fenômeno local.

Seja agora $w = x0^n x^R$, onde x^R denota o reverso de x (x escrito ao contrário) e x é uma palavra incompressível de tamanho n (veja o lema anterior). Lembre-se que $t_M(w)$ é o número de passos de M com entrada w. Deve existir uma posição i da fita em um dos locais inicialmente ocupados pela parte da entrada correspondente a 0^n com número de cruzamentos m entre as células i e $i+1$ satisfazendo $m \leq t_M(w)/n$. Isso ocorre pois, caso contrário, teríamos uma violação do número total de passos $t_M(w)$ de M. Suponha que a sequência de cruzamentos entre essas células seja $S_i(M, w) = (q_1, \ldots, q_m)$.

Vamos agora provar que, em relação à máquina M, a tupla $(i, n, S_i(M, w))$ identifica de forma única a palavra original x utilizada na construção de w. Com o intuito de obtermos uma contradição, suponha que exista uma palavra distinta $w' = y0^n y^R$ de tamanho $3n$ com a mesma tupla anterior. Considere agora o processamento de M com a entrada $x0^n y^R$. É fácil perceber que essa entrada produz a mesma sequência de cruzamentos entre as células i e $i+1$ que as palavras w e w'. Por isso, durante a computação de M com essa nova entrada, M se comporta nas primeiras i posições da fita como se estivesse processando a palavra w, e nas outras posições superiores como se estivesse processando a palavra w'. Como w e w' são palíndromos, as duas palavras são aceitas por M. Isso implica que, independentemente do local onde ocorra o último passo de sua computação, a palavra $x0^n y^R$ também será aceita por M. Como M decide PALIN e $x0^n y^R$ não é um palíndromo, obtemos uma contradição.

Provaremos a seguir que existe um máquina de Turing M' que, dada a tupla $(i, n, S_i(M, w))$, imprime a palavra x. Essa máquina computa da seguinte maneira:

1) Gera em ordem lexicográfica todas as palavras de tamanho $3n$ na forma $z0^n z^R$.
2) Simula a máquina M em cada uma dessas entradas.
3) Se ao término de alguma simulação a sequência de cruzamentos entre as células i e $i+1$ for igual a $S_i(M, w)$, a máquina M' imprime a palavra z usada na simulação atual e termina seu processamento.

Como x é a única palavra compatível com a tupla de entrada, o algoritmo anterior imprime x ao término de sua execução. Podemos adicionar a tupla $(i, n, S_i(M, w))$ à memória interna de M' para obtermos uma máquina M_x tal que $M_x(\epsilon) = x$.

Para todo x, qual o tamanho da palavra que codifica a máquina M_x? Claramente,

isso depende apenas do tamanho da tupla ligada a x e de uma constante $c_{M'}$ relacionada aos detalhes de implementação da máquina M'. Na tupla de x, temos que os valores i e n podem ser representados por $\log n$ bits. Por outro lado, $S_i(M, w)$ pode ser codificado em no máximo $mc_M \leq t_M(w)c_M/n$ bits, onde a constante c_M depende apenas da máquina de Turing M. Como x é uma palavra incompressível, isso prova que:

$$n \leq K_\mu(x) \leq c_M t_M(w)/n + 2\log n + c_{M'}.$$

Reescrevendo essa desigualdade, obtemos:

$$t_M(w) \geq (n^2 - 2n\log n - nc_{M'})/c_M = \Omega(n^2).$$

O fato de que $|w| = 3n$ não altera o resultado assintótico e por isso $T_M(n) = \Omega(n^2)$, como queríamos demonstrar. □

Lembre-se que a linguagem PALIN pode ser decidida por máquinas de Turing com várias fitas em tempo $O(n)$. Portanto, esse teorema mostra que máquinas com várias fitas são mais poderosas do que máquinas de fita única.

Embora o método da incompressibilidade utilize um argumento muito elegante para provar limitantes inferiores, ele é muito engenhoso e não parece ser forte o suficiente para provar cotas inferiores exponenciais, por exemplo. Veremos na seção 3.6 que existem argumentos que, quando aplicáveis, são capazes de provar limitantes inferiores muito mais fortes.

O resultado provado nesta seção é fortemente dependente das sutilezas do modelo computacional escolhido (máquinas de Turing com uma única fita). Ele foi apresentado para ilustrar a grande diferença entre a prova de um limitante superior (através do desenvolvimento de um algoritmo) e a prova de um limitante inferior (através de uma demonstração matemática). Notamos que, apesar de resultados como esse serem interessantes, os pesquisadores em complexidade computacional estão mais interessados em provar resultados que não dependem de detalhes do modelo computacional em consideração.

1.9 Referências Adicionais

Uma excelente introdução à teoria da computação é o livro de Sipser [103]. Para um estudo mais avançado da teoria de computabilidade, recomendamos a leitura de Rogers [93] e Odifreddi [84, 85]. Diversos artigos clássicos em computabilidade estão reunidos em Davis [31]. O livro de Savage [95] discute diversos modelos computacionais estudados em computação. O livro de Carnielli e Epstein [19] é uma ótima referênca em português para leitores interessados na relação entre computabilidade e os fundamentos da matemática.

A definição de máquinas de Turing apresentada neste capítulo é baseada no livro de Arora e Barak [7]. Uma possível exceção para a eficiência universal das máquinas de Turing

em relação a outros modelos computacionais é o desenvolvimento de um computador quântico. Entretando, ainda não temos certeza de que o modelo teórico por trás dessas máquinas é fisicamente realizável. Recomendamos os livros de Hirvensalo [57] e Kaye et al. [67] para uma introdução ao tema.

O compêndio de matemática editado por Gowers et al. [48] contém um excelente artigo de Goldreich que discute os principais problemas e desafios enfrentados em complexidade computacional. O artigo de Fortnow e Homer [42] dá uma retrospectiva dos desenvolvimentos em complexidade. Outro artigo no mesmo estilo é Impagliazzo [62]. Uma discussão fascinante sobre a relação entre complexidade computacional e matemática pode ser encontrada em Wigderson [109]. Um livro clássico sobre complexidade computacional é Papadimitriou [88]. Outros livros mais recentes são Arora e Barak [7] e Goldreich [47].

Para uma discussão informal sobre o estado atual do problema P vs NP, veja o artigo recente de Fortnow [41]. A história do problema está contada em Sipser [102] e a sua importâcia é resumida em Cook [24]. Veja a seção final dos outros capítulos do texto para referências técnicas sobre o problema.

A complexidade de Kolmogorov e o método da incompressibilidade são discutidos no livro de Li e Vitányi [75]. O teorema 1.8 foi provado no artigo de Hennie [55]. Uma extensão das idéias utilizadas na demonstração desse resultado pode ser obtida em Maass [77].

Capítulo 2

Introdução ao Problema P vs NP

"There is the problem, seek the solution.
You can find it through pure thought."
David Hilbert.

Neste capítulo abordamos o principal problema em aberto da teoria de complexidade computacional: a relação entre as classes de complexidade P e NP. Informalmente, o problema P vs NP procura determinar se, para uma classe importante de problemas computacionais, a busca exaustiva por soluções é essencialmente a melhor alternativa algorítmica possível. Apresentamos diversas formulações equivalentes para esse problema, além de discutirmos sua importância e os principais métodos empregados na tentativa de resolvê-lo.

2.1 Introdução

A mera existência de um programa de computador capaz de resolver um problema não significa que sua solução por meios computacionais seja de fato viável. Considere, por exemplo, a tarefa de encontrar o menor caminho a ser percorrido de carro para viajar entre duas cidades. A solução imediata de procurar pelo caminho de menor distância entre todos os caminhos possíveis pode resultar em uma busca extremamente demorada. Para visualizar isso, basta levar em conta que as duas cidades podem estar localizadas nos dois extremos de um continente. A quantidade de caminhos possível se torna tão grande que, mesmo após várias horas de execução, pode ser que nenhum computador moderno encontre o trajeto ótimo. Por isso, embora o algoritmo proposto para essa tarefa seja correto, a busca exaustiva inviabiliza completamente a solução do problema. Veremos

na próxima seção que o problema do caminho mínimo possui uma solução muito mais eficiente. A classe de problemas que admitem algoritmos eficientes é denotada por P.

Em meados da década de sessenta, A. Cobham [22] e J. Edmonds [34] estavam interessados em muitas questões desse tipo. Em particular, Edmonds foi capaz de provar que a busca exaustiva não é a melhor solução computacional para muitos problemas de interesse prático. Para conseguir isso, é preciso explorar de forma inteligente a estrutura intrínseca de cada problema. Segundo Edmonds, um algoritmo capaz de tirar proveito das sutilezas do problema computacional em consideração poderia ser considerado um bom algoritmo. Apesar de ter mostrado que isso é possível para muitas tarefas, ele e outros pesquisadores da época não foram capazes de encontrar soluções mais elegantes que a busca exaustiva para muitos outros problemas importantes.

O principal problema da busca exaustiva é que sua implementação requer um número de passos computacionais proporcional ao número de possibilidades no espaço de busca, o que é em geral uma função de crescimento exponencial no tamanho das instâncias do problema. Com a descoberta de novos algoritmos, ficou claro que a existência de um algoritmo de complexidade exponencial para um problema não significava que a complexidade real do problema era exponencial. Cobham e Edmonds sugeriram que um algoritmo é eficiente quando seu número de passos é limitado por uma função algébrica, ou seja, por um polinômio. Como essa definição apresenta diversas propriedades interessantes e a experiência nos mostra que algoritmos polinomiais são computacionalmente viáveis, ela se tornou amplamente aceita.

Os pesquisadores logo perceberam que diversos problemas computacionais difíceis possuem uma propriedade muito especial: dada uma resposta para uma instância do problema, é possível verificar de modo eficiente se ela é de fato uma das soluções procuradas. A classe de problemas com soluções que podem ser *verificadas de modo eficiente* é denotada por NP.

Lembre que P representa a classe de problemas com soluções que podem ser *encontradas de modo eficiente*. Qual a relação entre as classes P e NP? Em outras palavras, se podemos verificar a solução de um problema em tempo polinomial, então também podemos encontrar a solução em tempo polinomial?

Vejamos, através de um exemplo simples, o que isso significa. É fácil verificar quando uma sequência de números é uma solução para um quebra-cabeças Sudoku[1]. Portanto, Sudoku é um problema pertencente à classe NP[2]. No entanto, existe algum modo inteligente de encontrar rapidamente a solução para um jogo de Sudoku? Ou seja, Sudoku pertence à classe P?

[1] A descrição do jogo Sudoku pode ser obtida em *http://pt.wikipedia.org/wiki/Sudoku*.

[2] Para essa afirmação fazer sentido, estamos considerando que os tabuleiros de Sudoku podem ser arbitrariamente grandes.

O problema P vs NP procura estabelecer essa questão para todos os problemas com soluções que podem ser verificadas de forma eficiente. Se as classes P e NP forem idênticas, então as soluções de Sudoku, e de milhares de outros problemas computacionais importantes, podem ser encontradas de forma eficiente. Não será necessária nenhuma criatividade adicional para resolver esses problemas. Descobrir uma solução será tão simples quanto verificar uma solução.

A classe NP possui muitos problemas relevantes. Considere, por exemplo, o conjunto de teorias científicas capazes de explicar um certo fenômeno. Em geral, temos uma idéia razoável do que pode ser considerado uma teoria. Além disso, usualmente é possível estabelecer uma medida de sucesso para a compatibilidade de cada teoria com os dados científicos disponíveis, sendo que esse teste pode ser feito de modo eficiente (uma maneira de fazer isso seria comparar as predições da teoria com os dados experimentais). Portanto, dado um nível de precisão desejado, é possível verificar de forma eficiente se uma teoria é adequada para explicar um determinado fenômeno. Por isso, esse problema pertence à classe NP. Se P = NP, então também podemos encontrar rapidamente as melhores teorias capazes de explicar os dados científicos disponíveis. Seria possível, por exemplo, encontrar uma teoria para explicar com precisão os dados coletados através da colisão de partículas em altas energias no novo acelerador LHC. Não só isso, como veremos na seção 4.1, seria viável encontrar a teoria com a menor descrição possível satisfazendo as mesmas propriedades. Muitas vezes, a teoria mais simples possível é a teoria correta.

Devido a essa e diversas outras implicações fascinantes, muitos pesquisadores acreditam que P $\neq$ NP. Entretanto, ainda não sabemos provar esse fato. Por isso, por mais improvável que possa parecer, não podemos descartar a possibilidade de que P = NP. Além do mais, já fomos surpreendidos diversas vezes com idéias brilhantes capazes de superar os melhores algoritmos existentes até então.

Para provarmos definitivamente que P $\neq$ NP, será necessário mostrar que existe um problema em NP muito especial. Especificamente, será preciso demonstrar que nenhuma idéia pode ser usada para resolver esse problema de modo eficiente. De modo surpreendente, se essas classes forem realmente distintas, já sabemos quais são os problemas candidatos (veja a seção 2.5).

Que tipo de matemática podemos utilizar para separar essas duas classes de problemas computacionais? Ainda não sabemos responder essa questão, mas somos capazes de provar que diversos métodos matemáticos poderosos não são suficientes. Um exemplo importante de resultado nesse sentido será apresentado no capítulo 5. Na próxima seção apresentamos o enunciado formal do problema P vs NP.

2.2 Definição

A descrição apresentada anteriormente para o problema P vs NP corresponde à sua versão de busca. No entanto, o estudo de problemas desse tipo pode ser reduzido ao estudo de problemas de decisão (i.e., problemas de pertinência em linguagens). Por isso, vamos considerar como enunciado formal do problema P vs NP a sua versão de decisão. Por ser mais simples, essa abordagem é amplamente utilizada na literatura. Na seção 2.3 mostraremos que as duas formulações são de fato equivalentes (versão de busca vs versão de decisão).

Definição 2.1. [Classe de Complexidade DTIME]. *Seja $T : \mathbb{N} \to \mathbb{N}$ uma função. Uma linguagem $L \subseteq \{0,1\}^\star$ pertence à classe* $\mathrm{DTIME}(T(n))$ *se existe uma máquina de Turing M que decide L em $O(T(n))$ passos.*

Observe que estamos interessados apenas no comportamento assintótico da função de complexidade. Em outras palavras, os termos aditivos inferiores e as constantes multiplicativas não são relevantes para a teoria de complexidade computacional. Isso torna os resultados muito mais simples e elegantes, além de ser uma convenção que pode ser justificada pelo seguinte teorema.

Teorema 2.2. [Teorema do Speedup Linear]. *Sejam M uma máquina de Turing, $T_M(n)$ sua função de complexidade de tempo e suponha que M decida a linguagem L. Para todo número real $\varepsilon > 0$, existe uma máquina de Turing M_ε que decide L em tempo $T_{M_\varepsilon}(n) \leq \varepsilon T_M(n) + n + 2$.*

Demonstração. Aumentamos o alfabeto de M de forma que uma sequência de símbolos do alfabeto original de M possa ser representado por um único símbolo em M_ε. Com isso, a máquina de Turing M_ε pode simular diversos passos de M em um único passo. Os termos adicionais que aparecem na expressão de $T_{M_\varepsilon}(n)$ resultam do processamento inicial necessário para converter a palavra de entrada para os novos símbolos. A demonstração completa desse resultado pode ser obtida no livro de Papadimitriou [88]. □

Vamos agora definir a classe de complexidade que abriga as linguagens que podem ser decididas eficientemente.

Definição 2.3. [Classe de Complexidade P].
$\mathrm{P} = \bigcup_{k \geq 1} \mathrm{DTIME}(n^k)$.

O próximo exemplo exibe um algoritmo eficiente para o problema discutido no início da seção anterior.

Exemplo 2.4. [Problema do Caminho Mínimo]. *Esse problema pode ser modelado através de um grafo com peso nas arestas. Seja $G = (V, E, p)$ um grafo, onde V é um conjunto de vértices (cidades), E é um conjunto de pares não-ordenados de vértices (cidades vizinhas) e $p : E \to \mathbb{R}^+$ é o peso de cada aresta (distância entre as cidades vizinhas). Se $C = (v_1, \ldots, v_n)$ é um caminho no grafo G, a distância de C é dada pela soma do peso das arestas $v_i v_{i+1} \in E$, onde $1 \leq i \leq n-1$. Dados dois vértices s e t em G, nosso algoritmo deve determinar qual é o caminho entre s e t com a menor distância total. Vamos a seguir descrever um algoritmo A que resolve de forma eficiente esse problema.*

Adotamos as seguintes convenções. Fixe uma ordem qualquer nas arestas de G. Vamos denotar por E_k o conjunto que contém as k primeiras arestas segundo essa ordem. Seja $G_k = (V, E_k, p|_{E_k})$ o subgrafo ponderado de G que contém todos os vértices originais de G e apenas as k primeiras arestas segundo essa ordem. O algoritmo A resolverá um problema mais geral que o proposto. Na k-ésima etapa, irá encontrar o menor caminho entre todos os pares de vértices do grafo G_k. O caso base G_0 é trivial. Assuma que A tenha executado por k etapas e seja $C_k(u, v)$ o menor caminho encontrado entre os vértices u e v no grafo G_k. A distância total do caminho $C_k(u, v)$ será denotada por $d_k(u, v)$. Como encontrar os melhores caminhos no grafo G_{k+1}?

Em primeiro lugar, se a nova aresta $e_{k+1} = \{w_1, w_2\}$ é tal que $p(e_{k+1}) \geq d_k(w_1, w_2)$, então os melhores caminhos e as melhores distâncias não são alterados. Caso contrário, para todos os pares de vértices já sabemos qual é o menor caminho entre eles em G_{k+1} que não utiliza a nova aresta. Sejam v_1 e v_2 dois vértices de G. Temos duas possibilidades para cada caminho ótimo em G_{k+1}. Ou ele é igual ao caminho anterior, ou utiliza a nova aresta. Como qualquer aresta só aparece uma única vez em um caminho ótimo, basta A comparar $d_k(v_1, v_2)$ com $d_k(v_1, w_1) + p(e_{k+1}) + d_k(w_2, v_2)$ e com $d_k(v_1, w_2) + p(e_{k+1}) + d_k(w_1, v_2)$ para determinar $C_{k+1}(v_1, v_2)$ e $d_{k+1}(v_1, v_2)$. Isso completa a descrição do algoritmo A.

O leitor pode verificar facilmente que esse algoritmo utiliza um número de passos polinomial em $|E| + |V|$. Considere agora a versão de decisão do problema anterior:

$$\text{DISTMIN} = \{\langle G, s, t, k\rangle : G \text{ é um grafo ponderado e } d_G(s, t) \leq k\}.$$

Podemos utilizar o algoritmo anterior para determinar em tempo polinomial o menor caminho entre os vértices s e t e comparar sua distância com o valor k. Por isso, temos que DISTMIN $\in$ P.

Apresentamos a seguir a definição da classe NP. Uma definição alternativa é discutida na seção 2.4.

Definição 2.5. [Classe de Complexidade NP].
Uma linguagem $L \subseteq \{0, 1\}^\star$ pertence a classe NP *se existe um polinômio $p(.) : \mathbb{N} \to \mathbb{N}$ e uma máquina de Turing V de tempo polinomial tal que para todo $x \in \{0, 1\}^\star$:*

$$x \in L \;\Leftrightarrow\; \exists w \in \{0,1\}^{p(|x|)} \text{ tal que } V(x,w) = 1.$$

Se $x \in L$, $w \in \{0,1\}^{p(|x|)}$ e $V(x,w) = 1$, dizemos que w é um certificado para x em relação a linguagem L e a máquina V. Além disso, dizemos que V é um verificador eficiente para a linguagem L.

Observe que a exigência na definição da classe NP de que o tamanho dos certificados seja exatamente $p(n)$ para entradas de tamanho n não é essencial. Basta que algum polinômio limite o tamanho dos certificados, uma vez que sempre podemos completá-los até que atinjam o tamanho desejado. Vale reforçar que se $L \in$ NP, então L é uma linguagem decidível ($x \in L$ se e somente se $V(x,w) = 1$ para alguma palavra w de tamanho $p(|x|)$).

Na classe NP residem as linguagens que possuem certificados sucintos e eficientemente verificáveis para a presença de uma palavra na linguagem. Vejamos um exemplo.

Exemplo 2.6. [COMPOSTOS $\in$ NP]. *Considere a seguinte linguagem:*

$$\text{COMPOSTOS} = \{\langle n \rangle : \exists k \in \mathbb{N},\ 1 < k < n \text{ e } k \text{ divide } n\}.$$

Como a existência de um divisor não trivial de n serve como certificado canônico para $\langle n \rangle \in$ COMPOSTOS e a relação de divisibilidade pode ser verificada em tempo polinomial, temos que COMPOSTOS $\in$ NP. *Recentemente, um algoritmo eficiente conhecido como* AKS *foi encontrado para esse problema, ou seja, na verdade temos também que* COMPOSTOS $\in$ P. *Veja a seção de referências deste capítulo para mais detalhes.*

A versão de busca do problema P vs NP pergunta se encontrar uma solução é mais difícil do que verificar as soluções. Por outro lado, a versão de decisão desse problema pergunta se verificar uma afirmação por conta própria é mais difícil do que checar a validade de uma prova eficiente. O exemplo anterior mostra que é fácil convencer alguém de que um número é composto através de um certificado. Neste caso, porém, também é possível verificar essa afirmação de forma eficiente utilizando o algoritmo AKS. E quanto ao caso geral? A principal questão em aberto da teoria de complexidade computacional pode ser enunciada da seguinte maneira.

Questão em Aberto 2.7. P = NP *?*

Na seção 2.7 apresentamos as principais abordagens utilizadas até agora na tentativa de resolver esse problema.

Suponha, por um momento, que não tivéssemos conhecimento sobre o algoritmo AKS. Como convencer alguém rapidamente de que um número não é composto, ou seja, é primo? Existe alguma prova sucinta e eficiente para a propriedade de ser número primo? A classe de complexidade coNP abriga as linguagens para as quais podemos apresentar um certificado eficiente de que uma dada palavra *não* está na linguagem.

Definição 2.8. [Classe de Complexidade coNP].
coNP $= \{ \overline{L} : L \in \text{NP} \}$.

Definição 2.9. *De forma geral, se $\mathcal{R}$ é uma classe de complexidade, definimos* co$\mathcal{R}$ $= \{ \overline{L} : L \in \mathcal{R}\}$.

Observe que a classe coNP não é o complemento do conjunto NP. Por definição, se uma linguagem L está em coNP, então podemos verificar eficientemente através de um certificado que uma certa palavra não pertence à linguagem L.

Por exemplo, enquanto é fácil provar que um dado sistema de equações quadráticas possui solução (basta apresentar a solução), como convencer alguém de que esse sistema não é satisfatível? Existe algum certificado eficiente que possa ser usado para comprovar a ausência de soluções? No caso geral, temos o seguinte problema.

Questão em Aberto 2.10. NP = coNP *?*

Esse problema também é muito importante, e sua solução ampliará nosso entendimento sobre a capacidade das provas eficientes. Além disso, temos a seguinte relação entre as conjecturas anteriores.

Definição 2.11. *Seja $\mathcal{R}$ uma classe de complexidade. Dizemos que $\mathcal{R}$ é fechada por complementação de linguagens se, para toda linguagem $L \in \mathcal{R}$, temos $\overline{L} \in \mathcal{R}$.*

Lema 2.12. *Se* NP $\neq$ coNP *, então* P $\neq$ NP.

Demonstração. Em primeiro lugar, observe que NP = coNP se e somente se NP é fechada por complementação de linguagens (isso vale em geral para qualquer classe de complexidade). Portanto, segue pela hipótese do lema que NP não possui essa propriedade. No entanto, invertendo os estados de aceitação e rejeição de uma máquina determinística, temos que P é uma classe fechada por complementação de linguagens. Isso prova que P $\neq$ NP. □

Veremos mais alguns resultados relacionando as classes NP e coNP na seção 4.4. Veja a seção de referências adicionais deste capítulo para saber mais sobre a questão NP vs coNP.

Temos ainda outro relacionamento básico entre as classes discutidas anteriormente.

Lema 2.13. P $\subseteq$ NP $\cap$ coNP.

Demonstração. Como P é uma classe fechada por complementação de linguagens, basta provar que P $\subseteq$ NP. Seja M uma máquina de Turing polinomial que decide L. Considere o verificador V que aceita a entrada $\langle x, w\rangle$ se e somente se $M(x) = 1$. É imediato que V é um verificador eficiente. Tomando, por exemplo, o polinômio $p(n) = n$, temos que $x \in L \;\Leftrightarrow\; \exists w \in \{0,1\}^{p(|x|)}$ tal que $V(x, w) = 1$. Isso prova que $L \in$ NP. □

Suponha agora que exista uma linguagem L que admite certificados eficientemente verificáveis tanto para presença de palavras na linguagem quanto para ausência de palavras. Em outras palavras, $L \in \mathrm{NP} \cap \mathrm{coNP}$. Podemos concluir, nesse caso, que L pode ser decidida em tempo polinomial?

Questão em Aberto 2.14. $\mathrm{P} = \mathrm{NP} \cap \mathrm{coNP}$ *?*

Similarmente, é possível demonstrar que se esse resultado for falso então $\mathrm{P} \neq \mathrm{NP}$.

Vimos nesta seção que outras duas perguntas interessantes surgem naturalmente a partir do enunciado do problema P vs NP. A solução de qualquer um desses problemas será um avanço importante para a teoria de complexidade computacional.

2.3 Decisão vs Busca

Nesta seção vamos demonstrar que a versão de busca do problema P vs NP discutida na seção inicial não é mais difícil do que a versão de decisão definida na seção anterior. Em primeiro lugar, descreveremos informalmente uma maneira de formalizar a versão de busca do problema. Depois argumentaremos que as duas formulações são de fato equivalentes.

Considere novamente o exemplo 2.4. A linguagem DISTMIN possui certificados eficientes muito simples: basta exibir um caminho de comprimento menor ou igual a k para comprovar a presença de uma palavra na linguagem. Colocado de outra forma, os certificados de NP podem ser visualizados como soluções para uma dada instância do problema. Por isso, as linguagens em NP são aquelas com soluções eficientemente verificáveis. Por outro lado, encontrar a solução de uma instância de um problema em NP é o mesmo que obter um certificado em relação a um verificador V.

É claro que se $\mathrm{P} = \mathrm{NP}$ para a versão de busca, então temos $\mathrm{P} = \mathrm{NP}$ para a versão de decisão. Basta utilizar o algoritmo que encontra as soluções (certificados) de modo eficiente e descartar a solução encontrada, uma vez que tudo que precisamos saber na versão de decisão é se existe um certificado válido para a entrada ou não. Por outro lado, suponha que $\mathrm{P} = \mathrm{NP}$ no sentido formal apresentado na seção anterior (versão de decisão). Podemos encontrar soluções (certificados) de modo eficiente?

Vamos obter o certificado procurado bit por bit. Primeiro, mostramos que o problema de verificar se uma palavra é o início de um certificado pertence à classe NP. A hipótese de que $\mathrm{P} = \mathrm{NP}$ fornece um algoritmo eficiente para essa tarefa. Finalmente, usamos esse algoritmo para buscar de modo eficiente algum certificado. Partimos agora para a descrição formal dessas ideias.

Teorema 2.15. *Suponha que* $\mathrm{P} = \mathrm{NP}$. *Seja* $L \in \mathrm{NP}$ *e* V *um verificador eficiente para* L. *Então existe uma máquina de Turing* M *de tempo polinomial tal que para todo* $x \in L$,

M imprime um certificado w para x (em relação ao verificador V). Em outras palavras, se $x \in L$ então $V(x, M(x)) = 1$. Além disso, M indica que não há certificado para x no caso em que $x \notin L$.

Demonstração. Seja $p(.) : \mathbb{N} \to \mathbb{N}$ o polinômio que descreve o tamanho dos certificados de L em relação ao verificador V. Considere a seguinte linguagem:

$$L' = \{\langle x, w' \rangle : \exists w'' \text{ tal que } |w''| = p(|x|) - |w'| \text{ e } V(x, w'w'') = 1\}.$$

É fácil perceber que $L' \in$ NP, uma vez que a própria palavra w'' serve como certificado para a entrada $\langle x, w' \rangle$. Mais ainda, note que $x \in L$ se e somente se $\langle x, \epsilon \rangle \in L'$.

Como temos por hipótese que P = NP, segue que $L' \in$ P. Seja M' uma máquina de Turing de tempo polinomial que decide L'. Vamos utilizá-la como subrotina para a construção da máquina M do enunciado. Com entrada x, M computa da seguinte maneira:

1) Se $M'(x, \epsilon) = 0$, M declara que $x \notin L$.
2) Caso contrário, M encontra bit por bit um certificado w para a entrada x. Inicialmente, $w := \epsilon$.
3) Enquanto $|w| < p(|x|)$: se $M'(x, w0) = 1$ então $w := w0$, senão $w := w1$.
4) Imprime w e encerra sua execução.

Como M faz no máximo $p(|x|) + 1$ chamadas à máquina polinomial M', o certificado procurado é encontrado de modo eficiente, caso ele exista. □

O argumento utilizado na demonstração anterior é conhecido como *método do prefixo*. Esse método será útil novamente na seção 4.6.

2.4 Uma Formulação Alternativa

A classe NP também pode ser definida através da introdução de um novo recurso nas máquinas de Turing: a habilidade de realizar múltiplas escolhas durante os passos de sua computação. A única diferença entre uma máquina de Turing não-determinística (MTND) e uma máquina usual (veja a definição 1.1) é que a MTND possui duas funções de transição δ_0 e δ_1, além de um estado especial denotado por $q_{aceitar}$. Durante cada passo da computação de uma MTND M, uma das duas funções de transição é aplicada de forma arbitrária. Dada uma entrada x, dizemos que $M(x) = 1$ se existe alguma sequência de aplicação das funções de transição δ_0 e δ_1 que faz M entrar no estado $q_{aceitar}$. Caso contrário, dizemos que $M(x) = 0$. Dizemos que uma MTND M decide a linguagem L quando $x \in L$ se e somente se $M(x) = 1$. Finalmente, dizemos que a MTND M termina sua computação em k passos com uma entrada $x \in \{0, 1\}^\star$ se existe uma sequência de

escolhas não-determinísticas para aplicação das funções de transição de M em que o número de passos executados por M é k, e em nenhuma outra sequência de escolhas M executa por mais que k passos.

Definição 2.16. [Classe de Complexidade NTIME]. *Seja $T : \mathbb{N} \to \mathbb{N}$ uma função. Uma linguagem $L \subseteq \{0,1\}^\star$ pertence a classe* NTIME$(T(n))$ *se existe uma máquina de Turing não-determinística M que decide L em tempo $O(T(n))$.*

A classe NP foi inicialmente definida através do uso de máquinas de Turing não-determinísticas. Enquanto a sigla P expressa tempo polinomial, a sigla NP tem origem em tempo polinomial não-determinístico.

Teorema 2.17. NP $= \bigcup_{k\geq 1}$NTIME(n^k).

Demonstração. Duas ideias simples são utilizadas na demonstração desse resultado. Em primeiro lugar, uma MTND é capaz de gerar em tempo polinomial qualquer certificado possível a partir das suas escolhas não-determinísticas e então verificar a presença de uma palavra na linguagem. Similarmente, a sequência de escolhas das funções de transição que levam ao estado $q_{aceitar}$ pode ser usada como um certificado.

Suponha que $L \in$ NP. Seja V um verificador para L, $p(x)$ o polinômio que descreve o tamanho dos certificados e assuma que V computa em tempo $O(q(n))$ para algum polinômio $q(x)$. Considere a seguinte MTND M. Com entrada x, M faz $p(|x|)$ escolhas arbitrárias de uso das funções de transição δ_0 e δ_1, imprimindo em um espaço auxiliar da sua fita os valores 0 e 1, respectivamente. Através disso, M gera um possível certificado w de tamanho $p(|x|)$ para a entrada x. Após essa primeira etapa, M simula através de uma computação determinística (ou seja, as duas funções de transição coincidem) o processamento de V com a entrada $\langle x, w\rangle$. M entra no estado $q_{aceitar}$ se e somente se $V(x,w) = 1$. Como V é um verificador para L, temos que $x \in L$ se e somente se $M(x) = 1$. A complexidade total de M é $O(p(n) + q(n))$, ou seja, existe $k \in \mathbb{N}$ tal que $L \in$ NTIME(n^k).

Suponha que $L \in$ NTIME(n^k) para algum inteiro positivo k. Seja M uma MTND que decide L em tempo $O(n^k)$. Vamos mostrar que existe um verificador eficiente V para L que trabalha com certificados de tamanho $O(n^k)$. Com entrada $\langle x, w\rangle$, V simula a execução de M com entrada x aplicando no i-ésimo passo da simulação a função de transição indicada pelo i-ésimo bit do certificado w, e aceita a entrada $\langle x, w\rangle$ se e somente se $M(x) = 1$ com essas escolhas. Segue imediatamente da definição de aceitação de uma palavra por uma MTND que V é um verificador eficiente para L, ou seja, $L \in$ NP. □

É claro que o uso de não-determinismo permite que diversos problemas sejam resolvidos de forma muito mais simples. A grande pergunta é se esse recurso adicional pode

nos levar a um salto significativo de poder computacional. Em outras palavras, o problema P vs NP pergunta se máquinas eficientes determinísticas e não-determinísticas são capazes de resolver os mesmos problemas. Pelo menos no caso mais trivial envolvendo determinismo e não-determinismo, sabemos provar que computações não-determinísticas são mais poderosas.

Teorema 2.18. $\mathrm{DTIME}(n) \subsetneq \mathrm{NTIME}(n)$.

Demonstração. A prova desse resultado pode ser obtida em Paul et al. [89]. □

2.5 Reduções e Problemas Completos

Ao final da década de sessenta, uma grande classe de problemas computacionais para os quais ninguém conhecia algoritmo eficiente foi identificada. A maioria deles são problemas de otimização como empacotamento ótimo, agendamento de horários e o problema do caixeiro viajante. O que todos possuem em comum é uma grande quantidade de soluções possíveis e nenhum método conhecido para buscar a solução ótima a não ser essencialmente a busca exaustiva. Após grande quantidade de tempo e esforço despendidos na tentativa de encontrar algoritmos eficientes para esses problemas, começou-se a suspeitar que tais algoritmos pudessem não existir.

Apesar disso, não havia motivos aparentes para a existência de qualquer relação entre esses problemas, ou porque um deveria ser "mais difícil" do que o outro. A teoria de NP-completude, introduzida por Cook [26] em 1971, trouxe exatamente a evidência necessária. O que Cook mostrou em seu artigo foi que a existência de um algoritmo eficiente para qualquer um desses problemas garantia a existência de algoritmos eficientes para todos os outros problemas. Diversos problemas naturais possuem essa propriedade e são chamados de problemas NP-completos. Um ano mais tarde, Karp [66] utilizou os resultados de Cook para incluir vinte novos problemas na lista de problemas NP-completos, demonstrando a importância do conceito descoberto. Mais tarde centenas de outros problemas NP-completos foram identificados. Muitos desses são de extrema importância industrial, como aqueles de roteamento e agendamento. Para entender esses resultados é preciso estudar o conceito de redução entre problemas computacionais.

A noção de redução entre problemas é amplamente utilizada em computação. Em geral, uma liguagem L_1 pode ser reduzida à linguagem L_2 se a existência de um algoritmo A_2 que decide L_2 pode ser usada para a criação de um algoritmo A_1 capaz de decidir L_1. Existem diversas maneiras de formalizar o conceito de redução. A forma de redução mais abrangente é aquela em que permitimos que o algoritmo A_1 invoque diversas vezes o algoritmo A_2 como subrotina.

Suponha que uma linguagem L_1 seja redutível a uma linguagem L_2. Se resolvemos o problema de decisão L_2, então também sabemos resolver o problema de decisão L_1. Por isso, podemos considerar que L_2 é pelo menos tão difícil quanto L_1. Em complexidade computacional estamos interessados principalmente em reduções eficientes, ou seja, computadas em tempo polinomial.

Suponha que $\mathrm{P} \neq \mathrm{NP}$. Para provarmos isso, precisamos exibir uma linguagem L em NP que não está em P. Se for possível determinar quais são as linguagens mais difíceis em NP, então essas serão boas candidatas para esse papel. De modo surpreendente, o que Cook provou no seu artigo é que existem problemas em NP que são tão difíceis quanto qualquer outro problema em NP. Em outras palavras, existe uma linguagem L em NP tal que qualquer outra linguagem em NP se reduz à L. Além do mais, uma noção de redução mais restrita é suficiente para provar esse resultado.

Definição 2.19. [Redução de Karp]. *Uma linguagem $L \subseteq \{0,1\}^\star$ é Karp-redutível a uma linguagem $L' \subseteq \{0,1\}^\star$ (denotado por $L \leq_p L'$) se existe uma função $f : \{0,1\}^\star \to \{0,1\}^\star$ computável em tempo polinomial tal que para todo $x \in \{0,1\}^\star$, $x \in L$ se e somente se $f(x) \in L'$.*

Como a composição de polinômios é novamente um polinômio, temos que a redução de Karp é uma relação transitiva entre pares de linguagens. Além disso, é fácil verificar que se $L \leq_p L'$ e $L' \in \mathrm{P}$, então $L \in \mathrm{P}$.

Definição 2.20. [NP-difícil e NP-completo]. *Dizemos que uma linguagem $L' \subseteq \{0,1\}^\star$ é* NP*-difícil se $L \leq_p L'$ para toda linguagem $L \in \mathrm{NP}$. Dizemos que L' é* NP*-completa se L' é* NP*-difícil e $L' \in \mathrm{NP}$.*

O seguinte resultado confirma o papel fundamental exercido pelos problemas NP-completos.

Teorema 2.21. *Se L é* NP*-difícil e $L \in P$, então* $\mathrm{P} = \mathrm{NP}$.

Demonstração. Seja $L' \in \mathrm{NP}$. Então, por hipótese, $L' \leq_p L$. Como $L \in \mathrm{P}$, segue através da composição do algoritmo eficiente para L e do algoritmo eficiente que computa a redução que $L' \in \mathrm{P}$. Isso prova que $\mathrm{NP} \subseteq \mathrm{P}$ e portanto $\mathrm{P} = \mathrm{NP}$. □

Em particular, se L é uma linguagem NP-completa, então $L \in \mathrm{P}$ se e somente se $\mathrm{P} = \mathrm{NP}$. O próximo teorema prova que existem linguagens com essa propriedade especial.

Teorema 2.22. *Existem linguagens* NP*-completas.*

Demonstração. Primeiro mostramos que existe uma linguagem L' que é NP-difícil. Precisamos provar que para todo $L \in \mathrm{NP}$, temos $L \leq_p L'$. Em outras palavras, temos que

mostrar que existe função $f_L : \{0,1\}^\star \to \{0,1\}^\star$ computável eficientemente tal que $x \in L$ se e somente se $f_L(x) \in L'$. Dada uma linguagem arbitrária $L \in$ NP, o que sabemos sobre ela? Apenas que existe um verificador V_L eficiente para L capaz de testar corretamente certificados de tamanho $p_L(|x|)$. Como precisamos acomodar todas as reduções possíveis utilizando apenas essas informações, considere a seguinte definição para L':

$$L' = \{\langle V, x, 1^k\rangle : \exists w \in \{0,1\}^k \text{ tal que } V(x,w) = 1\}.$$

Observe que $x \in L$ se e somente se $\langle V_L, x, 1^{p_L(|x|)}\rangle \in L'$. Além disso, a função $f_L(x) = \langle V_L, x, 1^{p_L(|x|)}\rangle$ pode ser computada eficientemente. Isso prova que L' é NP-difícil.

O principal impedimento para mostrarmos que L' é NP-completa, ou seja, pertence à classe NP, é o fato de que w não é um certificado eficientemente verificável para L'. Embora w seja sucinto, como a máquina V da instância de entrada é arbitrária, não podemos garantir que a saída de V com entrada $\langle x, w\rangle$ pode ser computada em tempo limitado por algum polinômio fixo. Para superar essa limitação, adicionamos um novo ingrediente à definição de L':

$$L' = \{\langle V, x, 1^k, 1^t\rangle : \exists w \in \{0,1\}^k \text{ tal que } V(x,w) = 1 \text{ em no máximo } t \text{ passos }\}.$$

Seja $q_L(x)$ o polinômio que limita o tempo de execução de V_L. Então $x \in L$ se e somente se $\langle V_L, x, 1^{p_L(|x|)}, 1^{q_L(|x|)}\rangle \in L'$. Além disso, a nova redução também pode ser computada em tempo polinomial. Portanto, L' permanece NP-difícil. Por outro lado, temos agora que w é um certificado sucinto e eficientemente checável para a presença de uma palavra em L'. Isso demonstra que $L' \in$ NP, ou seja, L' é uma linguagem NP-completa. □

Embora o problema utilizado na demonstração anterior seja importante conceitualmente, ele não é muito natural. Um segundo fato surpreendente é que existem problemas NP-completos bastante naturais e de interesse prático. Na verdade, atualmente são conhecidos milhares de problemas com essa propriedade (veja a seção de referências adicionais deste capítulo). O mais famoso deles é apresentado a seguir.

Definição 2.23. [Linguagem SAT]. *Uma fórmula proposicional φ em forma normal conjuntiva sobre as variáveis $x_1, \ldots, x_n$ consiste em uma expresssão do tipo:*

$$\varphi(x_1, \ldots, x_n) = \bigwedge_i \left(\bigvee_j u_{i_j}\right),$$

onde cada u_{i_j} é uma variável x_k ou a sua negação $\neg x_k$. Uma fórmula desse tipo induz naturalmente uma função booleana $\phi : \{0,1\}^n \to \{0,1\}$ obtida através da aplicação dos conectivos $\wedge$, $\vee$ e $\neg$ aos bits de entrada. Dizemos que uma fórmula é satisfatível se existe $\vec{x} \in \{0,1\}^n$ tal que $\varphi(\vec{x}) = 1$. Denotamos por SAT *o conjunto de fórmulas proposicionais satisfatíveis em forma normal conjuntiva.*

Teorema 2.24. [Teorema de Cook-Levin]. SAT *é uma linguagem* NP*-completa.*

Demonstração. O fato de que SAT $\in$ NP é fácil de ser provado, pois os certificados canônicos são as próprias sequências de bits que tornam verdadeira a fórmula de entrada. A demonstração de que SAT é NP-difícil também é conceitualmente simples. A principal dificuldade é descrever precisamente os detalhes técnicos da prova. Vamos apenas destacar os pontos importantes da demonstração.

Suponha que, para toda máquina de Turing M e para toda entrada x de tamanho n, exista uma fórmula proposicional $\varphi_{M,n}$ tal que $M(x) = 1$ se e somente se $\varphi_{M,n}(x) = 1$. Além disso, caso M seja uma máquina eficiente, assuma que a fórmula $\varphi_{M,n}$ pode ser computada em tempo polinomial. Seja $L \in$ NP e V um verificador para essa linguagem que computa em tempo polinomial. Sabemos que $x \in L$ se e somente se $V(x, w) = 1$ para algum certificado w adequado. Note que as hipóteses anteriores garantem que podemos criar uma fórmula $\varphi_{V,x}$ tal que $V(x, w) = 1$ se e somente se $\varphi_{V,x}(w) = 1$. Isso pode ser feito em tempo polinomial e portanto $L \leq_p$ SAT.

Embora o argumento anterior possa ser usado para provar que um problema similar envolvendo circuitos booleanos é NP-difícil, existe uma dificuldade adicional quando lidamos com fórmulas proposicionais. Ao contrário do que ocorre com os circuitos, as fórmulas não podem reaproveitar "computações parciais". Por isso, não está claro se é possível criar uma fórmula de tamanho polinomial que simula a computação de um algoritmo. No entanto, fórmulas proposicionais são suficientemente expressivas para verificar todas as *condições locais* de consistência em uma tabela que descreve a computação de M com entrada x. Por isso, obtemos para toda máquina de Turing M e para toda entrada x de tamanho n uma fórmula proposicional $\varphi_{M,n}$ tal que $M(x) = 1$ se e somente se $\varphi_{M,n}(C_x) = 1$, onde C_x é uma tabela que descreve consistentemente a computação de M com entrada x. Portanto, $M(x) = 1$ se e somente se a fórmula $\varphi_{M,n}$ é satisfatível. Como o verificador utilizado na redução é uma máquina eficiente, a tabela que descreve sua computação tem tamanho polinomial, assim como a fórmula proposicional resultante. Isso completa a demontração de que SAT é NP-difícil.

A redução anterior possui algumas propriedades adicionais muito interessantes. Primeiro, o número de certificados aceitos pelo verificador original é igual ao número de atribuições que satisfazem a fórmula proposicional obtida. Além disso, toda atribuição satisfatível pode ser convertida eficientemente em um certificado aceito pelo verificador original, e vice-versa.

Os detalhes técnicos da prova residem no modo de construção da fórmula proposicional a partir da máquina de Turing. Uma descrição detalhada pode ser obtida no livro de Sipser [103]. Uma técnica um pouco mais eficiente pode ser usada para diminuir o tamanho da fórmula proposicional obtida. Veja essa demonstração no livro de Arora e Barak [7]. □

Devido à transitividade da redução de Karp e ao teorema anterior, para provar que um problema arbitrário L é NP-completo, basta mostrar que $L \in$ NP e SAT $\leq_p L$. Um outro

exemplo de problema NP-completo é a versão generalizada do quebra-cabeças Sudoku (veja Yato e Seta [115]). Devido ao teorema 2.21, se P $\neq$ NP nenhum desses problemas admite algoritmos eficientes.

Embora a teoria de NP-completude tenha obtido resultados surpreendentes e tenha apontado novas direções frutíferas de pesquisa, questões muito simples ainda permanecem em aberto.

Questão em Aberto 2.25. SAT $\in$ DTIME(n)*?*

Entretanto, somos capazes de provar que uma outra classe interessante de algoritmos não é capaz de resolver SAT (veja a seção 4.3).

2.6 Importância Matemática

No início do século passado, descobriu-se que o uso intuitivo e indiscriminado de diversas noções matemáticas básicas poderia levar ao surgimento de inconsistências lógicas. Esse fato inusitado provocou um enorme desenvolvimento no estudo da matemática, e diversos sistemas formais foram sugeridos na tentativa de construir a matemática a partir de bases mais seguras. Dentre os diversos sistemas axiomáticos formais propostos, um dos mais importantes é a teoria de conjuntos conhecida como ZFC (Teoria de Conjuntos "Zermelo-Fraenkel with Choice").

Essa teoria possui três propriedades muito interessantes. Em primeiro lugar, todos os teoremas matemáticos usuais podem ser provados a partir dos axiomas e regras do sistema ZFC (veja Cohen e Davis [23]). Além disso, a linguagem utilizada na demonstração de teoremas a partir dos axiomas dessa teoria é completamente formalizada. Por último, no sistema ZFC podemos verificar a validade de uma prova em tempo polinomial no tamanho da demonstração. Essas propriedades garantem que um algoritmo ou computador pode ser utilizado para o estudo de diversas questões relacionadas com esse sistema e com a matemática em geral.

Em virtude disso, um matemático muito importante em sua época, chamado David Hilbert, estava interessado na criação de um algoritmo capaz de verificar automaticamente se uma dada afirmação matemática é um teorema. Essa questão ficou conhecida como o *Entscheidungsproblem*, ou seja, o problema de decisão. Para o espanto de todos, Alan Turing provou em 1936 [107] que não existe algoritmo para resolver este problema.

Apesar desse resultado de impossibilidade, a complexidade computacional inspirou o estudo de uma versão moderna desse problema. Considere a seguinte linguagem:

$$\text{MATEMÁTICA} = \{\langle \psi, 1^n \rangle : \text{existe uma prova de } \psi \text{ em ZFC de tamanho} \leq n\}.$$

Essa linguagem, além de ser decidível (basta testar todas as provas possíveis de tamanho menor ou igual a n), também pertence à classe de complexidade NP. Isso ocorre pois uma demonstração de tamanho $k \leq n$ para o teorema ψ serve como certificado para sua presença na linguagem MATEMÁTICA (lembre que podemos verificar eficientemente se uma demonstração é válida). Além disso, é possível provar que esse problema é NP-completo. Isso segue do fato de que uma fórmula proposicional é satisfatível se e somente se existe uma demonstração sucinta de sua satisfatibilidade no sistema ZFC.

Suponha que exista um algoritmo eficiente para o problema MATEMÁTICA. Então, para descobrirmos se somos capazes de demonstrar um teorema ψ basta executar esse algoritmo com a entrada $\langle \psi, 1^n \rangle$ para um inteiro n suficientemente grande (existe uma relação polinomial entre o tamanho de uma demonstração matemática usual e o tamanho de uma demonstração formal dentro do sistema ZFC). Se a resposta do algoritmo for negativa, então não precisamos nem perder nosso tempo à procura de uma demonstração para ψ (se existe alguma demonstração para esse resultado, ela é tão grande que seria humanamente impossível encontrá-la). Além disso, como essa linguagem é NP-completa, a existência de um algoritmo eficiente para esse problema implica que P = NP. Pelo teorema 2.15, segue que também seremos capazes de encontrar a demonstração procurada, caso ela exista.

Por isso, podemos dizer que P = NP se e somente se o trabalho de um matemático pode ser eficientemente mecanizado. Em outras palavras, a menos que essas duas classes coincidam, a criatividade é essencial no processo de construção de uma prova matemática.

2.7 Principais Abordagens

Suponha que P = NP e considere como isso poderia ser demonstrado. A maneira mais natural seria exibir um algoritmo polinomial para a linguagem SAT ou para algum outro problema NP-completo. Existe uma série de métodos algorítmicos que podem ser utilizados para isso. As principais técnicas podem ser encontradas em Cormen et al. [29]. Devido à grande importância industrial de diversos problemas NP-completos, um imenso número de programadores e engenheiros tentaram encontrar algoritmos eficientes para esses problemas durante as últimas décadas, mas não obtiveram sucesso.

Um método algorítmico interessante é a modelagem de problemas computacionais através de programação linear. Como esse último problema pode ser resolvido em tempo polinomial, se for possível reduzir eficientemente algum problema NP-completo à um problema de programação linear, ficará provado que P = NP. No entanto, Yannakakis [114] mostrou que um importante problema NP-completo conhecido como TSP (problema do caixeiro viajante) não possui formulação eficiente como um problema de programação linear com certas restrições. Isso ilustra o fato de que é possível demonstrar que certas

abordagens não são suficientes para provar que P = NP.

Por outro lado, caso P $\neq$ NP, existem muitos métodos que foram propostos na tentativa de provar esse resultado. Alguns deles foram amplamente estudados e algumas limitações importantes também foram descobertas. Discutimos a seguir os casos mais interessantes.

Simulação e Diagonalização

O método de diagonalização teve origem com a demonstração de Cantor de que o conjunto de números reais não é enumerável. Adaptado à teoria da computação, esse método foi utilizado inicialmente para provar que certos problemas computacionais são indecidíveis. Em complexidade, um argumento envolvendo simulação e diagonalização pode ser aplicado para demonstrar que existem problemas computacionais arbitrariamente difíceis. Veremos como isso pode ser feito no capítulo 3.

Além disso, a diagonalização pode ser usada para provar limitantes inferiores superexponencias na complexidade de alguns problemas importantes (veja por exemplo o artigo de Fischer e Rabin [36]). Apesar dessa técnica ser extremamente poderosa, há evidências fortes de que ela sozinha não é suficiente para separar as classes de complexidade P e NP. Ainda que a técnica de diagonalização possua essa importante limitação, quando combinada com outros métodos, ela é capaz de provar resultados interessantes em complexidade computacional. Vamos discutir essa questão em profundidade no capítulo 5.

Complexidade de Circuitos

Como veremos posteriormente, a relativização das técnicas baseadas apenas em simulação e diagonalização indica que devemos de fato analisar as computações envolvidas, e não apenas simulá-las. Alguns resultados importantes provados na década de oitenta tornaram interessante o estudo de certos problemas através do uso de circuitos booleanos.

Na complexidade de circuitos, as funções são classificadas de acordo com o tamanho (quantidade de portas lógicas) e a profundidade dos circuitos booleanos capazes de computá-las. Um aspecto interessante desse modelo computacional de computação é a falta de uniformidade. Isso significa que, ao contrário das máquinas de Turing, entradas com tamanhos diferentes são processadas por circuitos booleanos diferentes.

O problema P vs NP é abordado através da complexidade de circuitos da seguinte maneira. Em primeiro lugar, para toda linguagem em P existe uma família de circuitos booleanos com uma quantidade polinomial de portas lógicas (em função do tamanho da entrada) capaz de decidí-la. Por outro lado, há fortes evidências de que os problemas NP-completos não podem ser computados por famílias de circuitos de tamanho polino-

mial. Como circuitos booleanos são muito mais simples que máquinas de Turing, diversos métodos combinatórios e probabilísticos podem ser utilizados na tentativa de provar que alguma linguagem NP-completa não admite circuitos eficientes.

Embora essa abordagem tenha tido algum sucesso com modelos mais restritos de circuitos booleanos, os melhores resultados obtidos para o caso geral são muito fracos. Assim como ocorre com a diagonalização, também é possível justificar parte do fracasso dessa abordagem. Discutiremos novamente essa questão na seção 5.8.

Métodos Algébricos

Após a descoberta das barreiras descritas anteriormente, um novo método mostrou-se interessante para o estudo da relação entre classes de complexidade. Uma técnica baseada na aritmetização de fórmulas lógicas foi capaz de provar resultados importantes, como a caracterização de classes[3] IP = PSPACE [101]. Além disso, foram descobertas demonstrações que ultrapassam simultaneamente as barreiras envolvendo os métodos anteriores.

Caracterização Lógica das Classes de Complexidade

A questão P vs NP é equivalente a um problema de expressividade envolvendo duas classes distintas de expressões lógicas. Todas as linguagens em P podem ser representadas por meio de fórmulas lógicas de primeira ordem envolvendo um operador adicional de ponto fixo. Similarmente, as linguagens em NP são capturadas pelas expressões lógicas em linguagem de segunda-ordem existencial. Portanto, o problema P vs NP é equivalente à seguinte questão: a lógica de segunda-ordem existencial é capaz de descrever mais linguagens do que a lógica de primeira ordem com o operador de ponto fixo? Um ponto forte dessa abordagem é a disponibilidade de métodos bastante estudados proveninentes da lógica matemática.

GCT - "Geometric Complexity Theory"

Esse é um método recente proposto por Mulmuney e Sohoni em uma série de artigos publicados nos últimos anos. Esses autores argumentam que diversas dificuldades enfrentadas pelos métodos anteriores podem ser formalmente analisadas e superadas através da geometria algébrica e da teoria de representações.

Em alguns casos, a prova de um limitante superior implica na demonstração de um

[3]Para uma definição da classe probabilística IP, recomendamos a leitura do livro de Goldreich [47]. A classe PSPACE contém as linguagens que podem ser decididas em espaço polinomial. Veja o capítulo 3 para mais detalhes.

limitante inferior. Por exemplo, um resultado famoso de Kannan [64] mostra que, para todo inteiro k, existem linguagens na hierarquia polinomial (definida na seção 4.1) com complexidade de circuito $\Omega(n^k)$. Por outro lado, é fácil provar que se P = NP então a hierarquia polinomial coincide com a classe P (veja a seção 4.1). Por isso, basta demonstrar que existe um inteiro k' tal que toda linguagem em P possui circuitos de tamanho $O(n^{k'})$ para mostrar que P $\neq$ NP. Esse exemplo mostra que se os problemas em P são fáceis demais, então há problemas em NP que são difíceis.

A principal ideia por trás da abordagem GCT é baseada em um argumento similar, porém aplicado em um contexto diferente. Essencialmente, para provar que P $\neq$ NP basta demonstrar que diversos problemas de decisão em geometria algébrica e teoria de representações percentem à classe P. A abordagem utiliza técnicas matemáticas bastante avançadas e os autores admitem que o desenvolvimento completo desse projeto pode levar décadas. Veja a seção de referências deste capítulo para mais detalhes.

2.8 Resultados Básicos

Nesta seção introduzimos duas novas classes de complexidade e relacionamos através de alguns resultados simples todas as classes discutidas até agora.

Lembre que o problema P vs NP pode ser enunciado como o estudo da relação entre máquinas eficientes determinísticas e não-determiníticas. Para entendermos melhor a diferença entre determinismo e não-determinismo é interessante o estudo desses conceitos em máquinas com muito mais poder computacional. Para isso, introduzimos a seguir as classes EXP e NEXP.

Definição 2.26. [Classe de Complexidade EXP].
$\mathrm{EXP} = \bigcup_{k\geq 1}\mathrm{DTIME}(2^{n^k})$.

Definição 2.27. [Classe de Complexidade NEXP].
$\mathrm{NEXP} = \bigcup_{k\geq 1}\mathrm{NTIME}(2^{n^k})$.

Primeiro, enunciamos um fato muito simples envolvendo as novas classes.

Lema 2.28. $\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP} \subseteq \mathrm{NEXP}$.

Demonstração. A primeira inclusão foi provado no lema 2.13 e a última inclusão é imediata. Suponha que $L \in \mathrm{NP}$ e seja V um verificador para L de complexidade $O(n^c)$. Além disso, considere que os certificados aceitos por V possuam tamanho $O(n^d)$. Dada uma entrada $x \in \{0,1\}^\star$, podemos decidir se $x \in L$ deterministicamente enumerando todos os $O(2^{n^d})$ certificados possíveis e utilizando a máquina V para checar se alguns deles é um certificado válido para x. Isso pode ser feito em tempo determinístico $O(2^{n^d}n^d)$ e, portanto, em tempo determinístico $O(2^{n^{d+1}})$. Logo, temos $L \in \mathrm{EXP}$ e portanto $\mathrm{NP} \subseteq \mathrm{EXP}$. $\square$

De forma análoga ao problema P vs NP, temos a seguinte conjectura.

Questão em Aberto 2.29. EXP = NEXP *?*

As conjecturas envolvendo o poder do não-determinismo em computação podem ser relacionadas através do seguinte teorema.

Teorema 2.30. *Se* P = NP *então* EXP = NEXP.

Demonstração. Utilizaremos um argumento muito útil em complexidade computacional que é conhecido como *método do preenchimento*. Basicamente, adicionamos diversos bits redundantes às palavras que representam as instâncias de uma linguagem para reduzirmos sua complexidade total.

Suponha que $L \in$ NEXP, ou seja, existe uma MTND M que decide L em no máximo $k'2^{n^k}$ passos. Temos então que a linguagem

$$L_{preenchida} = \{\langle x, 1^{k'2^{|x|^k}} \rangle : x \in L\}$$

está em NP, uma vez que a seguinte MTND M' decide $L_{preenchida}$ eficientemente: dado $y \in \{0,1\}^\star$, primeiro M' verifica se $y = \langle z, 1^{k'2^{|z|^k}} \rangle$ para alguma palavra z. Se isso não ocorre, M' rejeita y. Caso contrário, M' simula a máquina M com entrada z por até $k'2^{|z|^k}$ passos e aceita ou rejeita a entrada y de acordo com a decisão de M. Devido ao nosso preenchimento, temos que M' computa em tempo polinomial no tamanho de y, ou seja, $L_{preenchida} \in$ NP. Porém, por hipótese, temos que P = NP e portanto $L_{preenchida} \in$ P. Utilizando uma máquina determinística de tempo polinomial que decide $L_{preenchida}$ podemos decidir L em tempo determinístico exponencial: basta adicionar os bits de preenchimento e aplicar a máquina eficiente de $L_{preenchida}$. Isso prova que NEXP $\subseteq$ EXP e portanto EXP = NEXP. □

Observe que para provar que P $\neq$ NP basta demonstrar que EXP $\neq$ NEXP.

O resultado anterior mostra que, para alguns recursos computacionais, a igualdade entre classes de complexidade inferiores pode ser estendida às classes de complexidade superiores. É possível demonstrar uma versão mais geral do teorema 2.30, válida tanto para tempo quanto para espaço. Veja o livro de Papadimitriou [88] para mais detalhes. Veremos no capítulo 3 que muitos resultados desse tipo podem ser unificados através de uma noção mais abrangente de complexidade de tempo e espaço.

Embora alguns problemas discutidos anteriormente permaneçam em aberto, provaremos no próximo capítulo que P $\neq$ EXP e NP $\neq$ NEXP.

2.9 Referências Adicionais

Os primeiros textos que consideram a questão da eficiência dos algoritmos são os artigos de Cobham [22] e Edmonds [34]. Em particular, o artigo de Edmonds demonstra a existência de um algoritmo eficiente para o problema do emparelhamento em grafos arbitrários. Diversos algoritmos para o problema do caminho mínimo são descritos no livro de Cormen et al. [29].

O enunciado do problema P vs NP pode ser obtido no artigo de Cook [28]. Historicamente, um enunciado equivalente desse problema apareceu pela primeira vez em uma carta de Kurt Gödel enviada para Von Neumann em 1956. Basicamente, Gödel indagou se existe algum algoritmo eficiente para o problema computacional NP-completo discutido na seção 2.6, notando que esse fato teria consequências fantásticas para a matemática. O conteúdo completo da carta está disponível no artigo de Sipser [102].

O algoritmo de primalidade AKS foi desenvolvido por Agrawal, Kayal e Saxena [6]. Ele foi o primeiro algoritmo de primalidade proposto que é ao mesmo tempo geral, determinístico, eficiente e incondicional. Uma descrição informal das principais ideias utilizadas no algoritmo AKS pode ser encontrada no artigo de Aaronson [3]. Uma referência em português sobre esse teste de primalidade é o livro de Coutinho [30]. Antes do desenvolvimento do algoritmo AKS, Miller [81] apresentou um algoritmo eficiente para o mesmo problema assumindo a hipótese de Riemann generalizada.

A questão NP vs coNP está intimamente relacionada com uma área de pesquisa chamada complexidade de prova. O principal objetivo dessa disciplina é demonstrar que não existem provas sucintas e facilmente verificáveis de que uma fórmula proposicional é uma tautologia. Isso separaria as classes NP e coNP, provando que $P \neq NP$. Uma referência recente sobre o tema é o livro de Cook e Nguyen [25].

A seção 2.4 é baseada no livro de Arora e Barak [7] e a seção 2.6 é motivada por uma discussão apresentada no livro de Immerman [61].

A descoberta da existência de problemas NP-completos foi feita independentemente por Cook [26] e Levin [74]. O livro de Garey e Johnson [45] é a referência clássica sobre problemas NP-completos.

O artigo de Sipser e Boppana [15], embora não seja muito recente, ainda descreve os pricipais resultados obtidos em complexidade de circuitos. O livro de Immerman [61] é inteiramente dedicado à relação entre lógica e complexidade computacional. A abordagem GCT para o problema P vs NP está resumida no artigo de Regan [91]. Na página pessoal de Ketan Mulmuney há uma série de artigos sobre o tema.

Ao longo de décadas diversas pessoas afirmaram ter uma solução para o problema P vs NP. Veja o site [112] para mais detalhes.

O artigo de Aaronson [4] ilustra algumas consequências interessantes para a física se

assumirmos que os problemas NP-completos não podem ser resolvidos de forma eficiente no universo físico. Veja também o site [1] mantido pelo mesmo autor para uma descrição de centenas de classes de complexidade estudadas em complexidade computacional.

Capítulo 3

Simulação e Diagonalização

"Time is the most valuable thing a man can spend."
Theophrastus.

Intuitivamente, esperamos que com mais recursos computacionais seja possível resolver mais problemas. Os teoremas de hierarquia de tempo e de espaço, alguns dos resultados mais importantes provados em complexidade computacional, estabelecem exatamente isso. O argumento utilizado na prova desses teoremas é conhecido como método da diagonalização. Neste capítulo vamos estudar essa técnica em profundidade. Veremos também como generalizar e unificar a demonstração dos teoremas de hierarquia e de outros resultados importantes em complexidade computacional.

3.1 Introdução

Em um artigo muito importante para a teoria de complexidade computacional, Hartmanis e Stearn [54] provaram que é possível resolver mais problemas computacionais quando se tem mais tempo disponível. Em outras palavras, eles demonstraram a existência de problemas computacionais decidíveis porém arbitrariamente difíceis. Por exemplo, existem linguagens que até podem ser decididas em tempo polinomial, mas nunca em tempo $O(n^3)$. Similarmente, pode-se provar que o mesmo fenômeno ocorre quando consideramos a complexidade de espaço ao invés da complexidade de tempo [104]. Esses resultados ficaram conhecidos como teoremas de hierarquia.

O principal argumento utilizado na demonstração desses resultados é uma combinação de simulação e diagonalização. Embora essas idéias não sejam novas, esse é o único

método conhecido capaz de separar certas classes de complexidade importantes. Por isso, entender os pontos fortes e os limites da diagonalização é fundamental para o estudo de complexidade computacional.

Antes de provarmos uma forma bastante geral do teorema da hierarquia, vamos melhorar nossa intuição através de um exemplo mais simples. Se necessário, revise a definição de complexidade de espaço apresentada na seção 1.7.

Definição 3.1. [Classe de Complexidade DSPACE]. *Seja* $S : \mathbb{N} \to \mathbb{N}$ *uma função. Uma linguagem* $L \subseteq \{0,1\}^\star$ *pertence a classe* $\mathrm{DSPACE}(S(n))$ *se existe uma máquina de Turing* M *que decide* L *e que tenha complexidade de espaço* $O(S(n))$.

Teorema 3.2. $\mathrm{DSPACE}(n) \subsetneq \mathrm{DSPACE}(n^3)$.

Demonstração. É imediato que $\mathrm{DSPACE}(n) \subseteq \mathrm{DSPACE}(n^3)$. Considere agora a seguinte linguagem:

$$L = \{\langle M\rangle : M \text{ é uma máquina de Turing que aceita } \langle M\rangle \text{ e } s_M(\langle M\rangle) \leq |\langle M\rangle|^{1.5}\}.$$

Por conveniência, na definição da linguagem L utilizamos a própria descrição da máquina de Turing como entrada para sua computação. Observe que, permitindo entradas arbitrárias para cada máquina de Turing, podemos obter uma linguagem L' capaz de codificar as instâncias de todos os problemas que podem ser resolvidos em tempo $n^{1.2}$, por exemplo (no sentido da demontração do teorema 2.22). Isso explica, de certa forma, porque L é uma linguagem tão difícil para computações que utilizam espaço linear. Vamos dividir a demonstração desse teorema em dois lemas.

Lema 3.3. $L \in \mathrm{DSPACE}(n^3)$.

Demonstração. A prova deste lema ilustra o conceito de simulação entre máquinas de Turing. Precisamos mostrar que existe uma máquina de Turing U satisfazendo as seguintes propriedades: (1) U aceita a palavra $\langle M\rangle$ se e somente se M aceita seu próprio código utilizando no máximo $|\langle M\rangle|^{1.5}$ células de memória; (2) U utiliza espaço $O(n^3)$, onde $n = |\langle M\rangle|$ é o tamanho da palavra de entrada; (3) U sempre termina sua computação.

Para simular a computação de M, a máquina U utiliza três fitas de memória. A primeira fita de U armazena a entrada $\langle M\rangle$. Na segunda fita, U guarda o estado atual de M durante a simulação e o número de células de memória acessadas por M até o momento. Por último, a terceira fita de U armazena de forma concatenada o conteúdo das fitas de memória de M durante a sua computação. Símbolos especiais são utilizados na terceira fita para indicar a posição de cada cabeça de leitura de M durante a simulação. No início da simulação, U copia a palavra de entrada para o início da terceira fita, de modo que a palavra $\langle M\rangle$ sirva como entrada para a máquina M.

Para simular um passo de M, a máquina U verifica o símbolo lido por cada cabeça de leitura de M (fita 3), o estado atual de M (fita 2) e sua tabela de transição (fita 1). Informações auxiliares utilizadas por U para processar essas informações são mantidas na segunda fita. A máquina U atualiza sua terceira fita de modo a refletir o conteúdo das fitas da máquina M após cada passo da simulação. Se necessário, algumas fitas de M são deslocadas para a direita na terceira fita de U para garantir espaço para outras fitas. U atualiza as informações na fita 2 e prossegue para o próximo passo da simulação. Se em algum momento M aceita ou rejeita seu próprio código, U toma a mesma decisão. Caso M acesse mais do que $n^{1.5}$ células de memória, a máquina U rejeita a entrada $\langle M \rangle$. Claramente, com essa descrição temos que U satisfaz o primeiro requisito desejado.

A primeira fita de U utiliza durante a computação n células de memória. A terceira fita não utiliza mais do que $n^{1.5}$ células, pois caso esse limite seja ultrapassado, a simulação é interrompida e U rejeita a entrada $\langle M \rangle$. Por último, o contador de células de memória utilizadas por M e os dados auxiliares armazenados na segunda fita não utilizam mais do que $c_1 n$ células de memória para alguma constante c_1 que depende dos detalhes de implementação da máquina U. Isso prova que U utiliza no máximo $O(n^{1.5})$ células de memória.

No entanto, observe que a máquina simulada pode executar indefinidamente enquanto utiliza uma quantidade finita de memória. Por isso, a descrição atual da máquina U não satisfaz o requisito (3). Veremos que não é difícil descobrir quando essa situação ocorre. Lembre que a descrição de M tem tamanho n. Durante a simulação: (1) M pode utilizar no máximo $n^{1.5}$ células de memória; (2) M possui no máximo n fitas de leitura; (3) o número máximo de estados de M e de símbolos no seu alfabeto é n. Observe que essas informações determinam completamente o estado atual da computação de M (também chamado de configuração de M). Além do mais, combinando o conteúdo das fitas de M, o seu estado atual e a posição de cada cabeça de leitura, temos que existem no máximo $(n^{nn^{1.5}})(n)(n^{1.5})^n$ configurações distintas possíveis. Isso significa que, após $n^{n^{2.5}+1.5n+1}$ passos simulados, podemos rejeitar a máquina de entrada, pois nesse caso sabemos que M repete alguma configuração e portanto computa indefinidamente, ou seja, $\langle M \rangle \notin L$. Alteramos a máquina U para que ela conte o número de passos realizados por M (na segunda fita) e verifique quando isso ocorre. O espaço adicional utilizado por U é de no máximo $k \log n^{n^{2.5}+1.5n+1}$ células, para alguma constante k adequada. Ou seja, o espaço total utilizado por M ainda é $O(n^3)$. Isso completa a demonstração de que $L \in \mathrm{DSPACE}(n^3)$. □

Lema 3.4. $L \notin \mathrm{DSPACE}(n)$.

Demonstração. A prova deste lema ilustra o argumento de diagonalização. Suponha que $L \in \mathrm{DSPACE}(n)$, ou seja, existe uma máquina de Turing A que decide L e $S_A(n)$ é $O(n)$.

É fácil perceber que DSPACE(n) é uma classe fechada por complementação de linguagens (basta inverter os estados de aceitação e rejeição da máquina de Turing). Portanto, seja B uma máquina de Turing que decide a linguagem $\overline{L}$ em espaço linear. Suponha que $\langle B \rangle$ seja uma palavra suficientemente grande que represente a máquina B (sempre podemos adicionar tuplas irrelevantes à descrição $\langle B \rangle$). Considere agora a computação de A com entrada $\langle B \rangle$. Temos que A aceita $\langle B \rangle$ se e somente se B aceita $\langle B \rangle$ e $s_B(\langle B \rangle) \leq |\langle B \rangle|^{1.5}$, uma vez que A decide L. Por outro lado, isso ocorre se e somente se B aceita $\langle B \rangle$, pois a condição $s_B(\langle B \rangle) \leq |\langle B \rangle|^{1.5}$ é verdadeira já que B computa em espaço linear e $\langle B \rangle$ é uma palavra suficientemente grande. Finalmente, a última condição ocorre se e somente se A rejeita $\langle B \rangle$, pois A decide L e B decide $\overline{L}$. Essa contradição mostra que $L \notin$ DSPACE(n). □

Isso completa a demonstração do teorema. □

O teorema 3.2 pode ser utilizado para demonstrar que máquinas determinísticas que utilizam espaço linear e máquinas não-determinísticas eficientes não são capazes de resolver os mesmos problemas computacionais.

Definição 3.5. *Uma classe de complexidade $\mathcal{R}$ é fechada por redução de tempo polinomial se $L \in \mathcal{R}$ e $L' \leq_p L$ implicam $L' \in \mathcal{R}$.*

Teorema 3.6. NP $\neq$ DSPACE(n).

Demonstração. Como a composição de uma redução computada em tempo polinomial com uma MTND eficiente é novamente uma MTND eficiente, temos que NP é uma classe fechada por redução de tempo polinomial. Por outro lado, vamos utilizar o teorema 3.2 e o método do preenchimento para provar que DSPACE(n) não possui essa propriedade. Isso prova que NP $\neq$ DSPACE(n).

De acordo com o teorema 3.2, existe uma linguagem $L \in$ DSPACE(n^3) $\setminus$ DSPACE(n). Suponha que L seja decidida por uma máquina de Turing M_L que nunca utiliza mais que kn^3 células de suas fitas. Através de uma simples modificação na máquina M_L (desprezamos a sequência final de 1s), temos que a linguagem $L_{preenchida} = \{\langle w, 1^{k|w|^3} \rangle : w \in L\}$ está em DSPACE(n). Além disso, é imediato que $L \leq_p L_{preenchida}$. Porém, por hipótese, $L \notin$ DSPACE(n). Isso mostra que DSPACE(n) não é uma classe de complexidade fechada por redução de tempo polinomial. □

Embora este possa parecer um resultado muito simples, ainda não sabemos provar qual inclusão entre essas classes é falsa (provavelmente, as duas inclusões não são válidas). O mesmo argumento pode ser utilizado para demonstrar que P $\neq$ DSPACE(n).

Outras duas classes de complexidade importantes são definidas a seguir.

Definição 3.7. [Classe de Complexidade PSPACE].
PSPACE $= \bigcup_{k\geq 1}$DSPACE(n^k).

Definição 3.8. [Classe de Complexidade EXPSPACE].
EXPSPACE $= \bigcup_{k\geq 1}$DSPACE(2^{n^k}).

Lema 3.9. PSPACE $\subseteq$ EXP.

Demonstração. Para demonstrar isso, mostraremos que todo algoritmo determinístico com complexidade de espaço $O(n^b)$ possui complexidade de tempo $O(2^{n^{b+1}})$, onde b é um inteiro positivo qualquer. Seja $L \in$ PSPACE e considere que M é uma máquina de Turing que decide a linguagem L em espaço $k_1 n^{k_2}$. Sejam k_3 o número de estados da máquina M, k_4 o número de fitas de M e k_5 o números de símbolos no seu alfabeto. Antes de qualquer passo de sua computação: (1) M pode estar em algum dos k_3 estados; (2) o número máximo de combinações possíveis para a localização das cabeças de leitura de M é $(k_1 n^{k_2})^{k_4}$; (3) existem no máximo $(k_5)^{k_4 k_1 n^{k_2}}$ possibilidades para o conteúdo escrito nas fitas de M. Por isso, o número máximo de configurações possíveis para M é $k_3(k_1 n^{k_2})^{k_4} k_5^{k_4 k_1 n^{k_2}} \leq c_1 2^{n^{k_2+1}}$, onde c_1 é uma constante que não depende de n. Se M computa com alguma entrada de tamanho n por mais do que $c_1 2^{n^{k_2+1}}$ passos, então M repete alguma configuração anterior, entrando portanto em um laço infinito. Como M sempre termina sua computação (M decide L), isso não é possível. Portanto, M computa em tempo $O(2^{n^{k_2+1}})$, ou seja, $L \in$ EXP. □

Por outro lado, observe que NP $\subseteq$ PSPACE. Isso ocorre pois, dada uma entrada, podemos *reaproveitar o espaço* utilizado por uma máquina de Turing determinística de tempo polinomial para checar todos as escolhas não-determinísticas possíveis de uma MTND de tempo polinomial. Essa é uma vantagem fundamental da complexidade de espaço em relação à complexidade de tempo. Segue através do mesmo argumento que NEXP $\subseteq$ EXPSPACE. De fato, é possível provar através de simulação que espaço é um recurso mais poderoso que tempo.

Teorema 3.10. *Seja* $T(n) : \mathbb{N} \to \mathbb{N}$ *uma função arbitrária.*
Então DTIME$(T(n)) \subseteq$ DSPACE$(T(n)/\log T(n))$.

Demonstração. A prova desse teorema utiliza uma simulação envolvendo blocos de memória e pode ser obtida em Hopcroft et al. [58]. □

Vamos resumir no próximo enunciado o que sabemos sobre as classes de complexidade estudadas até agora.

Teorema 3.11. P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXP $\subseteq$ NEXP $\subseteq$ EXPSPACE.

Demonstração. Vimos na discussão anterior que NP $\subseteq$ PSPACE e NEXP $\subseteq$ EXPSPACE. O resultado segue a partir dos lemas 2.28 e 3.9. □

3.2 Um Teorema Geral de Hierarquia

Nosso próximo passo é provar um teorema geral de hierarquia e obter como corolários os teoremas de hierarquia clássicos demonstrados por Hartmanis e Stearn [54, 104]. Por simplicidade, vamos considerar nesta seção apenas máquinas de Turing determinísticas com uma única fita (veja a seção 1.8). Além disso, assuma que todas as máquinas utilizam o alfabeto $\{0, 1\}$. Veremos na próxima seção como converter esse resultado para máquinas de Turing sem tais restrições.

Primeiro, vamos definir uma noção de complexidade um pouco mais geral do que a usada até agora.

Definição 3.12. [f-complexidade]. *Sejam M uma máquina de Turing e $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ uma função arbitrária. Considere a função $f\text{-ts}_M : \{0, 1\}^\star \to \mathbb{N}$ dada por $f\text{-ts}_M(x) = f(t_M(x), s_M(x))$. Dizemos que a f-complexidade de M é dada pela função $f\text{-TS}_M : \mathbb{N} \to \mathbb{N}$, onde $f\text{-TS}(n) = \max\{\ f\text{-ts}_M(x) : x \in \{0, 1\}^n\ \}$.*

A função f introduz uma nova medida de complexidade baseada nas complexidades de tempo e espaço da máquina de Turing. Claramente, se f é uma das projeções binárias, então a f-complexidade obtida é a complexidade usual de tempo ou espaço.

Definição 3.13. *Uma linguagem L é decidível em f-complexidade $O(g(n))$ se existe uma máquina de Turing M decidindo L tal que $f\text{-TS}_M$ é $O(g(n))$.*

O próximo resultado mostra que, para qualquer f-complexidade adotada, existem problemas arbitrariamente difíceis.

Definição 3.14. *Sejam $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ e $g : \mathbb{N} \to \mathbb{N}$ funções arbitrárias. Definimos a seguinte linguagem:*

$$L_{f,g} = \{\ \langle M\rangle \mid M \text{ é uma máquina de Turing que aceita } \langle M\rangle \text{ e } f\text{-ts}_M(\langle M\rangle) \leq g(|\langle M\rangle|)\ \}. \tag{3.1}$$

Teorema 3.15. *Não existe máquina de Turing que decide $L_{f,g}$ em f-complexidade $o(g(n))$.*

Demonstração. A fim de obtermos uma contradição, suponha que uma máquina de Turing A decida $L_{f,g}$ e que $f\text{-TS}_A(n)$ seja $o(g(n))$. Considere a máquina de Turing B obtida a partir de A através da troca de papéis entre os estados de aceitação e rejeição. Então B aceita w se e somente se A rejeita w, para todo $w \in \{0, 1\}^\star$.

Adicione diversas regras irrelevantes à função de transição de B, obtendo uma máquina de Turing B'. Claramente,

$$L(B') = L(B) = \{0, 1\}^\star \setminus L(A). \tag{3.2}$$

Além disso, existe algum $n_0 \in \mathbb{N}$ tal que $f\text{-TS}_A(n) \leq g(n)$ para todo $n \geq n_0$, uma vez que $f\text{-TS}_A(n)$ é $o(g(n))$. Por isso, temos $f\text{-ts}_A(x) \leq g(n)$ para todo $x \in \{0,1\}^\star$ com $n = |x| \geq n_0$. Porém, é imediato que $f\text{-ts}_A = f\text{-ts}_{B'}$, e portanto

$$f\text{-ts}_{B'}(\langle B' \rangle) \leq g(|\langle B' \rangle|) \tag{3.3}$$

se aumentarmos suficientemente a descrição de B até obtermos $|\langle B' \rangle| \geq n_0$.

Considere agora a computação de A com a entrada $\langle B' \rangle$:

— A aceita $\langle B' \rangle$ *sse* (usando 3.1)

— B' aceita $\langle B' \rangle$ e $f\text{-ts}_{B'}(\langle B' \rangle) \leq g(|\langle B' \rangle|)$ *sse* (usando 3.3)

— B' aceita $\langle B' \rangle$ *sse* (usando 3.2)

— A rejeita $\langle B' \rangle$.

Obtemos uma contradição. Isso prova que não existe máquina de Turing A capaz de decidir $L_{f,g}$ em f-complexidade $o(g(n))$. □

Como a função f pode ser qualquer combinação de tempo e espaço (f pode ser, inclusive, uma função não-computável), o teorema 3.15 prova essencialmente que a melhor máquina de Turing universal possível é aquela que realiza uma simulação passo a passo da máquina de entrada.

Para ilustrar, seja $g(n) = n^3$ e considere a projeção $f(x, y) = x$. Neste caso, a f-complexidade corresponde à complexidade usual de tempo. Por isso, a linguagem $L_{f,g}$ pode ser descrita como o conjunto de palavras que representam máquinas de Turing que aceitam seu próprio código em no máximo n^3 passos. O teorema 3.15 prova que não existe máquina de Turing que decide $L_{f,g}$ em tempo $o(n^3)$.

O próximo resultado mostra que podemos desprezar a complexidade das máquinas de Turing envolvidas se tomarmos f como sendo a função identicamente nula.

Corolário 3.16. [Existência de Problemas Indecidíveis]. *Sejam $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ a função identicamente nula e $g : \mathbb{N} \to \mathbb{N}$ uma função arbitrária. Para f e g tomadas dessa forma, a linguagem $L_{f,g}$ da definição 3.14 é indecidível, i.e., não existe máquina de Turing que decide $L_{f,g}$.*

Demonstração. Seja M uma máquina de Turing que decide $L_{f,g}$. Como f é identicamente nula e g possui imagem não-negativa, temos que $f\text{-TS}_M$ é $o(g(n))$. Por isso, M decide $L_{f,g}$ em f-complexidade $o(g(n))$. No entanto, segue pelo teorema 3.15 que a linguagem $L_{f,g}$ não pode ser decidida em f-complexidade $o(g(n))$. Essa contradição completa a demonstração de que $L_{f,g}$ é uma linguagem indecidível. □

No entanto, se a função f é uma medida de complexidade natural (veja abaixo), então é possível provar que a f-complexidade obtida dá origem a uma hierarquia de linguagens. Podemos obter essa hierarquia calculando a f-complexidade de uma máquina de Turing que decide $L_{f,g}$. Essa linguagem pode ser decidida simulando a máquina M com a entrada $\langle M \rangle$.

Definição 3.17. *Seja $f : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ uma função. Dizemos que f é uma medida de complexidade natural se f satisfaz as seguintes propriedades:* (1) *f é uma função não-decrescente em cada uma de suas coordenadas;* (2) *f é uma função computável;* (3) *para todo par de inteiros t e s, temos que $f(t, s) \geq \min\{t, s\}$.*

Vamos agora enunciar o teorema de hierarquia. Lembre que T_M e S_M denotam a complexidade de tempo e espaço da máquina de Turing M.

Teorema 3.18. [Hierarquia de Espaço-Tempo]. *Sejam f uma medida de complexidade natural e $g : \mathbb{N} \to \mathbb{N}$ uma função computável tal que $g(n) \geq n$. Assuma que as máquinas de Turing M_f e M_g computam as funções f e g, respectivamente. Considere a linguagem $L_{f,g}$ como na definição 3.14. Então $L_{f,g}$ pode ser decidida por uma máquina de Turing A com f-complexidade $O(\ f(T_A(n), S_A(n))\ )$, onde*

$$T_A(n) \leq c_4 \left[T_{M_g}(n) + g(n) \left[T_{M_f}(c_3 g(n)) + S_{M_g}(n) + g(n) + S_{M_f}(c_2 g(n))\right]\right] \quad (3.4)$$

$$S_A(n) \leq c_1 \left[S_{M_g}(n) + g(n) + S_{M_f}(c_2 g(n))\right] \quad (3.5)$$

e $c_i \in \mathbb{N}$ é uma constante, $1 \leq i \leq 4$. Além disso, $L_{f,g}$ não pode ser decidida por uma máquina de Turing com f-complexidade $o(g(n))$.

Demonstração. Em primeiro lugar, segue pelo teorema 3.15 que a linguagem $L_{f,g}$ não pode ser decidida por máquinas de Turing em f-complexidade $o(g(n))$. Por outro lado, a máquina de Turing A apresentada a seguir decide $L_{f,g}$. Com entrada $\langle M \rangle$, A computa da seguinte maneira, onde definimos $n = |\langle M \rangle|$:

1. Calcula $g(n)$.
2. Simula um passo de M com entrada $\langle M \rangle$, mantendo contadores do número de passos t e do total de espaço s utilizados até o momento por M.
3. Computa $f(t, s)$. Se $f(t, s) > g(n)$, rejeita a entrada.
4. Verifica se $t > ng(n)2^{g(n)}$. Se esse for o caso, rejeita a entrada.
5. Se este for o último passo de M, então A aceita a entrada se e somente se M aceita, caso contrário A rejeita a entrada.

6. Retorna para o passo 2.

O passo 4 é necessário pois, dependendo da função f, a máquina M pode entrar em um laço infinito enquanto usa uma quantidade finita de espaço, mantendo o valorde $f(t, s)$ constante. Primeiro provaremos que $L(A) = L_{f,g}$. Depois limitaremos a f-complexidade de A.

Lema 3.19. *Se a máquina M termina sua computação com a entrada $\langle M \rangle$, então a palavra $\langle M \rangle$ não é rejeitada por A no passo 4.*

Demonstração. Suponha que $\langle M \rangle$ seja rejeitada por A no passo 4. Como $\langle M \rangle$ não foi rejeitada no passo 3, sabemos que $f(t, s) \leq g(n)$. Se $s > g(n)$, então temos claramente $t > g(n)$, e portanto $f(t, s) > g(n)$, uma vez que f é uma medida de complexidade natural. Por isso, $s \leq g(n)$. Por esse motivo não existem mais do que $ng(n)2^{g(n)}$ configurações possíveis para a máquina M com entrada $\langle M \rangle$. Porém, a rejeição no passo 4 requer $t > ng(n)2^{g(n)}$. Isso mostra que se A rejeita $\langle M \rangle$ no passo 4, então M repete alguma configuração e portanto nunca termina sua computação com a entrada $\langle M \rangle$, fato que contradiz a hipótese inicial do nosso lema. □

Continuando com a prova do teorema, suponha que $\langle M \rangle \in L_{f,g}$. Em particular, isso significa que M termina sua computação com a entrada $\langle M \rangle$. Portanto, o lema anterior implica que $\langle M \rangle$ não é rejeitada por A no passo 4. Como $f\text{-}ts_M(\langle M \rangle) \leq g(n)$ e f é não-decrescente, temos que $\langle M \rangle$ não é rejeitada por A no passo 3. Portanto, a simulação termina no passo 5 e, como M aceita $\langle M \rangle$, o mesmo faz A. Isso prova que $\langle M \rangle \in L(A)$.

Agora suponha que $\langle M \rangle \in L(A)$. Então $\langle M \rangle$ é aceita por A no passo 5, e por isso M também aceita $\langle M \rangle$. Como $\langle M \rangle$ não é rejeitada por A no passo 3, podemos concluir que $f\text{-}ts_M(\langle M \rangle) \leq g(|\langle M \rangle|)$. Por esse motivo, $\langle M \rangle \in L_{f,g}$. Logo segue que $L(A) = L_{f,g}$.

A partir de agora vamos analisar a simulação realizada no passo 2. A máquina A divide sua fita em 7 trilhas (considere as células da fita divididas módulo 7), organizadas da seguinte maneira:

A trilha 1 guarda $g(n)$.
A trilha 2 armazena $ng(n)2^{g(n)}$.
A trilha 3 é responsável polo contador t.
A trilha 4 contém o contador s.
A trilha 5 é usada para computar $f(t, s)$.
A trilha 6 guarda o código de M e seu estado atual.
A trilha 7 reflete a fita de M durante sua computação.

A máquina A simula a máquina de Turing M e sempre mantém a informação de cada trilha próxima do local que contém a célula atual lida por M (a máquina A desloca o conteúdo das trilhas para a esquerda ou para a direita em cada passo da simulação).

Primeiro, vamos calcular a complexidade de espaço $S_A(n)$ de A. O valor $g(n)$ é computado em espaço $S_{M_g}(n)$, uma vez que M_g computa g. O valor $ng(n)2^{g(n)}$ pode ser facilmente computado e armazenado em espaço $O(g(n))$. O contador s é limitado pelo contador t que, por sua vez, é limitado pelo valor na trilha 2. Por esse motivo, as trilhas 3 e 4 são irrelevantes para a complexidade de espaço assintótica final. O valor $f(t, s)$ pode ser computado em espaço $S_{M_f}(c_2 g(n))$, já que os valorres t e s são limitados assintoticamentes por $g(n)$. A descrição de M na trilha 6 possui tamanho n e, como $g(n) \geq n$, o espaço utilizado é também irrelevante. Finalmente, o tamanho da trilha 7 é limitado por $g(n)$ (como na prova do lema 3.19). Segue que $S_A(n)$ satisfaz:

$$S_A(n) \leq c_1 \left[S_{M_g}(n) + g(n) + S_{M_f}(c_2 g(n)) \right]. \tag{3.6}$$

Falta somente encontrar um limitante superior para $T_A(n)$. Sabemos que A computa $g(n)$ em tempo $T_{M_g}(n)$. A multiplicação do passo 4 pode ser feita em tempo $O(g(n))$. Durante a simulação, A precisa computar $f(t, s)$. Sempre temos $s \leq t \leq ng(n)2^{g(n)} + 1$, que em binário pode ser representado por $O(g(n))$ bits. Por isso, $f(t, s)$ é computado em tempo $T_{M_f}(c_3 g(n))$, para alguma constante c_3 apropriada. Em cada passo da simulação, as trilhas precisam ser deslocadas e alguns valores são comparados (lembre que A mantém o conteúdo das trilhas por perto). Isso pode ser feito em tempo proporcional ao tamanho das trilhas, ou seja, $c_1[S_{M_g}(n)+g(n)+S_{M_f}(c_2 g(n))\,]$. Finalmente, não mais do que $O(g(n))$ passos são simulados. Portanto, $T_A(n)$ satisfaz:

$$T_A(n) \leq c_4 \left[T_{M_g}(n) + g(n) \left[T_{M_f}(c_3 g(n)) + S_{M_g}(n) + g(n) + S_{M_f}(c_2 g(n)) \right] \right]. \tag{3.7}$$

□

Os teoremas clássicos de hierarquia de tempo e espaço podem ser derivados como casos particulares desse teorema geral de hierarquia. Consulte os livros de Sipser [103] e Papadimitriou [88] para ver a demonstração individual de cada teorema de hierarquia. Esses teoremas utilizam o conceito de função construtível, definido a seguir.

Definição 3.20. [Função Tempo-Construtível]. *Uma função $g : \mathbb{N} \to \mathbb{N}$ é dita tempo-construtível se a função que mapeia 1^n para a representação binária de $g(n)$ pode ser computada em tempo $O(g(n))$.*

Definição 3.21. [Função Espaço-Construtível]. *Uma função $g : \mathbb{N} \to \mathbb{N}$ é dita espaço-construtível se a função que mapeia 1^n para a representação binária de $g(n)$ pode ser computada em espaço $O(g(n))$.*

Observamos que a maior parte das funções utilizadas em complexidade computacional são construtíveis no sentido das definições 3.20 e 3.21. Por exemplo, n^5, $2^{\sqrt{n}}$ e $n \log n$ são funções tempo-construtíveis.

Corolário 3.22. [Hierarquia de Tempo]. *Para qualquer função tempo-construtível $g : \mathbb{N} \to \mathbb{N}$ com $g(n) \geq n$, existe uma linguagem decidível em tempo $O(g(n)^2)$ que não pode ser decidida em tempo $o(g(n))$.*

Demonstração. Seja $f(x, y) = x$. Pelo teorema 3.18 e pela definição de f, $L_{f,g}$ pode ser decidida em tempo $O(T_A(n))$, onde $T_A(n)$ satisfaz (3.4). Também temos que $S_{M_g}(n) \leq T_{M_g}(n)$ e $T_{M_g}(n)$ é $O(g(n))$, uma vez que g é uma função tempo-construtível e $g(n) \geq n$. Dada a definição de f, podemos assumir que $T_{M_f}(n)$ e $S_{M_f}(n)$ são funções $O(n)$. Portanto, $L_{f,g}$ é decidível em tempo $O(g(n)^2)$. Pelo teorema 3.15, $L_{f,g}$ não pode ser decidida em tempo $o(g(n))$. □

Corolário 3.23. [Hierarquia de Espaço]. *Para qualquer função espaço-construtível $g : \mathbb{N} \to \mathbb{N}$ com $g(n) \geq n$, existe uma linguagem decidível em espaço $O(g(n))$ que não pode ser decidida em espaço $o(g(n))$.*

Demonstração. Seja $f(x, y) = y$. Então o teorema 3.18 pode ser aplicado para obtermos que $L_{f,g}$ é uma linguagem decidível em f-complexidade $O(S_A(n))$, onde $S_A(n) \leq c_1[S_{M_g}(n) + g(n) + S_{M_f}(c_2 g(n))\ ]$. Sendo g uma função espaço-construtível, temos que $S_{M_g}(n)$ é $O(g(n))$, já que $g(n) \geq n$. Pela definição de f, podemos assumir que $S_{M_f}(n)$ é $O(n)$. Por isso, $L_{f,g}$ pode ser decidida em espaço $O(g(n))$. Pelo teorema 3.15, $L_{f,g}$ não pode ser decidida em espaço $o(g(n))$. □

Observe que a hierarquia de espaço é mais forte que a hierarquia de tempo. Isso ocorre porque uma quantidade de tempo assintoticamente relevante é perdida com os detalhes da simulação. Note que simulações mais eficientes produzem resultados mais fortes.

Veremos na seção 3.4 que uma situação inesperada ocorre quando trabalhamos com funções que não são construtíveis.

3.3 Consequências do Resultado Anterior

As classes de complexidade estudadas até agora foram definidas a partir do modelo computacional de máquinas de Turing com várias fitas. Por isso, antes de aplicarmos os resultados da seção anterior (válidos para máquinas de Turing de fita única e alfabeto binário) precisamos traduzí-los para esse modelo computacional. Para mostrar a generalidade dos resultados obtidos, vamos discutir primeiramente o que ocorre quando os modelos computacionas envolvidos são arbitrários.

Um fato importante verificado para os modelos estudados em computação é a equivalência em termos de poder computacional entre eles. Esse fato é conhecido como a Tese de Church-Turing. Isso significa que, dados dois modelos computacionais suficientemente poderosos $\mathcal{C}_1$ e $\mathcal{C}_2$, é possível converter uma máquina do primeiro modelo em uma máquina equivalente do segundo modelo, i.e., capaz de decidir a mesma linguagem. Além disso, para todos os modelos fisicamente implementáveis conhecidos, o tempo de execução da máquina convertida é no máximo polinomialmente maior do que o tempo de execução da máquina original. Por isso, se $\mathcal{C}_1$ e $\mathcal{C}_2$ são dois modelos computacionais arbitrários, existe um polinômio $p_{12}(x)$ tal que se uma linguagem L pode ser decidida em tempo $O(t(n))$ no modelo $\mathcal{C}_1$, então L pode ser decidida em tempo $O(p_{12}(t(n)))$ no modelo computacional $\mathcal{C}_2$.

Definição 3.24. *Seja $\mathcal{C}$ um modelo computacional arbitrário. O conjunto de linguagens decidíveis em tempo determinístico $O(t(n))$ no modelo computacional $\mathcal{C}$ será denotado por* $\mathrm{DTIME}_{\mathcal{C}}(t(n))$.

Lema 3.25. *Seja $t : \mathbb{N} \to \mathbb{N}$ uma função arbitrária. Temos então que:*

(i) Se $L \in DTIME_{\mathcal{C}_1}(t(n))$, então $L \in DTIME_{\mathcal{C}_2}(p_{12}(t(n)))$.

(ii) Se $L \notin DTIME_{\mathcal{C}_1}(p_{21}(t(n)))$, então $L \notin DTIME_{\mathcal{C}_2}(t(n))$.

As funções $p_{12}(x)$ e $p_{21}(x)$ são os polinômios associados ao custo computacional adicionado na conversão entre as máquinas dos modelos computacionais $\mathcal{C}_1$ e $\mathcal{C}_2$.

Demonstração. A demonstração de *(i)* é imediata e a demonstração de *(ii)* segue pela contrapositiva. □

Esse fato simples mostra que a existência de hierarquias de complexidade é uma propriedade compartilhada por todos os modelos computacionais.

Considere a partir de agora que $\mathcal{C}_1$ é o modelo computacional de máquinas de Turing de fita única com alfabeto binário e denote por $\mathcal{C}_2$ o modelo computacional de máquinas de Turing arbitrárias. Portanto, temos daqui em diante que $\mathrm{DTIME}_{\mathcal{C}_2}(t(n)) = \mathrm{DTIME}(t(n))$. É imediato que $p_{12}(x) = x$, uma vez que $\mathcal{C}_1$ é um caso particular de $\mathcal{C}_2$. Segue do próximo resultado que podemos tomar $p_{21}(x) = x^2$.

Teorema 3.26. *Seja M_1 uma máquina de Turing que decide uma linguagem L em tempo $O(t(n))$. Então existe uma máquina de Turing de fita única e alfabeto binário que decide L em tempo $O(t(n)^2)$.*

Demonstração. A prova desse resultado segue a partir de uma simulação similar àquela utilizada na prova do lema 3.3, onde várias fitas são armazenadas em uma única fita da máquina simuladora. □

Temos finalmente tudo que precisamos para separar algumas classes de complexidade importantes.

Teorema 3.27. P $\neq$ EXP.

Demonstração. Observe primeiramente que $t(n) = 2^{n/2}$ é uma função tempo-construtível em máquinas de Turing de fita única. Portanto, pelo corolário 3.22, temos a inclusão própria de classes $\mathrm{DTIME}_{\mathcal{C}_1}(2^{n/4}) \subsetneq \mathrm{DTIME}_{\mathcal{C}_1}(2^n)$. Além disso, segue a partir do lema 3.25 que:

$$\mathrm{P} \subseteq \mathrm{DTIME}(2^{n/8}) \subseteq \mathrm{DTIME}_{\mathcal{C}_1}(2^{n/4}) \subsetneq \mathrm{DTIME}_{\mathcal{C}_1}(2^n) \subseteq \mathrm{DTIME}(2^n) \subseteq \mathrm{EXP},$$

como queríamos demonstrar. □

As mesmas ideias podem ser utilizadas para separar outras classes de complexidade importantes. O próximo resultado segue a partir dos teoremas de hierarquia de modo análogo.

Definição 3.28. [Classe de Complexidade E].
$\mathrm{E} = \bigcup_{c \geq 1} \mathrm{DTIME}(2^{cn})$.

Teorema 3.29.

(i) PSPACE $\neq$ EXPSPACE.

(ii) E $\neq$ EXP.

(iii) P $\neq$ E.

Além disso, o argumento utilizado na demonstração do teorema 3.6 pode ser usado para provar que NP $\neq$ E.

3.4 O Teorema da Lacuna

Veremos nesta seção que os teoremas de hierarquia podem não ser válidos quando utilizamos funções não-construtíveis. Por exemplo, existem funções $t(n)$ não-triviais tais que $\mathrm{DTIME}(t(n)) = \mathrm{DTIME}(2^{t(n)})$. Assim como demonstramos um teorema de hierarquia um pouco mais geral, vamos provar uma versão mais abstrata do resultado conhecido como Teorema da Lacuna (Borodin [16]).

Definição 3.30. *Uma linguagem L pertence à classe* $\mathrm{COMPLEX}_f(g(n))$ *se existe uma máquina de Turing A que decide L e, para todo inteiro n, temos que $f\text{-}TS_A(n) \leq g(n)$.*

Teorema 3.31. [Teorema da Lacuna]. *Seja f uma medida de complexidade natural. Existe uma função computável $g : \mathbb{N} \to \mathbb{N}$ com $g(n) \geq n$ tal que se $L \in \mathrm{COMPLEX}_f(2^{g(n)})$, então L pode ser decidida em f-complexidade $O(g(n))$.*

Demonstração. Considere uma ordenação lexicográfica das máquinas de Turing: M_0, M_1, M_2, ... Para $i, k \geq 0$, definimos que o predicado $P(i, k)$ é verdadeiro se:

> Para todo j, $0 \leq j \leq i$, temos que M_j com qualquer entrada x de tamanho i satisfaz $f\text{-ts}_{M_j}(x) < k$ ou então temos $f\text{-ts}_{M_j}(x) > 2^k$ ou $f\text{-ts}_{M_j}(x)$ é indefinido (M_j pode computar indefinidamente com a entrada x).

Note que $P(i, k)$ é uma propriedade decidível através de uma simulação análoga à apresentada na demonstração do teorema 3.18.

Continuamos com a definição da função g. Fixe um inteiro $i \geq 0$. Como o número de máquinas de Turing M_j com $j \leq i$ e o número de palavras x com $|x| = i$ são finitos, existe uma quantidade finita de pares (M_j, x) satisfazendo essas condições. Além disso, cada par (M_j, x) só pode tornar $P(i, k)$ falso para uma quantidade finita de inteiros k. Por isso, deve existir um inteiro $l \geq i$ tal que $P(i, l)$ é verdadeiro. Selecione o menor desses inteiros e defina $g(i) = l$. Como o predicado P é decidível, g é uma função computável. Por construção, $g(i) \geq i$ e $P(i, g(i))$ é verdadeiro, para todo i.

Suponha que $L \in \mathrm{COMPLEX}_f(2^{g(n)})$. Então L é decidida por alguma máquina de Turing M_j em f-complexidade $2^{g(n)}$. Considere agora qualquer palavra x com $|x| \geq j$. Por construção, $P(|x|, g(|x|))$ é verdadeiro. Temos que $j \leq |x|$, portanto para a máquina M_j: $f\text{-ts}_{M_j}(x) < g(|x|)$ ou $f\text{-ts}_{M_j}(x) > 2^{g(|x|)}$ ou $f\text{-ts}_{M_j}(x)$ é indefinido. Como M_j *decide* L em f-complexidade $2^{g(n)}$, segue que $f\text{-ts}_{M_j}(x) < g(|x|)$. Como o conjunto de palavras de tamanho menor do que j é finito, L pode ser decidida em f-complexidade $O(g(n))$. □

Observe que podemos trocar a função 2^n utilizada no teorema por qualquer outra função computável não-descrescente.

Quando tomamos f como sendo uma das funções de projeção, obtemos os teoremas clássicos da lacuna para tempo e espaço.

Corolário 3.32. [Teorema da Lacuna para Tempo]. *Existe uma função computável $t : \mathbb{N} \to \mathbb{N}$ tal que, para toda linguagem $L \subseteq \{0, 1\}^\star$, se L pode ser decidida em tempo $2^{t(n)}$, então L pode ser decidida em tempo $O(t(n))$.*

Corolário 3.33. [Teorema da Lacuna para Espaço]. *Existe uma função computável $s : \mathbb{N} \to \mathbb{N}$ tal que, para toda linguagem $L \subseteq \{0, 1\}^\star$, se L pode ser decidida em espaço $2^{s(n)}$, então L pode ser decidida em espaço $O(s(n))$.*

3.5 A Hierarquia Não-Determinística

Nosso próximo objetivo é obter resultados para classes não-determinísticas similares aos obtidos nas seções 3.2 e 3.3.

Existe um obstáculo fundamental impedindo que a demonstração do teorema de hierarquia determinístico seja traduzida para o caso não-determinístico. Observe que o fecho por complementação de linguagens existente no caso determinístico é essencial para a diagonalização ocorrida na prova do teorema 3.15. Porém, não esperamos em geral que o complemento de uma linguagem decidida por uma MTND seja decidida por outra MTND de mesma complexidade de tempo. Em particular, muitos pesquisadores acreditam que NP $\neq$ coNP. Apesar disso, o complemento de uma linguagem decidida por um MTND em tempo $f(n)$ pode ser decidida em tempo determinístico $2^{f(n)}f(n)$ através da enumeração de todas as escolhas não-determinísticas possíveis. Felizmente, esse fato pode ser explorado através de um argumento de diagonalização um pouco mais elaborado.

Teorema 3.34. [Hieraquia de Tempo Não-Determinística]. *Sejam $f, g : \mathbb{N} \to \mathbb{N}$ funções e assuma que f é uma função tempo-construtível não-decrescente com $f(n+1)\log f(n+1)$ sendo $o(g(n))$. Então* $\mathrm{NTIME}(f(n)) \subsetneq \mathrm{NTIME}(g(n))$.

Demonstração. Para não discutirmos mais uma vez detalhes de simulação, vamos apenas esboçar a demonstração desse resultado. Veja a seção de referências do capítulo para mais detalhes.

Considere uma ordenação lexicográfica das MTNDs: M_0, M_1, M_2, ... Para cada MTND M_i vamos associar um intervalo $\mathrm{I}_i = [\varphi(i)+1, \ldots, \varphi(i+1)]$, onde $\varphi : \mathbb{N} \to \mathbb{N}$ é uma função crescente que depende de f a ser determinada posteriormente.

Para provar o resultado, vamos exibir uma MTND D com complexidade de tempo $O(g(n))$ cuja linguagem $L(D) \notin \mathrm{NTIME}(f(n))$. Para isso, construiremos D de forma que, para todo inteiro i, se M_i computa em tempo $O(f(n))$ então existe $k \in I_i$ tal que $M_i(1^k) \neq D(1^k)$.

Isso pode ser conseguido se simularmos a máquina M_i de um modo mais inteligente e então negarmos a sua resposta. Porém, como discutido no início da seção, o grande desafio é negar a resposta de M_i sem ultrapassar o limite de complexidade $O(g(n))$. O argumento que vamos apresentar é conhecido como *método da diagonalização atrasada*. Lembre que queremos provar que se M_i é uma máquina não-determinística que computa em tempo $O(f(n))$, então $L(M_i) \neq L(D)$. Em outras palavras, D não precisa se preocupar com o seu comportamento em relação à máquinas de Turing que computam por mais do que cerca de $f(n)$ passos.

Seja M_i uma MTND que computa em tempo $O(f(n))$. A máquina D utiliza o seu não-determinismo para simular M_i por cerca de $f(k+1)$ passos, de forma a obter: $\forall k \in I_i \backslash \{\varphi(i+1)\}$, $D(1^k) = M_i(1^{k+1})$ *(i)*. Porém, observe que ainda não definimos o valor

de $D(1^{\varphi(i+1)})$. É exatamente aqui que diagonalizamos: com entrada $1^{\varphi(i+1)}$, D simula deterministicamente a máquina M_i com entrada $1^{\varphi(i)+1}$ por cerca de $f(\varphi(i+1))$ passos e nega a sua resposta *(ii)*. Se D computar dessa forma, não podemos ter $L(D) = L(M_i)$. Caso contrário, seguiria a partir disso e de *(i)* que D e M_i aceitam todas as palavras unárias formadas por uma sequência de 1s com tamanho em I_i ou rejeitam todas essas palavras. No entanto, segue por *(ii)* que D aceita $1^{\varphi(i+1)}$ se e somente se M_i rejeita $1^{\varphi(i)+1}$. Isso prova que $L(D) \notin \mathrm{NTIME}(f(n))$.

Finalmente, para mostrarmos que $L(D) \in \mathrm{NTIME}(g(n))$, notamos que:

(1) Dada uma entrada na forma 1^k, D descobre qual máquina de Turing simular através do cálculo da função $\varphi(k)$. Isso pode ser feito em tempo $O(g(n))$, pois φ pode ser definida de forma que sua complexidade dependa essencialmente da complexidade de f, que por sua vez é uma função tempo-construtível em $o(g(n))$ passos;

(2) A simulação utilizada em *(i)* pode ser feita no tempo desejado, pois $f(n+1)\log f(n+1)$ é $o(g(n))$[1];

(3) A simulação determinística que ocorre em *(ii)* não ultrapassa o limite $O(g(n))$, pois a função $\varphi(n)$ garante que a entrada $1^{\varphi(i)+1}$ para a máquina M_i é muito menor do que a entrada $1^{\varphi(i+1)}$ da máquina D. Por exemplo, podemos definir essa função como: $\varphi(1) = c$ e $\varphi(n+1) = 2^{2f(\varphi(n))}$, onde c é uma constante que depende dos detalhes de implementação de D. □

Teorema 3.35. NP $\neq$ NEXP.

Demonstração. Segue do teorema anterior que $\mathrm{NP} \subseteq \mathrm{NTIME}(2^n) \subsetneq \mathrm{NTIME}(2^{n^2}) \subseteq \mathrm{NEXP}$. □

Observe que classes como P, NP e PSPACE não podem ser separadas pelos teoremas de hierarquia apresentados. Esses resultados não relacionam classes de complexidade que envolvem recursos diferentes. Apesar disso, podemos combinar os resultados anteriores para provar alguns teoremas interessantes.

Teorema 3.36.

(i) P $\neq$ NP ou NP $\neq$ EXP.

(ii) PSPACE $\neq$ EXP ou EXP $\neq$ EXPSPACE.

[1] É possível simular $t(n)$ passos de uma máquina de Turing em tempo $t(n)\log t(n)$. Consulte a seção de referências deste capítulo para mais detalhes.

(iii) NP $\neq$ EXP ou EXP $\neq$ NEXP.

Demonstração. A demonstração desses resultados segue por teoria de conjuntos elementar a partir dos teoremas 3.11, 3.27, 3.29 e 3.35. □

3.6 O Método da Completude

Na seção 2.5 vimos que os problemas NP-completos são os mais difíceis da classe NP. Uma propriedade fundamental da noção de completude é que se demonstrarmos que um problema completo está em uma classe inferior, então ocorre um colapso de classes. Por isso, caso seja provado que P $\neq$ NP, sabemos que nenhum problema NP-completo admite algoritmos eficientes.

Os teoremas de hierarquia implicam que diversas classes de complexidade são distintas. Por exemplo, vimos que P $\neq$ EXP. Definindo adequadamente a noção de completude para a classe EXP, podemos obter que problemas EXP-completos não podem ser resolvidos em tempo polinomial. Interessantemente, muitos problemas práticos acabam se mostrando completos para alguma classe de complexidade. Isso nos dá uma compreensão muito boa da dificuldade do problema sendo estudado.

Definição 3.37. [Completude]. *Seja $\mathcal{R}$ uma classe de complexidade. Dizemos que uma linguagem $L \subseteq \{0,1\}^\star$ é $\mathcal{R}$-completa se:*

(i) $L \in \mathcal{R}$;

(ii) Para toda linguagem $L' \in \mathcal{R}$, temos $L' \leq_p L$.

Observe que as classes de complexidade EXP, PSPACE, coNP, coNEXP e EXPSPACE são fechadas por redução de tempo polinomial. Consequentemente, se provarmos que um problema completo de uma certa classe de complexidade $\mathcal{A}$ pertence à classe $\mathcal{B}$, temos $\mathcal{A} \subseteq \mathcal{B}$. Conversamente, se $\mathcal{A} \nsubseteq \mathcal{B}$ então problemas completos da classe $\mathcal{A}$ não pertencem à classe de complexidade $\mathcal{B}$.

Uma construção semelhante a que aparece na demonstração do teorema 2.22 pode ser usada para exibir linguagens completas para diversas dessas classes. Embora esses problemas não sejam muito úteis, problemas completos naturais podem facilitar certas demonstrações envolvendo classes de complexidade. Veremos a seguir como a noção de completude pode ser usada para classificar a dificuldade computacional de alguns problemas de interesse prático.

Uma expressão regular é uma forma sucinta de representar palavras (sequências de caracteres) que satisfazem um certo padrão. Por exemplo, uma palavra satisfaz a expressão regular $\mathrm{A}^\star\mathrm{bA}^\star$ se ela é formada por uma sequência arbitrária de As, seguido por um b, e

depois por mais uma sequência de As. Outro exemplo é a expressão regular $\{c \cup b\}$arro, satisfeita pelas palavras carro e barro. Portanto, uma expressão regular é uma palavra bem-definida e o conjunto de palavras que satisfazem uma certa expressão regular é uma linguagem. É possível combinar as operações básicas $\star$, $\cup$ e a concatenação de símbolos para produzir regras que codificam linguagens sofisticadas.

Expressões regulares são usadas por diversos utilitários, tais como editores de texto e interpretadores, para manipular texto baseado em padrões. Por exemplo, pode-se usar uma expressão regular para garantir que o dado digitado por um usuário é um endereço web válido. Outro uso importante de expressões regulares é a filtragem de informação em bancos de dados textuais. Vamos definir a seguir o conjunto de palavras que representam expressões regulares válidas. Por simplicidade, vamos considerar apenas o alfabeto binário.

Definição 3.38. [Expressões Regulares]. *O conjunto de expressões regulares pode ser definido indutivamente através das seguintes regras:* (1) *os símbolos* 0 , 1 *e* ϵ *são expressões regulares;* (2) *se* α_1 *e* α_2 *são expressões regulares, então* $\alpha_1\alpha_2$, $\alpha_1 \cup \alpha_2$ *e* $\alpha_1^\star$ *são expressões regulares. O significado de uma expressão regular é uma linguagem definida indutivamente da seguinte maneira:* $L(0) = \{0\}$, $L(1) = \{1\}$, $L(\epsilon) = \emptyset$, $L(\alpha_1\alpha_2) = L(\alpha_1)L(\alpha_2) = \{xy : x \in L(\alpha_1), y \in L(\alpha_2)\}$, $L(\alpha_1 \cup \alpha_2) = L(\alpha_1) \cup L(\alpha_2)$, *e* $L(\alpha_1^\star) = L(\alpha_1)^\star$. *Os operadores são aplicados de acordo com a ordem de precedência anterior. Finalmente, dizemos que duas expressões regulares* α_1 *e* α_2 *são equivalentes se* $L(\alpha_1) = L(\alpha_2)$.

Um importante problema computacional é decidir se duas expressões regulares são equivalentes. Através de algumas modificações na definição de expressão regular, essa tarefa fornece um grande número de problemas completos:

(1) O problema de decidir se duas expressões regulares são equivalentes é PSPACE-completo.

(2) Uma expressão regular é livre de estrela se não possui ocorrências do operador *. Decidir se duas expressões regulares livres de estrela são equivalentes é um problema coNP-completo.

(3) Suponha que adicionemos o operador ^{2} (elevar ao quadrado) no conjunto de operadores de expressões regulares, de forma que $L(\alpha^2) = L(\alpha)L(\alpha)$. Decidir se duas expressões regulares livres de estrela e com o operador ^{2} são equivalentes é um problema coNEXP-completo.

(4) Suponha que os operadores * e ^{2} sejam permitidos. Então o problema da equivalência de expressões regulares se torna EXPSPACE-completo. Por isso, sabemos pelo

teorema 3.29 que esse problema não pode ser resolvido de forma eficiente.

(5) Além disso, podemos adicionar um operador de negação $\neg$ às expressões regulares, de forma que $L(\neg\alpha) = \{0,1\}^\star \backslash L(\alpha)$. Nesse caso, o problema de equivalência não pode ser resolvido por nenhum algoritmo determinístico com tempo de execução limitado por uma torre de exponenciais de altura constante.

A prova desses resultados pode ser obtida em Meyer e Stockmeyer [79, 80]. Muitos outros problemas completos e demonstrações de completude podem ser obtidos no livro de Du e Ko [33].

Os resultados provados neste capítulo mostram que alguns problemas computacionais não podem ser resolvidos de modo eficiente. Podemos demonstrar isso porque tais problemas são completos em suas respectivas classes de complexidade. Entretando, veremos na seção 4.5 que é bem provável que diversos problemas computacionais interessantes não sejam completos. Um grande desafio para os pesquisadores em complexidade computacional é descobrir métodos capazes de provar que problemas nessa situação são computacionamente difíceis. Infelizmente, os argumentos utilizados neste capítulo provavelmente não são suficientes para isso. Discutiremos em profundidade os limites do método de diagonalização no capítulo 5.

3.7 Referências Adicionais

As demonstrações dos teoremas de hierarquia e da lacuna apresentadas aqui podem ser obtidas no artigo de Oliveira e Moura [87]. O artigo de Blum [12] introduz uma noção muito mais geral de medida de complexidade. Recomendamos também o artigo de Seiferas [98] para uma discussão sobre resultados em complexidade que são independentes do modelo computacional em consideração.

O teorema de hierarquia não-determinístico foi provado inicialmente por Cook [27]. Uma versão mais forte desse resultado aparece no artigo de Seiferas et al. [99]. A demonstração apresentada na seção 3.5 é devida à Zak [116].

É interessante notar que, embora ocorra uma perda quadrática de tempo na simulação de máquinas de Turing de várias fitas por uma máquina de fita única, é possível simular máquinas arbitrárias em apenas duas fitas em tempo $T(n)\log T(n)$, onde $T(n)$ é a complexidade de tempo da máquina simulada. Veja o artigo de Hennie e Stearns [56] para mais detalhes.

O artigo de Fortnow [39] descreve algumas aplicações mais recentes do método de diagonalização. No mesmo artigo, o autor sugere a possibilidade de usar diagonalização para separar a classe NP de classes mais fracas do que P. Como veremos no capítulo 5, é

improvável que o método de diagonalização sozinho seja capaz de separar as classes P e NP.

Recomendamos a leitura de outro artigo de Fortnow [40] para uma discussão sobre classes de complexidade mais poderosas que NP e problemas completos para essas classes.

A discussão sobre o problema da equivalência para expressões regulares é motivada por um exercício que aparece no livro de Papadimitriou [88]. A demonstração de completude da versão EXPSPACE-completa do problema pode ser obtida no livro de Sipser [103].

Capítulo 4

O Problema P vs NP em Profundidade

"May the force of P *and* NP *be with you."*
Sanjeev Arora.

Neste capítulo vamos discutir alguns tópicos mais avançados relacionados com o problema P vs NP. Inicialmente, veremos como a hierarquia polinomial generaliza a definição das classes de complexidade P e NP. Vamos mostrar também que algoritmos muito eficientes em tempo e espaço não são capazes de decidir a linguagem SAT. Discutiremos em seguida algumas propriedades estruturais das linguagens em NP. Por último, demonstraremos que existem algoritmos assintoticamente ótimos para todos os problemas da classe NP.

4.1 A Hierarquia Polinomial

Na seção 3.1 mostramos que NP $\subseteq$ PSPACE. Vamos a partir de agora explorar algumas classes de complexidade situadas entre NP e PSPACE.

A Hierarquia Polinomial (PH) foi definida inicialmente por Meyer e Stockmeyer [79, 105], a partir de uma generalização das classes P e NP. Muitos resultados importantes em complexidade computacional estão relacionados com a hipótese de que a hierarquia polinomial não colapsa em nenhum de seus níveis.

Diversos problemas naturais têm complexidade de tempo situada em PH. Por exemplo, seja $\varphi(x_1, \ldots, x_n)$ uma fórmula proposicional e $f : \{0,1\}^n \to \{0,1\}$ a função computada por essa fórmula. Seria φ a fórmula que computa a função f e em cuja descrição ocorrem

o menor número possível de conectivos? Por definição, uma fórmula proposicional é satisfatível se e somente se *existe* uma atribuição que a torne verdadeira. Por outro lado, uma fórmula φ é a menor possível se e somente se *para toda* fórmula proposicional φ' onde ocorrem menos conectivos, *existe* uma atribuição $\vec{x}$ tal que $\varphi(\vec{x}) \neq \varphi'(\vec{x})$. Observe que essa combinação de quantificadores aumenta significativamente nossa capacidade de expressar problemas computacionais. Vejamos mais um exemplo.

Exemplo 4.1. [Jogadas Finais em um Jogo Arbitrário]. *Consideramos neste exemplo um jogo de tabuleiro entre dois jogadores com as seguintes propriedades:*

(i) Dada uma descrição C de um tabuleiro do jogo, é possível verificar de modo eficiente todas as próximas jogadas possíveis, ou seja, sabemos verificar se um tabuleiro C' pode ser obtido a partir de C com uma única jogada;

(ii) Os jogadores alternam as jogadas;

(iii) Existe uma máquina de Turing eficiente M que, ao receber como entrada um tabuleiro C_0 e tabuleiros $C_1, \ldots, C_k$, aceita a entrada $\langle C_0, C_1, \ldots, C_k \rangle$ se e somente se: (1) *cada tabuleiro pode ser obtido a partir do tabuleiro anterior conforme descrito no item (i);* (2) *os jogadores alternam as jogadas;* (3) *algum tabuleiro C_i determina a vitória do primeiro jogador.*

Observe que certos jogos de tabuleiro clássicos como dama e xadrez satisfazem as propriedades anteriores. No entanto, jogos muito mais gerais podem ser codificados pela máquina de Turing M.

Considere agora o seguinte problema computacional. Dado um tabuleiro C, o primeiro jogador pode ganhar o jogo nas próximas k jogadas? Em outras palavras, existe uma jogada do primeiro jogador que leva a um tabuleiro válido C_1 tal que, para toda jogada do segundo jogador que leva a um tabuleiro válido C_2 tal que, existe um jogada do primeiro jogador ... de forma que em no máximo k jogadas o jogo é vencido pelo primeiro jogador? Isso é o mesmo que perguntar se existe uma estratégia vencedora para o primeiro jogador que pode ser aplicada às próximas k jogadas. Esse problema pode ser representado pela seguinte linguagem:

$$L_k = \{\langle C \rangle \in \{0,1\}^\star : \exists C_1 \ \forall C_2 \ldots Q_k C_k \ M(C, C_1, \ldots, C_k) = 1\},$$

onde Q_i é o quantificador $\exists$ ou $\forall$ de acordo com a paridade do inteiro k.

Intuitivamente, esse problema é muito mais difícil do que problemas em NP. Por exemplo, tomando $k = 1$ e considerando M como sendo um verificador eficiente para SAT, podemos considerar uma fórmula proposicional como sendo um jogo em que o primeiro jogador vence se consegue encontrar uma atribuição que satisfaça a fórmula de entrada.

Definição 4.2. [Hierarquia Polinomial]. *Para todo inteiro $k \geq 0$, dizemos que uma linguagem L pertence à classe Σ_k^p (k-ésimo nível da hierarquia polinomial) se existe uma máquina de Turing M que computa em tempo polinomial e um polinômio $q(.)$ tal que*

$$x \in L \;\Leftrightarrow\; \exists w_1 \in \{0,1\}^{q(|x|)}\ \forall w_2 \in \{0,1\}^{q(|x|)} \ldots\ Q_k w_k \in \{0,1\}^{q(|x|)}\ M(x, w_1, \ldots, w_k) = 1,$$

onde Q_k denota $\forall$ ou $\exists$ se k for par ou ímpar, respectivamente.

Além disso, definimos $\Pi_k^p = \mathrm{co}\Sigma_k^p = \{\overline{L} : L \in \Sigma_k^p\}$. Finalmente, definimos a hierarquia polinomial como sendo a classe de complexidade $\mathrm{PH} = \bigcup_{k \geq 0} \Sigma_k^p$.

Novamente, a restrição de que toda palavra w_i tenha tamanho exato $q(|x|)$ não é uma limitação. Basta que as palavras envolvidas sejam polinomialmente limitadas, uma vez que sempre podemos obter uma máquina de Turing equivalente que opera de acordo com as restrições de tamanho da definição anterior. Utilizaremos esse fato implicitamente a partir de agora.

A linguagem L_k discutida no exemplo 4.1 pertence à classe Σ_k^p. Por isso, temos que $\overline{L_k}$ está em Π_k^p. Considere a seguir jogos em que a vitória será determinada nas próximas k jogadas (ou seja, se $M(C, C_1, \ldots, C_k) = 0$ então o segundo jogador vence o jogo). Negando a definição de L_k, obtemos que $\overline{L_k}$ representa o problema de verificar se existe uma estratégia vencedora para o segundo jogador para as próximas k jogadas.

Observe que $\mathrm{P} = \Sigma_0^p = \Pi_0^p$, $\mathrm{NP} = \Sigma_1^p$ e $\mathrm{coNP} = \Pi_1^p$. Além disso, por definição, se $L \in \Pi_k^p$ então $\overline{L} \in \Sigma_k^p$, ou seja:

$$x \in \overline{L} \;\Leftrightarrow\; \exists w_1 \in \{0,1\}^{q(|x|)}\ \forall w_2 \in \{0,1\}^{q(|x|)} \ldots\ Q_k w_k \in \{0,1\}^{q(|x|)} M(x, w_1, \ldots, w_k) = 1,$$

onde Q_k denota $\forall$ ou $\exists$ se k for par ou ímpar, respectivamente. Portanto, negando os dois lados da equivalência anterior, obtemos que:

$$x \in L \;\Leftrightarrow\; \forall w_1 \in \{0,1\}^{q(|x|)}\ \exists w_2 \in \{0,1\}^{q(|x|)} \ldots\ Q_k w_k \in \{0,1\}^{q(|x|)} M(x, w_1, \ldots, w_k) = 0,$$

onde dessa vez Q_k denota $\forall$ ou $\exists$ se k for ímpar ou par, respectivamente. Seja M' uma máquina de Turing que aceita sua entrada se e somente se M rejeita sua entrada. Claramente:

$$x \in L \;\Leftrightarrow\; \forall w_1 \in \{0,1\}^{q(|x|)}\ \exists w_2 \in \{0,1\}^{q(|x|)} \ldots\ Q_k w_k \in \{0,1\}^{q(|x|)} M'(x, w_1, \ldots, w_k) = 1.$$

Isso mostra que, alterando a disposição dos quantificadores, podemos obter uma definição alternativa para a classe Π_k^p.

Utilizando a observação anterior, note que para todo inteiro $k \geq 0$ temos $\Sigma_k^p \subseteq \Pi_{k+1}^p \subseteq \Sigma_{k+2}^p$ (o novo quantificador não altera a validade da expressão se desprezarmos a palavra de entrada adicional). Portanto, $\mathrm{PH} = \bigcup_{k \geq 0} \Pi_k^p$. Similarmente, é fácil provar que $\Sigma_k^p \subseteq \Sigma_{k+1}^p$ e que $\Pi_k^p \subseteq \Pi_{k+1}^p$.

A hierarquia polinomial é definida para problemas de decisão, ou seja, linguagens. No entanto, podemos associar um problema de busca à máquina de Turing M utilizada na definição de Σ_k^p (como discutido na seção 2.3). É possível adaptar a prova do teorema 2.15 para mostrar que se $\Sigma_k^p = \mathrm{P}$, então podemos encontrar eficientemente uma palavra w_1 como na definição 4.2. De forma mais geral, dados i certificados ($\exists$) e i palavras ($\forall$) anteriores, podemos encontrar eficientemente o $(i+1)$-ésimo certificado.

Como discutido anterioremente, ainda é um problema em aberto provar que a hierarquia polinomial possui infinitos níveis distintos.

Questão em Aberto 4.3. *Existe um inteiro $k \geq 0$ tal que* $\mathrm{PH} = \Sigma_k^p$*?*

Veremos nos próximos teoremas que o colapso entre dois níveis adjacentes provoca o colapso da hierarquia polinomial.

Lema 4.4. *Para todo inteiro $k \geq 0$, se $\Sigma_{k+1}^p = \Sigma_k^p$ então $\Pi_{k+1}^p = \Pi_k^p$.*

Demonstração. Assuma que $L \in \Pi_{k+1}^p$. Então $\overline{L} \in \Sigma_{k+1}^p \subseteq \Sigma_k^p$. Por definição, isso significa que $L \in \Pi_k^p$, como queríamos demonstrar. □

Teorema 4.5. [Colapso Vertical].
Para todo inteiro $k \geq 0$, se $\Sigma_{k+1}^p = \Sigma_k^p$ então $\mathrm{PH} = \Sigma_k^p$.

Demonstração. Assuma que $\Sigma_{k+1}^p \subseteq \Sigma_k^p$. Vamos provar que $\Sigma_{k+2}^p \subseteq \Sigma_{k+1}^p$. O resultado segue por indução, pois teríamos então $\Sigma_j^p \subseteq \Sigma_k^p$, para todo inteiro $j \geq k$.

Seja $L \in \Sigma_{k+2}^p$, ou seja, existe uma máquina de Turing M_L de tempo polinomial e um polinômio $q(.)$ tal que

$$x \in L \;\Leftrightarrow\; \exists w_1\ \forall w_2\ \exists w_3\ \ldots\ Q_{k+2} w_{k+2}\ M_L(x, w_1, \ldots, w_{k+2}) = 1,$$

onde $w_i \in \{0,1\}^{q(|x|)}$ para todo $1 \leq i \leq k+2$. Considere agora a linguagem

$$L' = \{\langle x, w_1\rangle : |w_1| = q(|x|) \text{ e } \forall w_2\ \exists w_3\ \ldots\ Q_{k+2} w_{k+2}\ M_L(x, w_1, \ldots, w_{k+2}) = 1\}.$$

É imediato que $x \in L$ se e somente se existe w_1 de tamanho $q(|x|)$ tal que $\langle x, w_1\rangle \in L'$ (1). Além disso, segue a partir da definição de L' que $L' \in \Pi_{k+1}^p$. Portanto, pelo lema 4.4 temos que $L' \in \Pi_k^p$. Em outras palavras, existe uma máquina de Turing $M_{L'}$ de tempo polinomial e um polinômio $q'(.)$ tais que

$$\langle x, w_1\rangle \in L' \;\Leftrightarrow\; |w_1| = q(|x|) \text{ e } \forall w_2\ \exists w_3\ \ldots\ Q_k w_{k+1} M_{L'}(\langle x, w_1\rangle, w_2, \ldots, w_{k+1}) = 1 \quad (2),$$

onde $w_i \in \{0,1\}^{q'(|\langle x, 1^{q(|x|)}\rangle|)}$ para todo $2 \leq i \leq k+1$. Combinando (1) e (2), podemos obter um polinômio $p(.)$ adequado e modificar a máquina $M_{L'}$ de forma que:

$$x \in L \;\Leftrightarrow\; \exists w_1\ \forall w_2\ \ldots\ Q_{k+1} w_{k+1}\ M_{L'}(x, w_1, \ldots, w_{k+1}) = 1,$$

onde $w_i \in \{0,1\}^{p(|x|)}$ para $1 \leq i \leq k+1$. Isso completa a prova de que $L \in \Sigma_{k+1}^p$. □

Teorema 4.6. [Colapso Horizontal].
Para todo inteiro $k \geq 0$, se $\Sigma_k^p = \Pi_k^p$ então PH $= \Sigma_k^p$.

Demonstração. Vamos provar que se $\Sigma_k^p = \Pi_k^p$, então $\Sigma_{k+1}^p \subseteq \Sigma_k^p$. O resultado segue pelo teorema anterior, uma vez que $\Sigma_k^p \subseteq \Sigma_{k+1}^p$.

Seja $L \in \Sigma_{k+1}^p$, ou seja, existe uma máquina de Turing M_L de tempo polinomial e um polinômio $q(.)$ tal que

$$x \in L \;\Leftrightarrow\; \exists w_1\ \forall w_2\ \exists w_3\ \ldots\ Q_{k+1} w_{k+1}\ M_L(x, w_1, \ldots, w_{k+1}) = 1,$$

onde $w_i \in \{0,1\}^{q(|x|)}$ para todo $1 \leq i \leq k+1$. Considere agora a linguagem

$$L' = \{\langle x, w_1\rangle : |w_1| = q(|x|) \text{ e } \forall w_2\ \exists w_3\ \ldots\ Q_{k+1} w_{k+1}\ M_L(x, w_1, \ldots, w_{k+1}) = 1\}.$$

É imediato que $x \in L$ se e somente se existe w_1 de tamanho $q(|x|)$ tal que $\langle x, w_1\rangle \in L'$ (1). Além disso, segue a partir da definição de L' que $L' \in \Pi_k^p$. Portanto, por hipótese, temos que $L' \in \Sigma_k^p$. Em outras palavras, existe uma máquina de Turing eficiente $M_{L'}$ e um polinômio $q'(.)$ tal que:

$$\langle x, w_1\rangle \in L' \;\Leftrightarrow\; \exists u_1\ \forall u_2\ \exists u_3\ \ldots\ Q_k u_k\ M_{L'}(\langle x, w_1\rangle, u_1, \ldots, u_k) = 1 \quad (2),$$

onde $u_i \in \{0,1\}^{q'(|\langle x, 1^{q(|x|)}\rangle|)}$ para todo $1 \leq i \leq k$. Seja $p(.)$ um polinômio tal que $p(n) = q(n) + q'(|\langle 1^n, 1^{q(n)}\rangle|)$, para todo inteiro n. Combinando (1) e (2) e modificando a máquina $M_{L'}$, obtemos que existe uma máquina de Turing eficiente M tal que:

$$x \in L \;\Leftrightarrow\; \exists y_1 \in \{0,1\}^{p(|x|)}\ \forall y_2 \in \{0,1\}^{p(|x|)}\ \ldots\ Q_k y_k \in \{0,1\}^{p(|x|)}\ M(x, y_1, \ldots, y_k) = 1.$$

Portanto, $L \in \Sigma_k^p$, como queríamos demonstrar. □

Corolário 4.7.

(i) P = NP *se e somente se* PH = P.

(ii) NP = coNP *se e somente se* PH = NP.

Demonstração. A prova desses resultados segue imediatamente a partir dos teoremas anteriores e da observação de que P $= \Sigma_0^p$, NP $= \Sigma_1^p$ e coNP $= \Pi_1^p$. □

Portanto, para provar que P $\neq$ NP é suficiente demonstrar que existe um inteiro $k \geq 0$ tal que $\Sigma_k^p \subsetneq \Sigma_{k+1}^p$.

É muito improvável que possamos resolver os problemas em Σ_k^p (para k arbitrário) de modo eficiente. Logo, o corolário 4.7 fornece uma evidência forte de que P $\neq$ NP. No entanto, assim como ocorre com o problema P vs NP, apenas simulação e diagonalização não são suficientes para resolver o problema em aberto 4.3. Veremos o principal motivo por trás disso no capítulo 5.

O próximo resultado (em conjunto com o teorema 3.11) mostra como a hierarquia polinomial se relaciona com as classes de complexidade discutidas anterioremente.

Lema 4.8. PH $\subseteq$ PSPACE.

Demonstração. Seja $L \in$ PH. Então existe uma máquina de Turing M de tempo polinomial e um polinômio $q(.)$ de acordo com a definição 4.2. Vamos exibir um algoritmo recursivo A que decide L em espaço polinomial.

Com entrada x, A computa em cada nível da recursão da seguinte maneira: (1) verifica se a profundidade da recursão é par ou ímpar; (2) gera todas as palavras possíveis de tamanho $q(|x|)$; (3) realiza $2^{q(|x|)}$ chamadas recursivas; (4) retorna a conjunção ou disjunção do resultado dessas chamadas de acordo com a paridade verificada em (1). No caso base, nosso algoritmo recursivo invoca a máquina de Turing M com a entrada x e as k palavras geradas nas etapas anteriores. Note que, por construção, esse algoritmo decide a linguagem L.

Observe que só precisamos armazenar em cada um dos k níveis da recursão qual foi a última palavra gerada e um bit adicional que guarda o resultado intermediário da operação realizada em (4). Isso pode ser feito pois podemos invocar o algoritmo recursivamente logo após a geração da próxima palavra. Finalmente, uma vez que M computa em tempo polinomial (e portanto em espaço polinomial), o algoritmo utiliza espaço polinomial no tamanho da entrada, ou seja, $L \in$ PSPACE.

O teorema anterior também pode ser provado, por indução em k, se utilizarmos a linguagem L' obtida a partir de L da mesma forma como foi feito nas demonstrações anteriores. Por exemplo, por hipótese de indução, existe uma máquina de espaço polinomial que decide L'. Porém, sabemos que $x \in L$ se e somente se existe w_1 tal que $\langle x, w_1 \rangle \in L'$. Essa última condição pode ser facilmente verificada em espaço polinomial. $\square$

Questão em Aberto 4.9. PH = PSPACE *?*

Vamos analisar melhor essa relação de classes após estudarmos a existência de problemas completos para PH e suas subcasses Σ_k^p.

Definição 4.10. [Problemas Completos em PH]. *Dizemos que uma linguagem $L \subseteq \{0,1\}^\star$ é Σ_k^p-completa se $L \in \Sigma_k^p$ e toda linguagem em Σ_k^p é Karp-redutível à L. Analogamente, dizemos que L é* PH*-completa se $L \in$ PH e toda linguagem em* PH *é Karp-redutível à L.*

Teorema 4.11. *Para todo $k > 0$, a classe de complexidade Σ_k^p possui o seguinte problema completo:*

$$\Sigma_k^p\text{-SAT} = \{\langle \varphi \rangle : \exists w_1 \ \forall w_2 \ \exists w_3 \ldots \ Q_k w_k \ \varphi(w_1, \ldots, w_k) = 1\},$$

onde φ é uma fórmula proposicional arbitrária com kn variáveis, cada w_i é uma sequência de bits de tamanho n e Q_k é um quantificador que depende da paridade de k.

Demonstração. A prova desse resultado pode ser obtida no livro de Papadimitriou [88]. □

Para os primeiros níveis da hierarquia polinomial, diversos problemas naturais completos são conhecidos. O artigo de Schaefer e Umans [96] apresenta diversos exemplos. Em relação ao caso geral, temos os seguintes resultados.

Lema 4.12. [Σ_k^p é fechada por redução polinomial].
Se $L \in \Sigma_k^p$ e $L' \leq_p L$, então $L' \in \Sigma_k^p$.

Demonstração. Se L pertence à classe Σ_k^p, então existe uma máquina de Turing M que computa em tempo polinomial e um polinômio $q(.)$ tais que

$$x \in L \;\Leftrightarrow\; \exists w_1 \in \{0,1\}^{q(|x|)}\; \forall w_2 \in \{0,1\}^{q(|x|)} \ldots\; Q_k w_k \in \{0,1\}^{q(|x|)}\; M(x, w_1, \ldots, w_k) = 1,$$

onde Q_k denota $\forall$ ou $\exists$ se k for par ou ímpar, respectivamente. Por outro lado, como $L' \leq_p L$, existe uma função $f : \{0,1\}^\star \to \{0,1\}^\star$ computável em tempo polinomial tal que $x \in L'$ se e somente se $f(x) \in L$. Portanto,

$$x \in L' \;\Leftrightarrow\; f(x) \in L \;\Leftrightarrow\; \exists w_1\; \forall w_2\; \ldots\; Q_k w_k\; M(f(x), w_1, \ldots, w_k) = 1,$$

Observe que a máquina de Turing $M_f(x, w_1, \ldots, w_k) = M(f(x), w_1, \ldots, w_k)$ computa em tempo polinomial, uma vez que f e M são polinomiais. Isso completa a prova de que $L' \in \Sigma_k^p$. □

Lema 4.13. *Se existe uma linguagem* PH*-completa, então a hierarquia polinomial colapsa em algum nível finito, ou seja, existe $k \geq 0$ tal que* $\mathrm{PH} = \Sigma_k^p$.

Demonstração. Se L é uma linguagem PH-completa, então existe k tal que $L \in \Sigma_k^p$. Por hipótese, para toda linguagem $L' \in \mathrm{PH}$, temos que $L' \leq_p L$. Segue pelo lema anterior que $\mathrm{PH} \subseteq \Sigma_k^p$ e portanto $\mathrm{PH} = \Sigma_k^p$. □

O próximo teorema mostra que, se PH = PSPACE, então ocorre o colapso da hierarquia polinomial. A demonstração utiliza o método da completude.

Teorema 4.14. *Se* PH = PSPACE*, então existe k tal que* $\mathrm{PH} = \Sigma_k^p$.

Demonstração. A seguinte linguagem é PSPACE-completa:

$$L = \{\langle M, x, 1^s\rangle : M \text{ aceita a entrada } x \text{ utilizando no máximo } s \text{ células de memória }\}.$$

Através de simulação podemos mostrar que $L \in \mathrm{PSPACE}$. Por outro lado, se L' é uma linguagem decidida em espaço polinomial por uma máquina $M_{L'}$, podemos usar essa máquina para provar que $L' \leq_p L$. Isso mostra que a linguagem L é completa para a classe PSPACE. Portanto, se PH = PSPACE, temos que a classe de complexidade PH possui problemas completos. Pelo lema anterior, obtemos que existe k tal que $\mathrm{PH} = \Sigma_k^p$, como queríamos demonstrar. □

Vimos na seção 2.7 que as classes P e NP podem ser caracterizadas através de certos conceitos lógicos. De forma análoga, é possível caracterizar a classe PH em termos de expressibilidade lógica. A hierarquia polinomial corresponde à classe de propriedades expressáveis na lógica de segunda ordem. Veja o livro de Immerman [61] para mais detalhes.

Outras três definições equivalentes para a classe PH são conhecidas. Uma envolve o conceito de máquinas alternantes, que será discutido na próxima seção. As outras duas são basedas em oráculos e certos tipos de circuitos, respectivamente. Para mais informações, consulte o livro de Arora e Barak [7], por exemplo.

4.2 Alternância

Nesta seção veremos brevemente como a alternância de quantificadores utilizada na definição da hierarquia polinomial pode ser transferida para dentro das máquinas de Turing. Essas ideias serão úteis na demonstração de alguns resultados da seção 4.3. As máquinas de Turing alternantes foram definidas pela primeira vez no artigo de Chandra et al. [20].

Uma configuração de uma máquina de Turing é uma descrição completa do estado da computação da máquina. Note que, para determinarmos o próximo passo de uma máquina de Turing, basta conhecermos o conteúdo atual das fitas, o estado atual da máquina e a posição de cada cabeça de leitura. Se M é uma máquina de Turing, dizemos que esse conjunto de parâmetros é uma configuração de M.

A computação de uma máquina de Turing não-determinística M (que sempre termina sua computação) com entrada x pode ser vista como uma *árvore finita de configurações*, de modo que existe uma aresta direcionada ligando as configurações C e C' se e somente se C' pode ser obtida a partir de C através da aplicação de uma das funções de transição de M, ou seja, em um passo de computação. A configuração inicial descreve o estado de M antes do primeiro passo de computação. As folhas da árvore (vértices com grau de saída zero) representam passos finais da computação de M, ou seja, configurações onde o estado é de aceitação ou de rejeição. Observe que M aceita a entrada x se e somente se existe um caminho na árvore do vértice que representa a configuração inicial até um vértice que descreve uma configuração em que M se encontra no estado de aceitação.

As máquinas de Turing alternantes (MTAs) generalizam naturalmente o conceito de não-determinismo em computação. A única diferença entre uma MTND e uma MTA é que a última possui uma partição no seu conjunto de estados que induz uma nova regra de aceitação para a palavra de entrada.

Definição 4.15. [Máquinas de Turing Alternantes]. *Uma máquina de Turing alternante* (MTA) *é uma máquina de Turing não-determinística M onde o conjunto de estados é dado*

por $Q = Q_\exists \cup Q_\forall$*, sendo* $Q_\exists$ *e* $Q_\forall$ *conjuntos disjuntos. Seja* $x \in \{0,1\}^\star$ *uma palavra, e considere a árvore da computação de* M *com entrada* x*, ou seja, a árvore de configurações discutida anteriormente.*

Vamos utilizar essa árvore para definir recursivamente se M *aceita a entrada* x*. Primeiro, todas as folhas da árvore de configuração com o estado de aceitação são definidas como configurações de aceitação. Uma configuração* C *com estado* $q \in Q_\forall$ *é uma configuração de aceitação se e somente se* todos *os vértices sucessores de* C *na árvore são configurações de aceitação. Por outro lado, uma configuração* C *com estado* $q \in Q_\exists$ *é uma configuração de aceitação se e somente se* existe *na árvore algum vértice sucessor de* C *que é uma configuração de aceitação. Observe que isso rotula todos os vértices como configurações de aceitação ou rejeição. Finalmente, dizemos que* M *aceita* x *(ou seja,* $M(x) = 1$*) se a configuração inicial é uma configuração de aceitação.*

Uma linguagem L *é decidida por uma máquina de Turing alternante* M *quando* $x \in L$ *se e somente se* $M(x) = 1$*. A definição da complexidade de tempo e de espaço de uma* MTA *não sofre nenhuma alteração em relação às mesmas definições para* MTNDs.

O leitor interessado em exemplos de máquinas de Turing alternantes pode consultar o livro de Sipser [103].

Definição 4.16. [Alternância e Classes de Complexidade]. *Seja* $f : \mathbb{N} \to \mathbb{N}$ *uma função. Uma linguagem* $L \subseteq \{0,1\}^\star$ *pertence a classe* ATIME$(f(n))$ *se existe uma máquina de Turing alternante* M *que decide* L *em tempo* $O(f(n))$*. Similarmente, dizemos que* L *pertence a classe* ASPACE$(f(n))$ *se existe uma máquina de Turing alternante* M *que decide* L *em espaço* $O(f(n))$*.*

De modo análogo às classes de complexidade P e PSPACE, temos as classes AP e APSPACE.

Definição 4.17. [Classes Alternantes].
Definimos as seguintes classes de complexidade alternantes:

(i) AP $= \bigcup_{k\geq 1}$ATIME(n^k)*;*

(ii) APSPACE $= \bigcup_{k\geq 1}$ASPACE(n^k)*;*

(iii) AEXP $= \bigcup_{k\geq 1}$ATIME(2^{n^k})*.*

A alternância nos fornece uma nova caracterização para diversas classes de complexidade estudadas anteriormente.

Teorema 4.18. [Relação entre ATIME e ASPACE]. *Seja* $f(n) : \mathbb{N} \to \mathbb{N}$ *uma função com* $f(n) \geq n$ *para todo natural* n*. Temos as seguintes inclusões de classe:*

(i) $\mathrm{ATIME}(f(n)) \subseteq \mathrm{SPACE}(f(n)) \subseteq \mathrm{ATIME}(f^2(n))$.

(ii) $\mathrm{ASPACE}(f(n)) = \mathrm{TIME}(2^{O(f(n))})$.

Demonstração. A demonstração desse resultado pode ser encontrada no livro de Sipser [103]. □

Corolário 4.19.

(i) AP = PSPACE.

(ii) APSPACE = EXP.

(iii) AEXP = EXPSPACE.

O corolário anterior sugere que a introdução de alternância aumenta significativamente o poder das classes de complexidade envolvidas (compare o corolário com o teorema 3.11).

Limitando o número de alternâncias, podemos obter uma nova caracterização dos diversos níveis da hierarquia polinomial.

Definição 4.20. *Seja $T(n) : \mathbb{N} \to \mathbb{N}$ uma função arbitrária. Para todo inteiro positivo k, dizemos que uma linguagem L pertence à classe de complexidade Σ_k^p-TIME$(T(n))$ se existe uma máquina de Turing alternante M que decide L em tempo $O(T(n))$ tal que:* (1) *o estado inicial de M pertence ao conjunto $Q_\exists$;* (2) *para toda entrada x e para todo caminho na árvore de configurações da computação de M com entrada x, existe uma alternância de tamanho máximo $k-1$ entre configurações com estado em $Q_\exists$ e $Q_\forall$.*

Teorema 4.21. [Alternancia e PH].
Para todo $k \in \mathbb{N}$, temos $\Sigma_k^p = \bigcup_{c \geq 1} \Sigma_k^p$-TIME$(n^c)$.

Demonstração. Seja $L \in \Sigma_k^p$. Então existe uma máquina de Turing M de tempo polinomial e um polinômio $q(.)$ tal que:

$$x \in L \;\Leftrightarrow\; \exists w_1 \in \{0,1\}^{q(|x|)}\ \forall w_2 \in \{0,1\}^{q(|x|)} \ldots\ Q_k w_k \in \{0,1\}^{q(|x|)}\ M(x, w_1, \ldots, w_k) = 1,$$

A seguinte máquina "Σ_k^p-alternante" decide a linguagem L:

1) Gera não-deterministicamente com estados existenciais uma palavra $w_1 \in \{0,1\}^{q(|x|)}$.
2) Gera não-deterministicamente com estados universais uma palavra $w_2 \in \{0,1\}^{q(|x|)}$.

$\vdots$

k) Gera através do seu não-determinismo uma palavra $w_k \in \{0,1\}^{q(|x|)}$. O tipo dos estados utilizados durante a geração dessa palavra depende da paridade de k.

Simula *deterministicamente* com estados do mesmo tipo utilizados na etapa anterior a computação de M com entrada $\langle x, w_1, \ldots, w_k \rangle$. Nosso algoritmo aceita x se e somente se M aceita essa entrada. Observe que, em partes determinísticas da computação, o tipo de estado não altera o conjunto de configurações de aceitação final.

Note que esse algoritmo alternante computa em tempo polinomial. Além disso, ocorre exatamente $k-1$ alternâncias entre estados existencias e universais. Finalmente, segue pela definição de aceitação por MTAs que nosso algoritmo decide a linguagem L, ou seja, $L \in \Sigma_k^p\text{-TIME}(n^c)$ para alguma constante c adequada.

Suponha agora que $L \in \Sigma_k^p\text{-TIME}(n^c)$ para alguma constante $c \geq 1$. Em outras palavras, existe uma MTA A com no máximo $k-1$ alternâncias que decide L em tempo dn^c, para alguma constante d. Vamos exibir uma máquina de Turing determinística M que computa em tempo polinomial tal que:

$$x \in L \;\Leftrightarrow\; \exists w_1\, \forall w_2\, \exists w_3\, \ldots\, Q_k w_k\; M(x, w_1, \ldots, w_k) = 1 \quad (1),$$

onde $|x| = n$ e $w_i \in \{0,1\}^{dn^c}$, para $1 \leq i \leq k$.

Com entrada $\langle x, w_1, \ldots, w_k \rangle$, M simula deterministicamente a máquina alternante A, escolhendo a função de transição a ser aplicada através das palavras $w_1, \ldots, w_k$. Especificamente, M guarda durante a simulação de A o número k' de alternâncias ocorridas até a simulação do passo atual. Observe que $0 \leq k' \leq k-1$. No i-ésimo passo a ser simulado ($1 \leq i \leq dn^c$), M escolhe a função de transição de acordo com o i-ésimo bit da palavra de entrada $w_{k'+1}$. Por último, M aceita $\langle x, w_1, \ldots, w_k \rangle$ se e somente se A aceita x com as escolhas não-determinísticas obtidas dessa forma. É fácil provar por indução em k que M se comporta de modo que (1) seja verdadeiro (em particular, note que bits de entrada que não são acessados por M são irrelevantes). Isso completa a prova de que $L \in \Sigma_k^p$.

Observe que a complexidade de tempo determinística do verificador (calculada em função do tamanho da entrada x) é dada pela complexidade de tempo não-determinística da máquina alternante, e vice-versa. Isso ocorre pois a simulação pode ser embutida dentro do código das máquinas de Turing envolvidas, o que não aumenta a complexidade final dos algoritmos. □

O último teorema refina a definição da hierarquia polinomial, uma vez que cada nível de PH se torna a união de infinitas classes de complexidade.

4.3 Cotas Inferiores para SAT

Por conveniência, definimos na seção 1.7 a complexidade de espaço de um algoritmo levando em conta o número de células ocupadas pela palavra de entrada. Por isso, todo algoritmo analisado dessa forma possui complexidade de espaço $\Omega(n)$. No entanto, existem muitos algoritmos interessantes que não alteram o conteúdo das células ocupadas pela palavra de entrada e utilizam espaço adicional $o(n)$. Utilizar tão pouco espaço é possível pois com apenas $O(\log n)$ células podemos armazenar diversos apontadores e contadores. Embora os algoritmos sublineares sejam discutidos apenas nesta seção, eles são bastante estudados em complexidade computacional.

Especificamene nesta seção, considere a complexidade de espaço de uma máquina de Turing como sendo o número de células distintas acessadas pelas cabeças de leitura de M em fitas diferentes da fita de entrada. Além disso, assumimos que as máquinas de Turing não podem escrever dados na fita de entrada. Essa fita é usada apenas para a leitura da palavra de entrada.

Com essa alteração, podemos discutir a existência de máquinas de Turing que computam em espaço sublinear. Diversos problemas não-triviais podem ser resolvidos dessa forma. Por exemplo, Reingold [92] descobriu recentemente um algoritmo (determinístico) com complexidade de espaço $O(\log n)$ que decide se existe um caminho ligando dois vértices em um grafo não-direcionado. Notamos também que a maioria das reduções[1] utilizadas nas provas de NP-completude podem ser computadas em espaço $o(n)$.

Embora a maioria dos especialistas acredite que SAT $\notin$ P, esse resultado não parece próximo de ser provado. Intuitivamente, deve ser muito mais fácil demonstrar o seguinte:

1) Não existe algoritmo de tempo linear que decide SAT.
2) Para todo inteiro positivo b, não existe algoritmo de espaço $O((\log n)^b)$ que decide SAT.

Esses resultados parecem extremamente fracos. No entanto, não sabemos como prová-los. Veremos nesta seção que podemos eliminar pelo menos o mais trivial dos algoritmos. Especificamente, demonstraremos que não existe máquina de Turing determinística que decide SAT em tempo $O(n^\lambda)$ e espaço $O((\log n)^b)$, onde $\lambda > 1$ é uma constante e b é um inteiro positivo arbitrário.

O valor da constante λ foi melhorado diversas vezes. Atualmente, temos que $\lambda > 1.8$ (Williams [110]). O desafio atual é provar que $\lambda \geq 2$. É interessante notar que esses resultados também são válidos para máquinas de Turing mais poderosas, dotadas de acesso aleatório à memória. Além disso, teoremas mais fortes podem ser provados para problemas

[1] Consideramos que a palavra computada pela redução é armazenada em uma fita de saída utilizada somente para escrita. Assim como ocorre com a fita de entrada, o número de células escritas na fita de saída não é levado em conta no cálculo da complexidade de espaço do algoritmo.

computacionais em níveis superiores da hierarquia polinomial. Por simplicidade, demonstraremos nesta seção que λ pode ser tomado arbitrariamente próximo de $\sqrt{2}$ (Lipton e Viglas [76]).

A técnica utilizada nessas demonstrações apareceu pela primeira vez em um artigo de Kannan [65]. A fim de obtermos uma contradição, assumimos que uma determinada inclusão de classes é verdadeira. Utilizamos essa hipótese para mostrar que é possível acelerar certos tipos de computação. Por último, mostramos que isso viola algum teorema de hierarquia conhecido. Isso significa que, em última instância, estamos utilizando um argumento de diagonalização, já que esse método é utilizado para provar os teoremas de hierarquia. O conceito de alternância será crucial na hora de colocar em prática essas ideias. Observe que o enunciado do nosso resultado não diz nada sobre alternância.

Definição 4.22. [Classe de Complexidade DTISP]. *Sejam $T, S : \mathbb{N} \to \mathbb{N}$ duas funções arbitrárias. Dizemos que uma linguagem $L \in \{0,1\}^\star$ pertence à classe* $\mathrm{DTISP}(T(n), S(n))$ *se existe uma máquina de Turing determinística M que decide a linguagem L em tempo $O(T(n))$ e espaço $O(S(n))$.*

Primeiro mostraremos que $\mathrm{NTIME}(n) \nsubseteq \mathrm{DTISP}(n^c, n^d)$, onde $c(c+2d) < 2$. Este será o teorema principal desta seção. Em seguida, vamos utilizar a completude de SAT e tomar d arbitrariamente pequeno para provar que $\mathrm{SAT} \notin \mathrm{DTISP}(n^{\sqrt{2}-\varepsilon}, (\log n)^b)$, para todo $\varepsilon, b > 0$. A demonstração do resultado principal será baseada em diversas inclusões de classes. Assuma que $\mathrm{NTIME}(n) \subseteq \mathrm{DTISP}(n^c, n^d)$. Mostraremos que:

$$\mathrm{NTIME}(n^k) \subseteq \mathrm{DTISP}(n^{kc}, n^{kd}) \subseteq \Sigma_2^p\text{-}\mathrm{TIME}(n^{k(d+c/2)}) \subseteq \mathrm{NTIME}(n^{ck(d+c/2)}),$$

onde k é um inteiro positivo escolhido de forma que em nenhum momento tenhamos computações em tempo sublinear (precisamos pelo menos ler a palavra de entrada nas MTs envolvidas na demonstração). Note que se $ck(d+c/2) < k$, temos uma violação do teorema de hierarquia não-determinístico demonstrado na seção 3.5. Reescrevendo essa desigualdade, fica provado que, para $c(c+2d) < 2$, $\mathrm{NTIME}(n) \nsubseteq \mathrm{DTISP}(n^c, n^d)$.

Durante as demonstrações vamos utilizar o fato de que é possível converter verificadores de Σ_k^p em máquinas "Σ_k^p-alternantes" e vice-versa sem alterar as complexidades de tempo envolvidas (veja a prova do teorema 4.21). A primeira inclusão é a mais simples de ser provada. Utilizamos um argumento usual de preenchimento.

Lema 4.23. *Se* $\mathrm{NTIME}(n) \subseteq \mathrm{DTISP}(n^c, n^d)$, *então* $\mathrm{NTIME}(n^k) \subseteq \mathrm{DTISP}(n^{kc}, n^{kd})$, *onde k é qualquer inteiro positivo.*

Demonstração. Primeiro observe que o resultado é bastante intuitivo: se podemos computar n passos não-determinísticos em tempo (ou espaço) determinístico $g(n)$, então podemos computar n^k passos não-determinísticos em tempo (ou espaço) determinístico $g(n)^k$.

Suponha que $L \in \mathrm{NTIME}(n^k)$. Considere a linguagem $L' = \{\langle x, 1^{|x|^k}\rangle : x \in L\}$. Usando uma MTND que decide L em tempo $O(n^k)$, podemos construir outra MTND que decide L' em tempo $O(n)$. Da hipótese, temos que $L' \in \mathrm{DTISP}(n^c, n^d)$. Seja M' uma máquina de Turing determinística que decide L' com essa complexidade. Considere a máquina M que, com entrada x, cria a entrada $\langle x, 1^{|x|^k}\rangle$ e retorna $M'(\langle x, 1^{|x|^k}\rangle)$. Portanto:

$$M(x) = 1 \;\Leftrightarrow\; M'(\langle x, 1^{|x|^k}\rangle) = 1 \;\Leftrightarrow\; x \in L,$$

ou seja, M decide L. Fazendo $|x| = n$, temos que $|\langle x, 1^{|x|^k}\rangle|$ é $O(n^k)$. Como M' computa em tempo $O(n^c)$ e espaço $O(n^d)$, temos que M computa em tempo $O(n^{kc})$ e espaço $O(n^{kd})$. Isso completa a demonstração de que $L \in \mathrm{DTISP}(n^{kc}, n^{kd})$. □

A próxima inclusão mostra que podemos trocar tempo determinístico por alternância. Observe que o resultado não depende de nenhuma hipótese.

Lema 4.24. $\mathrm{DTISP}(n^{kc}, n^{kd}) \subseteq \Sigma_2^p\text{-}\mathrm{TIME}(n^{k(d+c/2)})$.

Demonstração. Utilizaremos pela primeira vez um argumento conhecido como *método das configurações*. Basicamente, vamos determinar se uma palavra é aceita por uma máquina de Turing verificando se existe um caminho válido entre a configuração inicial da máquina e uma configuração de aceitação. Além disso, quebraremos esse problema em diversas partes, resolvendo cada uma delas separadamente utilizando alternância.

Seja $L \in \mathrm{DTISP}(n^{kc}, n^{kd})$ e considere que M é uma máquina de Turing determinística que decide L em tempo $O(n^{kc})$ e espaço $O(n^{kd})$. Em relação à computação de M com uma entrada x de tamanho n, temos que $x \in L$ se e somente se *existem* $n^{kc/2}$ configurações de M tais que, *para todo* $i \in \mathbb{N}$ com $1 \leq i < n^{kc/2}$, a configuração C_{i+1} pode ser obtida a partir da configuração C_i em $k'n^{kc/2}$ passos (para alguma constante k' adequada) e a configuração $C_{n^{kc/2}}$ é uma configuração de aceitação. Portanto,

$$x \in L \;\Leftrightarrow\; \exists\, \langle C_1, \ldots, C_{n^{kc/2}}\rangle\ \forall\, \langle i\rangle\ [M'(x, \langle C_1, \ldots, C_{n^{kc/2}}\rangle, \langle i\rangle) = 1],$$

onde M' é uma máquina de Turing determinística que verifica a condição anterior. Como M computa em espaço $O(n^{kd})$, temos que $|\langle C_1, \ldots, C_{n^{kc/2}}\rangle|$ é $O(n^{kc/2}n^{kd})$. Por isso, apenas para ler a entrada e obter as configurações C_i e C_{i+1}, M' precisa de $O(n^{k(d+c/2)})$ passos. A computação de M' pode ser feita em tempo $O(n^{kc/2})$. Convertendo M' em uma máquina alternante, obtemos que $L \in \Sigma_2^p\text{-}\mathrm{TIME}(n^{k(d+c/2)})$. □

Finalmente, uma hipótese um pouco mais fraca do que a inicial nos permite trocar alternância por tempo não-determinístico.

Lema 4.25. *Se* $\mathrm{NTIME}(n^k) \subseteq \mathrm{DTIME}(n^{kc})$, *então* $\Sigma_2^p\text{-}\mathrm{TIME}(n^{k(d+c/2)}) \subseteq \mathrm{NTIME}(n^{ck(d+c/2)})$.

Demonstração. Primeiro observe que esse resultado é razoável: se podemos computar n^k passos não-determinísticos em tempo determinístico $(n^k)^c$, então podemos remover um nível de alternância com o mesmo acréscimo de complexidade (lembre que Σ_1^p-TIME$(f(n)) =$ NTIME$(f(n))$).

Formalmente, se $L \in \Sigma_2^p$-TIME$(n^{k(d+c/2)})$, então existe uma máquina de Turing determinística M tal que:

$$x \in L \;\Leftrightarrow\; \exists w_1\ \forall w_2\ [M(x, w_1, w_2) = 1],$$

onde M computa em tempo $O(n^{k(d+c/2)})$, onde $n = |x|$. O tamanho das palavras w_1 e w_2 é limitado pela complexidade de tempo de M.

Como, por hipótese, temos que NTIME$(n^k) \subseteq$ DTIME(n^{ck}), seque pelo lema 4.23 que NTIME$(n^{k(d+c/2)}) \subseteq$ DTIME$(n^{ck(d+c/2)})$. Considere a seguinte linguagem:

$$L' = \{\langle x, w_1\rangle : \exists w_2 \text{ tal que } M(x, w_1, w_2) = 0\}.$$

Observe que $L' \in$ NTIME$(n^{k(d+c/2)})$, e portanto $L' \in$ DTIME$(n^{ck(d+c/2)})$. Como classes determinísticas são fechadas por complementação, temos que $\overline{L'} \in$ DTIME$(n^{ck(d+c/2)})$. Além disso, $x \in L$ se e somente se existe uma palavra w_1 tal que $\langle x, w_1\rangle \in \overline{L'}$. Portanto, existe uma máquina de Turing determinística M' com complexidade de tempo $O(n^{ck(d+c/2)})$ que satisfaz:

$$x \in L \text{ se e somente se existe } w_1 \text{ tal que } M'(x, w_1) = 1.$$

Isso prova que $L \in$ NTIME$(n^{ck(d+c/2)})$. □

Teorema 4.26. NTIME$(n) \not\subseteq$ DTISP(n^c, n^d), *onde* $c(c + 2d) < 2$.

Demonstração. A demonstração é imediata a partir dos lemas 4.23, 4.24 e 4.25 e da discussão apresentada após a definição 4.22. □

Corolário 4.27. [Limitante Inferior para SAT].
Para todo inteiro positivo b e para todo $\varepsilon > 0$, SAT $\notin$ DTISP$(n^{\sqrt{2}-\varepsilon}, (\log n)^b)$.

Demonstração. É possível provar que o problema de decidir pertinência para linguagens em NTIME$(t(n))$ pode ser reduzido ao problema de verificar se uma fórmula proposicional de tamanho $O(t(n) \log t(n))$ é satisfatível. Além disso, cada bit de saída dessa redução pode ser computado em tempo linear e espaço poli-logaritmo (veja Arora e Barak [7]).

Suponha que existam ε e b tais que SAT $\in$ DTISP$(n^{\sqrt{2}-\varepsilon}, (\log n)^b)$. Sabemos que, para todo $\beta > 0$, uma função $O((\log n)^b)$ é também $O(n^\beta)$, e portanto SAT$\in$ DTISP$(n^{\sqrt{2}-\varepsilon}, n^\beta)$. Além disso, se $L \in$ NTIME(n), então $x \in L$ se e somente se a fórmula proposicional de tamanho $n \log n$ associada a x é satisfatível (veja a discussão do parágrafo anterior). Como essa redução pode ser computada em tempo linear e espaço poli-logaritmo e SAT$\in$ DTISP$(n^{\sqrt{2}-\varepsilon}, n^\beta)$, existe um polinômio $p(.)$ tal que:

$$\mathrm{NTIME}(n) \subseteq \mathrm{DTISP}((n \log n)^{\sqrt{2}-\varepsilon} + n, (n \log n)^{\beta} + p(\log n)) \subseteq \mathrm{DTISP}(n^{\sqrt{2}-\varepsilon/2}, n^{2\beta}).$$

Podemos tomar β arbitrariamente pequeno de modo que, para $c = \sqrt{2} - \varepsilon/2$ e $d = 2\beta$, tenhamos $c(c + 2d) < 2$. No entanto, isso contradiz o teorema 4.26. Por esse motivo, temos que $\mathrm{SAT} \notin \mathrm{DTISP}(n^{\sqrt{2}-\varepsilon}, (\log n)^b)$, como queríamos demonstrar. □

Recentemente, Williams [111] formalizou todas as provas conhecidas desses limitantes inferiores. Através disso, foi possível desenvolver um programa de computador capaz de melhorar o valor da constante λ obtido através desses métodos. Infelizmente, seu trabalho sugere que essas técnicas não são capazes de provar que $\lambda \geq 2$.

4.4 Reduções e NP ∩ coNP

Na seção 2.5 discutimos a noção de redução entre problemas computacionais e introduzimos o conceito de redução de Karp. Essa noção restrita de redução foi estudada porque ela é suficiente para a teoria de NP-completude. No entanto, em complexidade computacional é mais interessante desenvolver uma noção de redução que se comporte da seguinte maneira: um problema A se reduz a um problema B se a existência de um algoritmo eficiente para B implica a existência de um algoritmo eficiente para A. Por exemplo, pode ser interessante chamar diversas vezes o algoritmo eficiente que resolve o problema B quando se quer resolver o problema A. Além disso, podemos ter diferentes caminhos de execução de acordo com as respostas obtidas por essas chamadas. Essa noção mais geral de redução entre problemas é capturada pela noção de redução de Cook.

Esse conceito pode ser facilmente formalizado através do uso de máquinas de Turing com oráculo (MTO). As MTOs são máquinas de Turing usuais que possuem a habilidade adicional de realizar chamadas a um oráculo L. Mais especificamente, $L \subseteq \{0,1\}^\star$ é uma linguagem arbitrária e a máquina de Turing com acesso ao oráculo L é capaz de verificar instantaneamente se uma dada palavra pertence à linguagem L. Para isso, basta a MTO escrever essa palavra em uma fita adicional chamada fita do oráculo e utilizar uma instrução especial que obtém a resposta do oráculo. Esse procedimento pode ser realizado diversas vezes durante a computação da MTO.

Definição 4.28. [Máquinas de Turing com Oráculo (MTOs)]. *Uma máquina de Turing M com oráculo é uma máquina de Turing com uma fita especial adicional, chamada fita de oráculo, e três estados especiais: $q_{oráculo}$, q_{sim} e $q_{não}$. A computação de M é sempre relativa a um oráculo $L \subseteq \{0,1\}^\star$. Quando M entra no estado $q_{oráculo}$, seu próximo estado de execução depende do conteúdo w escrito na fita de oráculo. O novo estado será q_{sim} caso $w \in L$, e $q_{não}$ se $w \notin L$. O contéúdo da fita de oráculo permanece inalterado após essa chamada. Além disso, independente da linguagem L utilizada como oráculo, cada*

resposta obtida através do oráculo conta como apenas um passo computacional. Se M é uma máquina de Turing com oráculo, $L \subseteq \{0,1\}^\star$ é uma linguagem e $x \in \{0,1\}^\star$ é uma palavra de entrada, então o resultado da computação de M com entrada x e com acesso ao oráculo L será denotado por $M^L(x)$. As máquinas de Turing não-determinísticas com oráculo são definidas similarmente.

Definição 4.29. [Classes de Complexidade com Oráculo]. *Para toda linguagem $L \subseteq \{0,1\}^\star$, definimos a classe de complexidade P^L como sendo o conjunto de linguagens que podem ser decididas em tempo polinomial por máquinas de Turing determinísticas com acesso ao oráculo L. Similarmente, NP^L denota o conjunto de linguagens que podem ser decididas em tempo polinomial por máquinas de Turing não-determinísticas com acesso ao oráculo L.*

Definição 4.30. [Redução de Cook]. *Dizemos que uma linguagem L é Cook-redutível a uma linguagem L' (denotado por $L \leq_T L'$) se $L \in \mathrm{P}^{L'}$.*

Observe que a redução de Cook captura o conceito intuitivo de redução discutido no início da seção. Além disso, observe que se $L' \in \mathrm{P}$ e $L \leq_T L'$, então $L \in \mathrm{P}$. Isso é válido pois podemos substituir as chamadas ao oráculo L' pela própria computação de uma máquina eficiente que decide L'. Como a composição de polinômios é novamente um polinômio, obtemos a partir disso um algoritmo eficiente para L. Por último, se $L \leq_p L'$, então $L \leq_T L'$ (a redução de Karp é um caso particular da redução de Cook).

Veremos agora como a redução de Karp e a redução de Cook se relacionam com a classe $\mathrm{NP} \cap \mathrm{coNP}$.

Teorema 4.31. *Seja $L \subseteq \{0,1\}^\star$ uma linguagem e suponha que $L \in \mathrm{NP} \cap \mathrm{coNP}$. Se L é* NP*-completa, então* $\mathrm{NP} = \mathrm{coNP}$.

Demonstração. Primeiro vamos provar que a classe coNP é fechada por redução de Karp. Seja $L' \in \mathrm{coNP}$ e suponha que $L'' \leq_p L'$. Então $\overline{L''} \leq_p \overline{L'}$. Como $\overline{L'} \in \mathrm{NP}$ e NP é uma classe fechada por redução de Karp (a composição de uma redução eficiente com uma MTND eficiente é novamente um MTND eficiente), temos que $\overline{L''} \in \mathrm{NP}$. Isso prova que $L'' \in \mathrm{coNP}$.

Considere agora a linguagem L do enunciado do teorema. Como $L \in \mathrm{coNP}$, L é NP-completa e coNP é fechada por redução de Karp, temos que $\mathrm{NP} \subseteq \mathrm{coNP}$. Por outro lado, se $L' \in \mathrm{coNP}$ então $\overline{L'} \in \mathrm{NP}$. Como $\mathrm{NP} \subseteq \mathrm{coNP}$, segue que $\overline{L'} \in \mathrm{coNP}$. Por definição, isso significa que $L' \in \mathrm{NP}$, ou seja, $\mathrm{coNP} \subseteq \mathrm{NP}$. Isso completa a prova de que $\mathrm{NP} = \mathrm{coNP}$. □

Portanto, para provar que $\mathrm{NP} = \mathrm{coNP}$ basta demonstrar a existência de um problema NP-completo em coNP. No entanto, o que ocorre se esse problema for completo em relação à redução de Cook? Com um pouco mais de trabalho, é possível obter o mesmo resultado. O próximo teorema é devido à Selman [100].

Teorema 4.32. *Seja L uma linguagem em* NP $\cap$ coNP*. Se toda linguagem em* NP *for Cook-redutível à linguagem L, então* NP = coNP*.*

Demonstração. Seja $L \in$ NP $\cap$ coNP. Observe que, pelo mesmo argumento utilizado na prova do teorema 4.31, basta provar que se $L' \leq_T L$ então $L' \in$ coNP. Por sua vez, para provar isso é suficiente demontrar que se $L' \leq_T L$ então $L' \in$ NP, já que $L' \leq_T L$ se e somente se $\overline{L'} \leq_T L$.

Como $L \in$ NP $\cap$ coNP, existem verificadores eficientes V_1 e V_2 tais que:

(1) $z \in L$ se e somente se $\exists w \in \{0,1\}^{p_1(|x|)}$ tal que $V_1(z,w) = 1$;
(2) $z \notin L$ se e somente se $\exists w \in \{0,1\}^{p_2(|x|)}$ tal que $V_2(z,w) = 1$.

Ou seja, podemos provar eficientemente que uma palavra arbitrária $z \in \{0,1\}^\star$ pertence ou não à linguagem L.

Para demonstrarmos que a linguagem $L' \in$ NP precisamos exibir um verificador eficiente V' para L'. Seja M a máquina de Turing com oráculo que computa a redução $L' \leq_T L$. Portanto, temos que $z \in L'$ se e somente se $M^L(z) = 1$.

Observe que podemos *verificar eficientemente* cada resposta emitida pelo oráculo L usando os verificadores V_1 e V_2. Por isso, definimos V' de forma que, dada uma palavra $w = ((z_1, r_1, w_1), \ldots, (z_k, r_k, w_k))$, temos $V'(x,w) = 1$ se e somente se:

(a) M aceita a entrada x invocando o oráculo com as palavras $z_1, \ldots, z_k$ na fita de oráculo durante sua computação;
(b) As respostas obtidas por M nas chamadas ao oráculo são dadas pelos bits $r_1, \ldots, r_k$.

No entanto, como V' é uma máquina determinística que não possui acesso ao oráculo L, usamos as palavras $w_1, \ldots, w_k$ para confirmar a consistência das respostas codificadas pelos bits r_i. Isso significa que V' verifica se $V_1(z_i, w_i) = 1$ quando $r_i = 1$, e se $V_2(z_i, w_i) = 1$ quando $r_i = 0$.

É imediato a partir da construção de V' que se $x \in L'$ então existe palavra w tal que $V'(x,w) = 1$. Por outro lado, se $V'(x,w) = 1$ para alguma palavra w, então essa palavra codifica de forma consistente a computação de M e as respostas dadas pelo oráculo L, ou seja, $M^L(x) = 1$ e portanto $x \in L'$.

Finalmente, observe que como M computa em tempo polinomial e os certificados aceitos por V_1 e V_2 são polinomialmente limitados, temos que os certificados aceitos por V' são formados pela concatenação de uma quantidade polinomial de palavras de tamanho polinomial, ou seja, os certificados de V' são polinomialmente limitados. Além disso, V' computa em tempo polinomial pois as máquinas M, V_1 e V_2 são máquinas eficientes. Isso completa a prova de que V' é um verificador eficiente para L', ou seja, $L' \in$ NP. $\square$

As máquinas de Turing com oráculo serão utilizadas novamente no capítulo 5.

4.5 O Teorema de Ladner

Dada uma linguagem L em NP, se $L \notin$ P então L é NP-completa? Embora seja possível demonstrar que milhares de problemas são NP-completos, esse resultado não é válido. De fato, a menos que P = NP, é possível provar que existem linguagens intermediárias entre a classe P e a classe de linguagens NP-completas (NPC). Esse resultado é conhecido como Teorema de Ladner [73].

O problema computacional exibido por Ladner em sua demonstração é bastante artificial. No entanto, é bem provável que alguns problemas naturais possuam dificuldade intermediária entre as duas classes. Por exemplo, muitos pesquisadores acreditam que certos problemas ligados à criptografia (em particular, fatoração e logaritmo discreto) não pertençam às classes P e NPC, embora sejam membros da classe NP.

Além disso, há fortes razões para acreditarmos que algumas linguagens aparentemente difíceis de NP não são NP-completas. Por exemplo, algumas dessas linguagens estão em NP $\cap$ coNP. Se algum desses problemas for NP-completo, segue pelo Teorema 4.31 que NP = coNP, um resultado considerado improvável pela maior parte dos pesquisadores em complexidade computacional. Outro caso interessante ocorre com o problema de verificar se dois grafos arbitrários são isomorfos (ISO-GRAFOS). Se esse problema for NP-completo, então ocorre o colapso da hierarquia polinomial.

Teorema 4.33. *Se* ISO-GRAFOS *é uma linguagem* NP*-completa, então* $\Sigma_2^p = \Pi_2^p$.

Demonstração. A prova desse resultado pode ser encontrada no artigo original de Boppana et al. [14]. □

No entanto, tais resultados também servem como um indício de que esses problemas podem admitir algoritmos eficientes. Por exemplo, existe um algoritmo linear que verifica se dois grafos planares são isomorfos (veja o artigo de Hopcroft e Wong [59]). Um resultado muito mais geral aparece em um artigo recente de Grohe [49], que demonstra a existência de algoritmos eficientes capazes de verificar o isomorfismo em diversas classes interessantes de grafos.

Existem duas demontrações conhecidas do teorema de Ladner. A primeira envolve um argumento de diagonalização atrasada (veja a prova do teorema 3.34) e aparece no artigo original de Ladner. Ela também pode ser obtida nos livros de Goldreich [47] e Papadimitriou [88]. Uma segunda prova foi descoberta por Impagliazzo e pode ser obtida no livro de Arora e Barak [7]. Por utilizar uma técnica diferente, vamos apresentar esta última demonstração. Um esboço das duas provas está descrito em um manuscrito de Fortnow [37].

Teorema 4.34. [Teorema de Ladner]. *Suponha que* $\mathrm{P} \neq \mathrm{NP}$. *Então existe uma linguagem* $L \subseteq \{0,1\}^\star$ *tal que:*

(i) $L \in \mathrm{NP}$*;*

(ii) $L \notin \mathrm{P}$*;*

(iii) L *não é uma linguagem* NP*-completa.*

Demonstração. Seja $f : \mathbb{N} \to \mathbb{N}$ uma função arbitrária. Considere a seguinte linguagem:

$$\mathrm{SAT}_f = \{\psi 01^{n^{f(n)}} : |\psi| = n \text{ e } \psi \in \mathrm{SAT}\ \}.$$

Vamos definir uma função $H(n) : \mathbb{N} \to \mathbb{N}$ de forma que a linguagem SAT_H satisfaça os requisitos do teorema. Primeiro, observe que mais uma vez estamos utilizando o método do preenchimento, pois a complexidade de SAT_H depende essencialmente do crescimento da função H (assumindo que essa função pode ser computada de forma eficiente). Por outro lado, definiremos H de forma que seu crescimento dependa da dificuldade do próprio problema SAT_H (H será uma função bem-definida, uma vez que o valor $H(n)$ depende apenas de valores $H(n')$ para $n' < n$):

> Considere uma enumeração das máquinas de Turing. Para todo número natural n, definimos $H(n)$ como sendo o menor inteiro $i < \log\log n$ tal que, para toda palavra $x \in \{0,1\}^\star$ com $|x| \leq \log n$, a máquina de Turing M_i computa, tendo x como entrada, em no máximo $i|x|^i$ passos e $M_i(x) = 1$ se e somente se $x \in \mathrm{SAT}_H$. Se não existe tal i, então definimos $H(n) = \log\log n$.

A função H foi definida dessa forma para que possamos provar o seguinte resultado.

Lema 4.35. $\mathrm{SAT}_H \in \mathrm{P}$ *se e somente se* $H(n)$ *é* $O(1)$ *(ou seja, existe uma constante* C *tal que* $H(n) \leq C$ *para todo inteiro* n*).*

Demonstração. Primeiro, suponha que $\mathrm{SAT}_H \in \mathrm{P}$. Seja M uma máquina de Turing que decide essa linguagem em no maximo $c_1 n^{c_1}$ passos. Como adotamos uma codificação de máquinas de Turing em que cada máquina é representada por infinitas palavras, existe um inteiro $i \geq c_1$ tal que M é equivalente a M_i. Se $n > 2^{2^i}$, então $i < \log\log n$ e daí segue da definição de $H(n)$ que temos $H(n) \leq i$. Isso prova que $H(n)$ é uma função $O(1)$.

Suponha agora que $H(n)$ seja $O(1)$. Como H só pode assumir um número finito de valores, existe um inteiro i tal que $H(n) = i$ para valores arbitrariamente grandes de n. Segue pela definição da função $H(n)$ que a máquina de Turing M_i decide SAT_H em tempo in^i. Isso prova que $\mathrm{SAT}_H \in \mathrm{P}$. □

Vamos agora demonstrar que $H(n)$ pode ser computada eficientemente.

Lema 4.36. *A função $H(n)$ pode ser computada em tempo $O(n^5)$.*

Demonstração. Para computar $H(n)$ basta: (1) computar $H(i)$ para todo $i \leq \log n$; (2) decidir SAT em instâncias de tamanho limitado por $\log n$; (3) simular no máximo $\log\log n$ máquinas de Turing com entradas de tamanho limitado por $\log n$ por no máximo $\log\log n(\log n)^{\log\log n}$ passos. Para n suficientemente grande, temos:

$$\log\log n(\log n)^{\log\log n} = \log\log n(2^{\log\log n})^{\log\log n} \leq \log\log n\ (2^{\log n}) \leq n^2.$$

Por isso, podemos computar a função $H(n)$ em complexidade de tempo $T(n)$, onde a função $T(n)$ satisfaz:

$$T(n) \leq \log n\ T(\log n) + O(n^2) + O((\log\log n)nn^2).$$

O termo $\log n\ T(\log n)$ limita o número de passos realizados em (1). Além disso, em $O(n^2)$ passos podemos computar as instâncias de SAT (2). Finalmente, o último termo limita o tempo gasto com as simulações nas diversas entradas (3). Por conveniência, podemos escrever:

$$T(n) \leq \log n\ T(\log n) + O(n^4).$$

É fácil provar por indução que, para valores suficientemente grandes de n, temos $T(n) \leq kn^5$, onde k é uma constante adequada. Isso completa a demonstração de que $H(n)$ pode ser computada em tempo $O(n^5)$. □

O teorema de Ladner segue a partir dos próximos lemas.

Lema 4.37. $\mathrm{SAT}_H \in \mathrm{NP}$.

Demonstração. Os certificados para a linguagem SAT_H são os mesmos utilizados na demonstração de que SAT é uma linguagem NP-completa: as sequências de bits que satisfazem a fórmula ψ codificada na palavra de entrada x. A seguir descrevemos o verificador V para a linguagem SAT_H. Com entrada $\langle x, w\rangle$, V verifica primeiro se $x = \psi 01^k$ para algum inteiro k, e rejeita a entrada caso contrário. Seja $|\psi| = n$. O verificador V computa $H(n)$ e compara se $k = n^{H(n)}$, rejeitando a palavra de entrada se isso não ocorre. Finalmente, V aceita a entrada $\langle x, w\rangle$ se e somente se $\psi(w) = 1$.

Pelo lema 4.36, a função $H(n)$ pode ser computada em tempo polinomial em n. No nosso caso, n é dado pelo tamanho da fórmula ψ, e portanto é limitado pelo tamanho da entrada. Isso prova que V computa em tempo polinomial e portanto $\mathrm{SAT}_H \in \mathrm{NP}$. □

Lema 4.38. $\mathrm{SAT}_H \notin \mathrm{P}$.

Demonstração. Suponha que $\mathrm{SAT}_H \in \mathrm{P}$. Pelo lema 4.35, $H(n) \leq C$ para alguma constante C, ou seja, as palavras de SAT_H são apenas as palavras de SAT preenchidas com uma quantidade polinomial de bits adicionais. Por isso, segue facilmente que $\mathrm{SAT} \leq_p \mathrm{SAT}_H$, o que prova que $\mathrm{SAT} \in \mathrm{P}$ e portanto $\mathrm{P} = \mathrm{NP}$. Como isso contradiz a hipótese do teorema, temos que $\mathrm{SAT}_H \notin \mathrm{P}$. □

Lema 4.39. SAT_H *não é uma linguagem* NP*-completa.*

Demonstração. Suponha que SAT_H seja uma linguagem NP-completa. Então, existe uma função $g(x) : \{0,1\}^\star \to \{0,1\}^\star$ que computa a redução $\mathrm{SAT} \leq_p \mathrm{SAT}_H$ em tempo $O(n^{k-1})$, para algum inteiro k positivo. Isso garante que existe um inteiro n_0 tal que, para entradas de tamanho maior ou igual a n_0, a redução anterior pode ser computada em tempo n^k. Como provamos no lema 4.38 que $\mathrm{SAT}_H \notin \mathrm{P}$, segue pelo lema 4.35 que a função $H(n)$ assume valores arbitrariamente grandes. Seja n_1 o menor inteiro maior ou igual a n_0 tal que, para todo $n \geq n_1$, temos $H(n) > k$. Vamos mostrar que podemos construir um algoritmo recursivo A que decide a linguagem SAT em tempo polinomial. Com entrada φ de tamanho n, A computa da seguinte maneira:

(1) Se $n < n_1$, então A decide se a fórmula φ é satisfatível através de uma busca exaustiva.
(2) Caso contrário, A computa $w = g(\varphi)$. Se a palavra w obtida não for da forma $\psi 01^{|\psi|^{H(|\psi|)}}$, então A rejeita a fórmula φ. Caso contrário, recursivamente, A retorna $A(\psi)$.

Seja $|\psi| = n'$. Observe que se $n' < n_1$, então não ocorrem mais chamadas recursivas após a chamada realizada no passo 2. Se $n' \geq n_1$, o tamanho de w é limitado pela complexidade de tempo da redução computada por $g(\varphi)$, ou seja, $|w| \leq n^k$. Lembre que temos $|\varphi| = n$ e que $|w| = |\psi| + 1 + |\psi|^{H(|\psi|)} = n' + 1 + n'^{H(n')}$. Portanto, temos:

$$n'^{k+1} \leq n' + 1 + n'^{k+1} \leq n' + 1 + n'^{H(n')} = |w| \leq n^k.$$

Como k é fixo, para valores suficientemente grandes de n temos que $n' \leq n/2$, ou seja, $|\psi| \leq |\varphi|/2$. Isso mostra que o algoritmo A pode ser modificado de forma a realizar no máximo $\log n$ chamadas recursivas (adequamos o valor de n_1 na definição de A). Como as funções $g(\varphi)$ e $H(n)$ podem ser computadas em tempo polinomial, segue que a complexidade de tempo do algoritmo resultante é polinomial. Isso prova que $\mathrm{SAT} \in \mathrm{P}$, e portanto $\mathrm{P} = \mathrm{NP}$. Como isso contradiz a hipótese do teorema, fica demonstrado que SAT_H não é uma linguagem NP-completa. □

Isso completa a prova do Teorema de Ladner. □

É possível provar um resultado ainda mais forte. Se $\mathrm{P} \neq \mathrm{NP}$, então existem infinitos níveis de dificuldade entre as classes P e NP.

Teorema 4.40. *Suponha que* $\mathrm{P} \neq \mathrm{NP}$. *Então existe uma sequência infinita de linguagens* $L_1, L_2, \ldots$ *em* $\mathrm{NP} \setminus \mathrm{P}$, *de forma que* L_{i+1} *é Karp-redutível à* L_i, *mas* L_i *não é Cook-redutível à* L_{i+1}.

Demonstração. Veja a prova desse resultado no livro de Goldreich [47]. □

Do ponto de vista matemático, entender a complexidade computacional dos problemas intermediários é uma tarefa extremamente difícil. Se $\mathrm{P} \neq \mathrm{NP}$, sabemos que nenhum problema NP-completo possui algoritmo eficiente. No entando, continuamos sem saber, por exemplo, se o problema do isomorfismo entre grafos pode ser resolvido de forma eficiente.

4.6 NP-Completude, Isomorfismo e Densidade

Durante a computação de uma máquina de Turing ocorre apenas a manipulação de símbolos, através de regras bem definidas. Por isso, do ponto de vista da máquina, não existem grafos, matrizes ou mesmo números inteiros. Essas são interpretações que atribuímos aos símbolos manipulados.

Lembre que a demonstração de que SAT é um problema completo prova essencialmente que suas instâncias são capazes de codificar computação. Além disso, vimos na seção 2.5 que se L é uma linguagem em NP, então $\mathrm{SAT} \leq_p L$ se e somente se L é uma linguagem NP-completa. Em outras palavras, não existe linguagem NP-completa que não seja capaz de codificar computação.

Portanto, faz sentido investigar mais a fundo se existe realmente alguma diferença estrutural entre os problemas NP-completos. A menos de uma mudança facilmente computável na codificação das instâncias, não seriam todos eles o mesmo problema?

Definição 4.41. [Isomorfismo Polinomial entre Linguagens]. *Dizemos que as linguagens* $L_1, L_2 \subseteq \{0,1\}^\star$ *são polinomialmente isomorfas (denotado por* $L_1 \sim_p L_2$*) se existe uma função* $f : \{0,1\}^\star \to \{0,1\}^\star$ *tal que:*

(i) f *é uma bijeção, ou seja,* f *é uma função injetora e sobrejetora;*

(ii) Para todo $x \in \{0,1\}^\star$, *temos* $x \in L_1$ *se e somente se* $f(x) \in L_2$*;*

(iii) As funções f *e* f^{-1} *podem ser computadas em tempo polinomial.*

Nesse caso, dizemos que f *e* f^{-1} *são isomorfismos polinomiais.*

O leitor interessado pode observar na literatura que as reduções usuais utilizadas para provar a completude dos problemas quase nunca são isomorfismos polinomiais. Geralmente, isso ocorre porque as instâncias produzidas na redução são muito específicas, ao passo que o isomorfismo polinomial precisa ser sobrejetivo.

No entanto, existe um método relativamente simples que pode ser usado para converter diversas reduções em isomorfismos polinomiais. Em particular, é possível aplicar esse método para demonstrar que todos os problemas NP-completos conhecidos são polinomialmente isomorfos. Consulte o livro de Papadimitriou [88] para mais detalhes. Devido a essa enorme evidência experimental, Hartmanis e Berman [50] sugeriram a seguinte conjectura.

Questão em Aberto 4.42. [Conjectura do Isomorfismo de Berman-Hartmanis]. *Se $L_1, L_2 \subseteq \{0,1\}^\star$ são linguagens* NP*-completas, então L_1 e L_2 são polinomialmente isomorfas?*

Como o isomorfismo polinomial é uma relação de equivalência, essa conjectura procura estabelecer essencialmente que todos os problems NP-completos são equivalentes.

É simples perceber que a conjectura do isomorfismo implica que $\mathrm{P} \neq \mathrm{NP}$. Primeiro, suponha que $\mathrm{P} = \mathrm{NP}$. Então todas as linguagens não-triviais de NP são NP-completas, e portanto polinomialmente isomorfas. Por outro lado, uma linguagem finita e uma linguagem infinita não podem ser polinomialmente isomorfas, uma vez que não existe bijeção entre um conjunto finito e um conjunto infinito.

Apesar da conjectura do isomorfismo ser bastante intuitiva, alguns resultados indicam que ela pode ser falsa. Em particular, ela não é válida na presença de um oráculo aleatório (Kurtz et al. [71]). Veremos como resultados envolvendo oráculos aleatórios podem ser formalizados na seção 5.5.

A próxima definição introduz uma propriedade estrutural importante das linguagens.

Definição 4.43. [Densidade de uma Linguagem]. *Seja $L \subseteq \{0,1\}^\star$ uma linguagem arbitrária. A função de densidade de L é dada por $dens_L(n) = |\{x \in L : |x| \leq n\}|$. Ou seja, $dens_L(n)$ é o número de palavras de tamanho menor ou igual a n que pertecem à linguagem L.*

Observe que o crescimento da função de densidade é no máximo exponencial. Por isso, podemos particionar as linguagens em duas classes.

Definição 4.44. [Linguages Densas e Linguagens Esparsas]. *Seja $L \subseteq \{0,1\}^\star$ uma linguagem arbitrária e $dens_L(n) : \mathbb{N} \to \mathbb{N}$ sua função de densidade. Dizemos que L é uma linguagem esparsa se existe um polinômio $p(x)$ com $dens_L(n) \leq p(n)$, para todo inteiro positivo n. Caso contrário, dizemos que L é uma linguagem densa.*

A seguir vamos relacionar a noção de densidade com o conceito de isomorfismo polinomial.

Teorema 4.45. [Isomorfismo Polinomial e Densidade]. *Sejam $L_1, L_2 \subseteq \{0,1\}^\star$ duas linguagens arbitrárias e assuma que L_1 e L_2 são polinomialmente isomorfas. Então L_1 é uma linguagem densa se e somente se L_2 é uma linguagem densa.*

Demonstração. Como $L_1 \sim_p L_2$ se e somente se $L_2 \sim_p L_1$, basta provar que se L_1 é uma linguagem esparsa então L_2 é uma linguagem esparsa. Sejam $f : \{0,1\}^\star \to \{0,1\}^\star$ um isomorfismo polinomial entre as linguagens L_1 e L_2 e $p(.)$ um polinômio tal que $dens_{L_1}(n) \leq p(n)$. Como $f^{-1}(x)$ pode ser computada em tempo polinomial, existe um polinômio $q(.)$ tal que, para toda palavra x com $|x| \leq n$, temos $|f^{-1}(x)| \leq q(n)$. Como f^{-1} é uma função injetora e $x \in L_2$ implica que $f^{-1}(x) \in L_1$, segue que $dens_{L_2}(n) \leq p(q(n))$. Isso completa a prova de que L_2 é uma linguagem esparsa. □

Não é difícil provar que algumas linguagens NP-completas envolvendo grafos são densas. Por isso, se a conjectura do isomorfismo é verdadeira, não existem linguagens NP-completas esparsas. No entanto, não é necessário assumir essa conjectura para provar esse resultado. Mahaney [78] foi o primeiro a demonstrar que, a menos que P = NP, não existem linguagens NP-completas esparsas. Apresentaremos a seguir uma prova muito mais simples descoberta posteriormente por Ogihara e Watanabe [86].

Teorema 4.46. [Teorema de Mahaney]. *Se* P $\neq$ NP*, então toda linguagem* NP*-difícil é densa.*

Demonstração. Suponha que $S \subseteq \{0,1\}^\star$ seja um linguagem esparsa, ou seja, existe um polinômio $p(.)$ tal que $dens_S(n) \leq p(n)$ para todo inteiro positivo n. Vamos demonstrar que se SAT $\leq_p S$ (i.e., S é uma linguagem NP-difícil), então existe um algoritmo polinomial para SAT. Observe que, pela contrapositiva, isso demonstra o teorema de Mahaney. A prova utiliza uma variação do método do prefixo, encontrado pela primeira vez na demonstração do teorema 2.15.

Seja $\varphi(x_1, \ldots, x_n)$ uma fórmula proposicional em forma normal conjuntiva. Toda palavra com no máximo n bits pode ser considerada uma atribuição parcial para a fórmula φ. Se $\rho = \rho_1 \ldots \rho_k$ e $\tau = \tau_1 \ldots \tau_k$ são palavras de k bits, dizemos que $\rho \preceq \tau$ se τ não precede ρ segundo a ordem lexicográfica. Se φ é uma fórmula satisfatível, dizemos que a atribuição ρ é a menor atribuição que satisfaz φ se $\varphi(\rho) = 1$ e, para toda atribuição τ com $\varphi(\tau) = 1$, temos $\rho \preceq \tau$. Além disso, dizemos que uma atribuição parcial ρ pode ser estendida a uma atribuição que satisfaz a fórmula φ se existe $\rho' \in \{0,1\}^{n-|\rho|}$ tal que $\varphi(\rho\rho') = 1$. Considere a seguinte linguagem:

$$\text{SAT}^* = \{\langle \varphi(x_1, \ldots, x_n), \tau \rangle \ : \exists \rho \preceq \tau, \rho' \in \{0,1\}^{n-|\rho|} \text{ tal que } \varphi(\rho\rho') = 1\}.$$

A linguagem SAT* é formada por pares de fórmulas e atribuições parciais tais que existe uma atribuição parcial menor ou igual (segundo a ordem lexicográfica) à atribuição parcial de entrada, e que pode ser estendida a uma atribuição que satisfaz a fórmula de entrada. Observe que $\rho\rho'$ serve como certificado eficiente para a linguagem SAT*. Por isso, SAT* $\in$ NP. Como SAT é uma linguagem NP-completa, temos que SAT* $\leq_p$ SAT. Pela transitivade da redução de Karp, obtemos que SAT* $\leq_p S$. Seja $h : \{0,1\}^\star \to \{0,1\}^\star$ uma redução de tempo polinomial entre SAT* e S. Como para toda atribuição parcial ρ temos $|\rho| \leq |\varphi|$, existe uma constante c_1 tal que $|h(\langle\varphi,\rho\rangle)| \leq c_1|\varphi|^{c_1}$. Observe que essa constante é independente da atribuição parcial ρ. Por isso, segue a partir da definição do polinômio $p(.)$ que existem no máximo $p(c_1|\varphi|^{c_1})$ palavras *em* S que são imagem de $h(\langle\varphi,.\rangle)$. Seja N esse número.

Exibiremos um algoritmo polinomial que encontra a menor atribuição que satisfaz a fórmula de entrada $\varphi(x_1,\ldots,x_n)$, caso ela seja satisfatível. Associamos a essa fórmula uma árvore binária de atribuições parciais de altura $n+1$ (no nível inicial temos a palavra vazia ϵ, no próximo nível as palavras 0 e 1, e assim por diante). Vamos explorar essa árvore e encontrar a menor atribuição que satisfaz φ (caso ela exista). No entanto, nosso algoritmo verificará (em tempo polinomial) no máximo N nós em cada nível da árvore. Como esse valor é polinomial no tamanho da entrada, teremos um algoritmo eficiente que poderá ser usado para decidir a linguagem SAT. Desenvolveremos e provaremos a corretude do nosso algoritmo por indução:

> No i-ésimo nível da árvore, temos no máximo N nós (atribuições de tamanho i) a serem analisados. Além do mais, se a fórmula φ for satistatível, algum desses nós pode ser estendido à menor atribuição que satisfaz φ.

No caso $i = 0$, a hipótese de indução é claramente satisfeita, uma vez que se φ é satisfatível, então a palavra vazia pode ser estendida à menor atribuição que satisfaz φ.

Suponha que o algoritmo tenha executado até o i-ésimo nível da árvore e que a hipótese de indução seja verdadeira nesse nível. Seja $k \leq N$ o número de nós restantes ao fim dessa etapa. Inicialmente, geramos todos as $2k$ atribuições parciais a serem verificadas no nível $i+1$. Segue a partir da hipótese de indução e por exaustão que alguma delas pode ser estendida à menor atribuição que satisfaz a fórmula φ, caso φ seja satisfatível. Se $2k \leq N$, então a hipótese de indução continua verdadeira e partimos para o nível seguinte da árvore. Caso contrário, sejam $\rho_1,\ldots,\rho_{2k}$ as atribuições parciais obtidas nessa etapa.

Em primeiro lugar, computamos em tempo polinomial as palavras $w_j = h(\langle\varphi,\rho_j\rangle)$, para $1 \leq j \leq 2k$. Se existirem índices j e j' tais que $w_j = w_{j'}$, então $\langle\varphi,\rho_j\rangle \in$ SAT* se e somente se $\langle\varphi,\rho_{j'}\rangle \in$ SAT* (lembre que h é uma redução entre SAT* e S). Afirmamos que a maior dessas atribuições parciais não pode ser estendida à menor atribuição que satisfaz φ. A fim de obtermos uma contradição, suponha que isso seja possível. Assuma,

sem perda de generalidade, que $\rho_j \preceq \rho_{j'}$. Vamos considerar os dois cenários possíveis. Por um lado, se $\langle \varphi, \rho_{j'} \rangle \in$ SAT*, vimos anteriormente que $\langle \varphi, \rho_j \rangle \in$ SAT*. Como $\rho_j \preceq \rho_{j'}$ e $\langle \varphi, \rho_j \rangle \in$ SAT*, $\rho_{j'}$ não pode ser estendida à menor atribuição que satisfaz φ (note que $\rho_j \neq \rho_{j'}$). Por outro lado, se $\langle \varphi, \rho_{j'} \rangle \notin$ SAT*, então $\rho_{j'}$ não pode ser estendida à nenhuma atribuição que satisfaz φ, e muito menos à menor delas. Por isso, precisamos continuar a execução do algoritmo guardando apenas as atribuições parciais que possuem imagens distintas segundo a função h, dando preferência por aquelas que são menores segundo à ordem lexicográfica.

Seja k' o número de atribuições parciais (com imagens distintas por h) restantes. Se $k' \leq N$, passamos para o próximo nível da árvore. Caso contrário, eliminamos as $k' - N$ primeiras atribuições parciais segundo a ordem lexicográfica e passamos para a próxima etapa. Isso pode ser feito pois, caso alguma das $k' - N$ primeiras atribuições parciais possa ser estendida à uma atribuição que satisfaz φ, teremos mais do que N atribuições parciais restantes formando palavras da forma $\langle \varphi, . \rangle$ pertencentes à SAT*, e portanto mais do que N imagens distintas $h(\langle \varphi, . \rangle)$ em S, o que contradiz a definição de N. Portanto, o $i+1$-ésimo passo é completado e a hipótese de indução continua válida.

Finalmente, utilizamos o algoritmo anterior para encontrar em tempo polinomial um conjunto de no máximo N atribuições de tamanho n para a fórmula de entrada $\varphi(x_1, \ldots, x_n)$. Além disso, por hipótese, alguma dessas atribuições é a menor atribuição possível que satisfaz a fórmula φ, caso φ seja satisfatível. Verificamos o valor de $\varphi(\rho_j)$ para cada atribuição ρ_j e descobrimos se a fórmula de entrada é satisfatível. Isso completa a descrição do algoritmo de tempo polinomial que decide SAT. □

Um importante precursor desse teorema foi um resultado provado por Berman [11].

Corolário 4.47. [Teorema de Berman]. *Suponha que alguma linguagem unária* $L \subseteq \{1\}^\star$ *seja* NP*-completa. Então* P = NP.

Demonstração. Note que toda linguagem unária é esparsa. O resultado segue pelo teorema de Mahaney. □

É possível demonstrar que a dificuldade das linguagens de baixa densidade em NP depende da relação entre certas classes de complexidade mais fortes.

Teorema 4.48. EXP $\neq$ NEXP *se e somente se existe uma linguagem esparsa em* NP $\setminus$ P.

Demonstração. A prova desse resultado pode ser obtida no artigo original de Hartmanis et al. [53]. □

Observe que esse teorema fornece uma prova alternativa do teorema de Ladner através da hipótese mais forte de que EXP $\neq$ NEXP, uma vez que uma linguagem esparsa não pode ser NP-completa pelo teorema de Mahaney.

4.7 Algoritmos Ótimos para NP

Nesta seção vamos provar que todo problema de busca com soluções eficientemente verificáveis admite um algoritmo (assintoticamente) ótimo. Além disso, vamos exibir explicitamente o seu código. Embora seja possível provar a otimalidade do algoritmo, a demonstração apresentada não deixa nenhuma pista sobre sua complexidade.

Seja V um verificador eficiente para uma linguagem $L \in \mathrm{NP}$. Na seção 2.3 definimos o problema de busca relacionado ao verificador V: dada uma palavra de entrada x, encontrar uma palavra w tal que $V(x, w) = 1$, ou indicar que tal w não existe.

Definição 4.49. *Sejam V um verificador eficiente para uma linguagem $L \in \mathrm{NP}$ e $q(x)$ o polinômio que descreve o tamanho dos certificados aceitos por V. Denotamos por $S_V(x) = \{w \in \{0,1\}^{q(|x|)} : V(x, w) = 1\}$ o conjunto de soluções da entrada x em relação ao verificador V.*

Para simplificar a demonstração e o enunciado do resultado, não vamos considerar o modelo computacional de máquinas de Turing nesta seção. Assuma que todos os algoritmos discutidos podem ser implementados em um modelo computacional capaz de simular t passos de sua computação em no máximo kt passos, onde k é uma constante adequada. Essa simplificação não oferece qualquer risco, uma vez que o teorema desta seção não será utilizado no resto do texto. Se A é um algoritmo, então $t_A(x)$ representa o número de passos utilizados por A com entrada x.

Teorema 4.50. [Algoritmos de Busca "Ótimos" para NP] *Sejam V um verificador eficiente para uma linguagem $L \in \mathrm{NP}$, $q(x)$ o polinômio que descreve o tamanho dos certificados aceitos por V e $p(x)$ um polinômio que limita a complexidade de tempo de V. Existe um algoritmo A com as seguintes propriedades:*

(i) A resolve o problema de busca associado ao verificador V;

(ii) Para todo algoritmo A' que resolve o problema de busca associado ao verificador V e para toda entrada $x \in \{0,1\}^\star$ com $S_V(x) \neq \emptyset$, temos $t_A(x) = O(t_{A'}(x) + p(x))$;

(iii) Para entradas $x \in \{0,1\}^\star$ com $S_V(x) = \emptyset$, temos $t_A(x) = O(2^{q(|x|)}p(|x|))$.

Demonstração. Com entrada $x \in \{0,1\}^\star$, o algoritmo A simula em paralelo todos os algoritmos existentes (explicações adiante) e utiliza o algoritmo V para checar a solução fornecida por cada um deles. Se durante algum passo da simulação uma solução é encontrada, A termina imediatamente sua computação e imprime a resposta obtida. Por isso, sempre que A termina sua computação e imprime uma palavra w, temos que $V(x, w) = 1$. Por outro lado, para garantir que A termina sua computação com entradas

que não possuem solução, simulamos também em paralelo um algoritmo B que usa uma busca exaustiva para procurar todos os certificados aceitos pelo verificador V. Podemos escolher o algoritmo B de modo que $t_B(x) = O(2^{q(|x|)}p(|x|))$.

Suponha que os algoritmos estejam ordenadas de acordo com a ordem lexicográfica de suas descrições. A simulação em paralelo realizada por A ocorre por etapas. Na j-ésima etapa de simulação:

(1) A simula os primeiros $2^j - 1$ algoritmos com entrada x;
(2) O algoritmo A_i é simulado por $2^j/(i+1)^2$ passos, onde $1 \leq i \leq 2^j - 1$;
(3) O algoritmo B é simulado por $\sum_{i=1}^{2^j-1} 2^j/(i+1)^2$ passos, ou seja, pelo número total de passos simulados no item anterior.

A variação na taxa de simulação de cada algoritmo será útil para provarmos o item *(ii)* do teorema. A simulação do algoritmo B continua na próxima etapa a partir do último passo realizado na etapa anterior. Por outro lado, para não sobrecarregar o algoritmo A, as simulações do item (2) são perdidas ao fim de cada etapa. Assumimos que os passos utilizados para verificação da solução obtida pelo algoritmo A_i também são contados na simulação realizada em (2).

Devido à simulação paralela realizada em (3), o algoritmo A sempre termina sua computação. Por isso, os itens *(i)* e *(iii)* do teorema são satisfeitos.

Vamos finalmente provar que A satisfaz a segunda propriedade do enunciado. Seja A' um algoritmo que resolve o problema de busca associado ao verificador V. Além disso, suponha que $S_V(x) \neq \emptyset$. Vamos denotar por $t'(x) = t_{A'}(x) + p(x)$ o tempo gasto por A' para obter uma solução para a entrada x mais o tempo utilizado por V com a verificação da solução obtida. Observe que A' é no máximo o $(2^{|A'|+1} - 1)$-ésimo algoritmo segundo a ordenação lexicográfica adotada. Por isso, os $t'(x)$ passos necessários para obter e verificar a solução de A' são realizados na j-ésima etapa de simulação se:

(a) $2^j - 1 \geq 2^{|A'|+1} - 1$, ou seja, $j \geq |A|' + 1$; e
(b) $2^j/(2^{|A'|+1})^2 \geq t'(x)$, ou seja, $j \geq 2(|A'| + 1) + \log(t_{A'}(x) + p(|x|)$.

Portanto, o algoritmo A termina sua computação com a entrada x na k-ésima etapa, onde $k = 2(|A'| + 1) + \log(t_{A'}(x) + p(|x|)$. Por isso, o número total de passos simulados não é superior a:

$$\sum_{j=1}^{k}\sum_{i=1}^{2^j-1} 2\,\frac{2^j}{(i+1)^2} \leq \sum_{j=1}^{k} 2^{j+1}\sum_{i=1}^{2^j-1}\frac{1}{(i+1)^2} \leq \frac{\pi^2}{6}\sum_{j=1}^{k} 2^{j+1} \leq 2^{k+3} = 2^{2|A'|+5}[t_{A'}(x) + p(|x|)].$$

Usamos em uma das desigualdades anteriores o fato de que a série $\sum_{n=1}^{\infty} 1/n^2$ é convergente (Rudin [94]). Em particular, temos $\sum_{n=1}^{\infty} 1/n^2 = \pi^2/6$. Como assumimos que t passos podem ser simulados em tempo kt, obtemos que $t_A(x) = O(t_{A'}(x) + p(|x|))$, como queríamos demonstrar. □

Observe que a constante escondida na notação assintótica depende apenas do algoritmo A'. Além disso, essa constante cresce exponencialmente com a descrição de A'. Infelizmente, isso faz com que o algoritmo obtido seja inviável na prática. No entanto, a noção de otimalidade é pontual, ou seja, é válida para cada palavra de tamanho n que possui solução, e não apenas no pior caso.

Mostraremos a seguir que, se a constante anterior fosse apenas polinomial no tamanho do algoritmo A', então P = NP. Suponha que existisse um polinômio $q(.)$ de forma que $q(|A'|)$ limitasse o valor dessa constante. Primeiro, para toda fórmula proposicional φ satisfatível existe um algoritmo A_φ de tamanho $O(|\langle\varphi\rangle|)$ que, com entrada $\langle\varphi\rangle$, imprime uma atribuição $\vec{y}$ tal que $\varphi(\vec{y}) = 1$ (adicionamos a própria atribuição ao código de A_φ). Além disso, A_φ computa em tempo linear. Portanto, para $L = \text{SAT}$, a complexidade de tempo do algoritmo obtido na demonstração anterior seria $O(q(|A_\varphi|)[t_{A_\varphi}(x) + p(|x|)])$, onde $t_{A_\varphi}(x)$ é $O(x)$ e $|A_\varphi|$ é $O(|x|)$. Portanto, poderíamos encontrar uma atribuição satisfazendo a fórmula de entrada em tempo polinomial ou determinar que tal atribuição não existe após a simulação desse número de passos. Isso provaria que SAT $\in$ P, e portanto teríamos P = NP.

4.8 Referências Adicionais

A demonstração do Teorema de Ladner segue o livro de Arora e Barak [7]. Uma generalização desse resultado, aplicável à outras classes de complexidade, pode ser obtida no artigo de Schöning [97]. Notamos que um teorema similar existe em teoria da computabilidade [44, 82].

O teorema 4.50 foi provado por Levin [74]. Veja o artigo de Hutter [60] para uma extensão desse resultado. Por último, o artigo recente de Trevisan [106] discute o tamanho das constantes envolvidas nessa e em outras demonstrações similares.

A existência de problemas completos para todos os níveis de PH foi provada por Wrathall [113]. Alguns resultados citados neste capítulo e outros desenvolvimentos recentes em complexidade computacional são discutidos no artigo de Cai e Zhu [18].

Diversos temas importantes relacionados com o problema P vs NP não foram discutidos neste capítulo. Em particular, não abordamos o Teorema PCP (Provas Checáveis Probabilisticamente). Intuitivamente, esse resultado estabelece que qualquer prova matemática pode ser reescrita com certa redundância de forma que sua validade possa ser

checada em tempo polinomial e com um alto índice de certeza através da leitura de um número constante de bits escolhidos aleatoriamente. Esse teorema fornece uma nova caracterização para a classe NP. Além disso, ele pode ser usado para provar que, a menos que P = NP, não podemos computar eficientemente soluções aproximadas para diversos problemas computacionais importantes. Para um excelente tratamento desse tópico e referências adicionais, consulte o livro de Arora e Barak [7].

Capítulo 5

Os Limites da Diagonalização

"Oracles are subtle but not malicious."
Scott Aaronson.

Veremos neste capítulo que alguns métodos discutidos nesta dissertação não são capazes de resolver o problema P vs NP. Discutiremos em seguida como essa limitação se relaciona com resultados de independência formal em matemática. Além disso, vamos estudar o comportamento de diversos problemas em aberto da complexidade computacional em universos computacionais alternativos. Finalmente, discutiremos como métodos mais modernos superam a limitação enfrentada por algumas das técnicas estudadas anteriormente.

5.1 Introdução

Vimos no capítulo 3 que o método da diagonalização pode ser usado para separar diversas classes de complexidade interessantes. Em particular, discutimos na seção 3.6 como ele pode ser aplicado, em conjunto com a noção de completude, para provar que certos problemas computacionais naturais não podem ser resolvidos em tempo polinomial. Além disso, mostramos em diversas ocasiões que certas inclusões de classe podem ser provadas através de simulação.

O principal objetivo deste capítulo é demonstrar que essas ideias, emboram sejam muito poderosas, são insuficientes para resolver diversas questões em aberto discutidas nesta dissertação. Em particular, mostraremos que não podemos separar (ou colapsar) classes como P e PSPACE, NP e coNP, P e PH, e P e NP utilizando apenas simulação e diagonalização. A principal desvantagem desses métodos é que eles tratam os algo-

ritmos como se fossem caixas-pretas, não sendo capazes de analisar adequadamente as computações envolvidas.

Informalmente, uma prova utiliza diagonalização quando se baseia na enumeração dos algoritmos através de palavras e na simulação entre algoritmos sem uma perda significativa de tempo ou espaço. Geralmente, um algoritmo A ligado a uma classe de complexidade pode simular todos os algoritmos ligados à outra classe de complexidade mais fraca, negando suas respostas. Por exemplo, poderíamos provar o teorema 3.2 dessa forma, e a linguagem decidida por A seria o complemento da linguagem L definida naquela demonstração. Veremos uma demonstração apresentada dessa forma no fim desta seção.

Considere o seguinte argumento utilizado na tentativa de separar as classes de complexidade P e PSPACE.

Exemplo 5.1. [Argumento diagonal envolvendo P e PSPACE]. *Vamos tentar descrever uma* MT D *que simula todas as máquinas de Turing determinísticas de tempo polinomial, negando suas respostas. Com entrada* $\langle M_i \rangle$, D *computa da seguinte maneira:*

1) *Simula a máquina* M_i *com entrada* $\langle M_i \rangle$ *por* $\varphi(n)$ *passos, onde* $n = |\langle M_i \rangle|$ *e* $\varphi(.)$ *é uma função a ser definida posteriormente.*
2) *Se* M_i *aceita* $\langle M_i \rangle$ *durante a simulação, então* D *rejeita a entrada* $\langle M_i \rangle$.
3) *Caso* M_i *rejeite* $\langle M_i \rangle$ *ou não termine sua computação em* $\varphi(n)$ *passos,* D *aceita a entrada* $\langle M_i \rangle$.

Seja L_D *a linguagem decidida por* D. *Queremos provar que* $L_D \in$ PSPACE $\setminus$ P. *Para isso, precisamos mostrar que, para toda* MT *determinística que computa em tempo polinomial, temos* $L(M_i) \neq L_D$. *Se* M_i *computa em tempo* $c_i n^{c_i}$, *isso pode ser obtido se tomarmos* $\varphi(n) \geq c_i n^{c_i}$. *No entanto, uma* MT *com essa complexidade de tempo pode computar em espaço polinomial* $\Theta(n^{c_i})$. *Além disso, o teorema de hierarquia de tempo estabelece que a constante* c_i *pode ser arbitrariamente grande para linguagens decidíveis em tempo polinomial. Por isso, ao garantir que* L_D *difere de toda linguagem em* P, *não é possível provar que* D *computa em espaço polinomial, i.e., que* $L_D \in$ PSPACE. *Para conseguir isso, é preciso que a simulação realizada na etapa* 1 *explore alguma propriedade das computações que utilizam espaço polinomial. Porém, é exatamente isso o que queremos provar, ou seja, que essas computações são mais poderosas do que computações de tempo polinomial.*

Observe que o método da diagonalização atrasada (introduzido na prova do teorema 3.34) encontra o mesmo obstáculo, uma vez que precisamos simular entradas de tamanho $n + 1$ *quando recebemos entradas de tamanho* n.

O exemplo anterior mostra que é difícil separar através de diagonalização classes de complexidade que utilizam recursos diferentes na mesma proporção. Para conseguir essa

separação através de diagonalização é preciso explorar de modo inteligente alguma propriedade do recurso mais poderoso. A mesma dificuldade aparece quando o argumento anterior é adaptado para as classes P e NP, por exemplo.

Discutimos na seção 4.4 o conceito de máquinas de Turing com acesso a um oráculo. Vamos a seguir generalizar a definição 4.29. Se $\mathcal{R}$ é uma classe de complexidade e $L \subseteq \{0,1\}^\star$ é uma linguagem, denotamos por $\mathcal{R}^L$ o conjunto de linguagens decidíveis por máquinas de Turing que utilizam uma quantidade de recursos compatível com a definição da classe $\mathcal{R}$ e possuem acesso à linguagem L através de um oráculo. Por exemplo, $\mathrm{NP}^{\mathrm{SAT}}$ denota o conjunto de linguagens decidíveis em tempo polinomial não-determinístico por máquinas de Turing com acesso a um oráculo para a linguagem SAT.

A adição de um oráculo à definição de uma classe de complexidade cria um novo universo computacional. Enquanto vivemos em um universo onde nenhuma computação ocorre de graça, podemos imaginar mundos em que essa regra é violada. Por exemplo, $\mathrm{NP}^{\mathrm{SAT}}$ é uma classe aparentemente mais poderosa do que NP, uma vez que cada caminho não-determinístico pode resolver instantaneamente uma instância de SAT. Utilizando a completude de SAT e negando a resposta do oráculo, podemos obter a resposta de uma computação não-determinística baseada em quantificadores universais. Formalmente, é possível demonstrar que $\mathrm{NP}^{\mathrm{SAT}} = \Sigma_2^p$ (veja o livro de Arora e Barak [7], por exemplo).

Observamos que muitos textos definem classes de complexidade do tipo $\mathcal{R}_1^{\mathcal{R}_2}$. Nesse caso, as máquinas envolvidas podem ter acesso à qualquer linguagem da classe $\mathcal{R}_2$ (a linguagem é fixa, porém arbitrária). Na maioria dos casos, isso equivale a dizer que $\mathcal{R}_1^{\mathcal{R}_2} = \mathcal{R}_1^L$, onde L é uma linguagem completa para a classe $\mathcal{R}_2$.

Durante uma simulação entre máquinas de Turing, todas as chamadas a um oráculo feitas pela máquina simulada podem ser respondidas pela máquina simuladora, desde que ela possua acesso ao mesmo oráculo. Veremos a seguir um resultado que utiliza essa observação para estender um teorema demonstrado na seção 3.3.

Teorema 5.2. [Diagonalização e Oráculos]. *Para toda linguagem $O \subseteq \{0,1\}^\star$, temos que $\mathrm{P}^O \neq \mathrm{EXP}^O$.*

Demonstração. Claramente, temos que $\mathrm{P}^O \subseteq \mathrm{EXP}^O$. Vamos exibir uma máquina de Turing D com acesso ao oráculo O que decide uma linguagem L_D tal que $L_D \in \mathrm{EXP}^O \setminus \mathrm{P}^O$. Com entrada $\langle M_i \rangle$, D computa da seguinte forma:

1) Simula 2^n passos da computação da máquina M_i com entrada $\langle M_i \rangle$, onde $n = |\langle M_i \rangle|$.
2) Se M_i invoca o oráculo O com uma palavra z na sua fita de oráculo, D obtém a resposta dessa chamada através do seu próprio oráculo para a linguagem O.
3) A máquina D rejeita a entrada $\langle M_i \rangle$ se e somente se M_i aceita $\langle M_i \rangle$ durante a simulação anterior.

Por construção, para toda linguagem $L \in \mathrm{P}^O$, existe uma máquina de Turing M_i de tempo polinomial que decide L tal que $\langle M_i \rangle \in L$ se e somente se $\langle M_i \rangle \notin L_D$, i.e., $L \neq L_D$. Isso prova que $L_D \notin \mathrm{P}^O$. Além disso, a máquina determinística D computa em tempo exponencial. Portanto, $L_D \in \mathrm{EXP}^O \setminus \mathrm{P}^O$. □

Essencialmente, o oráculo O não possui nenhuma influência na prova do resultado anterior. Por isso, dizemos que a demonstração é insensível à presença de um oráculo. Em outras palavras, demonstrações que envolvem apenas simulação entre algoritmos e diagonalização provam resultados que são válidos em todos os universos computacionais possíveis. Em complexidade computacional, dizemos que provas e métodos com essa propriedade relativizam, i.e., demonstram resultados verdadeiros em relação à qualquer oráculo.

O leitor pode verificar que todas as demonstrações apresentadas no capítulo 3 são indiferentes à presença de oráculos. Portanto, em qualquer universo computacional, esses resultados são verdadeiros. Veremos na seção 5.2 que é exatamente por isso que esses métodos são incapazes (sem nenhum argumento adicional que não relativize) de resolver o problema P vs NP. Vamos exibir dois mundos computacionais distintos onde $\mathrm{P} = \mathrm{NP}$ e $\mathrm{P} \neq \mathrm{NP}$, respectivamente.

5.2 Uma Barreira Fundamental

Nesta seção vamos provar que existem linguagens $A, B \subseteq \{0,1\}^\star$ tais que $\mathrm{P}^A = \mathrm{NP}^A$ e $\mathrm{P}^B \neq \mathrm{NP}^B$. Esse resultado foi provado por Baker, Gill e Solovay [8]. Como provas por simulação e diagonalização demonstram resultados válidos em qualquer universo computacional, isso mostra que esses métodos não são suficientes para resolver o problema P vs NP.

Teorema 5.3. [Teorema de Relativização BGS]. *Existem linguagens $A, B \subseteq \{0,1\}^\star$ tais que $\mathrm{P}^A = \mathrm{NP}^A$ e $\mathrm{P}^B \neq \mathrm{NP}^B$. Além disso, temos que A,$B \in \mathrm{EXP}$.*

Demonstração. Seja A uma linguagem PSPACE-completa. Claramente, usando simulação temos que $\mathrm{P}^A = \mathrm{PSPACE} = \mathrm{NP}^A$. Pelo lema 3.9, obtemos que $A \in \mathrm{EXP}$.

Precisamos agora demonstrar a existência de linguagens B e L_B tais que $L_B \in \mathrm{NP}^B \setminus \mathrm{P}^B$. Para toda linguagem $B \subseteq \{0,1\}^\star$, seja L_B a seguinte linguagem unária:

$$L_B = \{1^n : \ \exists w \in \{0,1\}^n \text{ tal que } w \in B\}.$$

Observe que, para toda linguagem B, temos que $L_B \in \mathrm{NP}^B$ (um verificador eficiente com acesso à B pode usar w como certificado de que $1^n \in L_B$). Resta demonstrar que $L_B \notin \mathrm{P}^B$. Para obter isso, vamos definir B em etapas, de modo que essa linguagem

seja uma adversária para qualquer máquina de Turing de tempo polinomial com acesso ao oráculo B que tenta decidir L_B. Considere que $M_1, M_2, \ldots$ é uma lista de todas as máquinas de Turing, de forma que cada máquina de Turing apareça infinitas vezes nessa enumeração.

Até o início da etapa i, a pertinência em B de apenas uma quantidade finita de palavras foi determinada. Note que isso vale no início da etapa 1, quando B é totalmente indeterminada. Seja $n \geq i$ o menor número inteiro de forma que a presença em B de nenhuma palavra de tamanho maior ou igual a n tenha sido decidida. Considere a computação de até 2^{n-1} passos da máquina de Turing M_i com entrada 1^n (M_i pode terminar sua computação antes de completar 2^{n-1} passos). Se a máquina M_i indaga sobre a presença em B de uma palavra w já determinada, respondemos consistentemente. Se a presença de w ainda não foi decidida, respondemos negativamente e determinamos que $w \notin B$. Por último, se em 2^{n-1} passos a máquina M_i aceita a entrada 1^n, determinamos que todas as palavras restantes de tamanho menor ou igual a n ainda não consultadas não pertencem à linguagem B. Caso contrário, decidimos que essas palavras estão em B. Isso completa a descrição da linguagem B.

Note que se M_i aceita 1^n em até 2^{n-1} passos, então não existe palavra de tamanho n que pertence à B, ou seja, $1^n \notin L_B$. Por outro lado, se a palavra 1^n não é aceita por M_i em até 2^{n-1} passos, pelo menos metade das palavras de tamanho n estão em B, uma vez que em 2^{n-1} passos B só pode ter usado o oráculo para verificar a pertinência de até 2^{n-1} palavras. Por isso, neste caso temos que $1^n \in L_B$. Isso mostra que se M_i termina sua computação com a entrada 1^n em até 2^{n-1} passos, então M_i não decide L_B.

Seja M uma máquina de Turing que computa em tempo polinomial. Então existe um inteiro positivo n_0 tal que, para toda palavra w de tamanho $n' \geq n_0$, M aceita ou rejeita w em no máximo $2^{n'-1}$ passos. Lembre que cada máquina de Turing aparece infinitas vezes na enumeração adotada. Portanto, existe um índice $i \geq n_0$ tal que M é equivalente a M_i. Além disso, na i-ésima etapa de construção da linguagem B, tomamos $n \geq i \geq n_0$ e garantimos que se M_i computa em tempo 2^{n-1} com a entrada 1^n, então $L(M_i) \neq L_B$. Portanto, M não decide a linguagem L_B. Isso prova que não existe máquina de Turing de tempo polinomial com acesso ao oráculo B que decide L_B, ou seja, $L_B \notin \mathrm{P}^B$.

Observe que, como tomamos na i-ésima etapa $n \geq i$, a presença em B de palavras de tamanho menor ou igual a n é determinada pelo comportamento das primeiras i máquinas de Turing. Portanto, para decidir se uma palavra w de tamanho n pertence à B, basta simularmos a computação das primeiras i máquinas de Turing, onde $i \leq n$ e a i-ésima máquina nunca precisa ser simulada por mais do que 2^{n-1} passos. Como isso pode ser feito em tempo exponencial, temos que $B \in \mathrm{EXP}$, como queríamos demonstrar. □

O teorema anterior nos dá forte evidência de que, para resolver o problema P vs NP, precisamos investigar profundamente as computações envolvidas, e não apenas simulá-

las. É interessante notar que provamos que a diagonalização é uma técnica fraca contra o problema P vs NP utilizando basicamente um argumento por diagonalização.

Podemos estudar os oráculos em situações muito mais complexas. Por exemplo, existe algum universo computacional onde NP = coNP, mas P $\neq$ NP? Vamos enunciar diversos resultados desse tipo na seção 5.4.

5.3 P vs NP e Resultados de Independência

Quais as implicações do Teorema BGS para o problema P vs NP? Nesta seção vamos discutir algumas questões envolvendo oráculos e resultados de independência em matemática.

Dizemos que uma proposição matemática φ é independente de um conjunto de axiomas $\mathcal{A}$ se tanto φ quanto $\neg\varphi$ não podem ser provadas a partir de $\mathcal{A}$. Em primeiro lugar, como isso pode ser possível? Inicialmente, vamos discutir a diferença entre verdade e demonstrabilidade.

A noção de verdade nunca está relacionada com um conjunto de axiomas, mas sim com uma determinada estrutura que satisfaz os axiomas adotados. Por exemplo, dados os axiomas da teoria de grupos, um grupo arbitrário G_1 é uma estrutura que satisfaz tais axiomas. Seja φ uma fórmula que expressa que um grupo é comutativo, ou seja, $\forall x \forall y (x * y = y * x)$. A fórmula φ pode ser verdadeira para alguns grupos e falsa em relação a outros grupos. Portanto, a noção de verdade depende da estrutura em consideração.

Por outro lado, o conceito de demonstrabilidade depende apenas dos axiomas e das regras de derivação de um sistema formal. Uma fórmula φ pode ser provável a partir de um conjunto de axiomas $\mathcal{A}$ (e de certas regras de derivação) ou não. Um fato muito importante dos sistemas formais utilizados em matemática é que eles são *corretos*, ou seja, sempre que uma fórmula φ é demonstrável a partir de um conjunto de axiomas, ela é verdadeira em qualquer estrutura que satisfaça os axiomas. Por exemplo, quando provamos um teorema ψ a partir dos axiomas da teoria de grupos, sabemos que todo grupo (i.e., estrutura compatível com esses axiomas) satisfaz a proposição ψ. Devido à corretude, se existem duas estruturas diferentes que satisfazem certos axiomas, mas que não concordam em relação a uma certa propriedade expressa por uma fórmula ψ (em uma estrutura a propriedade é verdadeira e em outra não), então temos que tanto ψ quanto $\neg\psi$ são fórmulas que não podem ser provadas a partir de tais axiomas. Portanto, dados certos axiomas, é possível que uma proposição seja verdadeira (em relação a uma estrutura canônica que associamos ao sistema formal[1]), porém não seja demonstrável.

[1]Por exemplo, associamos às diversas regras da aritmética a estrutura canônica formada pelo conjunto de números inteiros. No entanto, existem diversas estruturas bastante distintas que satisfazem os mesmos axiomas.

Embora tenhamos discutido apenas informalmente esses conceitos, a distinção precisa entre verdade e demonstrabilidade provocou uma revolução na matemática durante o século passado. Notamos que a relação entre esses dois conceitos depende também do poder expressivo das fórmulas utilizadas para expressar as proposições matemáticas.

Lembre que a teoria de conjuntos ZFC foi discutida na seção 2.6, e que com esse sistema formal podemos provar todos os teoremas usuais demonstrados em matemática. No entanto, é possível demonstrar que certas questões bastante profundas são independentes desses axiomas. Por exemplo, embora a questão P vs NP seja verdadeira ou falsa (as duas classes coincidem ou não no modelo padrão de ZFC), pode ser que a relação entre as duas classes não possa der decidida (através de uma demonstração) dentro do sistema formal ZFC.

Intuitivamente, o teorema BGS pode ser visto como um mini-resultado de independência. Essencialmente, ele prova que o problema P vs NP é independente de qualquer teoria matemática que prove apenas resultados que relativizam, ou seja, que são válidos na presença de qualquer oráculo. Embora muitos métodos discutidos nesta dissertação não sejam capazes de ultrapassar a barreira imposta pelo teorema BGS, existem técnicas mais modernas que não possuem essa limitação. Apresentaremos na seção 5.8 uma discussão adicional sobre isso. Assim, tendo em vista a existência de métodos capazes de superar essa dificuldade, não podemos tirar ainda nenhuma conclusão à respeito da independência do problema P vs NP original em relação à sistemas fortemente expressivos como a teoria de conjuntos ZFC.

No entanto, veremos nesta seção que, de fato, existem certos resultados em complexidade computacional que são independentes dos axiomas usuais da matemática. Vamos demonstrar que existe uma máquina de Turing M tal que o problema $\mathrm{P}^{L(M)}$ vs $\mathrm{NP}^{L(M)}$ é independente de ZFC. Isso ilustra as sutilezas envolvidas quando abandonamos o mundo "real" e estudamos universos relativizados.

Teorema 5.4. *Existe uma máquina de Turing determinística M que sempre termina sua computação tal que o problema $\mathrm{P}^{L(M)}$ vs $\mathrm{NP}^{L(M)}$ é independente de* ZFC. *Além disso, temos que $L(M) = \emptyset$.*

Demonstração. Vamos utilizar na demonstração o teorema da recursão, um resultado que garante que podemos alterar toda máquina de Turing de modo que seu próprio código esteja disponível durante a computação. Veja o livro de Sipser [103] para mais detalhes.

De acordo com o teorema 5.3, existem linguagens *decidíveis* A e B tais que $\mathrm{P}^A = \mathrm{NP}^A$ e $\mathrm{P}^B \neq \mathrm{NP}^B$. Além disso, é possível mostrar que esse teorema pode ser provado dentro do sistema formal ZFC. Seja M uma máquina de Turing que aceita uma entrada x se e somente se:

1) $x \in A$ *e* existe uma prova de tamanho no máximo $|x|$ em ZFC de que $\mathrm{P}^{L(M)} \neq \mathrm{NP}^{L(M)}$;
ou
2) $x \in B$ *e* existe uma prova de tamanho no máximo $|x|$ em ZFC de que $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$.

Na descrição anterior, a máquina de Turing M tem disponível seu próprio código para construir uma fórmula φ que expressa $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$.

Suponha que $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$ seja provável em ZFC. Por construção, a menos de um número finito de palavras, temos que $L(M) = B$. Por isso, ZFC também demonstra que $\mathrm{P}^B = \mathrm{NP}^B$. No entanto, isso viola a consistência dessa teoria, uma vez que ZFC prova que $\mathrm{P}^B \neq \mathrm{NP}^B$. Portanto, não existe demonstração em ZFC de que $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$. Similarmente, a menos que ZFC seja inconsistente, não existe prova de que $\mathrm{P}^{L(M)} \neq \mathrm{NP}^{L(M)}$ dentro desse sistema formal. Isso mostra que a relação entre as classes $\mathrm{P}^{L(M)}$ e $\mathrm{NP}^{L(M)}$ não pode ser determinada em ZFC.

Por último, observe que como $\mathrm{P}^{L(M)} = \mathrm{NP}^{L(M)}$ é independente de ZFC, segue a partir da definição de M que $L(M) = \emptyset$. □

O teorema 5.4 foi provado por Hartmanis e Hopcroft [52]. No mesmo artigo são exibidos alguns algoritmos cuja complexidade de tempo é independente de ZFC. Notamos que $L(M) = \emptyset$ não implica que o problema P vs NP original seja independente de ZFC. Pode não ser possível provar dentro desse sistema que as classes P e $\mathrm{P}^{L(M)}$ são equivalentes, por exemplo. Em particular, se ZFC fosse capaz de provar que $L(M) = \emptyset$, então ZFC provaria sua própria consistência (veja a definição da máquina de Turing M), o que violaria um resultado importante da lógica matemática.

O resultado de independência anterior é baseado na máquina de Turing M. Existem máquinas de Turing muito mais simples que decidem a linguagem vazia, e não está claro se o mesmo resultado de independência pode ser provado quando substituimos M por tais máquinas. Afinal de contas, não sabemos realmente se o problema P vs NP original é independente de ZFC. No entanto, existe um oráculo decidível O tal que o problema P^O vs NP^O é independente de ZFC em relação a qualquer máquina de Turing utilizada para representar O. Veja o artigo de Kurtz et al. [72] para mais detalhes.

Finalmente, notamos que a maioria dos pesquisadores acredita que questões combinatoriais como o problema P vs NP não são independentes dos axiomas usuais da matemática. Veja diversos pontos de vista interessantes sobre essa questão no artigo de Gasarch [46].

5.4 Resultados Adicionais

Nesta seção veremos que existem universos computacionais onde as mais variadas situações envolvendo classes de complexidade podem acontecer. Embora tais resultados indiquem que não devemos esperar por soluções simples para muitas questões em complexidade computacional, eles podem nos fornecer algumas informações interessantes. Por exemplo, considere o seguinte problema em aberto.

Questão em Aberto 5.5. *Se* $\mathrm{P} = \mathrm{NP}$*, então* $\mathrm{P} = \mathrm{PSPACE}$ *?*

Considere a seguinte abordagem para provar esse resultado. Vimos na seção 4.2 que a classe PSPACE contém as linguagens decidíveis por máquinas de Turing de tempo polinomial com um número arbitrário de alternâncias (veja o corolário 4.19). Poderíamos utilizar a hipótese de que $\mathrm{P} = \mathrm{NP}$ e simulação para eliminar cada alternância, uma de cada vez, sem aumentar muito a complexidade da máquina de Turing obtida. Isso provaria que $\mathrm{P} = \mathrm{PSPACE}$.

Embora seja bastante intuitiva, essa abordagem não é viável. Isso se deve ao seguinte teorema.

Teorema 5.6. *Existe um oráculo* A *tal que* $\mathrm{P}^A = \mathrm{NP}^A$ *e* $\mathrm{NP}^A \neq \mathrm{PSPACE}^A$*.*

Demonstração. Veja o artigo de Ko [68]. □

Portanto, qualquer prova da questão em aberto 5.5 envolvendo apenas simulação poderia ser usada para demonstrar também que se $\mathrm{P}^A = \mathrm{NP}^A$, então $\mathrm{P}^A = \mathrm{PSPACE}^A$, contradizendo o teorema 5.6. Na verdade, um resultado mais geral foi provado por Ko [68].

Teorema 5.7. [Oráculos e a Hierarquia Polinomial]. *Para todo inteiro* $k \geq 0$*, existem oráculos* A *e* B *tais que:*

(i) $\mathrm{PH}^A = \Sigma_k^{pA}$ *e* $\mathrm{PH}^A \neq \mathrm{PSPACE}^A$*.*

(ii) $\mathrm{PH}^B = \Sigma_k^{pB}$ *e* $\mathrm{PH}^B = \mathrm{PSPACE}^B$*.*

O teorema 5.6 fica demonstrado quando tomamos $k = 0$. Em particular, o teorema anterior mostra que não podemos separar ou colapsar os níveis da hierarquia polinomial através de uma demonstração que envolva apenas simulação e diagonalização.

Vamos enunciar a seguir diversos resultados envolvendo oráculos. A maioria dos oráculos são construídos por etapas usando diagonalização, como na prova do teorema 5.3. Alguns deles precisam satisfazer diversos requisitos simultaneamente, o que torna as demonstrações um pouco mais complicadas. Por simplicidade, as provas foram omitidas.

O próximo resultado mostra que, em universos relativizados, as mais diversas situações podem ocorrer envolvendo as classes P, NP e coNP.

Teorema 5.8. [P, NP, coNP e Relativização]. *Existem oráculos A,B e C tais que:*

(i) $\mathrm{P}^A \neq \mathrm{NP}^A \cap \mathrm{coNP}^A$ *e* $\mathrm{NP}^A \cap \mathrm{coNP}^A \neq \mathrm{NP}^A$.

(ii) $\mathrm{P}^B \neq \mathrm{NP}^B$ *e* $\mathrm{NP}^B = \mathrm{coNP}^B$.

(iii) $\mathrm{P}^C = \mathrm{NP}^C \cap \mathrm{coNP}^C$ *e* $\mathrm{NP}^C \cap \mathrm{coNP}^C \neq \mathrm{NP}^C$.

Demonstração. Veja o artigo original sobre relativização de Baker, Gill e Solovay [8]. □

Vimos na seção 2.8 que se P = NP, então EXP = NEXP (teorema 2.30). É possível provar a equivalência entre essas duas afirmações? Utilizando apenas simulação e diagonalização, a resposta é negativa.

Teorema 5.9. *Existe um oráculo A tal que* $\mathrm{P}^A \neq \mathrm{NP}^A$ *e* $\mathrm{EXP}^A = \mathrm{NEXP}^A$.

Demonstração. Veja o artigo de Dekhtyar [32]. □

Além disso, nesse mesmo artigo é demonstrado que existem oráculos que separam as classes PSPACE e EXP.

Discutimos na seção 4.6 a conjectura do isomorfismo. Notamos que é possível construir um oráculo que dá origem a um universo computacional onde essa conjectura é verdadeira. Veja o artigo de Fenner et al. [35] para mais detalhes. Entretanto, veremos na próxima seção que a conjectura do isomorfismo é falsa na maioria dos universos computacionais relativizados.

Em suma, resultados envolvendo oráculos servem como um guia para as pesquisas atuais. Precisamos descobrir métodos que não relativizam se quisermos resolver os problemas em aberto mais importantes da teoria de complexidade computacional.

5.5 Oráculos Aleatórios

Considere a seguinte partição da classe de linguagens: $\mathcal{A} = \{L \subseteq \{0,1\}^\star : \mathrm{P}^L = \mathrm{NP}^L\}$ e $\mathcal{B} = \{L \subseteq \{0,1\}^\star : \mathrm{P}^L \neq \mathrm{NP}^L\}$. Qual desses conjuntos é maior? Existe alguma relação entre essa questão e o problema P vs NP original?

Para formalizar a primeira questão, definimos um espaço de probabilidade no conjunto de oráculos $\mathcal{C} = \{L : L \subseteq \{0,1\}^\star\}$. Para isso, escolhemos os oráculos de modo aleatório[2], de forma que cada palavra x pertença ou não ao oráculo de modo independente e com probabilidade 1/2. Portanto, a primeira questão discutida no parágrafo anterior pode ser enunciada da seguinte maneira: qual a probabilidade (medida) do conjunto $\mathcal{A}$?

[2]Formalmente, para estudar as probabilidades no conjunto infinito $\mathcal{C}$ precisamos de diversas definições de uma disciplina matemática conhecida como teoria da medida . Por simplicidade, vamos discutir todos os resultados desta seção de modo informal. O leitor interessado pode consultar o livro de Du e Ko [33].

Existe um resultado em teoria da medida conhecido como lei zero-um que garante que, para praticamente qualquer propriedade S envolvendo oráculos, S ocorre com probabilidade um ou com probabilidade zero. Em particular, sabemos que $\text{Prob}(\mathcal{A}) = 1$ ou $\text{Prob}(\mathcal{A}) = 0$. No entanto, qual dos dois casos é verdadeiro[3]? Bennett e Gill [10] demonstraram que, em praticamente todos os universos computacionais possíveis, as classes P e NP são distintas.

Teorema 5.10. [P vs NP e Oráculos Aleatórios].
$\text{Prob}_{\mathcal{C}}[\text{P}^L \neq \text{NP}^L] = 1$.

Demonstração. A prova desse resultado considera a linguagem $L' = \{0^n : \exists w \in \{0,1\}^n$ tal que $w1, w11, \ldots, w1^n \in L\}$. Em primeiro lugar, w serve como certificado de que $0^n \in L'$ (o verificador tem acesso ao oráculo L). Portanto, para toda linguagem L, temos que $L' \in \text{NP}^L$. Por outro lado, é possível demonstrar que a probabilidade de L' ser decidida por uma máquina de Turing determinística de tempo polinomial com acesso à L é zero. Uma prova completa desse resultado pode ser obtida no livro de Du e Ko [33]. □

Corolário 5.11. *Existe um oráculo B tal que* $\text{P}^B \neq \text{NP}^B$.

É possível demonstrar que no espaço dos oráculos $\mathcal{C}$ a conjectura do isomorfismo é falsa com probabilidade um. Em outras palavras, o conjunto $\mathcal{I} = \{L \in \mathcal{C} :$ A conjectura do isomorfismo é verdadeira em $\text{NP}^L\}$ possui medida zero. Esse resultado foi provado por Kurtz et al. [71].

Finalmente, o que podemos deduzir sobre os problemas originais a partir desses resultados envolvendo oráculos aleatórios? Essa questão levou Bennet e Gill [10] a formularem a Hipótese do Oráculo Aleatório:

"Se um resultado em complexidade computacional é verdadeiro com probabilidade um para oráculos aleatórios, então ele é verdadeiro no universo não-relativizado."

Se essa hipótese é verdadeira, então o teorema 5.10 prova que $\text{P} \neq \text{NP}$. Além disso, a conjectura do isomorfismo seria falsa. No entanto, Kurtz [70] demonstrou que a Hipótese do Oráculo Aleatório é falsa. Diversos contra-exemplos adicionais foram obtidos posteriormente [21, 51]. Isso significa que não é possível extrair nenhuma relação entre o teorema 5.10 e a questão P vs NP original.

Finalmente, notamos que os resultados envolvendo oráculos aleatórios são úteis na criação de universos computacionais onde diversas situações ocorrem simultaneamente. Por exemplo, no espaço dos oráculos $\mathcal{C}$ temos com probabilidade um que $\text{NP} \neq \text{PSPACE}$

[3] Esse fato não contradiz o teorema BGS, uma vez que em espaços infinitos eventos com probabilidade zero podem ocorrer.

e PSPACE $\subsetneq$ EXP (veja o livro de Du e Ko [33]). Portanto, existe uma linguagem L tal que $\mathrm{NP}^L \neq \mathrm{PSPACE}^L$ e $\mathrm{PSPACE}^L \subsetneq \mathrm{EXP}^L$.

5.6 Relativização Positiva

Nesta seção vamos discutir uma modificação na definição da classe de complexidade NP relativizada. Veremos que o teorema BGS não pode ser provado com a alteração introduzida. O tipo de relativização resultante é conhecida como relativização positiva.

Na demonstração do teorema BGS, podemos converter o verificador eficiente apresentado para a linguagem L_B em uma máquina de Turing não-determinística M que computa em tempo polinomial (veja a prova do teorema 2.17). Além disso, note que a máquina M precisa realizar apenas uma chamada ao oráculo B em cada um dos 2^n caminhos de sua árvore de computação.

Observe que enquanto os algoritmos determinísticos de tempo polinomial que tentam decidir L_B podem fazer apenas uma quantidade polinomial de perguntas ao oráculo B, os algoritmos não-determinísticos de tempo polinomial podem realizar um número exponencial de chamadas ao mesmo oráculo. Por isso, a definição do mecanismo de acesso aos oráculos permite que MTNDs obtenham uma quantidade exponencial de informações sobre a linguagem B, o que naturalmente as torna mais poderosas do que as máquinas determinísticas. Para comparar de forma mais justa a relação entre computação determinística e não-determinística em um universo relativizado, devemos garantir que os dois modelos tenham acesso à mesma quantidade de informações.

Definição 5.12. [Relativização Positiva]. *Para todo oráculo $O \subseteq \{0,1\}^\star$, denotamos por NP_+^O o conjunto de linguagens decidíveis por* MTNDs *de tempo polinomial que realizam uma quantidade total de chamadas ao oráculo O limitada por um polinômio no tamanho da entrada.*

Interessantemente, quando adicionamos essa restrição à classe NP relativizada, podemos provar que o problema P vs NP original é independente do universo computacional em consideração.

Teorema 5.13. [P vs NP e Relativização Positiva].
$\mathrm{P} = \mathrm{NP}$ *se e somente se, para todo oráculo $O \subseteq \{0,1\}^\star$, temos que* $\mathrm{P}^O = \mathrm{NP}_+^O$.

Demonstração. É possível demonstrar a parte não-trivial desse teorema através de uma ideia similar àquela utilizada na prova do teorema de Mahaney (seção 4.46). A demonstração original pode ser obtida no artigo de Brook et al. [13]. □

Portanto, as restrições de acesso ao oráculo tem um papel central na definição de classes de complexidade relativizadas. Levando em conta o teorema anterior, podemos concluir

que o teorema BGS e os outros resultados envolvendo relativização não são interessantes? Como vimos nas seções anteriores, esses teoremas são essenciais para demonstrarmos que através de simulação e diagonalização não podemos resolver diversos problemas importantes em complexidade computacional.

5.7 O Argumento Contrário de Kozen

Veremos agora um resultado que parece contradizer o argumento apresentado neste capítulo de que uma prova envolvendo apenas diagonalização não pode separar as classes P e NP.

Um pouco após a demonstração do teorema BGS, Dexter Kozen [69] publicou um artigo bastante controverso em que ele argumenta que se existe uma demonstração de que $\mathrm{P} \neq \mathrm{NP}$, então existe uma demonstração por diagonalização[4].

Definição 5.14. [Linguagem Diagonal de uma Função]. *Seja $M_1, M_2, \ldots$ uma enumeração das máquinas de Turing determinísticas de tempo polinomial. Seja $h : \{0,1\}^\star \to \mathbb{N}$ uma função arbitrária. Dizemos que* diag_h *é a linguagem diagonal da função h se* $\mathrm{diag}_h = \{x \in \{0,1\}^\star : M_{h(x)}$ *rejeita a entrada x*$\}$.

Em seu artigo, Kozen mostrou que as linguagens utilizadas na demonstração dos teoremas de hierarquia originais são linguagens diagonais de determinadas funções computáveis. No entanto, os argumentos de Kozen são mais gerais e podem ser usados para provar certas consequências intrigantes.

Lema 5.15. *Seja $h : \{0,1\}^\star \to \mathbb{N}$ uma função arbitrária. A linguagem* $\mathrm{diag}_h \notin \mathrm{P}$ *se:*

(i) Para toda linguagem $L \in \mathrm{P}$, existe uma palavra x tal que $L = L(M_{h(x)})$; ou

(ii) Para toda linguagem $L \in \mathrm{P}$, existe uma linguagem L' que difere de L em apenas um número finito de palavras tal que, para infinitas palavras x, temos que $L' = L(M_{h(x)})$.

Demonstração. Vamos provar que se qualquer uma das condições for satisfeita, então $\mathrm{diag}_h \notin \mathrm{P}$. Suponha que $\mathrm{diag}_h \in \mathrm{P}$, ou seja, existe $L \in \mathrm{P}$ tal que $\mathrm{diag}_h = L$.

Assuma que a primeira condição do lema seja válida. Então existe $w \in \{0,1\}^\star$ tal que $\mathrm{diag}_h = L(M_{h(w)})$. Logo, $w \in \mathrm{diag}_h$ se e somente se $w \in L(M_{h(w)})$. Por sua vez, $w \in L(M_{h(w)})$ se e somente se $M_{h(w)}$ aceita w. Pela definição de diag_h, essa última condição ocorre se e somente se $w \notin \mathrm{diag}_h$. Essa contradição mostra que $\mathrm{diag}_h \notin \mathrm{P}$.

Por último, assuma que a segunda condição do lema seja verdadeira. Seja L' uma linguagem satisfazendo as hipóteses do lema. Como diag_h e L' diferem apenas em uma

[4]Em seu artigo original, uma teoria mais geral é desenvolvida. Nesta seção vamos discutir apenas os aspectos relacionados com o problema P vs NP.

quantidade finita de palavras, existe uma palavra w tal que $w \in \text{diag}_h$ se e somente se $w \in L(M_{h(w)})$. Como no caso anterior, obtemos uma contradição. Isso completa a prova de que $\text{diag}_h \notin \text{P}$. □

Teorema 5.16. [Classe P e Linguagens Diagonais]. *Para toda linguagem $L \notin \text{P}$, existe uma função computável h tal que $L = \text{diag}_h$. Além disso, h pode ser escolhida de forma que as duas condições do lema 5.15 sejam satisfeitas.*

Demonstração. Consulte o artigo original de Kozen [69]. □

Corolário 5.17. $\text{P} \neq \text{NP}$ *se e somente se* $\text{SAT} = \text{diag}_h$*, onde h é uma função computável que satisfaz as duas condições do lema 5.15.*

Devido ao corolário anterior, Kozen conclui que se $\text{P} \neq \text{NP}$ é demonstrável, então isso pode ser feito através de uma prova por diagonalização (corolário 6.2 do artigo [69]). O que podemos concluir a partir dessa afirmação aparentemente contraditória?

Fortnow [39] argumenta que o principal problema com a interpretação original de Kozen é que o conceito de prova por diagonalização não está definido. No entanto, vamos pensar um pouco mais sobre o enunciado do corolário 5.17. Note que, *assumindo* que $\text{P} \neq \text{NP}$, obtemos que $\text{SAT} \notin \text{P}$. Por isso, o teorema 5.16 pode ser aplicado, mostrando que existe uma função h computável tal que $\text{SAT} = \text{diag}_h$. Isso prova que podemos separar P e NP utilizando diagonalização? Não, pois assumimos a hipótese de que $\text{P} \neq \text{NP}$ para produzir uma função h com as propriedades desejadas. Nada garante que a prova original que separa as classes P e NP utiliza diagonalização. Portanto, embora o teorema demonstrado por Kozen seja interessante, a interpretação original dos resultados não parece estar correta.

5.8 É Possível Superar a Relativização?

Vimos nas seções anteriores que a resposta para diversas questões em aberto da teoria de complexidade computacional depende do universo computacional em consideração. Mostramos também que as provas envolvendo apenas diagonalização e simulação relativizam, ou seja, provam resultados que são válidos na presença de qualquer oráculo. Em particular, diversos métodos discutidos nesta dissertação não são capazes de resolver o problema P vs NP.

Após a demonstração do teorema BGS, as pesquisas em complexidade computacional se concentraram na busca por métodos capazes de superar a relativização. Uma possibilidade é utilizar problemas completos nas demonstrações. Por exemplo, a prova do teorema de Ladner (teorema 4.34) não pode ser estendida trivialmente para qualquer oráculo O. A demonstração desse resultado utiliza a linguagem SAT, que é completa para a classe

NP. Para generalizar esse resultado, precisamos substituí-la por uma linguagem completa para NP^O. Nesse caso, linguagens dessa forma poderiam ser obtidas, de modo similar àquele usado na demonstração do teorema 2.22.

A relativização das técnicas baseadas em simulação e diagonalização indica que para resolver o problema P vs NP devemos de fato analisar as computações envolvidas, e não apenas simulá-las. Como vimos na seção 2.7, uma das abordagens propostas foi a utilização de métodos combinatóricos baseados na complexidade de circuitos.

Por qual motivo esses métodos não foram capazes de resolver a questão P vs NP? Em 1994, Razborov e Rudich [90] apresentaram uma limitação crucial para as abordagens envolvendo limitantes inferiores no tamanho dos circuitos. Eles definiram a noção de "prova matemática natural", mostrando que todas as abordagens por circuitos utilizadas até então satisfaziam esse conceito. Em seguida, provaram que se fosse possível obter um limitante inferior forte com tais provas matemáticas, isso violaria uma forma mais forte da conjectura P vs NP que muitos pesquisadores acreditam ser verdadeira.

A classe de complexidade IP contém as linguagens que admitem provas interativas eficientes. Para uma definição precisa, veja o livro de Goldreich [47]. Fortnow e Sipser [43] demonstraram que em certos universos relativizados, algumas linguagens em PSPACE não possuem provas interativas eficientes, ou seja, existe um oráculo A tal que $\mathrm{IP}^A \neq \mathrm{PSPACE}^A$. Portanto, qualquer demonstração de que IP = PSPACE precisaria utilizar alguma técnica nova. No mesmo artigo, Fortnow e Sipser conjecturaram que as classes IP e PSPACE são diferentes.

Em 1992, A. Shamir [101] surpreendeu a comunidade científica ao demonstrar que ocorre o colapso entre essas duas classes, ou seja, IP = PSPACE. Sua prova utiliza novas ideias algorítmicas baseadas em métodos algébricos, capazes de superar a barreira imposta pela existência de um mundo relativizado contraditório. A grande novidade utilizada na demonstração foi a aritmetização de certas fórmulas lógicas basedas em um problema completo para a classe PSPACE.

Em 1998, Buhrman et al. [17] demonstraram a primeira separação de classes de complexidade que não relativiza. O resultado obtido por esses pesquisadores mostra que podemos separar classes de complexidade através de diagonalização se utilizarmos na demonstração algum resultado que não relativiza. O argumento não-relativizante utilizado nessa prova também é baseado em métodos algébricos. Portanto, ainda que a técnica de diagonalização por si só possua muitas limitações, quando combinada com outros métodos que não relativizam, ela continua capaz de provar resultados interessantes em complexidade computacional.

É possível resolver o problema P vs NP através dessas novas técnicas algébricas? Em 2008, Aaronson e Wigderson [5] mostraram que esses métodos também não são suficientes, e que diversos resultados importantes, como o já citado IP = PSPACE, possuem uma

propriedade essencial que os tornam compatíveis com tais métodos. Por outro lado, é possível provar que o problema P vs NP é resistente ao uso dessas técnicas.

Resumindo, é possível superar a relativização. No entanto, assim como ocorre com o método da diagonalização e simulação, diversas barreiras foram descobertas para as técnicas mais recentes. Por um lado, a possibilidade de provar que determinados métodos não são capazes de resolver certos problemas é um aspecto fascinante da teoria de complexidade computacional. Por outro, isso mostra que precisamos de novas ideias para resolver os problemas em aberto mais importantes na área.

5.9 Referências Adicionais

Diversos tópicos discutidos neste capítulo são abordados nos artigos de Fortnow [38, 39] e no livro de Du e Ko [33]. Consulte esses textos para mais detalhes.

O artigo de Joseph e Young [63] discute a possibilidade da independência formal de certas questões em ciência da computação. O artigo de Ben-David e Halevi [9] aborda a independência do problema P vs NP. Além desses textos, recomendamos fortemente a leitura do artigo de Aaronson [2].

Diversos resultados adicionais envolvendo relativização positiva aparecem no artigo de Book et al. [13]. Para saber mais sobre oráculos aleatórios, leia o artigo de Vollmer e Wagner [108]. Por último, um trabalho relativamente recente de Remmel et al. [83] discute certas questões relacionadas com os resultados de Kozen [69] envolvendo diagonalização.

Capítulo 6

Conclusão

Em complexidade computacional existem mais perguntas intrigantes do que respostas. Como provar que P $\neq$ NP? Quais são os limites da computação eficiente? Qual a relação entre determinismo e não-determinismo? A complexidade computacional não é uma disciplina sobre computadores, mas sim sobre computação. Infelizmente, não sabemos muito sobre esse conceito.

Vimos no início desta dissertação que, para mostrar que certos algoritmos não existem, precisamos de uma demonstração matemática. No entanto, na maior parte do texto, o que usamos realmente de matemática, além de raciocínio lógico? Essencialmente, construímos novos algoritmos para provar que certos algoritmos não existem, deixando de lado teorias e teoremas matemáticos mais avançados.

Discutimos os teoremas de hierarquia na seção 3.2. A prova desses teoremas utiliza um argumento de diagonalização muito simples. Analisando a essência desses resultados, notamos que eles estabelecem o fato fundamental de que não é possível acelerar a computação de máquinas de Turing universais. Pode parecer ridículo, mas dado nosso conhecimento atual, sem diagonalização a principal questão em aberto da teoria de complexidade computacional seria: existe alguma linguagem que não pode ser decidida em tempo linear?

Felizmente, sabemos mais do que isso. Além disso, olhando para certos universos computacionais alternativos, fomos capazes de perceber que apenas simulação e diagonalização não são suficientes para resolver diversos problemas importantes. Estimulados por esse resultado, descobrimos novos métodos, assim como novas barreiras.

Todas essas limitações sugerem que o problema P vs NP é sutil demais para as técnicas conhecidas. É bem provável que seja preciso utilizar métodos muito mais avançados para resolver esse problema. Vimos na seção 2.7 que abordagens mais recentes são baseadas nessa intuição. É muito difícil prever quando respostas definitivas serão encontradas. A única certeza é que muitas surpresas e resultados extraordinários estão por vir.

Referências Bibliográficas

[1] S. Aaronson. The complexity zoo. Disponível em http://qwiki.caltech.edu/wiki/ComplexityZoo.

[2] S. Aaronson. Is P Versus NP Formally Independent? *Bulletin of the EATCS*, 81:109–136, 2003.

[3] S. Aaronson. The prime facts: From Euclid to AKS, 2003. Disponível em http://www.scottaaronson.com/writings/prime.pdf.

[4] S. Aaronson. Guest Column: NP-complete problems and physical reality. *ACM SIGACT News*, 36(1):30–52, 2005.

[5] S. Aaronson e A. Wigderson. Algebrization: A new barrier in complexity theory. *ACM Transactions on Computation Theory*, 1(1):2, 2009.

[6] M. Agrawal, N. Kayal, e N. Saxena. PRIMES is in P. *Annals of Mathematics*, páginas 781–793, 2004.

[7] S. Arora e B. Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, New York, NY, 2009.

[8] T. Baker, J. Gill, e R. Solovay. Relativizations of the P $=^?$ NP Question. *Journal of the ACM*, 42:401–420, 1975.

[9] S. Ben-David e S. Halevi. On the independence of P versus NP. Relatório técnico, Relatório Técnico.

[10] C.H. Bennett e J. Gill. Relative to a Random Oracle A, $\mathrm{P}^A \neq \mathrm{NP}^A \neq \mathrm{coNP}^A$ with Probability 1. *SIAM Journal on Computing*, 10(1):96–113, 1981.

[11] P. Berman. Relationship between density and deterministic complexity of NP-complete languages. *Fifth International Colloquium on Automata, Languages, and Programming*, páginas 63–71, 1978.

[12] M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14(2):336, 1967.

[13] R.V. Book, T.J. Long, e A.L. Selman. Quantitative relativizations of complexity classes. *SIAM Journal on Computing*, 13:461, 1984.

[14] R.B. Boppana, J. Håstad, e S. Zachos. Does co-NP have short interactive proofs? *Information Processing Letters*, 25(2):127–132, 1987.

[15] R.B. Boppana e M. Sipser. The complexity of finite functions. *Handbook of Theoretical Computer Science*, 14:757–804, 1990.

[16] A. Borodin. Computational complexity and the existence of complexity gaps. *Journal of the ACM*, 19(1):158–174, 1972.

[17] H. Buhrman, L. Fortnow, e T. Thierauf. Nonrelativizing separations. *Proceedings of the Thirteenth Annual IEEE Conference on Computational Complexity*, páginas 8–12, 1998.

[18] J.Y. Cai e H. Zhu. Progress in computational complexity theory. *Journal of Computer Science and Technology*, 20(6):735–750, 2005.

[19] W.A. Carnielli e R.L. Epstein. Computabilidade - Funções Computáveis, Lógica e os Fundamentos da Matemática. *Editora Unesp*, 2006.

[20] A.K. Chandra, D.C. Kozen, e L.J. Stockmeyer. Alternation. *Journal of the ACM*, 28(1):114–133, 1981.

[21] B. Chor, O. Goldreich, e J. Hastad. The random oracle hypothesis is false. *Manuscrito*, 1990.

[22] A. Cobham. The Intrinsic Computational Difficulty of Functions. *Proceedings of the Third International Congress for Logic, Methodology and Philosophy of Science*. North-Holland Pub. Co., 1965.

[23] P.J. Cohen e M. Davis. *Set theory and the continuum hypothesis*. Addison-Wesley, 1966.

[24] S. Cook. The importance of the P versus NP question. *Journal of the ACM*, 50(1):27–29, 2003.

[25] S. Cook e P. Nguyen. *Logical foundations of proof complexity*. Cambridge University Press, 2010.

[26] S.A. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.

[27] S.A. Cook. A hierarchy for nondeterministic time complexity. *Journal of Computer and System Sciences*, 7(4):343–353, 1973.

[28] Stephen Cook. The P versus NP problem, 2001. Disponível em http://www.claymath.org/prize_problems/p_vs_np.pdf.

[29] T.H. Cormen, C.E. Leiserson, e R.L. Rivest. *Introduction to algorithms*. The MIT press, 2001.

[30] S.C. Coutinho. *Primalidade em tempo polinomial: uma introdução ao algoritmo AKS*. IMPA, 2003.

[31] Martin Davis. *The Undecidable*. Raven Press, Hewlett, NY, 1965.

[32] M. Dekhtyar. On the relativization of deterministic and nondeterministic complexity classes. *Mathematical Foundations of Computer Science*, páginas 255–259, 1976.

[33] D.Z. Du e K.I. Ko. *Theory of computational complexity*. Wiley New York, 2000.

[34] J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.

[35] S. Fenner, L. Fortnow, e S.A. Kurtz. The isomorphism conjecture holds relative to an oracle. *Proceedings of the 33rd Annual Symposium on Foundations of Computer Science*, páginas 30–39, 1992.

[36] M.J. Fischer e M.O. Rabin. Super-Exponential Complexity of Presburger Arithmetic, 1974.

[37] L. Fortnow. Two Proofs of Ladner's Theorem.

[38] L. Fortnow. The role of relativization in complexity theory. *Bulletin of the EATCS*, 52:229–243, 1994.

[39] L. Fortnow. Diagonalization. *Current trends in theoretical computer science*. World Scientific Publishing Company, 2001.

[40] L. Fortnow. Beyond NP: the work and legacy of Larry Stockmeyer. *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, páginas 120–127. ACM, 2005.

[41] L. Fortnow. The status of the P versus NP problem. *Communications of the ACM*, 52(9):78–86, 2009.

[42] L. Fortnow e S. Homer. A short history of computational complexity. *Bulletin of the EATCS*, 80:95–133, 2003.

[43] L. Fortnow e M. Sipser. Are there interactive protocols for co-NP languages? *Information processing letters*, 28(5):249–251, 1988.

[44] R.M. Friedberg. Two recursively enumerable sets of incomparable degrees of unsolvability. *Proceedings of the National Academy of Sciences of the United States of America*, 43(2):236–238, 1957.

[45] M.R. Garey e D.S. Johnson. Computers and intractability: a guide to NP-completeness, 1979.

[46] W.I. Gasarch. The P = NP Poll. Disponível em http://www.cs.umd.edu/~gasarch/papers/poll.pdf.

[47] O. Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[48] T. Gowers, J. Barrow-Green, e I. Leader. *The Princeton companion to mathematics*. Princeton University Press, 2008.

[49] M. Grohe. Fixed-Point Definability and Polynomial Time on Graphs with Excluded Minors.

[50] J. Hartmanis e L. Berman. On isomorphisms and density of NP and other complete sets. *Proceedings of the eighth annual ACM symposium on Theory of computing*, páginas 30–40. ACM, 1976.

[51] J. Hartmanis, R. Chang, D. Ranjan, e P. Rohatgi. Structural complexity theory: Recent surprises. *SWAT'90*, páginas 1–12, 1990.

[52] J. Hartmanis e J.E. Hopcroft. Independence results in computer science. *ACM SIGACT News*, 8(4):13–24, 1976.

[53] J. Hartmanis, N. Immerman, e V. Sewelson. Sparse sets in NP-P: EXPTIME versus NEXPTIME. *Information and Control*, 65(2-3):158–181, 1985.

[54] J. Hartmanis e R.E. Stearns. On the computational complexity of algorithms. *Transactions of the American Mathematical Society*, 117:285–306, 1965.

[55] F.C. Hennie. One-tape, off-line Turing machine computations. *Inf. and Control*, 8:553–578, 1965.

[56] F.C. Hennie e R.E. Stearns. Two-tape simulation of multitape Turing machines. *Journal of the ACM*, 13(4):533–546, 1966.

[57] M. Hirvensalo. *Quantum computing*. Natural computing series. Springer Verlag, 2004.

[58] J. Hopcroft, W. Paul, e L. Valiant. On time versus space. *Journal of the ACM*, 24(2):332–337, 1977.

[59] J. E. Hopcroft e J. K. Wong. Linear time algorithm for isomorphism of planar graphs. *STOC '74: Proceedings of the sixth annual ACM symposium on Theory of computing*, páginas 172–184, New York, NY, USA, 1974. ACM.

[60] M. Hutter. The Fastest and Shortest Algorithm for All Well-Defined Problems. *International Journal of Foundations of Computer Science*, 13(3):431–443, 2002.

[61] N. Immerman. *Descriptive complexity*. Springer Verlag, 1999.

[62] R. Impagliazzo. Computational complexity since 1980. *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, páginas 19–47.

[63] D. Joseph e P. Young. Independence results in computer science? *Journal of Computer and System Sciences*, 23(2):205–222, 1981.

[64] R. Kannan. Circuit-size lower bounds and non-reducibility to sparse sets. *Information and Control*, 55(1-3):40–56, 1982.

[65] R. Kannan. Towards separating nondeterminism from determinism. *Theory of Computing Systems*, 17(1):29–45, 1984.

[66] R.M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, páginas 85–103, 1972.

[67] P. Kaye, R. Laflamme, e M. Mosca. *An introduction to quantum computing*. Oxford University Press, USA, 2007.

[68] K.I. Ko. Relativized polynomial time hierarchies having exactly k levels. *Proceedings of the twentieth annual ACM symposium on Theory of computing*, páginas 245–253. ACM, 1988.

[69] D. Kozen. Indexings of subrecursive classes. *Theoretical Computer Science*, 11(3):277–301, 1980.

[70] S.A. Kurtz. On the random oracle hypothesis. *Information and Control*, 57(1):40–47, 1983.

[71] S.A. Kurtz, S.R. Mahaney, e J.S. Royer. The isomorphism conjecture fails relative to a random oracle. *Journal of the ACM*, 42(2):420, 1995.

[72] S.A. Kurtz, M.J. O'donnell, e J.S. Royer. How to prove representation-independent independence results. *Information Processing Letters*, 24(1):5–10, 1987.

[73] R.E. Ladner. On the structure of polynomial time reducibility. *Journal of the ACM*, 22(1):155–171, 1975.

[74] L.A. Levin. Universal search problems. *Problemy Peredachi Informatsii*, 9(3):265–266, 1973.

[75] M. Li e P. Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer-Verlag New York Inc, 2008.

[76] R.J. Lipton e A. Viglas. On the complexity of SAT. *40th Annual Symposium on Foundations of Computer Science*, páginas 459–464, 1999.

[77] W. Maass. Quadratic lower bounds for deterministic and nondeterministic one-tape Turing machines. *Proceedings of the sixteenth annual ACM symposium on Theory of computing*. ACM, 1984.

[78] S.R. Mahaney. Sparse complete sets for NP: Solution of a conjecture of Berman and Hartmanis. *Journal of Computer and System Sciences*, 25(2):130–143, 1982.

[79] A.R. Meyer e L.J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. *IEEE Conference Record of 13th Annual Symposium on Switching and Automata Theory*, páginas 125–129, 1972.

[80] A.R. Meyer e L.J. Stockmeyer. Word problems requiring exponential time. *Proc. 5th ACM Symp. on the Theory of Computing*, páginas 1–9, 1973.

[81] G.L. Miller. Riemann's hypothesis and tests for primality. *Journal of computer and system sciences*, 13(3):300–317, 1976.

[82] A.A. Muchnik. On the unsolvability of the problem of reducibility in the theory of algorithms. *Dokl. Akad. Nauk SSSR, NS*, volume 108, 1956.

[83] A. Nash, R. Impagliazzo, e J. Remmel. Universal languages and the power of diagonalization. *Proc. 18th IEEE Conference on Computational Complexity*, páginas 337–346, 2003.

[84] P. Odifreddi. *Classical Recursion Theory.* Elsevier, Amsterdam, 1999.

[85] P. Odifreddi. *Classical Recursion Theory. Volume 2.* Elsevier, Amsterdam, 1999.

[86] M. Ogiwara e O. Watanabe. On polynomial time bounded truth-table reducibility of NP sets to sparse sets. *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, páginas 457–467. ACM, 1990.

[87] I.C. Oliveira e A.V. Moura. A New Look at Some Classical Results in Computational Complexity. *ECCC – Electronic Colloquium on Computational Complexity*, TR09–025, 2009.

[88] C.H. Papadimitriou. *Computational complexity.* John Wiley and Sons Ltd., 2003.

[89] W.J. Paul, N. Pippenger, E. Szemeredi, e W.T. Trotter. On determinism versus non-determinism and related problems. *24th Annual Symposium on Foundations of Computer Science*, páginas 429–438, 1983.

[90] A.A. Razborov e S. Rudich. Natural proofs. *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, páginas 213. ACM, 1994.

[91] K.W. Regan. Understanding the Mulmuley-Sohoni Approach to P vs. NP. *Bulletin of the EATCS*, 78:86–99, 2002.

[92] O. Reingold. Undirected st-connectivity in log-space. *Proceedings of the thirty-seventh annual ACM Symposium on Theory of computing*, páginas 385. ACM, 2005.

[93] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* McGraw-Hill, 1967.

[94] W. Rudin e J. Cofman. *Principles of mathematical analysis.* McGraw-Hill New York, 1964.

[95] J.E. Savage. *Models of computation: Exploring the power of computing.* Addison-Wesley Publishing Company, Reading, MA, 1998.

[96] M. Schaefer e C. Umans. Completeness in the polynomial-time hierarchy: A compendium. *Sigact News*, 33(3):32–49, 2002.

[97] U. Schöning. A uniform approach to obtain diagonal sets in complexity classes. *Theoretical Computer Science*, 18(1):95–103, 1982.

[98] J.I. Seiferas. Machine-Independent Complexity Theory, Handbook of Theoretical Computer Science (vol. A): Algorithms and Complexity, 1991.

[99] J.I. Seiferas, M.J. Fischer, e A.R. Meyer. Separating nondeterministic time complexity classes. *Journal of the ACM*, 25(1):146–167, 1978.

[100] A. Selman. On the structure of NP. *Notices Amer. Math. Soc*, 21(6):310, 1974.

[101] A. Shamir. IP = PSPACE. *Journal of the ACM*, 39(4):869–877, 1992.

[102] M. Sipser. The history and status of the P versus NP question. *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, páginas 603–618. ACM New York, NY, USA, 1992.

[103] M. Sipser. *Introduction to the Theory of Computation*. PWS, Boston, MA, 1996.

[104] R.E. Stearns, J. Hartmanis, e P.M. Lewis. Hierarchies of memory limited computations. *Proceedings of the 6th Annual Symposium on Switching Circuit Theory and Logical Design*, páginas 179–190. IEEE Computer Society, 1965.

[105] L.J. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):1–22, 1976.

[106] L. Trevisan. The Program-Enumeration Bottleneck in Average-Case Complexity Theory. *ECCC – Electronic Colloquium on Computational Complexity*, TR10–034, 2010.

[107] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 2(1):230, 1937.

[108] H. Vollmer e K.W. Wagner. Measure one results in computational complexity theory. *Advances in Algorithms, Languages, and Complexity*, páginas 285–312, 1997.

[109] A. Wigderson. P, NP and Mathematics - A Computational Complexity Perspective. *Proc. of the International Congress of Mathematicians*, 2006.

[110] R. Williams. Time-space tradeoffs for counting NP solutions modulo integers. *Computational Complexity*, 17(2):179–219, 2008.

[111] R. Williams. Alternation-trading proofs, linear programming, and lower bounds. *STACS*, 2010.

[112] G.J. Woeginger. The P vs NP Page. Disponível em http://www.win.tue.nl/~gwoegi/P-versus-NP.htm.

[113] C. Wrathall. Complete sets and the polynomial-time hierarchy. *Theoretical Computer Science*, 3(1):23–33, 1976.

[114] M. Yannakakis. Expressing combinatorial optimization problems by linear programs. *Journal of Computer and System Sciences*, 43(3):441–466, 1991.

[115] T. Yato e T. Seta. Complexity and completeness of finding another solution and its application to puzzles. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, 86(5):1052–1060, 2003.

[116] S. Zak. A Turing machine time hierarchy. *Theoretical Computer Science*, 26(3):327–333, 1983.