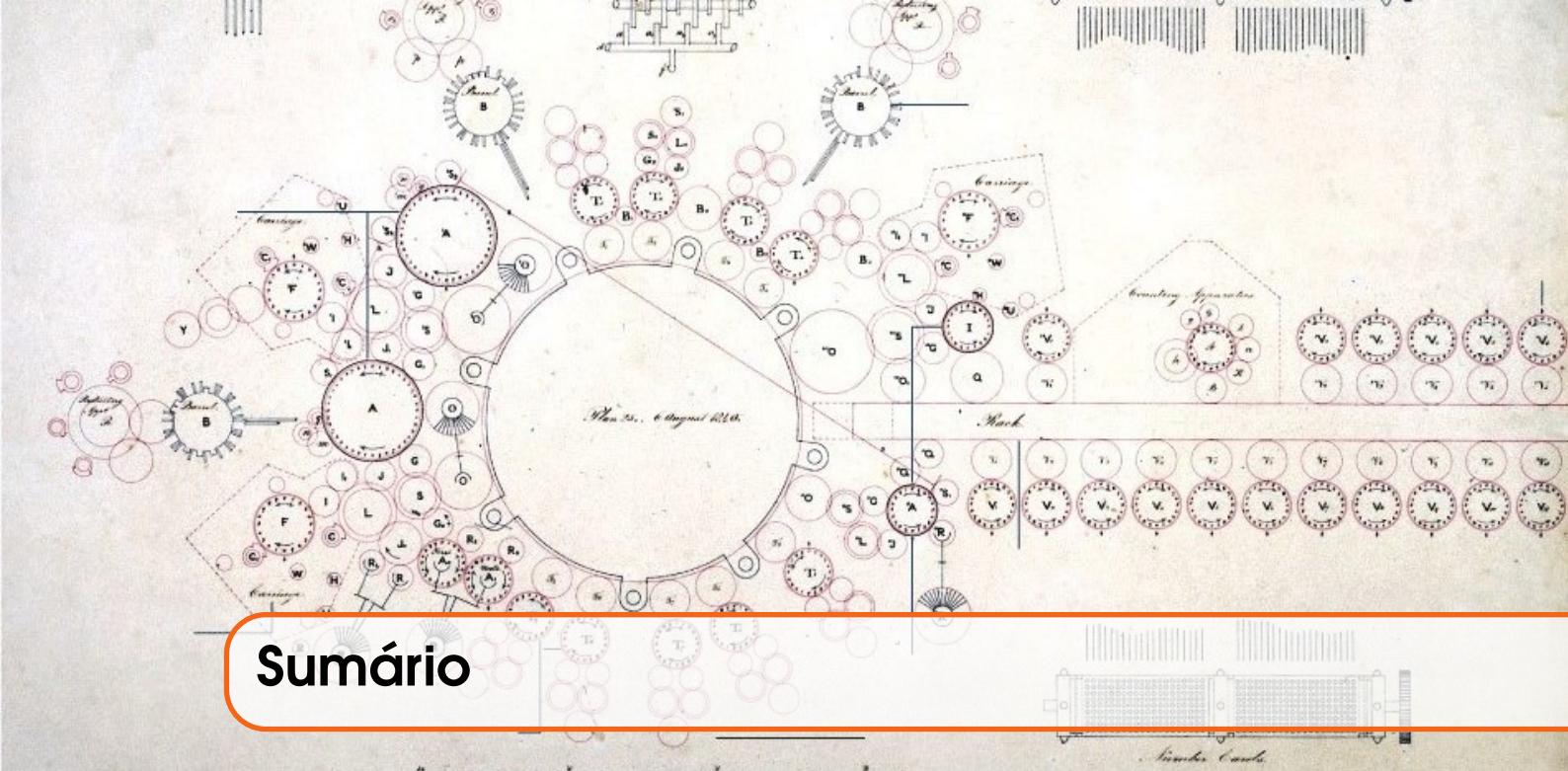


Autômatos, computabilidade e complexidade computacional

Murilo V. G. da Silva

AUTÔMATOS, COMPUTABILIDADE E COMPLEXIDADE COMPUTACIONAL
© MURILO VICENTE GONÇALVES DA SILVA 2017 – 2019

Este texto está licenciado sob a Licença *Attribution-NonCommercial 3.0 Unported License* (*the “License”*) da *Creative Commons*. Em resumo, você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra). Você não pode usar esta obra para fins comerciais. Se você alterar, transformar ou criar com base nesta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente. Para ver uma cópia desta licença, visite <http://creativecommons.org/licenses/by-nc/3.0>.



Sumário

1	Prólogo	7
1.1	O que é computação?	7
1.2	Algoritmos, problemas e computadores	8

I

Parte 1: Teoria de linguagens e autômatos

2	Alfabetos, Strings e Linguagens	15
2.1	Alfabetos e strings	15
2.1.1	Exercícios	18
2.2	Linguagens	18
2.2.1	Exercícios	21
2.2.2	Operações com linguagens	21
3	Autômatos e Linguagens Regulares	23
3.1	Autômatos Finitos Determinísticos (AFDs)	23
3.1.1	Modelando matematicamente autômatos	24
3.1.2	Aceitação e rejeição de strings	26
3.1.3	Definição formal para aceitação e rejeição de strings	27
3.1.4	Exercícios	28
3.2	Autômatos Finitos não Determinísticos (AFNs)	29
3.2.1	Definição formal para autômatos finitos não determinísticos	30
3.2.2	Aceitação e rejeição de strings por AFNs	32
3.2.3	Exercícios	33

3.3	Equivalência entre AFDs e AFNs	34
3.3.1	Algoritmo de construção de conjuntos	34
3.3.2	Algoritmo de construção de conjuntos: versão melhorada	35
3.3.3	Exercícios	37
3.4	Autômatos Finitos não Determinísticos com transições ϵ	37
3.5	Equivalência entre AFDs e ϵ-AFNs	40
3.5.1	Exercícios	41
3.6	Expressões Regulares (ERs)	42
3.6.1	Construindo Expressões Regulares	42
3.6.2	Expressões Regulares e Autômatos	44
3.6.3	Exercícios	44
4	Para além das Linguagens Regulares	47
4.1	O Lema do Bombeamento para Linguagens Regulares	47
4.2	Autômato com Pilha (AP)	49
4.2.1	O modelo matemático para autômatos com pilha	50
4.2.2	Computação com APs	51
4.2.3	Aceitação por pilha vazia	51
4.2.4	APs determinísticos	52
4.3	Gramáticas Livre de Contexto	53
4.3.1	Definição formal de gramáticas livre de contexto	53
4.3.2	Derivações de uma gramática	54
4.3.3	Derivação mais a direita e mais a esquerda	55
4.3.4	Árvores de análise sintática	55
4.3.5	Ambiguidade de Gramáticas	55
4.3.6	Equivalência entre APs e gramáticas livre de contexto	56
4.3.7	APDs e ambiguidade de gramáticas	56
4.3.8	Exercícios	56

II

Parte 2: Máquinas de Turing e Computabilidade

5	A Máquina de Turing	61
5.1	Revisão: problemas computacionais	61
5.2	Definição da Máquina de Turing	63
5.2.1	O funcionamento de uma Máquina de Turing	64
5.2.2	Diagrama de estados de uma Máquina de Turing	65
5.2.3	Linguagem de uma Máquina de Turing	66
5.3	Um Algoritmo é uma Máquina de Turing que sempre para	68
5.3.1	Exercícios	70
6	A Tese de Church-Turing	73
6.1	Perspectiva histórica	73
6.2	Máquinas de Turing são equivalentes a linguagens de programação	74
6.2.1	Programas Assembly	74

6.3	Máquinas de Turing e outros modelos de computação	76
6.4	A Tese de Church-Turing e suas interpretações	77
6.4.1	A TCT como definição matemática	78
6.4.2	A TCT como afirmação empiricamente verificável	78
6.5	A Tese de Church-Turing estendida	81
6.5.1	Exercícios	82
7	Computabilidade	83
7.1	Funções computáveis	83
7.2	Codificando objetos matemáticos em binário	84
7.2.1	Notação para Máquinas de Turing tomando vários argumentos de entrada .	84
7.2.2	Representando objetos matemáticos	85
7.2.3	Problemas de Decisão	86
7.3	Máquinas de Turing, pseudo-códigos, generalidade e especificidade	87
7.4	O problema da Parada	88
7.5	A Máquina de Turing Universal	89
7.6	Máquinas de Turing não determinísticas (MTN)	91
7.7	Exercícios	93

III

Parte 3: Complexidade Computacional

8	Complexidade de Tempo e Espaço	97
8.1	Complexidade de Tempo e de Espaço de Máquinas de Turing	98
8.2	As classes P, NP e P-space	100
8.3	Exercícios	102
9	A classe NP	103
9.1	Decidir ou verificar?	104
9.2	Certificados e verificação em tempo polinomial	105
9.2.1	Verificando o problema SAT em tempo polinomial	105
9.2.2	Redefinindo a classe NP	107
9.3	Exercícios	108
10	NP-completude	109
10.1	NP-completude e o Teorema de Cook-Levin	109
10.2	Lidando com problemas de busca e otimização	111
10.3	Provando a NP-completude de problemas	112
10.3.1	Provando que o problema do conjunto independente é NP-completo	112
10.4	Exercícios	114
	Bibliography	117
	Livros	117



1. Prólogo

1.1 O que é computação?

A maioria de nós tem uma noção intuitiva do que seja um algoritmo ou um computador. Estes dois conceitos são tão corriqueiros que é comum não pararmos para pensar a fundo sobre o que eles realmente significam. Entretanto, como veremos neste texto, quando começamos a fazer perguntas mais profundas a respeito de computação, as nossas noções intuitivas sobre algoritmos e computadores não nos serão suficientes para chegarmos a respostas adequadas. Um dos objetivos deste texto é apresentar conceitos fundamentais da computação de maneira matematicamente precisa e clara.

Um dos primeiros passos que precisamos tomar é pensar em algoritmos e computadores sem nos atermos à artefatos tecnológicos do nosso cotidiano, como laptops, smartphones, Linguagem C, ou Python, de forma que passaremos a enxergar computação como uma ciência que transcende a tecnologia corrente. Um bom ponto de partida para entendermos que isso é possível, é a observação de que o próprio conceito de *informação*, algo intimamente ligado à computação, já vem sendo estudado muito antes de termos computadores digitais. A ideia de informação, que é algo natural e antigo, começou a entrar de maneira mais regular no vocabulário dos cientistas no final do século XIX, especialmente no vocabulário dos físicos, quando estes começaram a esbarrar em conceitos como entropia e termodinâmica.

Uma série de descobertas científicas do último século, como por exemplo a descoberta das leis da mecânica quântica (leis em que a ideia de informação quântica é central), ou a descoberta da molécula de DNA (molécula que está intrinsecamente ligada a ideia de informação genética), começaram a solidificar a intuição que temos hoje de que informação é algo fundamental e que permeia o mundo natural em diferentes níveis de análise.

Com a popularização dos computadores na segunda metade do século XX, conceitos como informação e computação tornaram-se populares. Isso pode ser positivo por um lado, mas, por outro, acaba nos enviesando e nos induz a associar conceitos como informação e computação unicamente à artefatos tecnológicos, que é algo que queremos evitar neste livro. Para ilustrar o que queremos dizer, observe que não é incomum que em situações em que estamos descrevendo

processos computacionais no mundo natural, como moléculas de DNA *armazenando informação*, ou cérebros *processando informação*, nós venhamos a pensar que estamos meramente usando metáforas inspiradas pela nossa tecnologia corrente. Embora haja algum mérito em tal linha de pensamento¹, ela pode obscurecer o fato de que os conceitos de informação e de computação estão presentes no mundo (na matemática, na natureza, etc), e podem ser estudados como objetos em si, independentes de qualquer contingência tecnológica. Para isso, uma das tarefas essenciais que empreender neste texto é o entendimento do que são, fundamentalmente, algoritmos e computadores.

Além de algoritmos e computadores, um outro conceito que normalmente usamos de maneira intuitiva e que iremos tornar preciso neste texto é o conceito de *problema computacional*. Por exemplo, considere o problema de testar se um dado número é primo, ou o problema de testar se existe um caminho ligando dois vértices em um grafo. Em nosso estudo de teoria da computação uma questão que surgirá naturalmente neste contexto é definir exatamente o que é, de *maneira genérica*, um problema computacional. O quadro abaixo esclarece isso um pouco.

DEFINIÇÕES GENÉRICAS E DEFINIÇÕES ESPECÍFICAS

No Capítulo 2 iremos desenvolver algumas ferramentas matemáticas que nos ajudarão a dar definições precisas para o conceito de problema computacional. Entretanto, antes de termos estas ferramentas em mãos, vamos esclarecer aonde estamos querendo chegar com isso.

Vamos usar uma analogia com algo bastante familiar. Pense na função $f(x) = x^2$ e na função $g(x) = \log x$. Temos aqui dois casos *específicos* e matematicamente bem definidos de funções. Entretanto, sabemos que é perfeitamente possível sermos mais *genéricos* e fornecermos uma definição para o conceito *genérico* de função¹ tal que as funções $f(x)$ e $g(x)$ anteriores são casos particulares (ou instâncias) de tal definição. Ou seja, podemos falar de objetos matemáticos específicos, como $f(x) = x^2$ e $g(x) = \log x$, mas também do objeto matemático genérico que chamamos de *função*. A analogia que queremos fazer aqui é que é possível dar uma definição genérica para o conceito de problema computacional, tal que os problemas específicos, como testar primalidade de números ou testar a conectividade de um grafo, sejam *especificações* do conceito geral de problema computacional. Da mesma forma, neste curso vamos lidar não somente com algoritmos específicos, mas também com uma *definição matemática genérica para o próprio conceito de algoritmo*. Em nossos estudos, a flexibilidade mental de entender esta distinção entre casos gerais e casos particulares é bastante importante.

¹O leitor aqui pode estar um pouco enferrujado e não lembrar exatamente qual é definição matemática de uma função. Na verdade, isso não importa tanto aqui, pois o que queremos enfatizar neste ponto é que uma função, enquanto objeto matemático, *pode* ser precisamente definida. Curiosamente, funções podem ser definidas como casos particulares objetos matemáticos ainda mais genéricos, conhecidos como relações.

1.2 Algoritmos, problemas e computadores

Uma vez que temos em mãos definições precisas para algoritmos e para problemas computacionais, uma pergunta surge naturalmente é a seguinte: Será que existe algum problema para o qual não exista nenhum algoritmo que o resolva? Note que não estamos falando de problemas para os quais nós, hoje, ainda não conhecemos algoritmos que os solucionem, mas que no futuro os venhamos a descobrir. O que estamos perguntando aqui é se existem problemas que não podem ser resolvidos por nenhum algoritmo (entre os infinitos possíveis). Um dos resultados que veremos neste livro é que, de fato, existem problemas assim. Uma das razões para se estudar teoria da

¹Em particular, é preciso ter clareza filosófica em temas como este (no Capítulo 6 apresentamos uma breve discussão em torno de questões de interesse filosófico).

computação é a possibilidade de termos *certeza matemática* de tais afirmações a respeito das bases da ciência da computação.

No capítulo 5, veremos uma descrição matemática genérica para o conceito de algoritmo, de maneira que, qualquer algoritmo, com o naturalmente concebemos, acaba sendo um caso particular desta definição genérica. Mas antes disso, nos Capítulos 3 e 4, veremos definições matemáticas simplificadas de algoritmos (definições que “capturam” apenas um subconjunto de todos os possíveis algoritmos). Há duas razões para vermos estes modelos matemáticos simplificados: (1) Uma razão é pedagógica: primeiro ganhamos intuição estudando modelos mais simples, para depois ver a definição precisa para o conceito de algoritmo do capítulo 5; (2) A segunda razão é que estes modelos são úteis não apenas em teoria da computação, mas também em outras áreas da computação (e.g., processamento de texto, construção de compiladores, etc).

REFLETINDO UM POUCO: LIMITE TECNOLÓGICO OU PRINCÍPIO FUNDAMENTAL?

Na discussão anterior mencionamos a possibilidade de que existam problemas insolúveis. Mas a existência destes problemas não refletiria apenas uma limitação para o que atualmente entendemos por computadores e algoritmos? Ou seja, será que tais problemas não seriam apenas insolúveis apenas atualmente? Estes problemas não poderiam ser resolvidos no futuro por computadores “exóticos” descritos por modelos matemáticos que ainda não conhecemos? Ou a insolubilidade de certos problemas seria algo mais fundamental? Até que ponto podemos descartar a possibilidade de que seja possível construir objetos exóticos que poderíamos usar como computadores para resolver estes problemas que os computadores atuais não resolvem?

O consenso científico atual, chamado de Tese de Church-Turing (TCT), é que qualquer problema solúvel poderia ser resolvido, em princípio, por um computador como concebemos hoje. Este tipo de questão, é claro, como qualquer questão científica corrente, é passível de debate. Entretanto, a maioria das propostas que aparecem na literatura questionando a TCT, embora matematicamente bem definidas, tendem a cair, em última análise, em variações de ideias que podem ser inviáveis de ser realizadas fisicamente^a. O Capítulo 6 deste livro é dedicado a discussão da Tese de Church-Turing.

Entretanto, isso tudo não exclui a possibilidade de que possamos construir computadores *fundamentalmente* muito mais *eficientes* que os atuais. A pesquisa em computação quântica, por exemplo, investiga precisamente a possibilidade de construção de computadores *exponencialmente* mais rápidos do que os atuais na resolução de *alguns* problemas.

^aUm caso comum se refere a um modelo matemático que requer a existência de “oráculos mágicos” que executam passos computacionais sem nenhuma explicação. Um outro caso comum é o de um modelo matemático que requer a realização de computação “verdadeiramente analógica”, uma ideia que parece contradizer alguns pressupostos da física contemporânea.

Embora um curso de teoria da computação seja um curso de matemática, e não de física, é importante observarmos que objetos abstratos estudados em computação, como algoritmos e informação, se manifestam, de alguma forma no mundo físico².

O seu laptop rodando Windows é o caso mais óbvio do que podemos entender como um processo computacional ocorrendo em um meio físico. Entretanto, tal caso não é único, e este é exatamente o ponto que queremos ressaltar aqui. O cálculo do logaritmo de um número ocorrendo em uma máquina com engrenagens e graxa, como a máquina de Charles Babbage, do século XIX,

²O ponto de contato entre a natureza matemática da teoria da computação e questões de natureza empírica é bastante útil e será discutido com mais cuidado no Capítulo 6. Sem entrar em uma discussão filosófica mais profunda, o ponto que queremos chamar atenção aqui é simplesmente que existe conexão entre modelos matemáticos estudados em computação e objetos do “mundo físico”.

também é um outro exemplo de um processo computacional ocorrendo no meio físico. Poderíamos também pensar no cérebro de um primata processando sinais elétricos sensoriais, ou moléculas de DNA executando rotinas biológicas, ou mesmo em um punhado de átomos, elétrons e fótons interagindo sistematicamente em algum protótipo de computador quântico, de maneira que a manipulação da informação quântica de destas partículas fazem parte do processo de execução de um algoritmo quântico. A Figura 1.1 ilustra tais exemplos. O quadro “*Computação: instanciando objetos abstratos em objetos físicos*”, abaixo, descreve tais casos de maneira unificada.

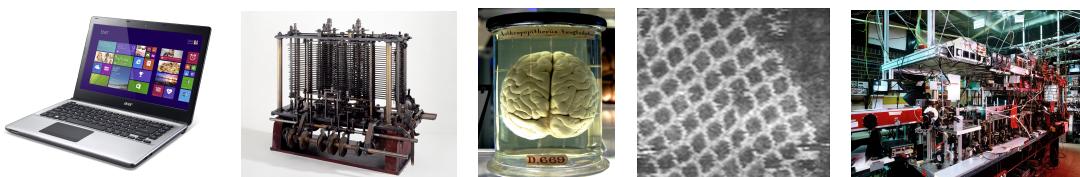


Figure 1.1: Computação ocorrendo em diferentes meios físicos: circuitos eletrônicos, engrenagens mecânicas, estruturas biológicas, moléculas de DNA e partículas subatômicas.

COMPUTAÇÃO: INSTANCIANDO OBJETOS ABSTRATOS EM OBJETOS FÍSICOS

Podemos pensar em um computador como uma maneira de instanciar objetos abstratos e o conjunto de possíveis relacionamentos matemáticos destes objetos abstratos (i.e., informação e algoritmos) em objetos físicos e o conjunto dos possíveis graus de movimento (ou graus de liberdade de movimento) destes objetos físicos.

O objetivo da teoria da computação é prover modelos matemáticos para processos computacionais que sejam *independentes do substrato físico* em que a computação esteja ocorrendo. O nosso objetivo principal é ter um modelo matemático que seja independente do substrato físico, mas que, ao mesmo tempo, seja abrangente e realista o suficiente para capturar qualquer possível computação ocorrendo em qualquer meio físico possível. Por quê isso seria o ideal? Por que uma vez que temos um modelo matemático com estas características, bastaríamos nos focar neste modelo para poder saber o que pode e o que não pode ser *efetivamente* computado.

Claramente este objetivo parece ser bastante ambicioso, pois estamos atrás de um modelo matemático que possa descrever todo e qualquer tipo de manipulação de informação em qualquer tipo de substrato físico! Entretanto, veremos que na década de 1930 foi proposto um modelo que, apesar de ser razoavelmente simples, parece ter tais propriedades. Este modelo é conhecido como *Máquina de Turing*. A tese científica que afirma que Máquinas de Turing atingem tal nível de generalidade em termos de poder representar qualquer processo computacional (i.e., processos para manipular/transformar informação) é conhecida hoje em dia como *Tese de Church-Turing*³.

Os computadores que usamos no dia a dia são objetos precisos e sabemos rigorosamente como eles funcionam (afinal, fomos nós quem os construímos!) e, portanto, temos modelos matemáticos

³Atualmente há duas interpretações para o que se entende por Tese de Church-Turing (TCT). Estas duas interpretações serão vistas com cuidado no Capítulo 6. Uma interpretação é que a TCT é uma *definição matemática* e a outra, que é a que nos referimos neste parágrafo, é que ela é uma *afirmação sobre a realidade física*. Um ponto fundamental desta segunda versão da TCT é que, na possibilidade dela ser falsa, em princípio existe meios de refutá-la. Isso é importante, pois para que uma hipótese que seja relevante no meio científico, especialmente em ciências empíricas, deve haver algum meio de refutá-la por experimentos, caso ela venha a ser falsa. Veremos também quais foram as razões que foram tornando esta tese mais e mais sólida com o tempo.

que os descrevem com exatidão. Veremos que os modelos matemáticos que descrevem nossos computadores atuais são equivalentes a certas Máquinas de Turing conhecidas como Máquinas de Turing Universais. A propriedade essencial de tais máquinas universais é que elas podem simular⁴ qualquer outra Máquina de Turing, e, portanto, de acordo com a TCT, poderiam simular qualquer outro processo computacional com alguma correspondência em algum substrato físico. Isso significa que, pelo menos em princípio, não existem instanciações físicas que possam resolver problemas que não possam ser resolvidos por computadores atuais devidamente programados e com as condições de eficiência e de memória adequadas⁵. A definição matemática de uma Máquina de Turing Universal é o modelo matemático para o que chamamos de *computador*.

O fato de que existem Máquinas de Turing Universais, que é um dos resultados mais importantes no artigo original escrito por Alan Turing (a Figura 1.2 mostra um fragmento de tal artigo), é conhecido como *universalidade* das Máquinas de Turing.

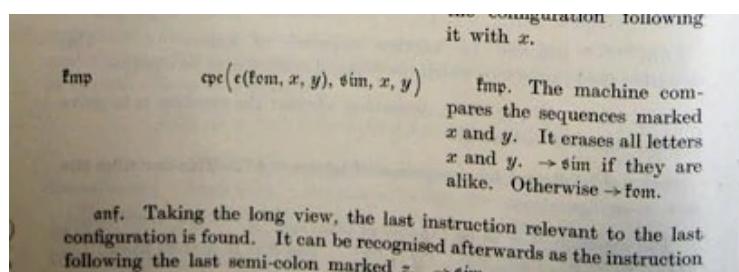


Figure 1.2: Um trecho do famoso artigo publicado em 1936 por Alan Turing, intitulado *On computable numbers, with an application to the Entscheidungsproblem*. Neste artigo é que foram estabelecidas as bases de toda ciência da computação.

A TESE DE CHURCH-TURING E A UNIVERSALIDADE DA COMPUTAÇÃO

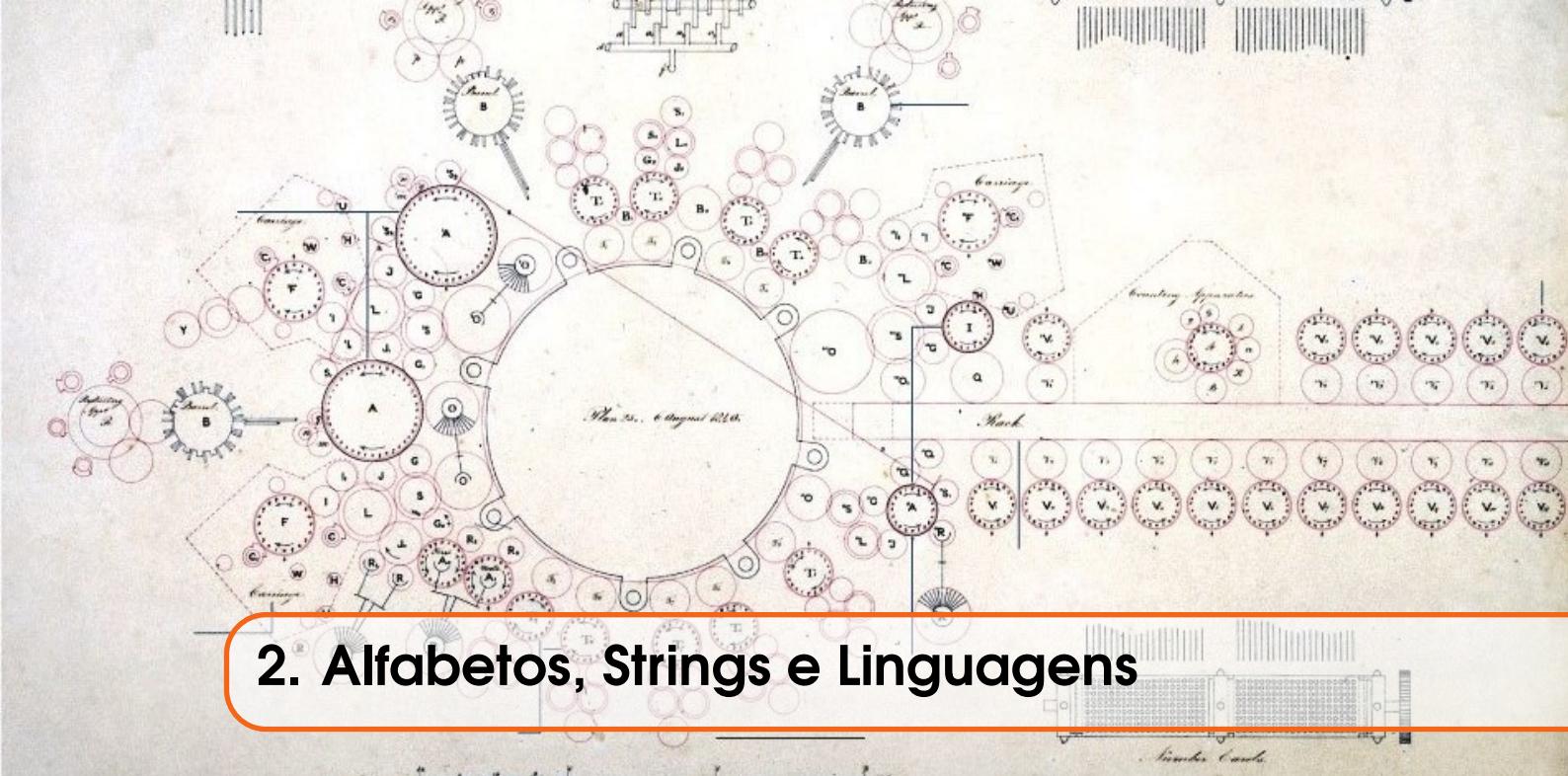
*“A lição que tomamos da universalidade da computação é que podemos construir um objeto físico, que chamamos de computador, que pode simular qualquer outro processo físico, e o conjunto de todos os possíveis movimentos deste computador, definido pelo conjunto de todos os possíveis programas concebíveis, está em **correspondência de um para um** com o conjunto de todos os possíveis movimentos de qualquer outro objeto físico concebível”.* – David Deutsch

⁴Aqui a palavra “simular” tem um significado matematicamente preciso que, a grosso modo, significa uma certa *correspondência de um para um* entre dois modelos matemáticos.

⁵Claramente afirmações assim podem gerar discussões de natureza filosófica (com direito a bastante confusão e imprecisão em muitos casos!).

Parte 1: Teoria de linguagens e autômatos

- 2 Alfabetos, Strings e Linguagens 15**
 - 2.1 Alfabetos e strings
 - 2.2 Linguagens
- 3 Autômatos e Linguagens Regulares .. 23**
 - 3.1 Autômatos Finitos Determinísticos (AFDs)
 - 3.2 Autômatos Finitos não Determinísticos (AFNs)
 - 3.3 Equivalência entre AFDs e AFNs
 - 3.4 Autômatos Finitos não Determinísticos com transições ϵ
 - 3.5 Equivalência entre AFDs e ϵ -AFNs
 - 3.6 Expressões Regulares (ERs)
- 4 Para além das Linguagens Regulares . 47**
 - 4.1 O Lema do Bombeamento para Linguagens Regulares
 - 4.2 Autômato com Pilha (AP)
 - 4.3 Gramáticas Livre de Contexto



2. Alfabetos, Strings e Linguagens

2.1 Alfabetos e strings

Algoritmos, computadores e problemas computacionais são objetos matemáticos complexos. Nossa primeiro passo será definir os conjuntos elementares de símbolos que usaremos para construir tais objetos. Ou seja, apresentar os tijolos básicos que serão necessários para construirmos todo o nosso edifício intelectual. Estes tijolos básicos serão chamados de símbolos e conjuntos finitos destes símbolos serão chamados de alfabetos.

Definição 2.1.1 — Alfabeto e símbolos. Um *alfabeto* é um conjunto finito qualquer e *símbolos* são elementos deste conjunto.

■ **Exemplo 2.1** Considere o conjunto $X = \{a, b, c, d\}$. Os elementos a, b, c, d do conjunto X são chamados de *símbolos do alfabeto X*.

■ **Exemplo 2.2** Considere o conjunto $\Sigma = \{0, 1\}$. Os elementos 0, 1 do conjunto Σ são chamados de *símbolos do alfabeto Σ* .

Note que os conceitos de alfabeto e símbolos nada mais são do que sinônimos dos conceitos matemáticos de conjuntos e elementos. Iremos usar vários alfabetos neste livro, entretanto, o mais utilizado será o alfabeto *alfabeto binário*, apresentando no Exemplo 2.2. Um exemplo de alfabeto bastante comum no mundo das linguagens de programação é o seguinte: $\Sigma_C = \{0, 1, 2, \dots, 9, a, b, \dots, z, A, B, \dots, Z, \dots, &, \#, \%, >, <, \dots\}$. A ideia é que este último alfabeto trata-se do conjunto de todos caracteres válidos em um programa escrito na linguagem C no padrão ANSI C. Poderíamos pensar em ou série de outros exemplos semelhantes.

Assim como não é possível construir um edifício usando apenas um tijolo, objetos matemáticos complexos não podem ser descritos usando-se apenas um símbolo. Para descrever objetos matemáticos complexos, iremos utilizar sequências de símbolos. A segunda definição que veremos neste capítulo se refere exatamente a isso.

Definição 2.1.2 — Strings. Dado um alfabeto Σ , uma *string sobre Σ* é uma sequência de símbolos de Σ justapostos.

Por exemplo, *aaba* é uma string sobre o alfabeto $X = \{a, b, c, d\}$, do Exemplo 2.1. A string *00101* é uma string sobre o alfabeto binário. Um outro exemplo de string é um programa escrito em linguagem C (note que tal programa é uma sequência de símbolos ANSI, incluindo os símbolos especiais usados para quebra de linha, indentação e espaçamento), que pode ser visto matematicamente como uma string sobre o alfabeto Σ_C , que mencionamos anteriormente.

Notação 2.1. Em geral usamos a letra grega Σ para denotar alfabetos. No caso em que o alfabeto não estiver explicitamente definido, a letra grega Σ denotará, por convenção, o alfabeto binário.

Terminologia 2.1. Embora estejamos usando “string” para se referir a uma sequência de símbolo, observamos que em textos em português é comum também que se use “palavra” para se referir a estes mesmos objetos matemáticos.

Definição 2.1.3 — Substring e concatenação de strings. Uma *substring* de w é uma subsequência de símbolos de w . A concatenação de duas strings x e y , denotada xy , é a string resultante da justaposição das strings x e y .

■ **Exemplo 2.3** As strings *ab*, *bb* e *bbc* são algumas das substrings da string *abbbbc*. ■

■ **Exemplo 2.4** Para um exemplo de concatenação, considere as strings $x = 111$ e $y = 000$. Neste caso, a string $xy = 111000$ é a concatenação de x e y e a string 000000 é a concatenação de y com ela mesma (i.e., $yy = 000000$). ■

Observe que concatenação não é uma operação comutativa. A partir do Exemplo 2.4, temos que $yx = 000111$, portanto $xy \neq yx$. Por outro lado, para quaisquer strings x, y, z , $(xy)z = x(yz)$, ou seja, a operação de concatenação de strings é associativa.

TEORIA DE LINGUAGENS FORMAIS

Em alguns livros de Teoria da Computação a definição de concatenação de strings é apresentada de maneira um pouco mais formal (veja o livro [Sud05], por exemplo) e em outros de maneira um pouco mais “intuitiva” (veja o livro [Sip06]), por exemplo). Esta variação é uma questão da ênfase que o autor quer dar ao assunto.

Na Definição 2.1.3 optamos pela versão mais intuitiva. De maneira semelhante, quando dissemos que concatenação é uma operação associativa nós não apresentamos uma demonstração matemática para isso (no livro [Sud05] tal demonstração é feita). Esta opção que fizemos aqui não tem a ver com a importância em sermos formais, pois seremos bastante formais no decorrer do texto. A ideia é que o nosso curso tem um enfoque mais na linha de cobrir em *amplitude as bases da teoria da computação* e menos na linha de cobrir *em profundidade a teoria de linguagens formais*.

Ainda assim, no Exercício 2.3 pedimos por uma definição mais formal para a operação de concatenação de strings. O intuito disso é mais na linha de fazer o aluno pensar um pouco no assunto e perceber que podemos dar definições diferentes para um mesmo conceito, e menos na linha de tratar em detalhes formais os desdobramentos do assunto em questão. Para os alunos interessados, um tratamento um pouco mais aprofundado a respeito de teoria linguagens formais pode ser visto em mais detalhes em [Sud05].

Definição 2.1.4 — Tamanho de uma string. O *tamanho* de uma string w é o número de símbolos de w , e é denotado por $|w|$.

■ **Exemplo 2.5** Considere as strings $aabc$ e 01 . Nestes casos escrevemos $|aabc| = 4$ e $|001| = 2$. ■

Definição 2.1.5 — String nula. Denotamos a *string nula*, ou seja, a string que não contém nenhum símbolo, por ϵ .

Observe que, de acordo com nossa definição, $|\epsilon| = 0$. Note também que ϵ é substring de qualquer possível string.

A seguir, definiremos o que é a reversa de uma string. Intuitivamente, a ideia é simples: dada uma string w , a reversa de w , denotada por w^R , é a string lida de trás para frente (e.g., se $w = 1110$, então $w^R = 0111$). Vamos formalizar isso usando uma definição indutiva¹:

Definição 2.1.6 — Reversa de uma string. A reversa de uma string w , denotada w^R , é definida recursivamente da seguinte maneira:

Base: Se $w = \epsilon$, então definimos $w^R = \epsilon$.

Caso Geral: Seja w uma string com pelo menos um símbolo. Se a string w tem a forma $w = xa$, tal que x é uma string e a é o último símbolo de w , então $w^R = ax^R$.

Notação 2.2. Seja Σ um alfabeto e $a \in \Sigma$. Diremos que uma string sobre este alfabeto é da forma a^n se a string é formada por n símbolos a consecutivos. De maneira análoga, se w é uma string sobre Σ , ao escrevermos w^n estamos nos referindo à concatenação de n cópias da string w .

■ **Exemplo 2.6** A string 11111 pode ser escrita como 1^5 . ■

■ **Exemplo 2.7** Considere a string $w = 10$. Com isso $w^6 = 101010101010$. ■

Note que, em particular, $x^0 = \epsilon$ para qualquer string x . No Exercício 2.3, pedimos para o aluno fornecer uma definição formal para o conceito de concatenação de strings.

Definição 2.1.7 — Potência de um alfabeto. Dado um alfabeto Σ , o conjunto Σ^i de strings w sobre Σ , tal que $|w| = i$, é dito uma *potência de Σ* . Definimos $\Sigma^0 = \{\epsilon\}$.

Por exemplo, dado o alfabeto binário Σ , o conjunto Σ^3 é o seguinte conjunto de strings: $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$.

Dado um alfabeto Σ e número natural i , note que Σ^i é um conjunto finito. A seguir vamos definir o conjunto infinito de todas as strings sobre Σ , ou seja, $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$. A definição formal é a seguinte.

Definição 2.1.8 — O conjunto Σ^* . Dado um alfabeto Σ , o conjunto de todas as strings sobre Σ é definido como

$$\Sigma^* = \bigcup_{i \geq 0} \Sigma^i.$$

¹Uma definição indutiva é a mesma coisa que uma definição recursiva. Em teoria da computação será bastante comum usar raciocínio indutivo. Este tipo de raciocínio será útil não apenas em demonstrações de teoremas, mas também em várias definições.

2.1.1 Exercícios

Exercício 2.1 Seja $\Sigma = \{0, 1\}$, $x \in \Sigma^*$ e $y \in \{a, b, c\}^*$. Responda verdadeiro ou falso.

- (a) Se w é uma string da forma $x01$, então $|w| > 2$?
- (b) Se w é uma string da forma $x010$, então $|w| \geq 3$?
- (c) Se w é uma string da forma $y010$, então w é uma string sobre Σ ?

Exercício 2.2 Seja Σ um alfabeto e i um número natural. Qual é a cardinalidade de Σ^i ? ■

Exercício 2.3 Nesta seção, fornecemos uma definição informal para a concatenação de uma string x com ela mesma n vezes, denotada x^n . Forneça uma definição formal induativa para x^n . ■

Exercício 2.4 Apresente uma definição formal recursiva para Σ^* diferente da Definição 2.1.8. ■

Exercício 2.5 É verdade que $x^0 = y^0$ para qualquer par de strings x e y ? ■

Exercício 2.6 Seja Σ um alfabeto qualquer, i um número natural e $w \in \Sigma^*$. Mostre que $(w^i)^R = (w^R)^i$? ■

2.2 Linguagens

Na seção anterior começamos com elementos fundamentais chamados símbolos e os usamos para construir objetos complexos chamados de strings. Agora vamos usar strings como elementos fundamentais para construir objetos ainda mais complexos chamados de *linguagens*.

Definição 2.2.1 — Linguagem. Seja Σ um alfabeto. Uma linguagem é um conjunto $L \subseteq \Sigma^*$.

Em outras palavras, dado um alfabeto, uma linguagem é um conjunto qualquer de strings sobre este alfabeto. Por exemplo, $L = \{1010, 11111, 0000001\}$ é uma linguagem sobre o alfabeto binário. Um exemplo de linguagem sobre o alfabeto Σ_{ANSI} é $A = \{a, aa, aaa, aaa, aaaa, \dots\}$, ou seja, o conjunto de strings da forma a^i , para $i \geq 1$.

Observe que strings e linguagens são objetos matemáticos diferentes. Enquanto strings são *sequências*, linguagens são *conjuntos*. Enquanto strings tem tamanho *finito*, linguagens podem ter tamanho *finito ou infinito*.

Considere o alfabeto binário $\Sigma = \{0, 1\}$. Nos Exemplos 2.8, 2.9, 2.10 e 2.11 apresentamos algumas linguagens binárias que serão bastante comuns neste curso:

■ **Exemplo 2.8** $L_{01} = \{\epsilon, 01, 0011, 000111, \dots\} = \{w \in \Sigma^* : w \text{ é da forma } 0^n 1^n, \text{ para } n \geq 0\}$. ■

■ **Exemplo 2.9** $L_{PAL} = \{\epsilon, 0, 1, 00, 11, 010, 101, 000, 111, \dots\} = \{w \in \Sigma^* : w \text{ é um palíndromo}\}$. ■

■ **Exemplo 2.10** $L_p = \{10, 11, 101, 111, 1011, \dots\} = \{w \in \Sigma^* : w \text{ é um número primo escrito em binário}\}$. ■

■ **Exemplo 2.11** $L_{SQ} = \{0, 1, 100, 1001, 10000, \dots\} = \{w \in \Sigma^* : w \text{ é um número quadrado perfeito escrito em binário}\}$. ■

LINGUAGENS NATURAIS E LINGUAGENS FORMAIS

Observe que chamamos conjuntos de strings de linguagens. A razão disso é que há uma certa analogia com o conceito que intuitivamente conhecemos como linguagem, usada naturalmente em nosso dia a dia. Vamos a alguns exemplos que nos ajudarão a tornar isso mais claro.

■ **Exemplo 2.12** Seja $\Sigma_p = \{a, b, c, \dots, z\}$ um alfabeto (i.e., o alfabeto da língua portuguesa). ■

Considere agora o conjunto L contendo todas as palavras da língua portuguesa. Note que este conjunto, por se tratar de um conjunto de strings sobre Σ_p , de acordo com Definição 2.2.1 é formalmente uma linguagem (por questão de simplicidade, estamos ignorando aqui acentos, hífens e outras sutilezas). Poderíamos chamar este conjunto de strings de “língua portuguesa”. Por exemplo, observe que $bola \in L$, $caneta \in L$, $inconstitucional \in L$.

■ **Exemplo 2.13** Seja $\Sigma_p = \{a, b, c, \dots, z, \sqcup\}$. ■

Ao incluirmos o símbolo “ \sqcup ” em nosso alfabeto, temos algo para ser usado de separador de palavras. Com isso, poderíamos montar um frase como $a \sqcup bola \sqcup verde \sqcup escura$. De maneira simplificada, novamente, poderíamos pensar na língua portuguesa como sendo o conjunto L' de todas as strings que correspondem a frases válidas em português.

Neste ponto, um linguista nos chamaria atenção (com razão!) para o fato que tentar definir matematicamente a língua portuguesa não é uma tarefa tão fácil e argumentaria que o conteúdo das linguagens L e L' que acabamos de ver não é tão bem definido como estamos sugerindo. Poderíamos até tentar contra-argumentar dizendo que, pelo menos no caso de L , seria possível sermos formais definindo a linguagem como sendo o conjunto de todas as palavras de algum dicionário específico fixado a priori (já para o caso de fornecer uma definição exata para L' a tarefa seria bem mais difícil).

Entretanto, o nosso foco neste livro não são linguagens naturais como o português ou qualquer outra língua natural, sendo que os exemplos vistos acima são úteis apenas para ilustrar a analogia que há entre o conceito matemático de linguagens e o conceito intuitivo de linguagens naturais. Ainda assim, existem muitas linguagens que nós, profissionais de computação, lidamos no dia a dia e que são completamente formais. Voltando à um exemplo que vimos anteriormente, considere o alfabeto Σ_C da Seção 2.1. A linguagem sobre Σ_C , definida abaixo é matematicamente precisa:

$$L_C = \{w \in \Sigma_C : "w \text{ é um programa válido em linguagem C}"\}$$

A linguagem L_C , definida acima, consiste de todos os (infinitos) possíveis programas válidos escritos em linguagem C. Embora tenhamos escrito textualmente a expressão informal “ w é um programa válido em linguagem C” na nossa especificação das strings contidas no conjunto L_C , esta linguagem pode, sim, ser matematicamente bem definida. Para tal precisamos de algumas ferramentas matemáticas, especialmente um conceito matemático conhecido como *gramática formal*. Na Seção 4.3 veremos formalmente o que é uma gramática formal e discutiremos brevemente algumas aplicações disto na especificação de linguagens de programação e na construção de compiladores.

Em diversas situações, nós queremos interpretar strings binárias não apenas como meras sequências de 0's e 1's, mas como números naturais representados em base binária. Em tais situações a seguinte notação será conveniente.

Notação 2.3. *O número natural representado pela string binária w é denotado por $N(w)$. Por convenção $N(\varepsilon) = 0$.*

Por exemplo, se $w_1 = 101$ e $w_2 = 1111$, então $N(w_1) = 5$ e $N(w_2) = 15$. Isso torna nossa notação mais concisa na definição de algumas linguagens. Por exemplo, a linguagem dos números primos e a linguagens dos números múltiplos de 3 podem ser denotadas, respectivamente, por $L_p = \{w : N(w) \text{ é um número primo}\}$ e $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$.

REFLETINDO UM POUCO: PROBLEMAS COMPUTACIONAIS E LINGUAGENS

O que significa “*o problema de testar se um número n é primo*”? Uma maneira de tentar formalizar isso é pensar que isso é equivalente ao seguinte problema: dada uma string w , decidir se w é a representação binária de um número primo, ou seja, testar se a string w pertence a linguagem L_p do Exemplo 2.10.

Para aqueles que não gostam de números binários, poderíamos alternativamente definir a linguagem dos números primos usando outros alfabetos (e.g., o alfabeto dos dígitos de 0 a 9), mas isso realmente não é tão importante aqui. O ponto central é que uma vez que fixamos um alfabeto, digamos o alfabeto binário, a linguagem L_p incorpora a propriedade “ser primo”, pois todos os objetos contidos neste conjunto tem tal propriedade e todos os objetos que não fazem parte deste conjunto não tem tal propriedade. Portanto, responder se um número é primo se reduz a distinguir se uma dada string pertence ou não ao conjunto L_p .

Em teoria da computação é bastante comum pensamos em linguagens como sinônimos de problemas computacionais. Embora esta ideia pareça simples, ela também é bastante poderosa. Existem outras maneiras de tornar formal o conceito de problema computacional e, no decorrer deste livro, veremos outras definições mais sofisticadas para tal. Entretanto, em boa parte do curso o uso de linguagens será o formalismo padrão para representar problemas.

Para quem ainda não está convencido de que existe tal conexão direta entre o conceito de linguagem e o conceito de problema, uma maneira intuitiva de se pensar a respeito disso segue na seguinte linha: No funcionamento interno de um computador, qualquer objeto matemático, em última análise, é uma sequência de bits. No caso particular do exemplo acima, quando escrevemos um programa para testar se um dado número é primo, o que o programa faz é testar se uma sequência de bits da memória do computador representa ou não representa um número primo em binário. Ou seja, no final das contas, o que o programa está fazendo é testar se uma string do alfabeto binário pertence ou não à linguagem L_p . Não é muito difícil de ver que esse tipo de raciocínio pode ser generalizado para uma série de outros problemas. De fato, qualquer problema cuja solução envolva testar se um dado objeto matemático tem uma certa propriedade pode ser modelado usando uma linguagem.

2.2.1 Exercícios

Exercício 2.7 Seja Σ um alfabeto arbitrário. Com relação a linguagens sobre Σ , responda:

- (a) Σ^* é uma linguagem?
- (b) \emptyset é uma linguagem?
- (c) $\{\varepsilon\}$ é uma linguagem?
- (d) O conjunto potência Σ^4 é uma linguagem?

Exercício 2.8 Forneça uma definição formal para a linguagem sobre $\Sigma = \{0, 1\}$, das strings que representam números múltiplos de 4 escritos em binário.

Exercício 2.9 Seja Σ um alfabeto qualquer. É verdade que a linguagem $\{w : \exists x \in \Sigma^* \text{ tal que } w = xx^R\}$ é a linguagem de todos os palíndromos construídos com símbolos de Σ ? Em caso afirmativo, prove. Em caso negativo, forneça um contra-exemplo e, em seguida, forneça uma definição formal adequada para a linguagem dos palíndromos construídos com símbolos de Σ .

2.2.2 Operações com linguagens

Assim como podemos criar strings maiores a partir da operação de concatenação de strings menores, vamos definir agora uma operação análoga para linguagens, chamada de concatenação de linguagens.

Definição 2.2.2 A concatenação de duas linguagens L e M é o conjunto de todas as strings que podem ser formadas tomando-se uma string x de L e uma string y de M e fazendo a concatenação xy . Denotamos a linguagem resultante por LM ou $L \cdot M$.

■ **Exemplo 2.14** Considere as linguagens $L = \{00, 0\}$ e $L' = \{1, 111\}$. A concatenação destas duas linguagens é $LL' = \{001, 00111, 01, 0111\}$.

■ **Exemplo 2.15** Seja $L = \{a, aa, aaa, \dots\}$ e $M = \{\varepsilon, x, xx, xxx, \dots\}$. A linguagem LM é $\{a, ax, axx, axxx, \dots, aa, aax, aaxx, \dots, aaxxx, \dots\}$.

Exercício resolvido 2.1 Seja Σ o alfabeto binário e L_3 e R as linguagens sobre Σ definidas a seguir: $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$ e $R = \{0\}$. Qual é a linguagem L_3R ?

Solução: As strings contidas na linguagem da concatenação L_3R são as strings de L_3 adicionadas de um 0 ao final. Note que quando adicionarmos 0 ao final de um número binário o resultado é um novo número binário que é dobro do número original. Portanto a linguagem resultante é: $L_3R = \{w : N(w) \text{ é um múltiplo de } 6\}$.

Exercício 2.10 Seja Σ o alfabeto binário e L_3 e R as linguagens sobre Σ definidas a seguir: $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$ e $R = \{\varepsilon, 0\}$. Qual é a linguagem L_3R ?

Exercício resolvido 2.2 Seja Σ o alfabeto binário. Seja $R = \{0\}$ uma linguagem sobre Σ . Qual é a linguagem Σ^*R ? E qual é a linguagem Σ^*RR ?

Solução: Lembrando que as strings binárias terminadas em 0 são precisamente os múltiplos de 2 em binário e a multiplicarmos múltiplos de 2, temos múltiplos de 4, temos que:

- $\Sigma^*R = \{w : N(w) \text{ é um múltiplo de } 2\}$.
- $\Sigma^*RR = \{w : N(w) \text{ é um múltiplo de } 4\}$.

A operação de concatenação de uma linguagem com ela mesma pode ser aplicada um número arbitrário de vezes. A definição a seguir formaliza isso:

Definição 2.2.3 Seja L uma linguagem. O Fechamento (ou Fecho de Kleene) de L , denotado por L^* , é o conjunto de strings que podem ser formadas tomando-se qualquer número de strings de L (possivelmente com repetições) e as concatenando. Isto é,

$$L^* = \bigcup_{i \geq 0} L^i, \text{ sendo que } L^0 = \{\varepsilon\} \text{ e } L^i = \underbrace{LL\dots L}_{i \text{ vezes}}$$

■ **Exemplo 2.16** Seja $L = \{000, 111\}$. Então $L^* = \{\varepsilon, 000, 111, 000000, 000111, 111000, 111111, 000000000, 000000111, 000111000, \dots\}$. ■

Observe que ao escrevermos A^* , devemos especificar exatamente o que significa A . Se A é um alfabeto, então A^* é a união infinita de conjuntos potência de A . Por outro lado se A é uma linguagem, então A^* é o fecho de A , ou seja, a sequência infinita de concatenações de A consigo mesma. Entretanto, em ambos os casos, note que A^* é uma linguagem. Mais precisamente, a linguagem de todas as strings construídas usando os elementos do conjunto A como “tijolos básicos”. Caso A seja um alfabeto, estes tijolos básicos são símbolos, caso A seja uma linguagem, estes tijolos básicos são strings.

Exercício resolvido 2.3 Considere as linguagens $L = \{1\}$ e $R = \{0\}$. Qual é a linguagem LR^* ?

Solução: A linguagem é $LR^* = \{w : N(w) \text{ é uma potência de } 2\}$

Exercício 2.11 Nas questões abaixo, A é sempre um alfabeto e L é sempre uma linguagem. Responda verdadeiro ou falso e justifique sua resposta:

- Para todo A , é verdade que $A = A^*$?
- Existe A , tal que $A = A^*$?
- Para todo A , é verdade que $A^* = (A^*)^*$?
- Para todo L , é verdade que $L^* = (L^*)^*$?
- Para todo L e A tal que L é uma linguagem sobre A , é verdade que $A^* = L^*$?
- Existe uma linguagem L sobre A tal que $A^* = L^*$?

3. Autômatos e Linguagens Regulares

Um dos objetivos que queremos alcançar neste livro é prover uma definição formal para o conceito de algoritmo. Nós alcançaremos este objetivo no Capítulo 5, com a definição das Máquinas de Turing. O que veremos agora é a definição de *autômatos finitos*, que é um modelo matemático com poder de expressão menor do que as Máquinas de Turing. Por poder de expressão menor, queremos dizer que apenas um subconjunto de todos os possíveis algoritmos podem ser representados por autômatos finitos. Entretanto, estudar este modelo será útil para nos familiarizarmos com os conceitos matemáticos que serão muito utilizados no Capítulo 5. Além disso, por si só, o assunto que veremos agora é bastante útil em aplicações práticas, como busca de padrões em textos e construção de analisadores léxicos para compiladores.

3.1 Autômatos Finitos Determinísticos (AFDs)

Considere uma máquina que vende chocolates que custam 6 reais. Abaixo mostramos uma figura ilustrando tal máquina juntamente com um diagrama que descreve o mecanismo de funcionamento interno dela.

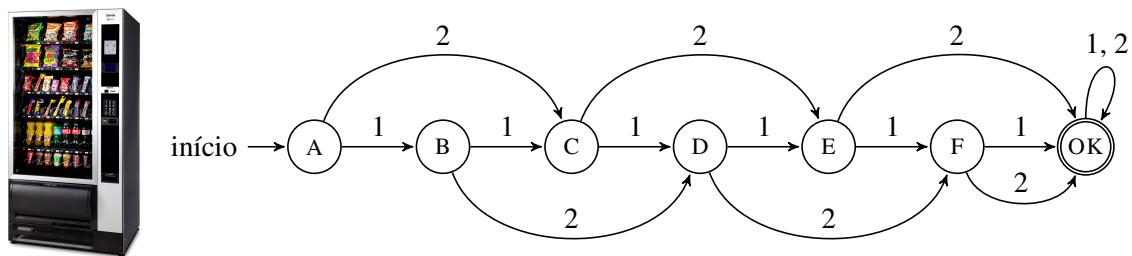


Figure 3.1: Uma máquina que vende chocolates de R\$ 6 e aceita moedas de R\$ 1 e R\$ 2. Ao lado o diagrama simplificado descrevendo o seu mecanismo de funcionamento interno. Por exemplo, se inserirmos uma moeda de R\$ 1 e uma moeda de R\$ 2, a máquina, atingirá o estado D. A máquina atingirá o estado OK se for inserida uma sequência de moedas cuja soma seja pelo menos R\$ 6.

Diagramas como o da Figura 3.1 são conhecidos como *máquinas de estados*, *máquinas de estados finitos* ou *autômatos finitos determinísticos*. A ideia é entender que a entrada da máquina é uma sequência de moedas (podemos também pensar que a entrada é uma “string de moedas”), que podem ser de 1 ou 2 reais. A máquina libera o chocolate se a sequência de moedas somar pelo menos 6 reais. O estado A é marcado com a indicação *start* por que este é o estado inicial da máquina. A partir do estado A, a máquina faz uma transição para cada moeda que é fornecida como entrada. A máquina libera o chocolate ao final do processo se, e somente se, ela atingir o *estado OK*. Note que estamos em um cenário simplificado em que a máquina não precisa fornecer troco.

REFLETINDO UM POUCO: MODELOS MATEMÁTICOS E OBJETOS FÍSICOS

No Capítulo 1 vimos que podemos pensar em computadores como “instanciações de objetos abstratos e seus possíveis relacionamentos matemáticos em objetos físicos e seu conjunto de possíveis graus de movimento”. Note que a Figura 3.1 apresenta exatamente isso, pois:

- (1) A máquina de chocolates é um objeto físico (embora devemos estar atentos que ela não é um computador de propósito geral, pois trata-se de uma máquina que implementa um único algoritmo: o algoritmo que testa se a soma das moedas de entrada é maior ou igual a 6 reais).
- (2) O diagrama faz o papel do objeto matemático abstrato^a, em particular, um objeto matemático que consiste de um conjunto de estados e um conjunto de símbolos (o conjunto {1, 2} dos símbolos que aparecem nos rótulos das transições). O diagrama também representa certas relações matemáticas, como os relacionamentos entre estados e símbolos indicados pelas setas saindo de um estado e indo para outro. A definição exata destas relações ficará clara no decorrer desta seção.

O ponto central aqui é que *o objeto matemático e suas relações matemáticas estão “amaradas” em uma correspondência de um para um com o objeto físico e seus graus de movimento*. Os graus de liberdade de movimento são os tipos de movimentações que os mecanismos internos da máquina de vender chocolate tem.

^aO leitor mais atento aqui pode fazer objeção a isso, pois um desenho ou um diagrama não é, estritamente falando, um objeto matemático. Mas, por hora, vamos ignorar isso.

3.1.1 Modelando matematicamente autômatos

O diagrama da seção anterior foi apresentado sem maiores detalhes. Veremos agora uma definição precisa que corresponde a tais diagramas.

Definição 3.1.1 — Autômato Finito Determinístico (AFD). Uma Autômato Finito Determinístico, também chamado de AFN, é uma 5-tupla $D = (Q, \Sigma, \delta, q_0, F)$, tal que:

- Q é o conjunto de *estados*;
- Σ é o conjunto de *símbolos de entrada*;
- δ é a *função de transição* $\delta : Q \times \Sigma \rightarrow Q$;
- $q_0 \in Q$ é o *estado inicial*;
- $F \subseteq Q$ é o conjunto de *estados finais*.

Normalmente, diremos simplesmente *autômatos* ou usaremos simplesmente a sigla *AFD* para se referir aos autômatos finitos determinísticos¹. Algo importante de se observar é que esta definição é genérica, ou seja, se quisermos uma definição matemática particular para o diagrama da nossa

¹Algumas vezes usa-se o acrônimo *DFA*, que vem da expressão em inglês “*deterministic finite automata*.”

máquina de chocolates, teríamos que dar uma *instância específica da Definição 3.1.1*. O exemplo a seguir ilustra o que queremos dizer com isso.

■ **Exemplo 3.1 — Máquina de vender chocolates.** Uma definição matemática para o exemplo da Figura 3.1 é o autômato finito determinístico $C = (Q_c, \Sigma_c, \delta_c, A, F_c)$, sendo que os componentes da 5-tupla são:

$$Q = \{A, B, C, D, E, F, OK\};$$

$$\Sigma = \{1, 2\};$$

$\delta_c : Q \times \Sigma \rightarrow Q$ é a função definida caso a caso a seguir:

$$\begin{aligned} \delta_c(A, 1) &= B, \quad \delta_c(A, 2) = C, \quad \delta_c(B, 1) = C, \quad \delta_c(B, 2) = D, \quad \delta_c(C, 1) = D, \quad \delta_c(C, 2) = E, \\ \delta_c(D, 1) &= E, \quad \delta_c(D, 2) = F, \quad \delta_c(E, 1) = F, \quad \delta_c(E, 2) = OK, \quad \delta_c(F, 1) = OK, \quad \delta_c(F, 2) = OK, \\ \delta_c(OK, 1) &= OK, \quad \delta_c(OK, 2) = OK. \end{aligned}$$

O estado A é o estado inicial;

O conjunto unitário $\{OK\}$ é o conjunto de estados finais. ■

Neste momento pode ser instrutivo lembrar da analogia que fizemos no Capítulo 1, quando discutimos definições genéricas e definições específicas. Naquela ocasião mencionamos a definição matemática genérica de “função” e as definições matemáticas de funções particulares, como x^2 ou $\log x$. Na definição 3.1.1 temos a definição para um AFD em geral. Por outro lado, no exemplo 3.1, a 5-tupla $C = (Q_c, \Sigma_c, \delta_c, A, F_c)$ é uma definição matemática do AFD especificamente correspondente a máquina de vender chocolates.

Exercício resolvido 3.1 Desenhe o diagrama do autômato A definido a seguir:

$A = (Q, \Sigma, \delta, p, \{r\})$, tal que $Q = \{p, q, r\}$, $\Sigma = \{0, 1\}$ e a função δ é definida abaixo:

$$\delta(p, 1) = p$$

$$\delta(p, 0) = q$$

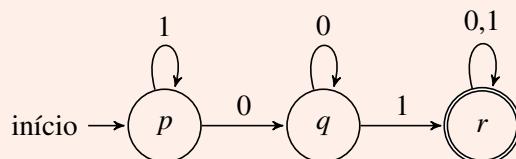
$$\delta(q, 0) = q$$

$$\delta(q, 1) = r$$

$$\delta(r, 0) = r$$

$$\delta(r, 1) = r$$

Solução:



A função δ do enunciado do Exercício 3.1 é um tipo de função que é definida ponto a ponto (ou seja, ela não tem uma regra geral que descreve o comportamento da função, como é o caso de funções que estamos acostumados a estudar em cálculo, como, por exemplo, $f(x) = x^2$). Em teoria da computação, funções que não podem ser facilmente descritas por regras gerais serão bastante comuns. Para facilitar a nossa vida, vamos descrever δ usando uma tabela, como esta apresentada no exemplo a seguir:

	0	1
$\rightarrow p$	q	p
q	q	r
$*r$	r	r

Table 3.1: Tabela de transições da função δ do AFD do Exercício 3.1.

Na primeira coluna da Tabela 3.1.1, em negrito, temos os estados p, q, r do autômato. A seta ao lado do estado p indica que ele é o estado inicial e o asterisco ao lado do estado r indica que ele é um estado final. No topo da tabela, também em negrito, temos os símbolos do alfabeto, ou seja, 0 e 1. Preenchemos a posição correspondente a linha do estado p e a coluna do símbolo 0 com o valor q , pois $\delta(p, 0) = q$. As demais linhas da tabela são preenchidas de maneira semelhante.

No decorrer deste livro, ao nos referirmos à um AFD, estritamente falando, estaremos nos referindo à uma 5-tupla. Porém, por questões de conveniência, será bastante comum apresentarmos a tabela de transições ou mesmo o diagrama, para descrever o AFD em questão.

Exercício 3.1 Projete uma máquina que venda chocolates que custem 2 reais. Ela deve aceitar moedas de 10 e 50 centavos e moedas de 1 real. Você deve tanto desenhar o diagrama quanto apresentar a definição formal. ■

3.1.2 Aceitação e rejeição de strings

Agora que nós já temos o nosso modelo matemático para autômatos finitos, vamos interpretá-lo. Mas o que queremos dizer com “interpretá-lo”? O que vamos fazer é entender como este modelo matemático pode ser visto como um *modelo de computação* e não uma mera 5-tupla.

A ideia central é enxergar o AFD como um objeto que receba uma string $w = w_1w_2\dots w_n$ como entrada, processe um a um os símbolos de w , e produza uma saída. Durante o processamento de w , cada passo consiste em ler um símbolo w_i , fazer uma mudança de estado e descartar w_i . O AFD irá, passo a passo, descartar símbolos de w e irá também mudando de estados neste processo. O ponto central é que podemos fazer a seguinte pergunta:

- Em que estado o AFD se encontra após o descarte do último símbolo de w ?

Digamos que depois da última transição de estados o AFD atinja um dado estado q . A ideia aqui é que se o estado q for um estado final, nós iremos dizer que o AFD *aceitou* a string w . Por outro lado, se q não for um estado final, nós iremos dizer que o AFD *rejeitou* a string w . Os estados finais também são chamados de estados de aceitação do autômato.

Exercício resolvido 3.2 Seja $\Sigma = \{0, 1\}$. Considere a linguagem binária L das strings que contém a substring 01, ou seja, $L = \{w : w \text{ é da forma } x01y, \text{ sendo que } x, y \in \Sigma^*\}$. Construa um AFD que aceite todas e somente as strings de L .

Solução: Podemos usar o mesmo AFD usado no Exercício 3.1. ■

Exercício 3.2 Seja $\Sigma = \{a, b, c\}$. Seja $L = \{w \in \Sigma^* \mid w \text{ é da forma } xbbya, \text{ sendo que } x, y \in \Sigma^*\}$. Construa um AFD que aceite a string w fornecida como entrada se, e somente se, $w \in L$. ■

Exercício 3.3 Seja $\Sigma = \{0, 1\}$. Construa AFDs que aceite a string fornecida como entrada se, e somente se, a string pertence a linguagem L , para cada um dos casos abaixo:

- $L = \{w : w \text{ tem ao mesmo tempo um número ímpar de } 0's\}$.
- $L = \{w : |w| \leq 3\}$.
- $L = \{w : w \text{ tem ao mesmo tempo um número par de } 0's \text{ e um número par de } 1's\}$.
- $L = \{w : \text{se } w' \text{ é uma substring de } w \text{ com } |w'| = 5, \text{ então } w' \text{ tem pelo menos dois } 0's\}$.
- $L = \{w : w \text{ é terminada em } 00\}$.
- $L = \{w : \text{o número de } 0's \text{ em } w \text{ é divisível por } 3 \text{ e o número de } 1's \text{ é divisível por } 5\}$.
- $L = \{w : w \text{ contém a substring } 011\}$.
- $L = \Sigma^*$

O que precisamos fazer agora é transformar esta ideia intuitiva de que um AFD aceita algumas strings e rejeita outras strings em definições matemáticas precisas. Ou seja, queremos saber matematicamente o que significa um certo AFD $D = (Q, \Sigma, \delta, q_0, F)$ aceitar ou rejeitar uma dada string w .

3.1.3 Definição formal para aceitação e rejeição de strings

Voltando a nossa máquina de vender chocolates, considere o seguinte: Se a máquina estiver no estado B e receber 3 moedas de 1 real, em que estado a máquina vai parar? A resposta é que a máquina irá atingir o estado E . O mais importante aqui é observar a *estrutura* da pergunta que fizemos: *dado um estado e uma sequência de moedas, qual é o estado resultante?* O que vamos fazer agora é formalizar esta idéia de que dado um estado q e uma sequência de símbolos w , obtemos o estado resultante é q' . O objeto matemático para modelar esta ideia é uma função $f : Q \times \Sigma^* \rightarrow Q$ de forma que $f(q, w) = q'$. Uma vez a definição de tal função depende da função δ original da máquina, vamos chamar esta nova função de $\hat{\delta}$.

Definição 3.1.2 — Função de transição estendida. Dado um AFD $D = (Q, \Sigma, \delta, q_0, F)$, a função de transição estendida $\hat{\delta}$ de D é a função $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ definida indutivamente:

Caso base: $w = \epsilon$.

$$\hat{\delta}(q, \epsilon) = q$$

Indução: $|w| > 0$.

Seja w uma string da forma $w = xa$, sendo que $x \in \Sigma^*$ e $a \in \Sigma$. Então $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.

A razão de termos apresentado a definição 3.1.2 é que agora temos uma maneira precisa de dizer o que significa um AFD D aceitar ou rejeitar uma string. A maneira precisa é a seguinte:

Definição 3.1.3 — Aceitação e rejeição de strings. Seja $D = (Q, \Sigma, \delta, q_0, F)$ um AFD e $w \in \Sigma^*$. Se $\hat{\delta}(q_0, w) \in F$, então dizemos que D aceita a string w . Se $\hat{\delta}(q_0, w) \notin F$, então dizemos que D rejeita a string w .

Podemos generalizar a ideia de aceitação de string para a ideia de aceitação de uma linguagem. Se L é o conjunto de todas as strings aceitas por D , então dizemos que D aceita a linguagem L . Neste caso dizemos que L é a linguagem de D . Isto é definido formalmente a seguir.

Definição 3.1.4 — Linguagem de um AFD. A linguagem de um AFD D , denotada por $L(D)$, é definida como $L(D) = \{w : \hat{\delta}(q_0, w) \in F\}$.

Se L é a linguagem de um AFD D , dizemos que D *decide* a linguagem L . Dizemos também que D *aceita* a linguagem L . Agora segue a definição mais importante deste capítulo:

Definição 3.1.5 — Liguagem Regular. Dada uma linguagem L , se existe um AFD D tal que $L = L(D)$, então L é dita uma *linguagem regular*.

3.1.4 Exercícios

Exercício 3.4 Seja um AFD com alfabeto Σ e conjunto de estados Q . Sejam $x, y \in \Sigma^*$, $a \in \Sigma$ e $q \in Q$ quaisquer.

- (a) Mostre que $\hat{\delta}(q, xy) = \hat{\delta}(\hat{\delta}(q, x), y)$.
- (b) Mostre que $\hat{\delta}(q, ax) = \hat{\delta}(\delta(q, a), x)$.

Exercício 3.5 Considere o seguinte AFD $D = (Q, \Sigma, \delta, q_0, F)$ tal que

- $Q = \{q_0, q_1, q_2, \dots, q_6\}$
- $\Sigma = \{0, 1, 2, \dots, 6\}$
- $F = \{q_0\}$
- Função de transição δ definida a seguir:
 $\delta(q_j, i) = q_k$, sendo que $k = (j + i) \bmod 7$

Qual é a linguagem aceita por D ?

Exercício 3.6 Forneça um AFD com alfabeto $\Sigma = \{0, 1\}$ que aceite a seguinte linguagem:

$$L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}.$$

Em outras palavras, o AFD deve aceitar strings como 0, 11, 110, 1001, etc. Para simplificar vamos assumir que a string ε representa o número zero (ou seja, $N(\varepsilon) = 0$).

Exercício 3.7 Prove formalmente que a sua solução para o Exercício 3.6 é correta.

Exercício 3.8 Seja uma linguagem L sobre o alfabeto Σ . O complemento da linguagem L , denotado por \bar{L} , é definido da seguinte maneira: $\bar{L} = \Sigma^* \setminus L$. Prove que se L é regular, então \bar{L} também é regular.

Exercício 3.9 Considere os seguintes AFDs: $D_Q = (Q, \Sigma, \delta_Q, q_0, F_Q)$ e $D_P = (P, \Sigma, \delta_P, p_0, F_P)$ sendo $Q = \{q_0, q_1, \dots, q_h\}$ e $P = \{p_0, p_1, \dots, p_h\}$.

Construiremos agora um terceiro AFD D_A a partir dos dois AFDs anteriores. A ideia é que dada qualquer string w , o AFD D_A vai simular ao mesmo tempo a computação de D_Q com w e D_P com w . Por exemplo, se na terceira transição D_Q estiver no estado q_1 e D_P estiver no estado p_5 , então na terceira transição o AFD D_A estará em um estado chamado (q_1, p_5) . A definição formal dos componentes de $D_A = (A, \Sigma, \delta_A, a_0, F_A)$ segue abaixo:

- $A = Q \times P$ (ou seja, para cada $q_i \in Q$ e $p_j \in P$, temos um estado $(q_i, p_j) \in A$)
- $a_0 = (q_0, p_0)$
- $F_A = \{\text{“conjunto de todos os elementos } (q_i, p_j) \text{ tal que } q_i \in F_Q \text{ e } p_j \in F_P\”\}$.
- Definição de δ_A : Para todo $q_i, q_j \in Q$ e todo $p_r, p_t \in P$ e todo símbolo $s \in \Sigma$ temos que:
 Se $\delta_Q(q_i, s) = q_j$ e $\delta_P(p_r, s) = p_t$, então $\delta_A((q_i, p_r), s) = (q_j, p_t)$.

Sendo $L_Q = L(D_Q)$ e $L_P = L(D_P)$, responda: Qual é a linguagem aceita por D_A ?

3.2 Autômatos Finitos não Determinísticos (AFNs)

A computação com AFDs é completamente determinística, ou seja, para cada par (q, a) , sendo q um estado e a um símbolo, temos exatamente uma transição definida no ponto (q, a) . E se quiséssemos definir um modelo abstrato de autômato que em certos momentos possa escolher uma entre várias transições possíveis de maneira não determinística? E se este autômato tivesse uma habilidade “mágica” de advinhar qual é a transição correta a ser executada no momento? Um exemplo de um diagrama de um autômato com as propriedades que queremos é o seguinte:

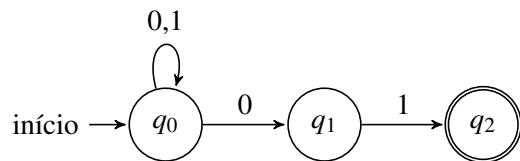


Figure 3.2: Um autômato finito não determinístico.

Imediatamente notamos uma diferença no autômato da Figura 3.2 em relação aos autômatos da seção anterior. Neste exemplo, se o autômato estiver no estado q_0 e o símbolo lido for 0, existem duas possibilidades: (1) o autômato pode continuar no estado q_0 ; (2) o autômato pode mudar para o estado q_1 . A pergunta chave é a seguinte: dado um autômato com estas características, quais são as strings que o autômato aceita?

Para responder esta pergunta, vamos agora ser mais precisos sobre o que queremos dizer com “o autômato advinha” qual transição fazer. A ideia é que, dada uma string w , se existe uma sequência de passos que leve o autômato a atingir um estado final ao finalizar o processamento de w , então o autômato irá escolher, a cada momento, uma transição que leve a computação em um caminho correto. Em tais casos, dizemos que string pertence à linguagem do autômato.

Mesmo com esta habilidade nova, pode ocorrer que não exista nenhuma sequência de transições que leve o autômato a aceitar certas strings. Por exemplo, o autômato da Figura 3.2 não “consegue” aceitar a string 100. Neste caso diremos que tais strings não estão na linguagem do autômato.

Se prestarmos atenção no autômato da Figura 3.2, veremos que ele tem a habilidade de aceitar exatamente as strings terminadas em 01. Dada uma string da forma $x01$, tal que x é uma substring qualquer, o autômato pode processar toda a substring x usando o “loop” sobre o estado q_0 . Quando faltarem apenas os dois símbolos finais, o autômato utiliza a transição que vai ao estado q_1 e depois a transição que vai ao estado q_2 . Observe que se a string não é terminada em 01, o autômato não consegue atingir o estado final em hipótese alguma.

Observe que a habilidade do autômato poder escolher entre duas transições possíveis não é a única diferença que este novo modelo tem em relação aos autômatos determinísticos da seção anterior. Uma outra situação que pode ocorrer neste modelo e que não ocorria no caso determinístico é aquela em que o autômato não tenha nenhuma transição definida para um determinado par (estado, símbolo). Por exemplo, se o autômato da Figura 3.2 estiver no estado q_1 e próximo símbolo a ser lido for 0, o autômato não terá nenhuma opção de transição para realizar. Em tal caso, diremos que o autômato *morre*.

NÃO DETERMINISMO?

Claramente, do ponto de vista prático, um autômato não determinístico não parece ser um modelo realista de computação correspondendo a algo concreto do mundo real. Ainda assim, o estudo deste modelo matemático será bastante útil.

Em teoria da computação este tipo de situação é razoavelmente comum. O ponto chave é que estes modelos podem ser pensados, em última análise, como *ferramentas matemáticas úteis* no estudo de modelos concretos de computação. No Capítulo 8, veremos que Máquinas de Turing não determinísticas podem ser usadas para definir um conjunto de linguagem conhecido como *NP*, que é parte do famoso (e concreto) problema *P vs NP*. Em um curso mais aprofundado de complexidade computacional a definição de modelos “irreais” de computação, mas que ainda assim sejam matematicamente úteis para lidar com problemas ou modelos concretos de computação, acontece com muita frequência.

3.2.1 Definição formal para autômatos finitos não determinísticos

Autômatos finitos não determinísticos têm uma definição formal muito parecida com a definição dos AFDs. A única diferença é que dado um par (q, a) , sendo q um elemento do conjunto Q de estados do autômato e a um símbolo, pode ser que exista zero, um, ou mais estados possíveis de serem atingidos por uma transição com rótulo a . Mais precisamente, a função de transição tem a forma $\delta(q, a) = S$, sendo que S é um conjunto qualquer de estados. Por exemplo, a função δ do autômato da Figura 3.2, quando aplicada a $(q_0, 0)$, deve ter a forma $\delta(q_0, 0) = \{q_0, q_1\}$. Note que, como os elementos da imagem de δ são conjuntos, o contradomínio da função δ é conjunto de todos os subconjuntos de Q , ou seja, o conjunto potência $\mathcal{P}(Q)$.

Definição 3.2.1 — Autômato Finito não Determinístico (AFN). Um Autômato Finito não Determinístico, também chamado de AFN, é uma 5-tupla $A = (Q, \Sigma, \delta, q_0, F)$, tal que:

- Q é o conjunto de *estados*;
- Σ é o conjunto de *símbolos de entrada*;
- δ é a *função de transição* $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q)$;
- $q_0 \in Q$ é o *estado inicial*;
- $F \subseteq Q$ é o conjunto de *estados finais*.

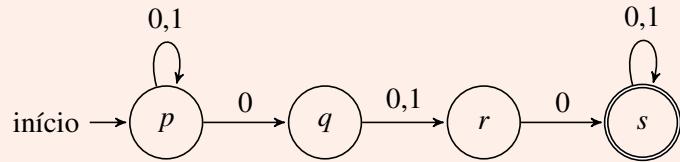
■ **Exemplo 3.2** A definição formal para o diagrama da Figura 3.2 é o AFN $N = (Q, \Sigma, \delta, q_0, F)$, tal que $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$ e a função δ é definida abaixo:

$$\begin{aligned}\delta(q_0, 0) &= \{q_0, q_1\} \\ \delta(q_0, 1) &= \{q_1\} \\ \delta(q_1, 1) &= \{q_2\} \\ \delta(q_1, 0) &= \delta(q_2, 0) = \delta(q_2, 1) = \emptyset\end{aligned}$$

Para simplificar, nós usaremos tanto 5-tuplas como tabelas de transições como definições formais para AFNs.

Exercício resolvido 3.3 Desenhe o diagrama do AFN definido pela seguinte tabela de transições:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	\emptyset
$*s$	$\{s\}$	$\{s\}$

Solução:

■

Uma maneira bastante útil para se descrever os ramos possíveis de computação de um AFN A com uma string w é a utilização de uma árvore, chamada de *árvore de computações possíveis de A com w* . Antes de definir este conceito, vamos apresentar a definição um pouco mais geral.

Definição 3.2.2 — (A, q, w) -árvores. Seja (A, q, w) uma tripla tal que A é um AFN, q um é estado de A e w um string do alfabeto de A . Uma (A, q, w) -árvore é uma árvore enraizada em q definida da seguinte maneira:

Base: $w = \epsilon$

A árvore contém apenas o nó raiz q

Indução: $w \neq \epsilon$

Suponha que w é uma string da forma ax , sendo que $a \in \Sigma$ e $x \in \Sigma^*$ e seja δ a função de transição do AFN. Se $\delta(q, a) = \{p_1, p_2, \dots, p_k\}$, então então a árvore contém o nó q e, além disso, q possui os filhos p_1, p_2, \dots, p_k que são raízes de uma (A, p_i, x) -árvore.

Definição 3.2.3 — Árvore de computações possíveis. Seja $A = (Q, \Sigma, \delta, q_0, F)$ um AFN, $w \in \Sigma^*$. Uma árvore de computações possíveis de A com w é uma (A, q_0, w) -árvore.

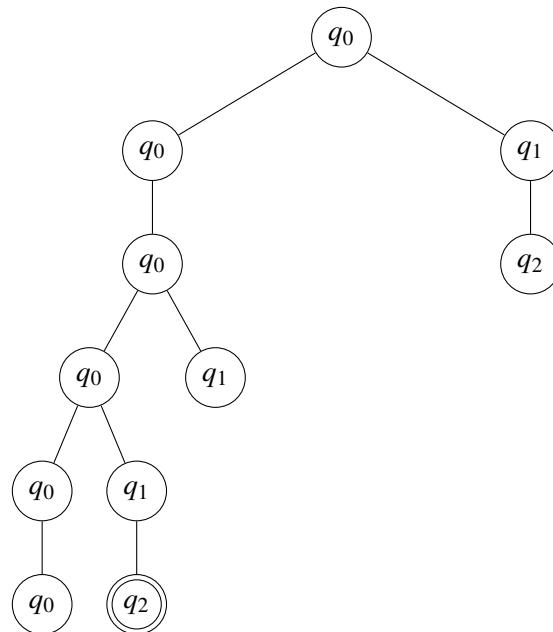


Figure 3.3: A árvore de computações possíveis do AFN N da Figura 3.2 com a string 01001.

Observe que o *nível 0* da árvore (i.e., a raiz) é o estado inicial do AFN e o nível i contém todos os estados que o AFN pode estar depois de i transições do autômato ao processar os i primeiros símbolos da string de entrada. Em particular, dada uma string de tamanho n , o AFN aceita a string se e somente se o n -ésimo nível da árvore contém o estado q_2 . O fato de que neste nível existe apenas um nó que é um estado final significa que o AFN tem exatamente uma computação possível que aceita a string 01001. A computação é definida pela sequência de estados do caminho ligando a raiz ao estado final em questão.

Exercício resolvido 3.4 Apresente uma definição formal para um AFN cujo diagrama seja idêntico ao diagrama do AFD obtido na solução do Exercício 3.1.

Solução: Relembramos que o AFD do Exercício 3.1 é definido pela seguinte tabela:

	0	1
$\rightarrow p$	q	p
q	q	r
$*r$	r	r

Para obtermos um AFN cujo diagrama seja idêntico ao diagrama do AFD definido pela tabela acima, basta fazer o seguinte: se no AFD a função de transição é $\delta(x,y) = z$, defina o AFN de maneira que sua função de transição seja $\delta(x,y) = \{z\}$. A definição formal do AFN é dada pela seguinte tabela:

	0	1
$\rightarrow p$	$\{q\}$	$\{p\}$
q	$\{q\}$	$\{r\}$
$*r$	$\{r\}$	$\{r\}$

A Figura 3.4 mostra o diagrama do AFN do exercício 3.4.

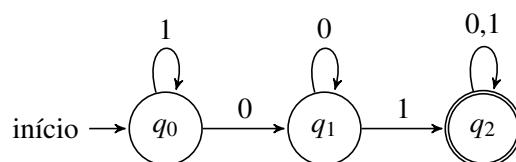


Figure 3.4: Um caso particular em que um AFN que se comporta de maneira determinística.

Exercício 3.10 Desenhe a árvore de computações possíveis para o AFN do Exercício 3.4 com a string de entrada 111010. ■

3.2.2 Aceitação e rejeição de strings por AFNs

A definição da função de transição estendida de um AFN é um pouco mais complicada que a definição que vimos no caso dos AFDs. A ideia básica é a seguinte: dado um estado q e uma string x , queremos saber qual é o conjunto de estados que o autômato pode estar após o processamento da string x se a computação começou no estado q .

Definição 3.2.4 Dado um AFN $N = (Q, \Sigma, \delta, q_0, F)$, definimos $\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$:

Base: $w = \varepsilon$.

$$\hat{\delta}(q, \varepsilon) = \{q\}$$

Indução: $|w| > 0$.

Seja $w \in \Sigma^*$ da forma xa , onde $x \in \Sigma^*$ e $a \in \Sigma$. Suponha $\hat{\delta}(q, x) = \{p_1, p_2, \dots, p_k\}$. Então

$$\hat{\delta}(q, w) = \bigcup_{i=1}^k \delta(p_i, a)$$

Exercício 3.11 Calcule $\hat{\delta}(q_0, 00101)$ do AFN do Exemplo 3.2. ■

Definição 3.2.5 — Linguagem de um AFN. Se $N = (Q, \Sigma, \delta, q_0, F)$ é um AFN, então $L(N) = \{w; \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ é a linguagem de N .

Se L é a linguagem de um AFN N , dizemos que N *decide* a linguagem L . Dizemos também que N *aceita* a linguagem L .

Exercício resolvido 3.5 Seja $D = (Q, \Sigma, \delta, q_0, F)$ um AFD. Apresente um AFN N , tal que $L(D) = L(N)$.

Solução: A ideia é generalizar o que fizemos no Exercício Resolvido 3.4. O AFN N que aceita a mesma linguagem de D é o seguinte: $N = (Q, \Sigma, \delta', q_0, F)$, tal que a função δ' é a seguinte: se $\delta(x, y) = z$, então $\delta'(x, y) = \{z\}$. ■

3.2.3 Exercícios

Exercício 3.12 Suponha que queiramos mudar a definição de AFNs para que a *função de transição* tenha a seguinte forma: $\delta : Q \times \Sigma \rightarrow (\mathcal{P}(Q) \setminus \emptyset)$. Observe que autômatos segundo esta nova definição nunca morrem. Prove que dado um AFN N , podemos construir um AFN N' segundo nossa nova definição que aceita a mesma linguagem de N . ■

Exercício 3.13 No caso particular de AFNs que são determinísticos, qual é o formato da árvore de computações possíveis? ■

3.3 Equivalência entre AFDs e AFNs

Algo que vamos lidar com freqüência neste curso é a diferença de expressividade entre diferentes modelos de computação. Veremos no Capítulo 4 que existem algoritmos que, embora não possam ser expressos na forma de um AFD, podem ser expressos em outros modelos matemáticos. Em tais casos normalmente dizemos que estes outros modelos são mais poderosos (ou mais expressivos) que o modelo de AFD.

EXPRESSIVIDADE DE MODELOS DE COMPUTAÇÃO

A ideia de expressividade de um modelo de computação é central em Teoria da Computação. Embora AFDs sejam capazes de realizar tarefas interessantes, como testar se um dado número é divisível por 3 (ou mesmo divisibilidade por qualquer k fixado a priori), veremos no Capítulo 4 que AFDs não são capazes de “testar primalidade” de números. De maneira formal, provaremos que não existe um AFD D tal que $L(D) = L_p$, ou seja, que tome uma string binária e decida se ela representa um número primo. Como sabemos que podemos escrever algoritmos (expressando eles em linguagem C, por exemplo) para testar primalidade de números, concluímos que o formalismo de AFD não é o formalismo mais geral possível para expressar algoritmos.

A limitações dos AFDs, neste momento do curso, não é o mais importante aqui. O que queremos observar aqui é que sempre que apresentamos um novo modelo de computação, como o modelo de AFNs, algo que sempre nos preocuparemos é como tal modelo se compara com outros modelos de computação já conhecidos, com AFDs.

No *Exercício Resolvido 3.5* o objetivo foi mostrar que qualquer linguagem que um AFD reconheça, também pode ser reconhecida por um AFN. Isso significa que AFNs são pelo menos tão poderosos como AFDs. Isso é natural, pois AFNs são generalizações de AFDs. A pergunta óbvia que devemos fazer é se AFNs são *estritamente* mais poderosos que AFDs. Veremos nesta seção que a resposta é não. Ou seja, podemos mostrar que se uma linguagem pode ser aceita por um AFN, então existe algum AFD que aceita a mesma linguagem. Isso pode parecer um pouco surpreendente, pois AFNs, em algumas situações, tem a capacidade de advinhar qual transição deve fazer, uma capacidade que AFDs não tem.

3.3.1 Algoritmo de construção de conjuntos

O Algoritmo 1 apresentado a seguir recebe um AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ como entrada e retorna um AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ como saída tal que $L(N) = L(D)$. A ideia central é construir um AFD D que possa “simular” N usando a seguinte ideia: No momento em que o AFN N vai processar o i -ésimo símbolo da string de entrada $w = w_1 \dots w_i \dots w_n$, os possíveis estados que N pode estar em tal instante são dados pelos nós do i -ésimo nível da árvore de computações possíveis de N com w . Seja P_i o conjunto dos estados do i -ésimo nível desta árvore e P_{i+1} o conjunto de nós do $(i+1)$ -ésimo nível desta mesma árvore. O AFD D construído pelo algoritmo, terá uma transição de um estado chamado P_i para outro chamado P_{i+1} (tais estados correspondem aos conjuntos de estados P_i e P_{i+1} da árvore de computações possíveis mencionados anteriormente), e esta transição terá o rótulo w_i . Em outras palavras, $\delta_D(P_i, w_i) = P_{i+1}$. O que o algoritmo faz é usar força bruta e aplicar a função δ_N em todas as combinações possíveis de pares (P_i, a) , tal que $P_i \subseteq Q$, $a \in \Sigma$ para que possa determinar P_{i+1} .

Algorithm 1 Construindo um AFD a partir de um AFN.

AFN_AFD ($Q_N, \Sigma, \delta_N, q_0, F_N$)

- 1: $Q_D = \mathcal{P}(Q_N)$
- 2: $F_D = \{S \subseteq \mathcal{P}(Q_N) \text{ tal que } S \cap F_N \neq \emptyset\}$
- 3: **for all** $S \subseteq Q_D$ **do**
- 4: **for all** $a \in \Sigma$ **do**
- 5: Defina a função δ_D no par (S, a) da seguinte maneira: $\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$
- 6: $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$
- 7: **Return** D

Vamos aplicar o algoritmo no autômato da Figura 3.2, cuja tabela de transições é a seguinte.

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	\emptyset	$\{q_2\}$
$*q_2$	\emptyset	\emptyset

Ao aplicarmos o Algoritmo 1 no AFN da tabela anterior, obtemos como saída o AFD definido pela seguinte tabela de transições:

	0	1
\emptyset	\emptyset	\emptyset
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	\emptyset	$\{q_2\}$
$*\{q_2\}$	\emptyset	\emptyset
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	\emptyset	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Observe que os estados do AFD obtido são conjuntos. Isso não é um problema, pois os estados de um AFD podem ser qualquer coisa, desde que os elementos da imagem da função δ sejam do mesmo tipo que os estados do AFD (note que neste caso os próprios estados são conjuntos). O que não devemos fazer é confundir com o caso dos AFNs em que os elementos da imagem da função δ são de natureza diferente dos estados.

3.3.2 Algoritmo de construção de conjuntos: versão melhorada

Alguns alunos mais observadores devem ter notado que não precisamos de todos os estados do AFD resultante. Precisamos apenas dos estados “alcançáveis” a partir do estado inicial $\{q_0\}$. Olhando a tabela, notamos que na linha do estado inicial $\{q_0\}$, atingimos o estado $\{q_0, q_1\}$ e o próprio estado $\{q_0\}$, dependendo do símbolo lido. Ao olharmos a tabela, na linha do estado $\{q_0, q_1\}$, vemos que atingimos o estado $\{q_0, q_2\}$ se o símbolo lido for 1. Se o autômato estiver no estado $\{q_0, q_2\}$, independente do símbolo lido, os únicos estados alcançáveis são estados que já mencionamos (i.e., $\{q_0\}$, $\{q_0, q_1\}$ e $\{q_0, q_2\}$). Portanto, como a computação sempre começa no estado $\{q_0\}$, os únicos três estados atingíveis para qualquer string de entrada são $\{q_1\}$, $\{q_0, q_1\}$ e

$\{q_0, q_2\}$. Com isso, podemos eliminar os estados desnecessários e simplificar a tabela da seguinte maneira:

	0	1
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

Não vamos nos preocupar tanto com formalismo neste ponto, mas se quiséssemos fornecer uma definição formal para o conceito de estado alcançável poderíamos usar a ideia de um vértice alcançável por uma busca em largura em um grafo direcionado. Pense no AFD obtido pela saída do Algoritmo 1 como um grafo direcionado em que os vértices são os estados e as arestas direcionadas do grafo ligam o vértice p ao vértice q caso ocorra que para algum símbolo a , $\delta(p, a) = q$. Com isso, os *estados alcançáveis* são os vértices alcançáveis por algum caminho direcionado a partir do vértice correspondente ao estado inicial.

A partir de agora, sempre que formos construir um AFD equivalente a um dado AFN, vamos manter apenas os estados alcançáveis. A versão aprimorada do algoritmo é o seguinte:

Algorithm 2 Construindo um AFD a partir de um AFN: algoritmo melhorado.

AFN_AFD ($Q_N, \Sigma, \delta_N, q_0, F_N$)

- 1: $Q_D = \mathcal{P}(Q_N)$
 - 2: $F_D = \{S \subseteq \mathcal{P}(Q_N) \text{ tal que } S \cap F_N \neq \emptyset\}$
 - 3: **for all** $S \subseteq Q_N$ **do**
 - 4: **for all** $a \in \Sigma$ **do**
 - 5: Defina a função δ_D da seguinte maneira: $\delta_D(S, a) = \bigcup_{p \in S} \delta_N(p, a)$
 - 6: Elimine os estados não alcançáveis
 - 7: **Return** $(Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$
-

Teorema 3.3.1 Se o AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ é obtido pelo Algoritmo 1 (ou pelo Algoritmo 2) a partir de um AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$, então $L(N) = L(D)$.

Exercício 3.14 Prove o Teorema 3.3.1

(Dica: prove por indução que $\forall w \in \Sigma^*, \hat{\delta}_D(q_0, w) \in F_N \Leftrightarrow \hat{\delta}_N(\{q_0\}, w) \cap F_N$)

■

O próximo teorema enuncia a equivalência entre AFDs e AFNs.

Teorema 3.3.2 Uma linguagem L é aceita por um AFD se e somente se L é aceita por um AFN.

Prova: Consequência do Teorema 3.3.1 e do *Exercício Resolvido 3.5*.

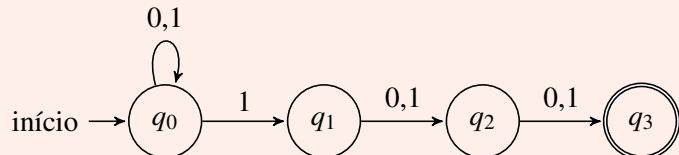
3.3.3 Exercícios

Exercício 3.15 Considere o o AFN dado pela tabela abaixo:

	0	1
$\rightarrow p$	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	\emptyset
$*s$	$\{s\}$	$\{s\}$

Apresente um AFD que aceite a mesma linguagem do AFN acima. ■

Exercício 3.16 Qual a linguagem aceita pelo AFN abaixo?



Exercício 3.17 Forneça um AFD equivalente ao AFN do Exercício 3.16. ■

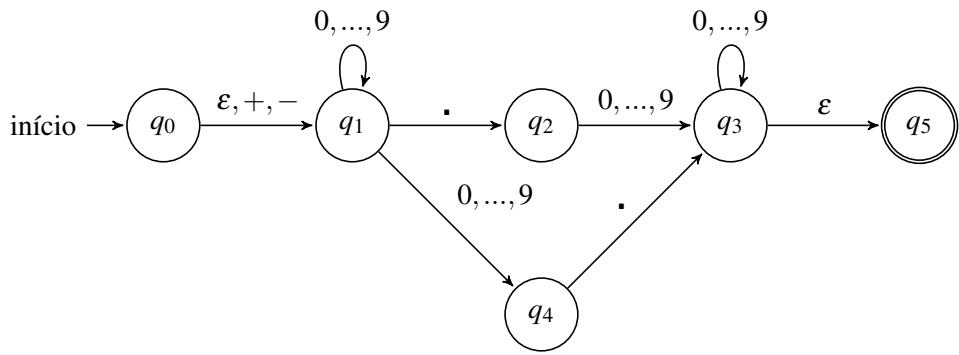
3.4 Autômatos Finitos não Determinísticos com transições ϵ

Vamos considerar agora um diagrama de um AFN que contém algumas transições com o rótulo ϵ . Estas transições são chamadas de *transições ϵ* e um autômato que tenha tais transições será chamado de ϵ -AFN.

A ideia é tentar incrementar ainda mais o poder do nosso modelo de computação. A nova habilidade que vamos incluir em nosso autômato é a possibilidade de arbitrariamente fazer certas mudanças de estado sem que nenhum símbolo seja consumido. Para apresentar este conceito vamos usar um exemplo do livro de Hopcroft, Motwani e Ullman [HNU06]. A ideia é construir um ϵ -AFN que aceita strings que representam números decimais que estejam na forma descrita a seguir:

- (1) O número pode ter sinal “+” ou “-”, mas este sinal é opcional;
- (2) Em seguida, há um string de dígitos (os dígitos da parte inteira do número), que também é opcional;
- (3) Após esta sequência de dígitos há um ponto decimal “.” obrigatório;
- (4) Opcionalmente, há outra string de dígitos (os dígitos da parte decimal do número);
- (5) Faz-se a restrição extra de que pelo menos uma das strings de dígitos de (2) e (4) é não seja vazia.

Observe que o alfabeto do autômato é $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, ., +, -\}$. Alguns exemplos de strings aceitas são 5.72 , $+5.72$, $-12.$, -1 . e -5 enquanto alguns exemplos de strings não aceitam são $+5.72$, -12 , $8.0-$ e $.56$. O autômato é o seguinte:

Figure 3.5: Um ϵ -AFN que aceita números decimais.

A definição formal para ϵ -AFNs é bastante semelhante a definição dos AFNs:

Definição 3.4.1 — ϵ -AFN. Um ϵ -AFN é uma 5-tupla $E = (Q, \Sigma, \delta, q_0, F)$ tal que cada um dos componentes tem a mesma interpretação que um AFN, exceto pelo fato que δ é uma função do tipo $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow \mathcal{P}(Q)$, sendo que $\epsilon \notin \Sigma$.

■ **Exemplo 3.3** O ϵ -AFN do exemplo anterior é $E = (\{q_0, \dots, q_5\}, \{., +, -, 0, \dots, 9\}, \delta, q_0, \{q_5\})$ tal que a tabela de transições de δ é a seguinte:

	ϵ	sinal	.	dígito
q_0	$\{q_1\}$	$\{q_1\}$	\emptyset	\emptyset
q_1	\emptyset	\emptyset	$\{q_2\}$	$\{q_1, q_4\}$
q_2	\emptyset	\emptyset	\emptyset	$\{q_3\}$
q_3	$\{q_5\}$	\emptyset	\emptyset	$\{q_3\}$
q_4	\emptyset	\emptyset	$\{q_3\}$	\emptyset
$*q_5$	\emptyset	\emptyset	\emptyset	\emptyset

Observe que no Exemplo 3.3, se quiséssemos ser totalmente rigorosos, a tabela de transições teria que ter 14 colunas: exatamente 13 colunas para o alfabeto do autômato (uma coluna para cada um dos 10 dígitos, duas colunas para os sinais e uma para o ponto) além de uma coluna extra para ϵ). Entretanto, para simplificar, agrupamos todos os dígitos em apenas uma coluna e os dois possíveis sinais em uma outra coluna, pois transições para cada elemento destes grupos são as mesmas.

O nosso próximo passo agora é definir o conceito de função de transição estendida. Para que possamos apresentar tal definição, vamos antes definir o que é o ϵ -Fecho de um estado. A ideia é que o fecho de um estado q é o conjunto de todos os possíveis estados que podem ser atingidos a partir de q (incluindo o próprio q) utilizando-se uma quantidade arbitrária de transições ϵ .

Definição 3.4.2 — ϵ -Fecho de estados. Seja um ϵ -AFN com conjunto de estados Q e seja $q \in Q$. Vamos definir o conjunto ϵ -Fecho(q) de maneira indutiva:

Base: $q \in \epsilon$ -Fecho(q)

Indução: se $p \in \epsilon$ -Fecho(q) e $r \in \delta(p, \epsilon)$, então $r \in \epsilon$ -Fecho(q).

Agora que temos a Definição 3.4.2 em mãos, podemos definir o conceito de função de transição

estendida de um ε -AFN. A ideia é parecida com a definição de função de função de transição estendida de um AFN, com o cuidado extra de adicionar os estados que podem ser atingidos por transições ε .

Definição 3.4.3 — Função de transição estendida em ε -AFNs. Dado um ε -AFN $N = (Q, \Sigma, \delta, q_0, F)$, definimos $\hat{\delta} : Q \times \Sigma^* \rightarrow \mathcal{P}(Q)$ indutivamente:

Base: $\hat{\delta}(q, \varepsilon) = \varepsilon\text{-Fecho}(q)$

Indução: Seja $w \in \Sigma^*$ da forma xa , onde $x \in \Sigma^*$ e $a \in \Sigma$. Suponha $\hat{\delta}(q, x) = \{p_1, p_2, \dots, p_k\}$.

Suponha $\bigcup_{i=1}^k \delta(p_i, a) = \{r_1, r_2, \dots, r_m\}$

Então • $\hat{\delta}(q, w) = \bigcup_{j=1}^m \varepsilon\text{-Fecho}(r_j)$

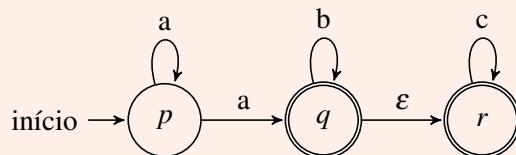
O próximo passo é definir o que é a linguagem de um ε -AFN.

Definição 3.4.4 — Linguagem de ε -AFNs. Seja $E = (Q, \Sigma, \delta, q_0, F)$ um ε -AFN. Definimos $L(E) = \{w; \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$ como sendo a linguagem de E .

Dada uma linguagem L , se existe um ε -AFN tal que $L = L(E)$, então dizemos que E aceita L .

Exercício resolvido 3.6 Seja $\Sigma = \{a, b, c\}$. Apresente um ε -AFN para a linguagem das strings sobre Σ que têm a forma $a^n b^m c^l$, tal que $n > 0, m \geq 0, l \geq 0$.

Solução:



■

3.5 Equivalência entre AFDs e ε -AFNs

Nesta seção vamos construir um AFD D a partir de um ε -AFN $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. A ideia aqui é praticamente a mesma da que vimos na seção 3.3. A única diferença é que temos que tomar cuidado com as transições ε .

Algorithm 3 Construindo um AFD a partir de um ε -AFN

AFN_AFD ($Q_E, \Sigma, \delta_E, q_0, F_E$)

- 1: $Q_D = \mathcal{P}(Q_E)$
 - 2: $F_D = \{S; S \subseteq Q_D \text{ e } S \cap F_E \neq \emptyset\}$
 - 3: $q_0 = \varepsilon\text{-Fecho}(q_0)$
 - 4: **for all** $S \subseteq Q_N$ **do**
 - 5: **for all** $a \in \Sigma$ **do**
 - 6: $R = \bigcup_{q_i \in S} \delta_E(q_i, a)$
 - 7: Defina a função δ da seguinte maneira: $\delta_D(S, a) = \bigcup_{r_j \in R} \varepsilon\text{-Fecho}(r_j)$
 - 8: Elimine os estados não alcançáveis
 - 9: **Return** $(Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$
-

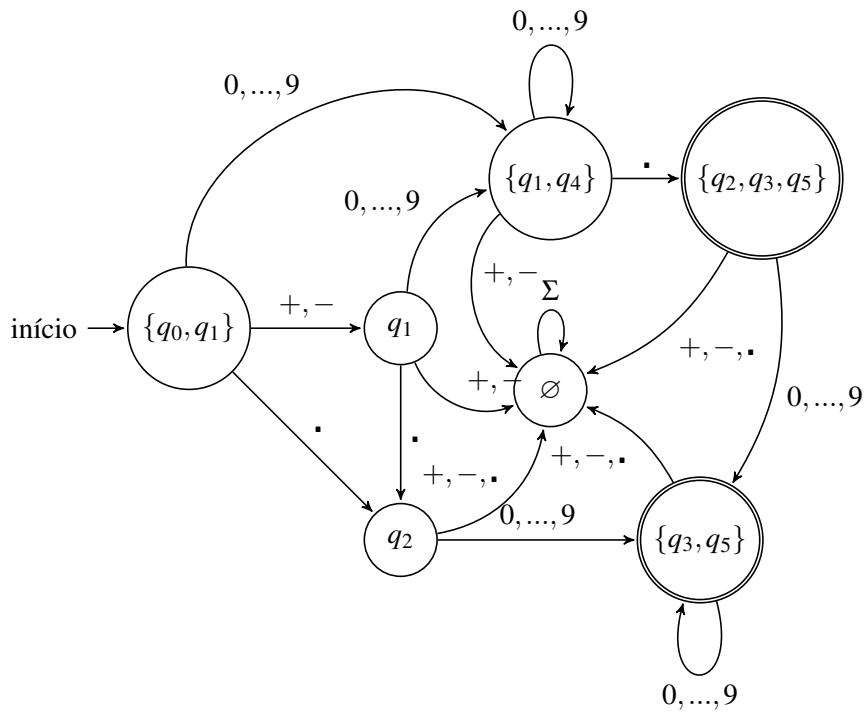
Teorema 3.5.1 Se o AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ é obtido pelo Algoritmo 3 quando toma como entrada o ε -AFN $E = (Q_N, \Sigma, \delta_N, q_0, F_N)$, então $L(D) = L(N)$.

Teorema 3.5.2 Uma linguagem é aceita por um AFD se e somente se é aceita por um ε -AFN.

Vamos construir um AFD equivalente ao ε -AFN da Figura 3.5 usando o Algoritmo 3. A tabela do AFD que o algoritmo retorna é o seguinte:

	$+, -$	$0, \dots, 9$	\cdot
$\rightarrow \{q_0, q_1\}$	$\{q_1\}$	$\{q_1, q_4\}$	$\{q_2\}$
$\{q_1\}$	\emptyset	$\{q_1, q_4\}$	$\{q_2\}$
$\{q_1, q_4\}$	\emptyset	$\{q_1, q_4\}$	$\{q_2, q_3, q_5\}$
$\{q_2\}$	\emptyset	$\{q_3, q_5\}$	\emptyset
\emptyset	\emptyset	\emptyset	\emptyset
$*\{q_2, q_3, q_5\}$	\emptyset	$\{q_3, q_5\}$	\emptyset
$*\{q_3, q_5\}$	\emptyset	$\{q_3, q_5\}$	\emptyset

O diagrama do AFD obtido pelo Algoritmo 3 é o seguinte:



3.5.1 Exercícios

Exercício 3.18 Considere o ε -AFN dado pela tabela abaixo:

	ε	a	b	c
$\rightarrow p$	\emptyset	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	\emptyset
$*_r$	$\{q\}$	$\{r\}$	\emptyset	$\{p\}$

- (a) Desenhe o diagrama do ε -AFN
 - (b) Calcule o ε -fecho de cada estado.
 - (c) Forneça todas strings w tal que $|w| \leq 2$ aceitas pelo autômato.
 - (d) Converta o ε -AFN em um AFD.

Exercício 3.19 Repita a questão 1 para o ε -AFN dado pela tabela abaixo:

	ε	a	b	c
$\rightarrow p$	$\{q, r\}$	\emptyset	$\{q\}$	$\{r\}$
q	\emptyset	$\{p\}$	$\{r\}$	$\{p, q\}$
$*r$	\emptyset	\emptyset	\emptyset	\emptyset

Exercício 3.20 Apresente um AFD que decida a mesma linguagem do ε -AFN do Exercício Resolvido 3.6 ■

Exercício 3.21 Apresente a árvore de computações possíveis para ε -AFN da figura 3.5 com a string -0.95 . ■

3.6 Expressões Regulares (ERs)

Expressões regulares são expressões matemáticas que representam linguagens. A ideia é que podemos usar uma expressão como, por exemplo, 01^*0 para denotar uma determinada linguagem. A expressão deste exemplo, em particular, representa o conjunto de todas as strings que contém “um único 0 seguido por um número arbitrário de 1’s seguido de um único 0”.

De maneira mais geral, veremos como representar **qualquer linguagem regular** usando tais expressões. Adicionalmente, veremos também que **apenas linguagens regulares**, ou seja, linguagens de autômatos, podem ser representadas por expressões regulares. Com isso, a conclusão que tiramos é que existe uma certa equivalência entre expressões regulares e autômatos. Mais precisamente, expressões regulares tem o poder de expressar exatamente os problemas que podem ser resolvidos pelos algoritmos expressos na forma de AFDs.

3.6.1 Construindo Expressões Regulares

O nosso objetivo agora é definir indutivamente que tipos de expressões são consideradas expressões regulares válidas. O ponto chave é que cada expressão regular corresponda a exatamente uma linguagem. Portanto, ao mesmo tempo que iremos definir quais expressões são válidas, vamos também definir exatamente qual linguagem corresponde a esta expressão.

Definição 3.6.1 — Expressões Regulares.

Seja Σ um alfabeto qualquer. Uma expressão regular R é uma expressão matemática, correspondente a uma linguagem $L(R) \subseteq \Sigma^*$, definida indutivamente a seguir.

Base (expressões regulares elementares):

- (1) Tanto ε quanto \emptyset são expressões regulares que correspondem, respectivamente, às linguagens $L(\varepsilon) = \{\varepsilon\}$ e $L(\emptyset) = \emptyset$.
- (2) Para todo símbolo a de Σ , a expressão a é uma expressão regular correspondente a linguagem $L(a) = \{a\}$.

Indução:

- (1) Se E e F são expressões regulares, então $E + F$ é uma expressão regular representando a união de $L(E)$ e $L(F)$. Isto é, $L(E + F) = L(E) \cup L(F)$.
- (2) Se E e F são expressões regulares, então EF é uma expressão regular denotando a concatenação de $L(E)$ e $L(F)$. Isto é, $L(EF) = L(E)L(F)$.
- (3) Se E é uma expressão regular, então E^* é uma expressão regular que representa o fechamento de $L(E)$. Isto é, $L(E^*) = (L(E))^*$.
- (4) Se E é uma expressão regular, então (E) também é uma expressão regular, representando a mesma linguagem que E representa. Isto é, $L((E)) = L(E)$.

■ **Exemplo 3.4** Seja $\Sigma = \{0, 1\}$. Considere a linguagem das strings binárias $\{\epsilon, 1, 11, 111, 1111, \dots\}$, i.e., strings que consistem de um número arbitrário de símbolos 1. A expressão regular $R = \underline{1}^*$ corresponde a esta linguagem, pois:

- Primeiramente, como $1 \in \Sigma$, observe que a expressão regular $\underline{1}$ é uma expressão válida pela base da Definição 3.6.1, item (2). Segundo a indução, item (3), da Definição 3.6.1, se $\underline{1}$ é uma expressão válida, então $\underline{1}^*$ também é uma expressão válida.
- Note que $L(\underline{1}) = \{1\}$ pelo item (2) da base da definição. Aplicando agora a indução, item (3), veja que $L(\underline{1}^*) = \{1\}^* = \{\epsilon, 1, 11, 111, 1111, \dots\}$.

Exercício resolvido 3.7 A expressão regular para a linguagem L das “strings contendo 0’s e 1’s alternados” é $(01)^* + (10)^* + 0(10)^* + 1(01)^*$. Justifique isto passo a passo.

Solução: Como ainda estamos em um dos primeiros exemplos de expressões regulares, vamos ser bastante cuidadosos em nossa demonstração.

Passo 1: Segundo a Base da Definição 3.6.1 (item 2), se $0 \in \Sigma$, então $\underline{0}$ é uma expressão regular válida. Ainda segundo esta definição, a expressão $\underline{0}$ representa a linguagem $\{0\}$.

Passo 2: Usando o mesmo argumento do passo anterior, $\underline{1}$ é uma expressão regular válida que representa a linguagem $\{1\}$.

Passo 3: Até este ponto já sabemos que $\underline{0}$ e $\underline{1}$ são expressões regulares válidas que representam as linguagens $\{0\}$ e $\{1\}$, respectivamente. Segundo a Definição 3.6.1 (Indução, item 2), concluímos que $\underline{0}\underline{1}$ também é uma expressão válida. A linguagem que esta expressão representa é $L(\underline{0}\underline{1}) = \{0\} \cdot \{1\} = \{01\}$.

Passo 4: A partir da expressão obtida no passo Passo 3, podemos aplicar a Definição 3.6.1 (Indução, item 3) e concluir que $\underline{(01)}^*$ é uma expressão válida, e $L(\underline{(01)}^*) = \{\epsilon, 01, 0101, 010101, \dots\}$.

Passo 5: Usando argumentos semelhantes aos anteriores, concluímos que $\underline{(10)}^*$ também é uma expressão válida, e $L(\underline{(10)}^*) = \{\epsilon, 10, 1010, 101010, \dots\}$.

Passo 6: Pelos Passos 4 e 5, $\underline{(01)}^*$ e $\underline{(10)}^*$ são expressões válidas que representam as linguagens $\{\epsilon, 01, 0101, 010101, \dots\}$ e $\{\epsilon, 10, 1010, 101010, \dots\}$, respectivamente. Pela Definição 3.6.1 (Indução, item 1), $\underline{(01)}^* + \underline{(10)}^*$ é uma expressão válida e $L(\underline{(01)}^* + \underline{(10)}^*) = \{\epsilon, 01, 0101, 010101, \dots\} \cup \{\epsilon, 10, 1010, 101010, \dots\} = \{\epsilon, 01, 10, 0101, 1010, 010101, 101010, \dots\}$.

Passo 7: Podemos concluir que $\underline{0}(10)^*$ é uma expressão regular a partir do fato que $\underline{0}$ e $\underline{(10)}^*$ são expressões válidas (concluímos isso nos Passos 1 e 5) e a linguagem que expressão representa é $\{0, 010, 01010, 0101010, \dots\}$. Usando argumentos semelhantes, podemos concluir que $\underline{1}(01)^*$ é uma expressão regular válida e $L(\underline{1}(01)^*) = \{1, 101, 10101, 1010101, \dots\}$. Com isso, podemos aplicar a Definição 3.6.1 (Indução, item 1) nas expressões $\underline{0}(10)^*$ e $\underline{1}(01)^*$ para concluir que $\underline{0}(10)^* + \underline{1}(01)^*$ é uma expressão regular válida representando a linguagem $\{0, 010, 01010, 0101010, \dots\} \cup \{1, 101, 10101, 1010101, \dots\}$.

Passo 8: Aplicando a Definição 3.6.1 (Indução, item 1) nas expressões obtidas no Passo 6 e 7, concluímos que $E = \underline{(01)}^* + \underline{(10)}^* + \underline{0}(10)^* + \underline{1}(01)^*$ é uma expressão regular válida e $L(E) = \{\epsilon, 0, 1, 01, 10, 010, 101, 0101, 1010, 010101, \dots\}$. ■

Assim como ocorre em expressões aritméticas, operadores em expressões regulares obedecem regras de precedência. As regras são as seguintes:

- O operador $*$ tem precedência mais alta e, portanto, deve ser o primeiro a ser aplicado. Com isso, por exemplo, a expressão $\underline{01}^*$ é equivalente a $\underline{0}(1)^*$;
- O operador com segunda maior precedência é o operador de concatenação. Com isso, por exemplo, a expressão $a + \underline{bc}$ é equivalente a $\underline{a} + (\underline{bc})$;
- O operador de menor precedência é o operador $+$;
- Como de custume, usamos parentesis para alterar a precedência de operadores.

Exercício 3.22 Seja $\Sigma = \{a, b, c\}$. Apresente a expressão regular para a linguagem das strings sobre Σ que começam e terminam com a e têm pelo menos um b .

Exercício 3.23 Seja $\Sigma = \{a, b\}$. Apresente a expressão regular para a linguagem das strings sobre Σ que tem tamanho ímpar.

3.6.2 Expressões Regulares e Autômatos

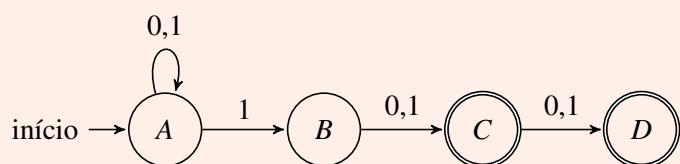
Os dois seguintes teoremas são centrais em teoria de linguagens regulares:

Teorema 3.6.1 Seja $L(D)$ a linguagem de um AFD D qualquer. Então existe uma expressão regular R tal que $L(R) = L(D)$.

Teorema 3.6.2 Seja $L(R)$ é a linguagem representada por uma expressão regular R qualquer. Então existe um AFD D tal que $L(D) = L(R)$.

3.6.3 Exercícios

Exercício 3.24 Obtenha uma expressão regular que represente a linguagem do seguinte ϵ -AFN:



Exercício 3.25 Obtenha um ϵ -AFN cuja linguagem seja a mesma representada pela expressão regular $\underline{(0+1)}^*1\underline{(0+1)}$.

Exercício 3.26 Considere os alfabetos $\Sigma_1 = \{a, b, c\}$ e $\Sigma_2 = \{0, 1\}$. Forneça expressões regulares para as seguintes linguagens:

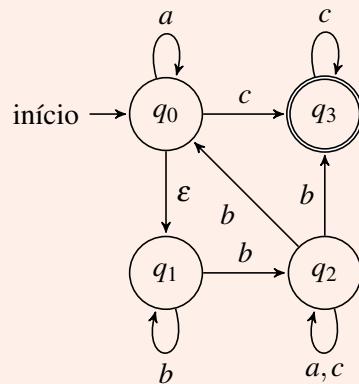
- $L \subseteq \Sigma_1^*$ tal que toda string de L tem pelo menos um a e um b .
- Conjunto de strings sobre Σ_2 tal que o terceiro símbolo de trás para frente é 1.
- $L \subseteq \Sigma_2^*$ definido por $L = \{w \mid w \text{ tenha um número par de } 0's \text{ e um número par de } 1's\}$.

Exercício 3.27 Dado o AFD abaixo, encontre uma expressão regular equivalente. Você deve mostrar o desenvolvimento passo a passo de solução.

	0	1
$\rightarrow^* p$	s	p
q	p	s
r	r	q
s	q	r

■

Exercício 3.28 Considere o ε -AFN $E = (\{q_0, q_1, q_2, q_3\}, \{a, b, c\}, \delta, q_0, \{q_3\})$ abaixo:



(a) Apresente a tabela de transições da função δ do autômato E .

(b) Forneça uma expressão regular R tal que $L(R) = L(E)$. Mostre passo a passo o desenvolvimento da sua solução.

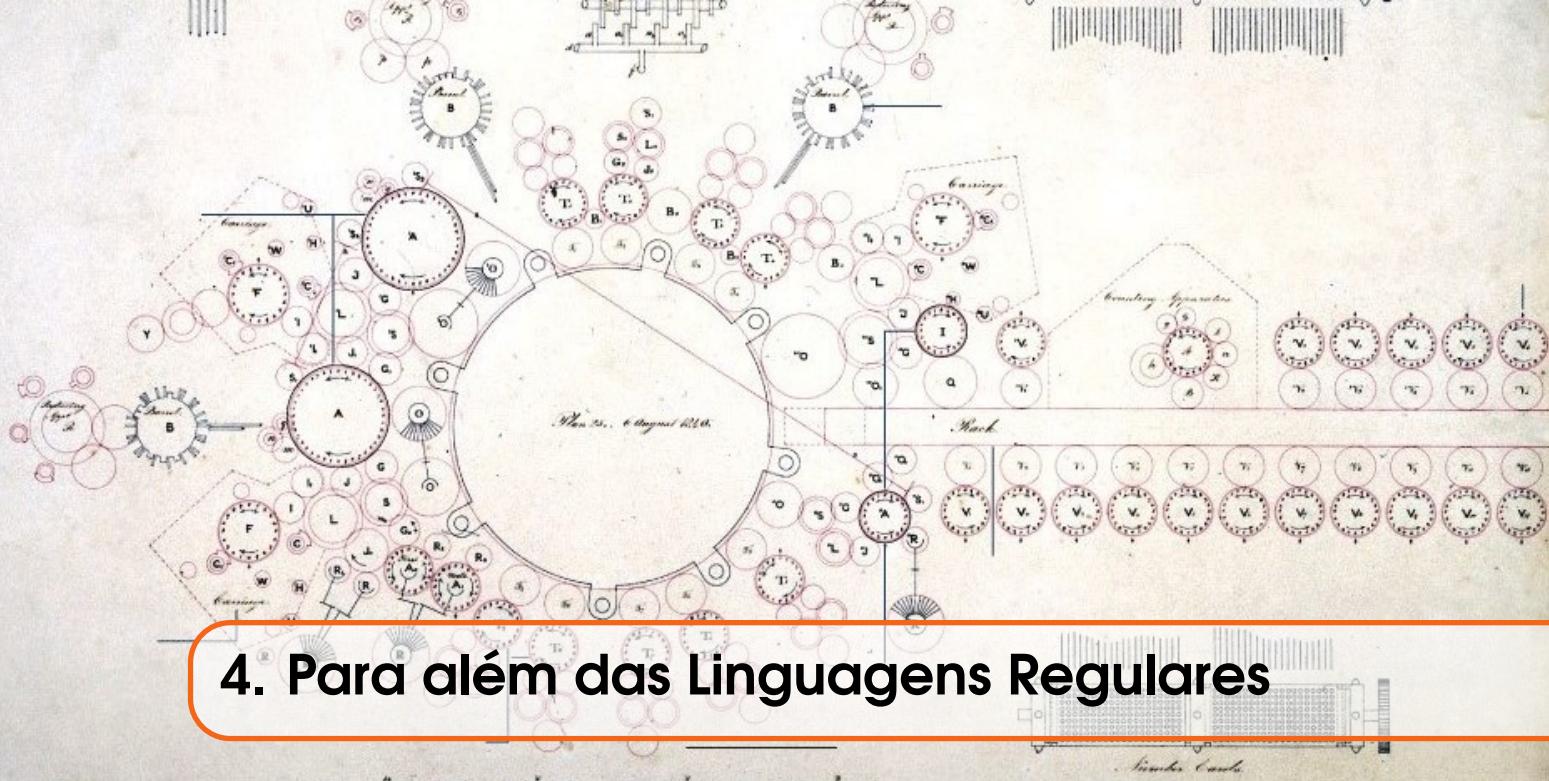
■

Exercício 3.29 Forneça um ε -AFN que aceite a mesma linguagem da expressão regular $(00 + 11)^* + (111)^*0$.

■

Exercício 3.30 Construa um ε -AFN que aceite a mesma linguagem da expressão regular $00(0 + 1)^*$.

■



4. Para além das Linguagens Regulares

Existem linguagens não regulares? Nas seções anteriores nós já insinuamos que a resposta é sim. Veremos neste capítulo que algumas linguagens extremamente simples como, por exemplo, $L_{01} = \{0^n 1^n \mid n \geq 1\}$ não são regulares.

Lembramos que, por definição, uma linguagem é regular se existe um AFD que a aceite. Portanto, dada uma linguagem L , se quisermos mostrar que L é regular, basta explicitamente apresentarmos um AFD a aceite. Entretanto, se quisermos mostrar que L não é regular, temos que provar que não existe nenhum AFD que aceite L . Ou seja, precisamos usar um argumento que exclua logicamente a possibilidade de que cada um dos infinitos possíveis AFDs tenha a propriedade de ser um AFD que aceite L . Este tipo de resultado tende a ser bem mais difícil de se obter. Nossa primeiro objetivo neste capítulo é apresentar uma ferramenta matemática, conhecida como *Lema do Bombeamento*, que será extremamente útil para demonstrarmos que certos AFDs não existem.

4.1 O Lema do Bombeamento para Linguagens Regulares

O seguinte lema será bastante útil para provarmos que certas linguagens não são regulares:

Lema 4.1.1 — Lema do Bombeamento (LB). Seja L uma linguagem regular infinita. Então existe uma constante t tal que $\forall w \in L$, com $|w| \geq t$, o seguinte é verdadeiro:

$\exists x, y, z \in \Sigma^*$ tal que $w = xyz$ e as três condições abaixo são satisfeitas:

- $$(1) y \neq \varepsilon \quad (2) |xy| \leq t \quad (3) \forall k \geq 0, xy^k z \in L$$

No decorrer deste capítulo nos referiremos ao Lema 4.1.1 como Lema do Bombeamento ou simplesmente LB. Agora vamos mostrar como podemos fazer uso do LB para mostrar que uma linguagem não é regular. A vantagem de se usar o LB é que não precisamos mostrar explicitamente que um certo AFD não existe. O que acontece aqui é que todo trabalho da prova de inexistência do AFD fica “encapsulada” dentro da demonstração do LB, que iremos omitir aqui. Segue um exemplo

de como fazer uso do Lema do Bombeamento para provar que uma determinada linguagem não é regular.

Teorema 4.1.2 $L_{01} = \{0^i 1^i \mid i \geq 1\}$ não é regular.

Prova: Suponha que L_{01} é regular. Portanto, usando o LB, sabemos que existe $t \in \mathbb{N}$, tal que se tomarmos uma string w de L “grande o suficiente” ou seja, tal que $|w| \geq t$, deve existir $x, y, z \in \Sigma^*$ tal que w pode ser escrita como $w = xyz$ de maneira que as três afirmações abaixo são verdadeiras:

- (1) $y \neq \epsilon$
- (2) $|xy| \leq t$
- (3) $\forall k \geq 0, xy^k z \in L_{01}$.

Considere a string $w = 0^t 1^t$. Note que $w \in L_{01}$ e $|w| \geq t$. Portanto podemos aplicar o LB e, com isso, w pode ser escrita na forma $w = xyz$ tal que as três afirmações acima são verdadeiras.

Pela condição (2), temos que $|xy| \leq t$ e portanto a string xy contém apenas 0's. Portanto, todos os símbolos 1 da string w estão contidos em z (note não **necessariamente** z contém apenas símbolos 1, mas isso não é relevante aqui).

Pela condição (3), a string $xy^k z$ deve pertencer a L_{01} para qualquer $k \geq 0$. Portanto, em particular, $xy^0 z \in L_{01}$. Com isso, temos que $xz \in L_{01}$.

Note que $|xyz| = 2t$. Pela condição (1), $|xz| < |xyz|$ e portanto $|xz| < 2t$. Como z tem t símbolos 1, a string xz pode ter no máximo $t-1$ símbolos 0. Isso é uma contradição, pois $xz \in L_{01}$. Logo L_{01} não é regular. \square

Vamos utilizar agora o LB em uma linguagem um pouco mais interessante:

Teorema 4.1.3 $L_p = \{1^p \mid p \text{ é um número primo}\}$ não é regular.

Prova: Suponha que L_p é regular. Então o LB nos diz que existe $t \in \mathbb{N}$, tal que se tomarmos uma string w de L_p tal que $|w| \geq t$, então $\exists x, y, z \in \Sigma^*$ tal que w pode ser escrita como $w = xyz$ e:

- (1) $y \neq \epsilon$
- (2) $|xy| \leq t$
- (3) $\forall k \geq 0, xy^k z \in L_p$.

Considere a string $w = 1^p$ para algum primo $p \geq t$. Note que w é uma string para a qual podemos aplicar o LB, pois $w \in L_p$ e $|w| \geq t$. Portanto w pode ser escrita na forma $w = xyz$ satisfazendo condições acima.

Pela condição (3), a string $xy^k z$ deve pertencer a L_p para qualquer $k \geq 0$. Em particular, para $k = p+1$, podemos concluir que $xy^{p+1} z \in L_p$.

Note que $|xy^{p+1} z| = |xz| + |y^{p+1}|$. Seja $|y| = n$. Com isso temos:

$$\begin{aligned} |xy^{p+1} z| &= |xz| + |y^{p+1}| \\ &= (p-n) + n \cdot (p+1) \\ &= p - n + n + tp \\ &= p + np \\ &= p \cdot (1+n) \end{aligned}$$

Como p é primo, $p \geq 2$. Além disso, a condição (1) diz que $n \geq 1$, e portanto $(1+n) \geq 2$. Como ambos p e $(1+n)$ são maiores ou iguais a dois, o produto $p \cdot (1+n) = |xy^{p+1} z|$ não é um número primo. Isso contradiz o fato que $xy^{p+1} z \in L_p$. \square

O Teorema 4.1.3 mostra que o problema de reconhecer se um determinado número é primo não pode ser solucionado usando um algoritmo (ou uma máquina) cujo funcionamento possa ser

descrito por um autômato finito. Entretanto, observe que aqui estamos permitido que o alfabeto contenha apenas símbolos 1, de maneira que os números primos são representados pelas strings 1^p , onde p é um primo. Também podemos provar algo mais “natural”: supor que o alfabeto de entrada é $\Sigma = \{0, 1\}$ e os primos são as strings binárias que representem números primos. Embora a prova disto seja um pouco complicada (ela fica como Exercício Opcional), vamos enunciar este teorema abaixo. No enunciado do teorema, relembramos que $N(w)$ é o número natural que a string binária w representa (e.g., se $w = 1001$, então $N(w) = 9$).

Teorema 4.1.4 $L_p = \{w \mid N(w) \text{ é um número primo}\}$ não é regular.

Prova: Exercício 4.5. \square

Exercício 4.1 Prove que $L_{RR} = \{\text{“strings da forma } ww^R\text{”}\}$ não é regular. ■

Exercício 4.2 Prove que $L_{EQ} = \{w \mid w \text{ tem o mesmo número de 0's e 1's}\}$ não é regular. ■

Exercício 4.3 Prove que $L_{EQ} = \{w \mid \text{o número de 0's em } w \text{ é o dobro do número de 1's}\}$ não é regular. ■

Exercício 4.4 $L_p = \{1^p \mid p \text{ é um quadrado perfeito}\}$ não é regular. ■

Exercício 4.5 (OPCIONAL) Prove que $L_p = \{w \mid N(w) \text{ é um número primo}\}$ não é regular. ■

4.2 Autômato com Pilha (AP)

No capítulo 3 nós apresentamos a definição de AFDs e definimos linguagens regulares como sendo linguagens aceitas por AFDs. Depois disso, demos uma habilidade a mais ao autômatos: a possibilidade de “advinhar” qual transição fazer. Em seguida, adicionamos a possibilidade do autômato fazer transições ϵ . Ainda assim, o conjunto de linguagens aceitas por tais autômatos continuou sendo as linguagens regulares. Agora vamos encontrar um cenário diferente. Vamos incrementar o nosso ϵ -AFN com uma habilidade que, conforme veremos adiante, o tornará mais poderoso, isto é, o tornará capaz de aceitar linguagens não regulares. A habilidade que vamos incluir ao autômato é a possibilidade de armazenar informação e recuperar informação em uma memória. O ponto chave aqui é que, embora o autômato tenha acesso a um tipo de memória bastante restrito (uma pilha de dados), este fato será suficiente para aumentar o seu poder de computação.

Primeiramente, observe que este novo modelo é capaz de realizar tarefas que um ϵ -AFN já realizava, ou seja, reconhecer linguagens regulares, pois podemos computar simplesmente ignorando o fato que temos agora uma pilha de dados. Entretanto, fazendo uso da pilha de dados, veremos que este modelo de computação é capaz de reconhecer linguagens que não são regulares.

4.2.1 O modelo matemático para autômatos com pilha

Definição 4.2.1 — Autômatos com Pilha (AP). Um autômato com pilha P é uma 7-tupla $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tal que:

Q, Σ, q_0, F : Tem a mesma interpretação que em um ϵ -AFN.

Γ é o alfabeto da pilha.

$Z_0 \in \Gamma$ é o símbolo inicial. Adicionalmente, temos que $Z_0 \notin \Sigma$.

$\delta : Q \times \Sigma \cup \{\epsilon\} \times \Gamma \rightarrow S$, tal que S é um conjunto de pares (q, γ) , tal que $q \in Q$ e $\gamma \in \Gamma^*$.

Vamos agora fazer a interpretação de como o modelo que acabamos de definir computa. Relembreamos antes que, a cada passo, um ϵ -AFN executava duas ações: (1) consumia um símbolo (possivelmente ϵ); (2) mudava de estado. Agora no caso dos APs, a cada passo, o autômato executa quatro ações: (1) consome um símbolo da string de entrada; (2) desempilha o símbolo do topo da pilha; (3) muda de estado; (4) empilha uma sequência de símbolos na pilha.

No caso dos ϵ -AFNs, cada passo da computação depende de um par (*estado atual, símbolo consumido*). No caso dos APs¹, cada passo da computação do AP é determinado por uma tripla com a seguinte forma: (*estado atual, símbolo consumido, símbolo desempilhado*). Por este motivo, observe que o domínio da função δ da Definição 4.2.1 é $Q \times \Sigma \cup \{\epsilon\} \times \Gamma$.

Neste modelo, após a mudança de estado o AP tem a habilidade de empilhar uma sequência de símbolos na pilha. Por este motivo, o contradomínio da função δ é definido como sendo $Q \times \Gamma^*$. Observe que se em uma determinada situação quisermos que o AP apenas faça uma transição sem empilhar nada, basta que façamos que o AP empilhe ϵ . O símbolo Z_0 da definição do modelo é, por convenção, o único símbolo presente na pilha no início da computação.

Considere a linguagem $L_{RR} = \{“\text{conjunto das strings da forma } ww^R”\}$. Observamos que esta linguagem não é regular (a demonstração deste fato é o objetivo do Exercício 4.1). O AP P_{RR} , da Figura 4.1 reconhece L_{RR} .

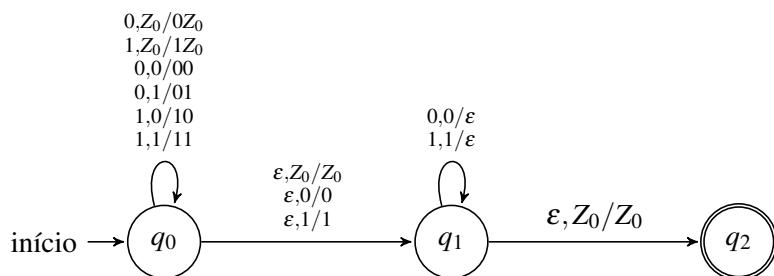


Figure 4.1: Diagrama do AP P_{RR} que aceita a linguagem L_{RR} .

Note que sobre as transições os rótulos aparecem da forma $a, b / w$, onde $a, b \in \Sigma$ e $w \in \Sigma^*$. Estes rótulos devem ser interpretados da seguinte maneira: a transição é executada consumindo o símbolo a da string, desempilhando o símbolo b e empilhando a string w . A sequência da símbolos de w é empilhada de trás para frente. Por exemplo, se $w = 01$, o símbolo 1 é empilhado primeiramente e fica abaixo do símbolo 0 na pilha.

Quando projetamos um AP para realizar uma tarefa é comum tomar cuidado para que o símbolo Z_0 sempre esteja no “fundo” da pilha para que possamos saber que a pilha está vazia. Isso não é obrigatório, mas é uma boa prática no projeto de APs, pois o AP, a cada passo, sempre remove

¹Estamos usando “AP” para nos referir aos autômatos com pilha. Entretanto, em alguns textos usa-se o acrônimo PDA, que vem do inglês *Pushdown Automata*.

um símbolo da pilha por padrão. Mas o que acontece quando o AP tenta fazer uma transição e a pilha está vazia? Aqui temos uma situação que não é muito diferente ε -AFNs tentando realizar uma transição que não está definida: o autômato morre. Isso nos faz lembrar de algo importante: o modelo da Definição 4.2.1 não é determinístico, uma vez que é uma generalização de um modelo não determinístico. Uma pergunta importante é se este não determinismo faz diferença, ou seja, se podemos fornecer uma versão determinística de um AP que aceite as mesmas linguagens que os APs que acabamos definir. Na Seção 4.2.4 daremos uma definição para APs determinísticos e veremos que tais APs **não** aceitam as mesmas linguagens aceitas por APs não determinísticos, embora ainda aceitem um conjunto maior de linguagens do que as linguagens regulares.

4.2.2 Computação com APs

A cada instante da computação de um AP, podemos descrever a situação em que o processo computacional se encontra, desde que saibamos o seguinte: (1) O estado q em que o AP se encontra; (2) A string w de símbolos restante que o AP ainda tem que processar; (3) A string γ de símbolos que está na pilha de dados. Durante a computação, a situação que um AP se encontra em um determinado instante é descrito por uma tripla contendo estes três elementos, chamada de *Descrição instantânea* do AP. Formalizamos isso a seguir.

Definição 4.2.2 — Descrição instantânea (ID). Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP e $q \in Q, w \in \Sigma^*, \gamma \in \Gamma^*$. Uma tripla (q, w, γ) é chamada de descrição instantânea (ID) de P .

Vamos usar agora o conceito de ID definir um passo computacional do autômato.

Definição 4.2.3 Seja um AP $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. O símbolo \vdash_P representa um passo computacional que leva um ID I_1 para um ID I_2 e é definido da seguinte maneira:

Se $(p, \alpha) \in \delta(q, a, X)$, então $\forall w \in \Sigma^*$ e $\forall \beta \in \Gamma^*$, temos $(q, aw, X\beta) \vdash_P (p, w, \alpha\beta)$.

No Capítulo 3, a definição da função $\hat{\delta}$ foi útil para que pudéssemos descrever matematicamente uma sequência arbitrariamente longa de transições de um autômato finito. O símbolo \vdash_P^* , que vamos definir a seguir, terá utilidade semelhante para APs.

Definição 4.2.4 Definimos agora \vdash_P^* indutivamente:

Base: $I \vdash_P^* I$ para qualquer ID I .

Indução: $I \vdash_P^* J$ se $\exists K$ tal que $I \vdash_P K$ e $K \vdash_P^* J$.

Exemplo 4.1 Sabemos que o AP P_{RR} da Figura 4.1 aceita a string 010010. Portanto, podemos escrever o seguinte: $(q_0, 010010, Z_0) \vdash_{P_{RR}}^* (q_2, \varepsilon, Z_0)$. ■

A linguagem de um AP é conjunto de strings que ele aceita. Usando o símbolo \vdash^* , temos uma maneira precisa de definir este conceito

Definição 4.2.5 Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP. Definimos a linguagem de P como sendo $L(P) = \{w \in \Sigma^* \mid (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha) \text{ para } q \in F \text{ e } \alpha \in \Gamma^* \text{ qualquer}\}$.

4.2.3 Aceitação por pilha vazia

Relembreamos que quando projetamos um AP, normalmente tomamos cuidado para que o fundo da pilha sempre contenha o símbolo Z_0 , pois o AP morre quando tenta fazer uma transição com

a pilha vazia. Entretanto, uma vez que temos ao nosso dispor uma pilha de dados, em algumas aplicações é interessante projetar o AP de maneira que ele morra exatamente quando terminar de ler a string. Alguns de nós com mais experiência em programação de computadores sabe que uma maneira bastante natural de se resolver certos problemas é usar uma pilha de dados e certificar-se que no final ela foi completamente esvaziada.

Neste contexto, a pergunta chave que queremos fazer agora é a seguinte: quais são as strings que fazem com que um determinado AP esvazie completamente sua pilha (e por consequência morra no passo seguinte)? Dado um AP P , vamos definir a seguir $N(P)$ como sendo o conjunto contendo toda string que faz o AP morrer com a pilha vazia. Observe que o conjunto $N(P)$ não é a linguagem do AP, embora, em algumas situações, este possa ser o caso.

Definição 4.2.6 Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP. Então $N(P) = \{w ; (q_0, w, Z_0) \vdash_P^* (q, \epsilon, \epsilon) \text{ para } q \in Q \text{ qualquer}\}$.

■ **Exemplo 4.2** Como o AP P_{RR} da Figura 4.1 nunca esvazia a pilha, então $N(P_{RR}) = \emptyset$. ■

Observe que as strings de entrada de um dado AP P caem em dois casos: as strings que esvaziam a pilha de P e as strings que não esvaziam a pilha de P . Assim como temos pensado no AP P como tendo um certo *poder de computação* capaz de distinguir se uma dada string pertence ou não pertence a $L(P)$, podemos também pensar que P também tem um certo poder de computação para distinguir se a dada string pertence ou não pertence a $N(P)$. A partir de agora nós vamos formalizar esta ideia e dizer que as strings de $N(P)$ são as strings que P aceita por pilha vazia. Observe que o fato de que P aceite por pilha vazia uma dada string w , não implica necessariamente que P aceite (no sentido da string estar na linguagem do AP) esta mesma string w . Entretanto, os seguintes teoremas mostram há uma certa equivalência entre os dois modos de aceitação de strings.

Teorema 4.2.1 Seja L a linguagem de um AP P . Então existe um AP P' tal que $N(P') = L$.

Teorema 4.2.2 Seja $N(P)$ o conjunto de strings aceitas por pilha vazia de AP P . Então existe um AP P' tal que $L(P') = N(P)$.

4.2.4 APs determinísticos

Vamos finalizar esta seção sobre APs seguindo a direção oposta que víhamos fazendo desde o Capítulo 3. Até agora víhamos generalizado o nosso modelo de computação. Agora, vamos definir um modelo de computação um pouco mais restrito do que a nossa definição de APs, que chamaremos de autômatos determinísticos com pilha de dados. Chamaremos este modelo de *Autômato Finito Determinístico com Pilha (APD)*.

Definição 4.2.7 — Autômatos Finitos Determinísticos com Pilha (APDs). Um APD é um AP cuja função de transição δ tem as seguintes restrições:

1. $|\delta(q, a, X)| \leq 1$
2. Se $\delta(q, a, X) \neq \emptyset$, então $\delta(q, \epsilon, X) = \emptyset$.

Sempre que definimos um novo modelo de computação a pergunta natural que nos vem é: qual o poder de computação deste novo modelo? O teorema a seguir responde tal pergunta.

Teorema 4.2.3 Existe uma linguagem L tal que $L = L(P)$ para algum AP P tal que não existe nenhum APD que aceite L .

Embora APDs tenham menos poder computacional do que APs, ainda assim o conjunto de linguagens decididas por APDs é maior do que o conjunto de linguagens regulares.

4.3 Gramáticas Livre de Contexto

Na Seção 3.6 nós vimos que podemos usar expressões regulares (ER) para representar linguagens. Mais precisamente, vimos que as linguagens representáveis por ERs são exatamente as linguagens aceitas por autômatos finitos. Neste seção vamos fazer algo semelhante. Vamos apresentar um outro formalismo matemático, chamado de *Gramáticas Livres de Contexto* (GLC), que também pode ser usado para representar linguagens. Na sequência veremos que as linguagens representáveis por GLCs são precisamente as linguagens aceitas por APs.

4.3.1 Definição formal de gramáticas livre de contexto

O objeto matemático que usaremos para representar linguagens nesta seção é uma quádrupla. Vamos primeiramente apresentar a definição matemática deste objeto e, em seguida, explicaremos como associamos uma linguagem a estas quádruplas.

Definição 4.3.1 Uma *Gramática Livre de Contexto* é uma quádrupla $G = (V, T, P, S)$ tal que:

- V é o conjunto de *variáveis*
- T é o conjunto de *símbolos terminais*.
- P é o conjunto de *regras de produção*, sendo que cada elemento de P é uma expressão da forma $X \rightarrow W$, onde $X \in V$ e W é uma string de $(V \cup T)^*$.
- S é a variável inicial, onde $S \in V$.

Para simplificar, chamaremos as vezes uma Gramática Livre de Contexto apenas de “gramática” e uma regra de produção apenas de “regra”.

■ **Exemplo 4.3** $G_P = (\{P\}, \{0, 1\}, A, P)$ é uma gramática onde A contém as seguintes regras:

$$\begin{aligned} P &\rightarrow \epsilon \\ P &\rightarrow 0 \\ P &\rightarrow 1 \\ P &\rightarrow 0P0 \\ P &\rightarrow 1P1 \end{aligned}$$

■

Na Seção 4.3.2 veremos como formalmente podemos associar linguagens a gramáticas, como a que acabamos de ver no Exemplo 4.3. Por enquanto vamos apenas nos certificar que entendemos exatamente que tipo de objeto matemático é uma gramática. Vamos a mais um exemplo:

■ **Exemplo 4.4** $G_1 = (\{I, E\}, \{a, b, 0, 1, +, *, (,)\}, A_1, E)$ onde as regras de A_1 são:

$$\begin{aligned} E &\rightarrow I \\ E &\rightarrow E + E \\ E &\rightarrow E * E \\ E &\rightarrow (E) \\ I &\rightarrow a \\ I &\rightarrow b \end{aligned}$$

$$I \rightarrow Ia$$

$$I \rightarrow Ib$$

$$I \rightarrow I0$$

$$I \rightarrow I1$$

■

Para simplificar, em vez de escrever a lista completa de regras de um dado conjunto de regras, é bastante comum utilizarmos uma notação mais compacta. No caso do Exemplo 4.4, ao invés de gastar dez linhas para descrever o conjunto de regras A_1 , poderíamos ter usado apenas duas linhas e o descrito da seguinte maneira:

$$E \rightarrow I|E+E|E*E|(E)$$

$$I \rightarrow a|b|Ia|Ib|I0|I1$$

De maneira semelhante, o conjunto de regras do Exemplo 4.3 é descrito por $P \rightarrow \epsilon|0|1|0P0|1P1$.

4.3.2 Derivações de uma gramática

Dada uma gramática G , veremos agora qual é a linguagem $L(G)$ que esta gramática descreve. Para chegarmos a tal definição, vamos definir o conceito de *derivação* de strings. A ideia é que as strings deriváveis de G são as strings que pertencem a linguagem $L(G)$.

Definição 4.3.2 Seja $G = (V, T, P, S)$ uma gramática e seja uma string $\alpha A\beta$ onde $A \in V$ e $\alpha, \beta \in (V \cup T)^*$. Seja $A \rightarrow \gamma$ uma regra de P . Então dizemos que $\alpha A\beta \Rightarrow_G \alpha\gamma\beta$

No exemplo acima, dizemos que a string $\alpha\gamma\beta$ pode ser *derivada* da string $\alpha A\beta$. Observe que tanto $\alpha\gamma\beta$ quanto $\alpha A\beta$ são strings cujos símbolos pertencem ao conjunto $V \cup T$. Estas strings são chamadas de *termos sentenciais*. As strings cujos símbolos pertencem apenas ao conjunto T são chamadas de *strings terminais*. Note que uma derivação pode ser aplicada apenas a strings que são termos sentenciais. Por outro lado, quando formos definir quais são as strings que pertencem a uma dada gramática, estaremos interessados apenas em strings terminais.

Antes de seguir em frente, vamos mostrar como usar a Definição 4.3.2 dizer formalmente que a $a*(a+b00)$ pode ser derivada da gramática G_1 do Exemplo 4.4.

■ **Exemplo 4.5** A string $a*(a+b00)$ pode ser derivada da gramática G_1 , pois:

$$\begin{aligned} E &\Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow a*(a+I) \Rightarrow \\ &a*(a+I0) \Rightarrow a*(aI00) \Rightarrow a*(a+b00) \end{aligned}$$

■

A próxima definição será útil para indicar que uma string pode derivada de uma dada gramática usando uma ou mais derivações.

Definição 4.3.3 Sejam $\alpha, \beta, \gamma \in (V \cup T)^*$, definimos indutivamente \Rightarrow_G^* :

BASE: $\alpha \Rightarrow_G^* \alpha$

INDUÇÃO: se $\alpha \Rightarrow_G^* \beta$ e $\beta \Rightarrow_G \gamma$, então $\alpha \Rightarrow_G^* \gamma$

No Exemplo 4.5 vimos que podemos derivar $a*(a+b00)$ de G_1 Usando uma série de derivações. Agora temos uma maneira formal de dizer isso: $E \Rightarrow_G^* a*(a+b00)$. Com isso podemos definir qual é linguagem associada a uma dada gramática.

Definição 4.3.4 — A Linguagem de uma Gramática. Dada uma gramática $G = (V, T, P, S)$, a linguagem de G é $L(G) = \{w \in T^* \mid S \Rightarrow_G^* w\}$.

4.3.3 Derivação mais a direita e mais a esquerda

Em algumas situações pode ser útil fixarmos que em uma derivação de uma string α nós sempre escolhemos aplicar a regra de produção à variável mais a esquerda presente em α (no caso da string α ter mais de uma variável). Tal derivação é dita *mais a esquerda*. Escrevemos \Rightarrow_{lm} e \Rightarrow_{lm}^* (assumindo que G é conhecida). Similarmente temos *derivações mais a direita* e podemos representá-las usando os símbolos \Rightarrow_{rm} e \Rightarrow_{rm}^* .

Exercício 4.6 Verifique que a derivação de G_1 do Exemplo 4.5 é mais a esquerda ■

Exercício 4.7 Forneça uma derivação mais a direita para a gramática G_1 do Exemplo 4.4. ■

Definição 4.3.5 — Linguagens Livre de Contexto. Se L é a linguagem de uma gramática livre de contexto G , então dizemos que L é uma *Linguagem Livre de Contexto*. Neste casos dizemos que a gramática G *gera* a linguagem L .

■ **Exemplo 4.6** As linguagens $L(G_p)$ e $L(G_1)$ das gramáticas dos Exemplos 4.3 e 4.4 são linguagens livre de contexto, pois são geradas por gramáticas livre de contexto. ■

4.3.4 Árvores de análise sintática

Exercício 4.8 Pesquise a respeito de árvores de análise sintática (também conhecidas como árvores de derivação ou *parse trees*) de gramáticas livres de contexto. ■

4.3.5 Ambiguidade de Gramáticas

Seja $G = (V, T, P, S)$ uma gramática e $w \in L(G)$. Se existem duas árvores de derivação diferentes para w , então dizemos que G é ambígua. Se existe exatamente uma árvore de derivação para cada string $w \in L(G)$, então G é uma gramática *não ambígua*.

OBSERVAÇÕES IMPORTANTES:

- Mesmo que uma string tenha várias derivações diferentes, isso não necessariamente quer dizer que a gramática seja ambígua. A gramática somente é ambígua se existir uma string que admita mais do que uma árvore de derivação.
- Por outro lado, no caso de apenas permitirmos derivações mais a esquerda e ainda assim pudermos obter mais do que uma derivação de uma dada string, podemos concluir que a gramática é ambígua. O mesmo pode ser dito no caso em que permitimos apenas derivações mais a direita.

Dada uma linguagem livre de contexto L , por definição existe uma gramática livre de contexto G tal que $L(G) = L$. Podemos nos perguntar se sempre é possível obter G que gere L tal que G não seja ambígua. A resposta é não, pois existem linguagens livre de contexto que são ditas *inerentemente* ambíguas. Mais precisamente, isso quer dizer que existem linguagens livre de contexto L tal que para toda gramática G tal que $L(G) = L$, a gramática G é ambígua. A linguagem $L = \{a^n b^n c^m d^m; n > 0, m > 0\} \cup \{a^n b^m c^m d^n; n > 0, m > 0\}$ é um exemplo de tal linguagem. Entretanto não vamos apresentar aqui a demonstração de que esta linguagem é *inerentemente* ambígua, pois isto é bastante complicado e está fora do escopo deste curso.

4.3.6 Equivalência entre APs e gramáticas livre de contexto

Enunciamos abaixo o teorema mais importante a respeito de APs e gramáticas:

Teorema 4.3.1 Uma linguagem L é livre de contexto \Leftrightarrow existe um AP que aceita L .

4.3.7 APDs e ambiguidade de gramáticas

O Teorema 4.3.1 mostra que o conjunto de linguagens aceitas por APs é exatamente o mesmo conjunto das linguagens expressa por gramáticas. Por outro lado, vimos anteriormente que o conjunto das linguagens aceitas por APs não é o mesmo que o conjunto de linguagens aceitas por APDs. Além disso, vimos que algumas gramáticas são inherentemente ambíguas. Em outras palavras, o mundo das linguagens livres de contexto é bem mais sutil do que o mundo das linguagens regulares. Enunciamos a seguir dois teoremas que mostram a relação entre ambiguidade de gramáticas e APDs.

Teorema 4.3.2 Se $L = L(P)$ para algum APD P , então existe uma gramática livre de contexto não ambígua G tal que $L(G) = L$.

Teorema 4.3.3 Existe uma gramática livre de contexto não ambígua G tal que não existe nenhum APD que aceite G .

4.3.8 Exercícios

Exercício 4.9 Use o Lema do Bombeamento (LB) para provar que a seguinte linguagem sobre $\Sigma = \{0, 1\}$ não é regular: $L = \{0^i 1^j \mid i > j\}$. ■

Exercício 4.10 Forneça um AP que aceite a linguagem $\{0^n 1^n \mid n \geq 1\}$. ■

Exercício 4.11 Considere o alfabeto Σ que contém os símbolos de “abertura” e “fechamento” de parêntesis, ou seja $\Sigma = \{(,)\}$. Forneça um AP que aceite a linguagem das strings de parêntesis平衡ados sobre σ . Por exemplo, a string $((())()$) deve ser aceita, enquanto a string $(())($ não deve ser aceita. ■

Exercício 4.12 Seja $A = \{a, b, c\}$ e seja L a linguagem $\{a^n b c^n \mid n \geq 1\}$ sobre o alfabeto A . Forneça um AP que aceite a linguagem L . ■

Exercício 4.13 Apresente um AP para a linguagem das strings palíndromas sobre o alfabeto $\Sigma = \{0, 1\}$. ■

Exercício 4.14 Forneça uma CFG para a linguagem $\{0^n 1^n \mid n \geq 1\}$. ■

Exercício 4.15 Observe que a linguagem da expressão regular $0^*1(0+1)^*$ é a mesma linguagem da gramática regular $G = (V, T, P, S)$ com as regras abaixo:

$$\begin{aligned}S &\rightarrow A1B \\A &\rightarrow 0A \mid \epsilon \\B &\rightarrow 0B \mid 1B \mid \epsilon\end{aligned}$$

Forneça uma derivação mais a direita e uma derivação mais a esquerda da string 00101. ■

Exercício 4.16 Considere a gramática $G = (V, T, P, S)$, onde $V = \{S\}$, $T = \{0, 1\}$ e as regras de P são:

$$S \rightarrow \epsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$$

Considere agora a seguinte definição: $S(G) = \{w \in (V \cup T)^* \mid S \Rightarrow_G^* w\}$. Com isso em mente, forneça um exemplo de string w tal que $w \in S(G) \setminus L(G)$. ■

Exercício 4.17 Forneça uma gramática que tem como linguagem expressões bem formadas em lógica proposicional. Lembre que em uma expressão bem formada em lógica proposicional pode ter variáveis x_1, x_2, x_3, \dots , operadores binários $\wedge, \vee, \Rightarrow, \Leftrightarrow$, operador unário \neg e abertura e fechamento de parênteses. ■

Exercício 4.18 Considere a gramática $G = (V, T, P, S)$, onde

- $V = \{ [\text{STMT}], [\text{IF-THEN}], [\text{IF-THEN-ELSE}], [\text{ASSIGN}] \}$
- $T = \{ a=1, \text{if, condition, then, else} \}$
- $S = [\text{STMT}]$

Onde as regras são:

$$\begin{aligned}[\text{STMT}] &\rightarrow [\text{IF-THEN}] \mid [\text{IF-THEN-ELSE}] \mid [\text{ASSIGN}] \\[\text{IF-THEN}] &\rightarrow \text{if condition then } [\text{STMT}] \\[\text{IF-THEN-ELSE}] &\rightarrow \text{if condition then } [\text{STMT}] \text{ else } [\text{STMT}] \\[\text{ASSIGN}] &\rightarrow a=1\end{aligned}$$

A gramática G acima é naturalmente o fragmento de uma linguagem de programação, entretanto G é ambígua. Mostre que a gramática é ambígua e forneça uma outra gramática não ambígua equivalente. ■

Exercício 4.19 Para responder as questões abaixo assuma que o alfabeto é $\Sigma = \{a, b, c\}$.

- Forneça um AP P tal que $L(P) = \{a^m b^n c^n; m, n \geq 0\}$.
- Forneça uma Gramática G tal que $L(G) = \{a^n b^n c^m; m, n \geq 0\}$.

Exercício 4.20 (OPCIONAL) Prove que $L = \{a^n b^n c^n; n \geq 0\}$ não é livre de contexto. ■

Exercício 4.21 A interseção de duas linguagens regulares também é uma linguagem regular. Por ter esta propriedade, as linguagens regulares são ditas *fechadas sob interseção*. Nesta questão você deve provar que esta propriedade de ser fechada sob interseção não vale para linguagens livre de contexto, ou seja, você deve provar que existem linguagens livre de contexto L_1 e L_2 tal que $L_1 \cap L_2$ não é uma linguagem livre de contexto. Dica: Tente usar o fato de que a linguagem da questão 4.20 não é livre de contexto. ■

Parte 2: Máquinas de Turing e Computabilidade

5 A Máquina de Turing 61

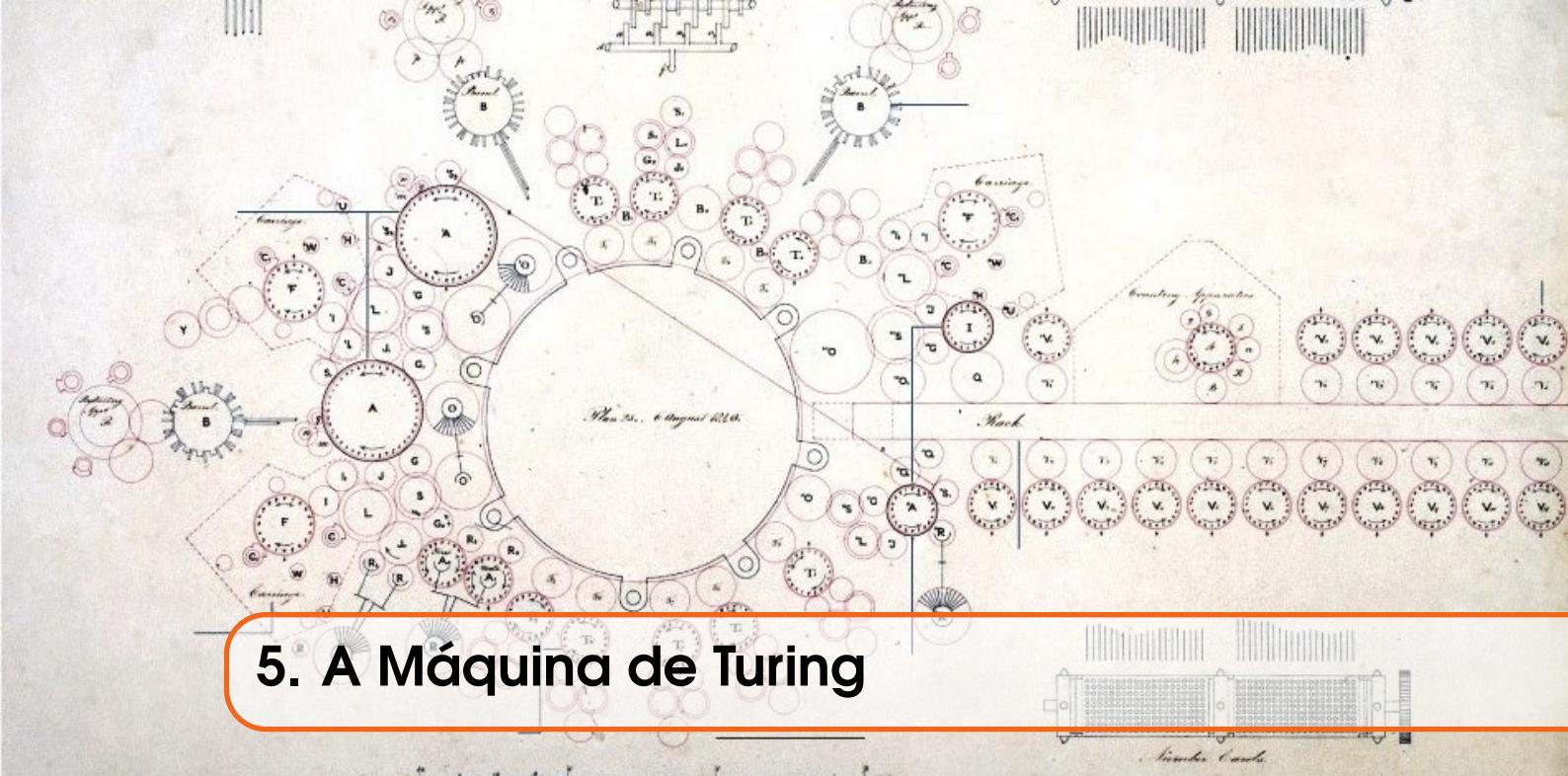
- 5.1 Revisão: problemas computacionais
- 5.2 Definição da Máquina de Turing
- 5.3 Um Algoritmo é uma Máquina de Turing que sempre para

6 A Tese de Church-Turing 73

- 6.1 Perspectiva histórica
- 6.2 Máquinas de Turing são equivalentes a linguagens de programação
- 6.3 Máquinas de Turing e outros modelos de computação
- 6.4 A Tese de Church-Turing e suas interpretações
- 6.5 A Tese de Church-Turing estendida

7 Computabilidade 83

- 7.1 Funções computáveis
- 7.2 Codificando objetos matemáticos em binário
- 7.3 Máquinas de Turing, pseudo-códigos, generalidade e especificidade
- 7.4 O problema da Parada
- 7.5 A Máquina de Turing Universal
- 7.6 Máquinas de Turing não determinísticas (MTN)
- 7.7 Exercícios



5. A Máquina de Turing

Neste capítulo iremos apresentar a definição do modelo matemático proposto por Alan Turing em 1936, conhecido como Máquina de Turing. Usaremos este modelo para fornecer uma definição precisa para o que entendemos como algoritmo. No Capítulo 7 veremos como usar este modelo para obter uma definição precisa para o que entendemos como computador.

5.1 Revisão: problemas computacionais

Antes de falarmos de Máquinas de Turing, vamos brevemente relembrar algo que discutimos no Capítulo 2, que é a ideia de que problemas computacionais podem ser vistos como linguagens. Vimos, por exemplo, que o problema da divisibilidade por 3 e o problema da primalidade podem ser modelados pelas seguintes duas linguagens sobre o alfabeto binário: $L_3 = \{w \mid N(w) \text{ é um múltiplo de } 3\}$ e $L_p = \{w \mid N(w) \text{ é um número primo}\}$. A ideia central é que são precisamente as strings de L_3 que incorporam a propriedade “ser divisível por 3” e são precisamente as strings que não estão em L_3 que incorporam a propriedade “não ser divisível por 3”. Portanto, podemos ver a linguagem L_3 como um objeto matemático que captura a essência do problema de divisibilidade por 3. De maneira análoga, a linguagem L_p captura a essência do problema de teste de primalidade.

REFLETINDO UM POUCO: PROBLEMAS SÃO SEMPRE LINGUAGENS?

Uma simplificação que estamos fazendo aqui é que sempre estamos lidando com problemas para os quais a resposta é SIM ou NÃO, ou seja, problemas para os quais as respostas consistem de apenas um bit de informação (afinal, SIM ou NÃO podem ser vistos como 1 ou 0). Problemas com estas características são conhecidos como *problemas de decisão*.

Nós sabemos perfeitamente que nem todo problema computacional é um problema de decisão. Por exemplo, considere o *problema do caminho mínimo em grafos*. Dada uma tripla (G, u, v) , sendo que G é um grafo e u, v são vértices de G , a resposta que queremos produzir é o menor caminho conectando os vértices u e v no grafo G . Neste caso, a resposta do problema é um objeto matemático que não pode ser representado por apenas um bit. Mais precisamente, a

resposta para este problema é uma sequência de vértices v_1, v_2, \dots, v_k tal que $v_i v_{i+1}$ são arestas presentes em G . Observe que podemos representar uma sequência de vértices usando algum esquema de codificação em binário, ou seja, podemos pensar que a resposta é uma string binária respeitando determinadas propriedades.

Um outro exemplo de problema que também não encaixa no modelo de problema de decisão é o problema de multiplicar dois números naturais a e b . A resposta para este problema é o produto $a \cdot b$. Neste caso, novamente, podemos pensar que a resposta desejada é uma string, mais precisamente a string w tal que $N(w) = a \cdot b$. Nos capítulos subsequentes lidaremos com tais tipos de problemas mais gerais (i.e., problemas para os quais a resposta pode ser uma string com mais de um bit de informação), entretanto, é importante salientar que mesmo no cenário restrito a problemas de decisão já seremos capazes de explorar fundamentos e limites da computação.

A seguir, vamos formalizar a correspondência entre linguagens e problemas:

Definição 5.1.1 Um *problema de decisão* é uma linguagem sobre o alfabeto binário.

Do ponto de vista formal, dizemos que encontramos uma solução para um problema de decisão L quando apresentamos um “objeto matemático” que possa ser usado para determinar sistematicamente se uma dada string pertence ou não pertence à linguagem L . Em outras palavras, dizemos que **solucionamos** o problema quando apresentamos uma definição matemática de algoritmo para resolver o problema. Por exemplo, sabemos que para o problema L_3 existe um AFD que é capaz de distinguir números divisíveis por 3 de números que não são divisíveis por 3. Ou seja, para apresentarmos uma solução algorítmica para L_3 , não precisamos do poder computacional de um AP e, menos ainda, do poder computacional de linguagens de programação modernas, uma vez que um mero AFD é o suficiente esta tarefa. Por outro lado, vimos que AFDs não são capazes de resolver o problema L_p , ou seja, não são capazes de fazer teste de primalidade.

No Capítulo 4 nós estudamos APs, que são modelos matemáticos capazes de expressar algoritmos que AFDs não são capazes. Nós não exploramos APs mais a fundo, pois isto está fora do escopo deste material, mas é importante ressaltar que é possível demonstrar existem vários problemas que também não podem ser resolvidos por APs, incluindo a tarefa de testar a primalidade de um número. Entretanto sabemos que é fácil escrever um algoritmo em nossa linguagem de programação favorita para testar se um dado número é primo. Em outras palavras, **existem algoritmos** que não podem ser escritos na forma de AFDs ou APs, mas que podem ser expressos de maneira formal (afinal, linguagens de programação são formais e precisas), e, portanto, AFDs e APs são modelos matemáticos que não são capazes de expressar todos os algoritmos possíveis e imagináveis.

Na próxima seção veremos a definição de modelo das Máquinas de Turing, um modelo capaz de expressar qualquer algoritmo que possamos escrever em qualquer linguagem de programação conhecida. No Capítulo 6 discutiremos a tese científica que afirma que Máquinas de Turing não são apenas equivalentes a qualquer linguagem de programação conhecida, mas que são capazes de representar qualquer computação concebível.

CONTEXTO HISTÓRICO: AFDs, APs E MÁQUINAS DE TURING

Neste texto nós fomos introduzindo modelos de computação que são incrementalmente mais poderosos. Começamos com AFDs (que são equivalentes a AFNs e ε -AFNs), seguidos por APs e, neste capítulo, vamos introduzir Máquinas de Turing. Apresentar estes modelos nesta sequência é interessante do ponto de vista pedagógico, mas não reflete a sequência histórica em

que estes modelos foram aparecendo na literatura científica. Curiosamente, Alan Turing propôs seu modelo na década de 30, enquanto que os modelos de computação vistos nos Capítulos anteriores apareceram na literatura por volta da década de 50 e 60.

5.2 Definição da Máquina de Turing

O modelo que veremos nesta seção, chamado de Máquina de Turing, é semelhante a um AFD adicionado de uma fita de dados. A Figura 5.1 ilustra abstratamente o funcionamento de um AFD, um AP e uma Máquina de Turing.

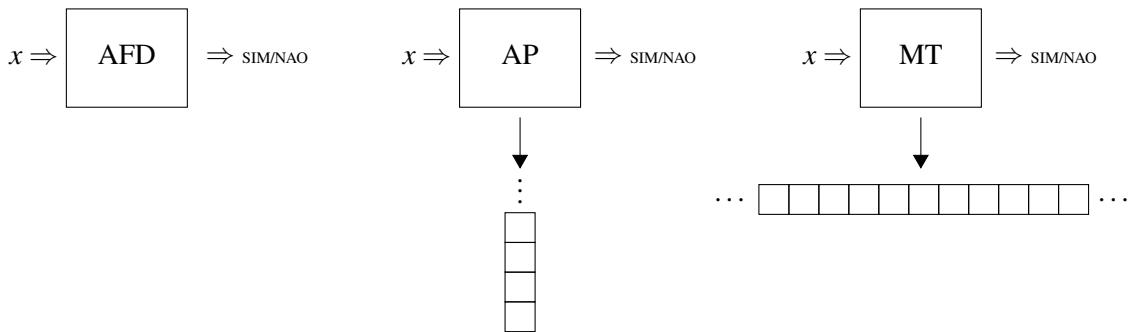


Figure 5.1: Comparação entre um AFD, um AP e uma Máquina de Turing. Os AFDs são máquinas de estados e APs são máquinas de estados com acesso a uma memória em forma de pilha de dados. A Máquina de Turing é essencialmente uma máquina de estados com uma memória em forma de *fita* de dados. A máquina pode acessar diferentes posições desta fita e ler/escrever um símbolos em tais posições. Esta figura compara os três modelos em um nível abstrato e não captura alguns detalhes de baixo nível do modelo (por exemplo, no modelo específico de Máquina de Turing que iremos trabalhar, por questões de conveniência, iremos assumir que a computação inicia com a string x posicionada na fita de memória).

Definição 5.2.1 — Máquina de Turing (MT). Uma *Máquina de Turing* é uma 7-tupla $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, tal que:

- Q é o conjunto finito de estados
- Σ é o alfabeto de entrada.
- Γ é o alfabeto da fita, tal que $\Sigma \subseteq \Gamma$.
- $\delta : (Q \setminus F) \times \Gamma \rightarrow Q \times \Gamma \times D$ é uma função parcial e $D = \{L, R, S\}$.
- q_0 é o estado inicial
- B é o símbolo especial chamado de *símbolo branco*.
- $F \subseteq Q$ é o conjunto de estados finais.

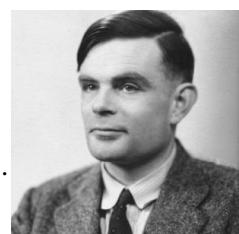


Figure 5.2: Alan Turing

Algo que mencionamos no início deste curso, mas que gostaríamos de reforçar, é que de agora em diante, exceto quando explicitamente dito o contrário, estaremos assumindo que Σ é sempre o alfabeto binário. Por questões de simplicidade, vamos assumir que a string x de entrada no início da computação encontra-se localizada na fita de dados¹. Isso simplifica um pouco a nossa definição

¹Na literatura podemos encontrar algumas variações de modelos de Máquinas de Turing diferentes do definido aqui. O importante é que todas estas variações acabam tendo o mesmo poder de computação do nosso modelo.

da função δ (lembre que o domínio da função δ de APs era uma tripla (estado, símbolo, símbolo) e agora o domínio da função δ de MTs é apenas um par (estado, símbolo). A cada momento a máquina estará em um determinado estado e terá acesso a uma posição específica da fita e, em tal situação, o vocabulário que usaremos é o seguinte: diremos que a *cabeça de leitura* da máquina está *escaneando* uma determinada *célula* da fita de dados.

5.2.1 O funcionamento de uma Máquina de Turing

Vamos agora interpretar em detalhes o modelo matemático para entender como o processo de computação ocorre. Suponha que a função $\delta(q, X)$ retorna (q', X', d) . Neste caso, o que acontece é que a se máquina estiver no estado q com o símbolo X na célula sendo escaneada na fita, então ela fará uma transição para o estado q' , sobreescrivendo X na fita pelo símbolo X' e moverá sua cabeça de leitura da seguinte maneira: (1) Se $d = L$, então a cabeça de leitura se moverá para a esquerda (ou seja, no próximo passo a máquina estará escaneando a célula da fita do lado esquerdo da célula atual); (2) Se $d = R$, então a cabeça de leitura se moverá para a direita; (3) Se $q = S$, então a cabeça de leitura continuará na posição corrente.

Se a string de entrada é $x = x_1x_2\dots x_n$, assumiremos que no início da computação x se encontra na fita e a cabeça de leitura da máquina esta posicionada sobre x_1 . Além disso, por definição, todos os demais símbolos da fita (antes de x_1 depois de x_n) são símbolos B . A Figura 5.3 exemplifica isso.

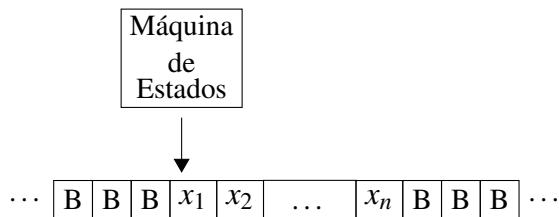


Figure 5.3: Máquina de Turing escaneando o primeiro símbolo da string $x = x_1x_2\dots x_n$ armazenada na fita.

Observe que os conceitos de fita e de cabeça de leitura da máquina não estão aparentes na definição da MT, que é apenas uma 7-tupla de objetos matemáticos, como conjuntos e elementos, e uma função “amarrrando” estes objetos de determinada maneira específica. As ideias de fita e cabeça de leitura podem ser vistas como parte da interpretação de como o modelo computa ou como intuições do que seria um objeto físico que o modelo matemático descreve. Algo importante de observar a respeito de Máquinas de Turing, é que o objeto físico correspondente ao objeto matemático é extremamente simples: máquina de estados finita com uma fita de memória.

VACAS ESFÉRICAS NO VÁCUO

Uma frase que físicos bem humorados gostam de usar é a seguinte: “Considere uma vaca esférica no vácuo”. A ideia é brincar um pouco com a ideia de que muitos argumentos são propostos usando objetos exageradamente simples em condições ideais. Um objeto simples como uma máquina de estados adicionada de uma fita de dados não deixa de parecer com a vaca esférica no vácuo dos cientistas da computação. A verdade é que, de fato, a simplicidade é um dos atrativos das Máquinas de Turing.

Entretanto, o que torna este objeto idealizado realmente útil não é apenas a sua simplicidade (afinal, AFDs são ainda mais simples!), mas o fato de que ele, **apesar** da simplicidade, é tão poderoso quanto qualquer outro modelo conhecido para representar algoritmos.

Dissemos que MTs são “semelhantes” a DFAs (ao invés “exatamente” AFDs) com a adição de uma fita de dados. O que ocorre é que há uma pequena diferença na máquina de estados de MTs em

relação de AFDs. Em Máquinas de Turing, a função δ não precisa estar definida em todos os pares (q, X) . Entretanto, é importante ressaltar que vamos tratar MTs como modelos determinísticos de computação (ao contrário de alguns outros modelos, como AFNs, por exemplo, em que a função δ poderia não estar definida para algumas entradas).

No caso de MTs, quando a função δ não está definida em um elemento de (q, X) , a MT irá finalizar sua execução. Neste caso, diremos que a MT *para* Observe que uma máquina “parar” pode ser visto como um passo puramente determinístico, i.e., a computação terminou, independente da entrada ter sido lida inteiramente ou não².

Algo importante de lembrarmos é que a causa do comportamento não determinístico de AFNs era a possibilidade de haver mais do que uma transição definida em algumas pares (q, a) e não o fato de algumas transições estarem indefinidas (este fato é explorado no Exercício 3.12). O que ocorre em nossa definição de MTs é que não há mais de uma transição definida em um certo par (q, X) , tornando o comportamento da máquina determinístico. Dada uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, a cada aplicação da função $\delta(q, X)$ ocorre exatamente um dos dois casos abaixo:

- (1) A função δ é definida em (q, X) . Neste caso o comportamento da MT é unicamente determinado pela tripla de $Q \times \Gamma \times D$ que a função δ retorna;
- (2) A função δ não é definida em (q, X) . Neste caso o comportamento da MT também é unicamente determinado, pois a máquina só tem uma escolha, que é parar sua execução. Na Seção 5.2.3 veremos que o estado em que MT parou sua execução (final ou não) é que vai definir se M aceita a string de entrada.

5.2.2 Diagrama de estados de uma Máquina de Turing

Podemos representar uma MT usando diagramas semelhantes aos diagramas de autômatos vistos anteriormente neste curso. A Figura 5.4, apresenta um exemplo de um diagrama de uma MT. Observe que neste diagrama há uma transição ligando q_0 à q_1 com o rótulo $0/XR$. Isto significa que se a máquina estiver no estado q_0 com a cabeça de leitura lendo um símbolo 0 na fita, então a máquina muda para o estado q_1 , reescreve o símbolo 0 com o símbolo X e move a cabeça de leitura para direita. De maneira geral, temos:

Notação 5.1. No diagrama de uma Máquina de Turing, uma transição de q_i para q_j com rótulo A/Bd indica que se a máquina estiver no estado q_i com a cabeça de leitura escaneando um símbolo A na fita de dados, a máquina muda para o estado q_j , reescreve o símbolo A com o símbolo B e faz o seguinte com a cabeça de leitura: (1) Se $d = R$, então move a cabeça para a direita; (2) Se $d = L$, então move a cabeça para a esquerda; (3) Se $d = S$, então deixa a cabeça de leitura imóvel.

Ainda não definimos o exatamente o que é a linguagem de uma dada MT e o que significa uma MT “aceitar” ou “rejeitar” uma string, mas antecipamos que daremos definições semelhantes às definições de outros modelos de computação. Ainda assim, observe que as únicas strings que fazem a MT da Figura 5.4 atingir seu estado final são as strings que pertencem a linguagem $L = \{0^n 1^n \mid n \geq 1\}$. Adicionalmente, note que uma vez que a máquina atinge o estado final, ela **obrigatoriamente** para sua execução (afinal, não há transições definidas no estado final).

Exercício 5.1 O exemplo da MT da Figura 5.4 é didático por ser simples. Entretanto, ele pode não ser tão interessante pelo seguinte motivo: A linguagem $L = \{0^n 1^n \mid n \geq 1\}$ é Livre de Contexto, ou seja, não precisaríamos de todo poder de uma Máquina de Turing para decidí-

²Isso é diferente do que ocorre com AFNs em que pensamos em termos de ramos que “morrem”, dentre os muitos ramos de computações possíveis. Mais adiante, usaremos também o termo *morrer* no contexto de Máquinas de Turing *não* determinísticas (em tais casos o conceito de morrer terá uma interpretação semelhante ao conceito visto em AFNs).

la. Por outro lado, é possível demonstrar que a linguagem $L_{abc} = \{a^n b^n c^n \mid n \geq 1\}$ não é livre de contexto (a demonstração deste fato está fora do escopo deste curso) sobre o alfabeto $\Sigma = \{a, b, c\}$. Apresente uma MT que decida a linguagem L_{abc} . ■

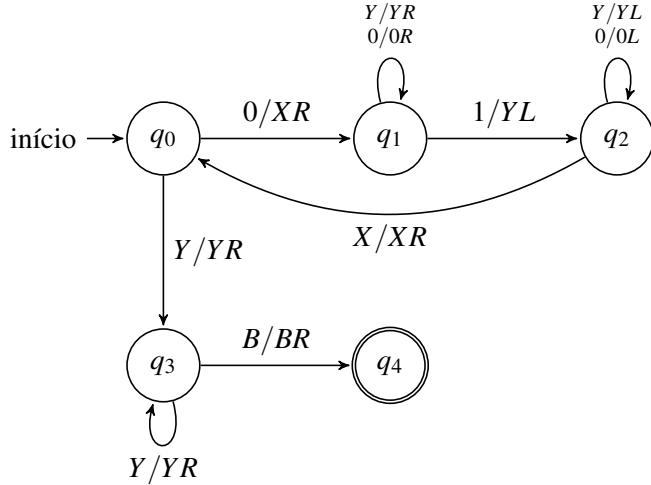


Figure 5.4: Diagrama de estados de uma Máquina de Turing.

5.2.3 Linguagem de uma Máquina de Turing

De maneira semelhante ao que fizemos com APs, veremos o processo de computação de uma Máquina de Turing como uma sequência de descrições instantâneas.

Definição 5.2.2 — Descrição Instantânea (ID). Dada uma Máquina de Turing $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, uma *Descrição Instantânea* de M é uma string $\alpha q \beta$ tal que $\alpha, \beta \in \Gamma^*$ e $q \in Q$.

Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma Máquina de Turing, $X_1 \dots X_n \in \Gamma^*$ e $q \in Q$. A descrição instantânea $X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n$ indica que M encontram-se no estado q , com a string $X_1 \dots X_n$ em sua fita de dados e com a cabeça de leitura posicionada sobre o símbolo X_i . Além disso, a fita contém uma sequência infinita de símbolos B tanto à esquerda de X_1 e quanto à direita de X_n . Note que alguns símbolos X_i podem ser eventualmente iguais a B .

O símbolo \vdash_M definido a seguir, representa um passos computacional de uma MT.

Definição 5.2.3 Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma Máquina de Turing e $X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n$ um ID de M . Se $\delta(q, X_i) = (p, Y, L)$, então escrevemos

$$X_1 X_2 \dots X_{i-1} q X_i X_{i+1} \dots X_n \vdash_M X_1 X_2 \dots X_{i-2} p X_{i-1} Y X_{i+1} \dots X_n.$$

Exceto no caso em que $i = 1$ e no caso em que $i = n$ e $Y = B$. Nestes casos temos o seguinte.

- (1) Se $i = 1$, então escreveremos $q X_1 \dots X_n \vdash_M p B Y X_2 \dots X_n$
- (2) Se $i = n$ e $Y = B$, então escreveremos $X_1 X_2 \dots X_{n-1} q X_n \vdash_M X_1 X_2 \dots X_{n-2} p X_{n-1}$

Note que a definição acima reflete um passo da computação da MT e o movimento para a esquerda. Precisamos definir também os casos em que é válido escrever $I_1 \vdash_M I_2$ no caso em que $\delta(q, X_i) = (p, Y, R)$ e no caso em que $\delta(q, X_i) = (p, Y, S)$. Este é o objetivo do exercício acima.

Exercício 5.2 Apresente definições de quando é possível escrever $I_1 \vdash_M I_2$, no caso em que $\delta(q, X_i) = (p, Y, R)$ e no caso em que $\delta(q, X_i) = (p, Y, S)$. ■

De maneira semelhante a PDAs, podemos definir o símbolo \vdash_M^* da seguinte maneira:

Definição 5.2.4 Dada uma MT M , o símbolo \vdash_M^* é definido indutivamente:

Base: $I \vdash_M^* I$ para qualquer ID I de M .

Indução: $I \vdash_M^* J$ se $\exists K$ tal que $I \vdash_M K$ e $K \vdash_M^* J$.

A seguinte definição será útil várias situações.

Definição 5.2.5 — IDs iniciais e finais. Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ e $w \in \Sigma^*$. O ID q_0w é chamado de *ID inicial de M com w*. Se $p \in F$, então qualquer ID da forma $\alpha p \beta$, tal que α, β são strings de Γ^* quaisquer, é chamado de *ID final de M*.

Exercício 5.3 Seja M a MT da Figura 5.2.2 e $w_1 = 000111$ e $w_2 = 011$. Responda as seguintes questões com relação a Descrições Instantâneas:

- Qual é o ID inicial de M com w_1 ?
- Apresente a sequência de IDs definida pela computação de M quando a string w_1 é fornecida como entrada. Faça o mesmo para M com a string w_2 fornecida como entrada.
- Existe um ID final de M com w_1 ?
- Existe um ID final de M com w_2 ?
- Existe mais do que um ID final de M com w_1 ?

O próximo passo é definir o conceito de aceitação de strings e de linguagens por Máquinas de Turing. Em seguida, vamos definir o conjunto de todas a linguagens aceitas por MTs como linguagens *recursivamente enumeráveis*.

Definição 5.2.6 — Strings aceitas por MTs. Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma Máquina de Turing. Dizemos que uma string $w \in \Sigma^*$ é *aceita* por M se $q_0w \vdash_M^* I_F$, tal que I_F é um ID final de M .

Definição 5.2.7 — Strings rejeitadas por MTs. Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma Máquina de Turing. Dizemos que uma string $w \in \Sigma^*$ é *rejeitada* por M se $q_0w \vdash_M^* I_N$, tal que I_N não é um ID final de M e máquina para ao atingir o ID I_N .

Observe que se uma string w não é aceita por uma máquina M , não podemos concluir imediatamente que w é **rejeitada** por M , pois pode ocorrer da MT continuar executando indefinidamente sem nunca parar. Porém, por definição, ou uma string é **aceita** ou **não é aceita**. O que ocorre é que quando dizemos que uma string w é rejeitada, estamos nos referindo a um caso particular de **não aceitação** da string.

Exercício 5.4 Apresente a definição formal e o diagrama de uma MT M tal que, para qualquer strings w fornecida como entrada, a máquina M não aceita nem rejeita w . ■

Definição 5.2.8 — Linguagens aceitas por MTs. Dada uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, a linguagem $L(M) = \{w \in \Sigma^* \mid q_0 w \vdash_M^* I_F\}$, tal que I_F é um ID final de M é chamada de *linguagem de M* ou *linguagem aceita* por M .

Definição 5.2.9 — Linguagens Recursivamente Enumeráveis. Se L é uma linguagem aceita por alguma Máquina de Turing, então a linguagem L é dita *Recursivamente Enumerável*. O conjunto de todas as linguagens recursivamente enumeráveis é denotado por \mathcal{RE} .

Neste momento é bom parar para refletir um pouco sobre os seguintes pontos:

- Ao contrário de AFDs e APs, na computação com Máquinas de Turing não existe o conceito de “terminar de ler a string de entrada”. Com isso, existe a possibilidade de que uma máquina continue computando indefinidamente em algumas circunstâncias, ou seja, a máquina pode ficar em *loop infinito*. Ou seja, existem strings que não são nem aceitas nem rejeitadas pela máquina em questão.
- Mais precisamente, como a função δ é sempre indefinida para estados finais (note que F é excluído do domínio de δ na Definição 5.2.1), a máquina sempre para quando a string é aceita. Uma consequência disso é que se $L \in \mathcal{RE}$, então existe uma MT M tal que $\forall w \in L$, a máquina M aceita w e para. Por outro lado, observe que se uma dada string $w \notin L$ que não é aceita por M , então isso pode significar duas coisas: M pode ter parado sua execução em um estado que não é final (i.e., M rejeitou w) ou M pode ter ficado em loop infinito.
- A nossa busca por uma definição matemática formal para um algoritmo é essencialmente uma busca por uma definição genérica do que seja um procedimento determinístico que retorne a solução para qualquer instância de um dado problema em um número **finito** de passos. Com isso, essa possibilidade das MTs continuarem computando indefinidamente em alguns casos não parece desejável. Da fato, estaremos particularmente interessado no conjunto das MTs que sempre param depois de uma quantidade finita de passos. Entretanto, a possibilidade deste nosso modelo matemático ser capaz de expressar procedimentos que possam rodar indefinidamente será útil adiante.

5.3 Um Algoritmo é uma Máquina de Turing que sempre para

A partir de agora passaremos usar os termos *MT que sempre para* e *Algoritmo* como sinônimos. Entretanto, como podemos especificar MTs fiquem em loop infinito, vamos tomar o seguinte cuidado: quando usarmos o apenas o termo “Máquina de Turing”, sem especificamente dizer que a máquina sempre para, não *necessariamente* estaremos nos referindo a um algoritmo.

Definição 5.3.1 — Algoritmo. Um *Algoritmo* é uma Máquina de Turing que sempre para.

Definição 5.3.2 — Decidindo Problemas. Seja L um problema de decisão. Se existe um Algoritmo M tal que $L(M) = L$, dizemos que M decide L .

Definição 5.3.3 — Linguagens Recursivas. Se existe MT que decide uma dada linguagem L , dizemos que L é uma *Linguagem Recursiva*. O conjunto de todas as linguagens recursivas é denotado por \mathcal{R} . Usaremos também o termo *Linguagem Decidível* para ser referir a uma Linguagem Recursiva.

Note que o conjunto das Linguagens Recursivas é o conjunto dos Problemas (de decisão) Decidíveis. Observe também que se L é **decidível** por uma MT M , então por definição a máquina M tal que $L = L(M)$ tem a propriedade de sempre parar (i.e., nunca fica em loop infinito). Isso

não necessariamente quando nos referimos a linguagens aceitas por Máquinas de Turing. Mais precisamente, observe que se $x \notin L(M)$, quando a string x é fornecida como entrada M rejeita a entrada.

Exercício 5.5 Apresente uma MT M_3 que aceite, mas que não decida a seguinte linguagem:
 $L_3 = \{w \in \Sigma^* \mid N(w) \text{ é um múltiplo de } 3\}$. ▀

Exercício 5.6 Com relação a MT M_3 e a linguagem L_3 do Exercício 5.5 responda:

- (a) A existência da máquina M_3 implica que $L_3 \in \mathcal{RE}$? Justifique sua resposta.
- (b) Como M_3 não decide L_3 (apesar de aceitar esta linguagem), é possível concluir que $L_3 \notin \mathcal{R}$? Justifique sua resposta.
- (c) Caso a afirmação do item (b) esteja incorreta, ou seja, a existência de M_3 não necessariamente diz algo sobre L_3 estar ou não estar em \mathcal{R} , diga qual a relação de L_3 com \mathcal{R} e justifique sua resposta (ou seja, prove que $L_3 \in \mathcal{R}$ ou prove que $L_3 \notin \mathcal{R}$, dependendo de qual caso for verdade). ▀

Exercício 5.7 Mostre que $\mathcal{R} \subseteq \mathcal{RE}$. ▀

MÁQUINAS DE TURING E ALGORITMOS

No Capítulo 6 veremos que o poder computacional de uma Máquina de Turing é equivalente ao poder computacional de um programa escrito em uma linguagem de programação moderna, ou seja, qualquer programa já escrito, ou que venha a ser escrito, em linguagens como as que usamos hoje, poderia ser escritos neste modelo matemático proposto em 1936.

Algo importante que devemos antentar é que, quando usamos Máquinas de Turing na definição de algoritmos, não estamos dizendo a Máquina de Turing é o formalismo mais conveniente para se escrever algoritmos (caso não esteja convencido disto, pegue um algoritmo escrito em uma linguagem de alto nível, como Python e tente reescrevê-lo em forma de Máquina de Turing!), mas, ao invés disso, estemos dizendo que MTs são capazes de expressar qualquer algoritmo que possamos conceber.

A vantagem de trabalharmos com Máquinas de Turings é precisamente o fato de que podemos provar teoremas genéricos sobre algoritmos usando uma mera 7-tupla (independente do fato de que existem infinitos algoritmos longos e complicados, e que podem fazer chamadas recursivas disparando *threads* em paralelo e diversas outras complicações que poderíamos ter que lidar, se estivéssemos usando uma linguagem de alto nível). Em outras palavras, todos os programas concebíveis, independente de quão complicado podem ser, são instâncias particulares de uma 7-tupla da Definição 5.2.1.

Uma pergunta que responderemos no Capítulo 7 é a seguinte. Será que existe uma linguagem que esteja em \mathcal{RE} , mas que não esteja em \mathcal{R} ? Uma outra pergunta próxima a esta é a seguinte: existe alguma linguagem qualquer que não esteja contida em \mathcal{RE} ?

5.3.1 Exercícios

Exercício 5.8 Forneça uma MT $M_{\text{COPY}} = (C, \Sigma, \Gamma, \delta_{\text{COPY}}, c_0, B, F_{\text{COPY}})$ que tenha o seguinte comportamento quando uma string x é fornecida como entrada. A máquina deve adicionar ao fim da string x mais uma cópia de x (ou seja, a fita deverá conter xx), retornar a cabeça de leitura para a posição inicial. Em seguida a máquina deve aceitar e parar. ■

Exercício 5.9 Responda se existe uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ tal que:

$$\forall x \in \Sigma^*, q_0 w \vdash_M^* pww, \text{ onde } p \in F$$

Exercício 5.10 Observe que a M_{COPY} a MT da Questão 5.8 não é o tipo máquina que normalmente temos projetado durante o curso, ou seja, não é uma máquina que testa se uma dada string de entrada tem ou não certa propriedade e no fim aceita ou rejeita a string de acordo com essa propriedade. Em outras palavras, o conceito de qual é a linguagem da máquina M_{COPY} não é algo central para o tipo de computação que ela realiza. Ainda assim, a definição matemática $L(M_{\text{COPY}})$ é bem precisa e, portanto o conjunto $L(M_{\text{COPY}})$ é bem definido. Pergunta: qual é linguagem $L(M_{\text{COPY}})$ da máquina M_{COPY} da Questão 5.8? ■

Exercício 5.11 Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma MT que decide a linguagem L . Mostre que é possível construir uma MT $M' = (Q', \Sigma, \Gamma, \delta, q'_0, B, F')$ com as seguintes especificações:

- M' também decide L
- Quando a máquina M' para, ela deixa na fita apenas um bit 1 ou 0, dependendo do caso quem aceitou ou rejeitou a string. De maneira mais precisa, a máquina se comporta da seguinte maneira:
 - Se M' aceita w , então $q'_0 w \vdash_M^* p1$, onde $p \in F'$
 - Se M' rejeita w , então $q'_0 w \vdash_M^* q1$, onde $q \notin F'$

Exercício 5.12 Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma MT que decide a linguagem L . Mostre que é possível construir uma MT $M' = (Q', \Sigma, \Gamma, \delta, q'_0, B, F')$ com as seguintes especificações:

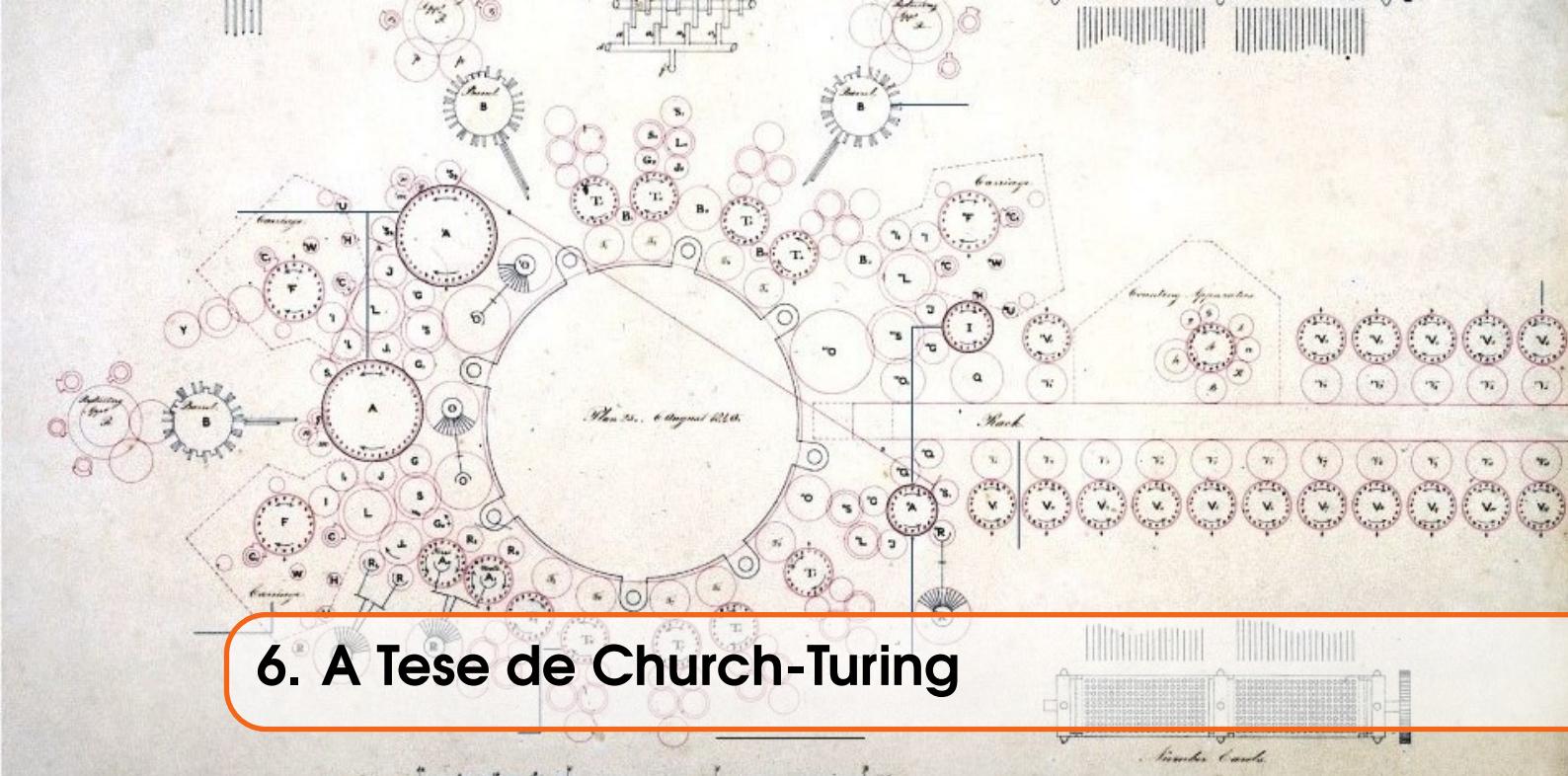
- M' aceita Σ^*
- Quando a máquina M' para, ela deixa na fita apenas um bit 1 ou 0, dependendo do caso que a máquina M aceita ou rejeita a string. De maneira mais precisa, a máquina M' se comporta da seguinte maneira:
 - Se M aceita w , então $q'_0 w \vdash_M^* p1$, onde $p \in F'$
 - Se M rejeita w , então $q'_0 w \vdash_M^* q1$, onde $q \notin F'$

Exercício 5.13 Forneça uma definição formal para uma MT com 3 fitas. ■

Exercício 5.14 Forneça uma definição formal para uma MT com 1 fita “read only” onde a string de entrada é posicionada e 1 fita “read/write” que a MT pode utilizar como memória. ■

Exercício 5.15 Forneça uma definição formal para uma MT com 1 fita “read only” onde a string de entrada é posicionada e 1 fita “read/write” que a MT pode utilizar como memória e uma fita “write only” onde a MT pode, por exemplo, escrever alguma string de “resposta”. ■

Exercício 5.16 Seja M uma MT que sempre para com as características da Questão 5.15. Mostre que podemos projetar uma MT M' que sempre para com $L(M) = L(M')$ com as seguintes características: a MT tem 1 fita “read only” de entrada e 1 fita “read/write” de memória e uma terceira fita “write only” de saída que a máquina escreve um bit de informação antes da máquina parar. O bit de informação é 0, se a string é rejeitada e 1 se a string é aceita. ■



6. A Tese de Church-Turing

Neste capítulo vamos discutir duas afirmações que, embora relacionadas, não são idênticas. A primeira delas é a afirmação de que Máquinas de Turing capturaram a noção intuitiva que temos do que seja um processo computacional. A segunda é a afirmação de que qualquer processo computacional **fisicamente realizável**, pode ser simulado por uma Máquina de Turing. As duas afirmações são comumente chamadas de Tese de Church-Turing.

6.1 Perspectiva histórica

No início do século XX alguns matemáticos observaram que o processo de se provar um teorema assemelhava-se a um processo mecânico: dada uma afirmação, o que fazemos é usar certos axiomas e regras válidas de inferência para, passo a passo, concluir que a afirmação é verdadeira (ou refutá-la, caso seja falsa). Em outras palavras, todo o raciocínio matemático começava a ser visto como um processo mecânico.

Isso motivou o matemático David Hilbert a desafiar a comunidade matemática em 1928 a encontrar um algoritmo que tome como entrada uma afirmação matemática e que responda SIM, se a afirmação é verdadeira (ou seja, a afirmação é um teorema) ou NÃO se a afirmação é falsa¹. O observe que, dada a maneira como Hilbert enunciou o problema, ele sequer considerava a possibilidade de que tal algoritmo não viesse a existir.

Resolver o problema proposto por Hilbert, conhecido como *Entscheidungsproblem* (problema da decisão, em alemão), era uma tarefa nada modesta, pois o objetivo de Hilbert era encontrar um algoritmo extremamente poderoso que automatizaria todo o processo de se “fazer matemática”. Qualquer pessoa seguindo este algoritmo seria capaz de provar qualquer teorema².

¹Estamos ignorando algumas complicações aqui. Em particular, estamos assumindo que os axiomas e as regras de inferência são algo fixo, conhecido a priori. Mas, poderíamos também pensar que o algoritmo, juntamente com a afirmação matemática, toma como entrada o conjunto finito de axiomas e regras de inferência que ele pode usar para provar ou refutar a afirmação matemática em questão.

²O matemático Gottfried Leibniz também já havia pensado sobre este mesmo problema no século XVII, entretanto, naquela época, a matemática não estava avançada ao ponto desta pergunta poder ser formulada com a precisão com que começava a ser formulada no começo do século XX.

Em 1936, primeiramente Alonzo Church, e, logo em seguida, Alan Turing provaram que tal algoritmo não existe. A questão importante que temos que ter em mente é que para se provar que um dado algoritmo não existe, a primeira coisa que deveria ser feita era tornar preciso exatamente o que é um algoritmo. Alonzo Church e Alan Turing provaram seus resultados praticamente ao mesmo tempo. Alonzo Church usou um formalismo matemático chamado *cálculo λ* e Alan Turing criou seu formalismo matemático, que hoje em dia é chamado de Máquina de Turing. Os dois modelos são equivalentes, mas a vantagem do modelo de Turing era a sua simplicidade e intuitividade, de maneira que ele tinha uma interpretação “física” que tornava mais convincente a ideia de que aquele era um processo mecânico que poderia representar um algoritmo qualquer.

Adicionalmente, Turing também mostrou que seu modelo tinha uma propriedade conhecida com *universalidade*. Esta propriedade não está tão relacionada a ideia de algoritmos em si, mas à ideia de computadores de propósito geral, ou seja, objetos que tomam algoritmos como entrada e os executam passo a passo. Veremos em detalhes o significado deste conceito no Capítulo 7. A definição da Máquina de Turing, o conceito de universalidade e a interpretação física destes conceitos marca início do que conhecemos por ciência da computação. Estes conceitos estabelecem também a base necessária para entendermos a Tese de Church-Turing, o que é o objetivo central deste capítulo.

6.2 Máquinas de Turing são equivalentes a linguagens de programação

Antes de entrarmos em uma discussão mais profunda para entender a afirmação de que Máquinas de Turing expressam o que queremos dizer algoritmos e computação, vamos começar com algo mais concreto. Nossa primeiro passo será apresentar um teorema que afirma que Máquinas de Turing são capazes de representar algoritmos escritos em linguagens de programação.

Primeiramente, não é difícil observar que projetar uma Máquina de Turing para realizar uma tarefa é muito mais trabalhoso do que escrever um programa usando linguagens de programação de alto nível para realizar a mesma tarefa. Entretanto, o fato de que temos mais trabalho escrevendo um algoritmo usando um dado formalismo em comparação outros formalismos não tem relação com o *poder comutacional* do formalismo em questão. Basta pensar que, embora escrever um programa em Linguagem Assembly é muito mais trabalhoso do que o mesmo programa em Python, tal programa **pode** ser escrito usando uma Linguagem Assembly.

Nesta seção nós vamos enunciar um teorema que afirma que Máquinas de Turing são equivalentes aos algoritmos escritos em linguagens de programação usadas hoje em dia. A demonstração do teorema não envolve nenhum conceito abstrato complicado, é muito longa e técnica e não é o nosso objetivo deste livro. O nosso objetivo é obsevar que isso é um **Teorema Matemático!** A demonstração, caso algum estudante curioso tenha interesse em verificar, pode ser vista na seção 2.6 do livro [PAP94]. Para que possamos enunciar o teorema, vamos definir um modelo matemático equivalente a programas escritos em assembly, uma vez que um programa escrito em uma linguagem de programação de alto nível pode sempre ser expresso por um programa em assembly³.

6.2.1 Programas Assembly

Um programa assembly pode ser definido como uma sequência de instruções. Nossa primeiro passo é definir exatamente o formato que uma instrução podem ter. Para que possamos apresentar tal definição, vamos definir os seguintes conjuntos de strings sobre o alfabeto $\{A, \dots, Z, 1, \dots, 9, \hat{\cdot}\}$:

- $A_1 = \{\text{HALF}, \text{HALT}\}$

³O trabalho de um compilador é converter um programa de alto nível em um programa assembly, que, por sua vez, é essencialmente uma sequência de intruções que o processador do computador é capaz de executar.

- $A_2 = \{\text{ADD}, \text{SUB}, \text{READ}, \text{STORE}, \text{LOAD}, \text{JUMP}, \text{JPOS}, \text{JZERO}, \text{JNEG}, \text{ZERO}\}$
- $B = \{j, \hat{j}\}$, onde j é uma string de $\{0, \dots, 9\}$.

Definição 6.2.1 — Instrução assembly. Uma *instrução* π é um objeto matemático que pode ter duas formas:

- (1) π pode ser um elemento do conjunto A_1
- (2) π_i pode ser um par (a, b) , $a \in A_2$ e $b \in B$

Além disso, há ainda as seguintes restrições sobre os tipos de intruções do tipo (2):

- Se $b = \hat{j}$, então obrigatoriamente $a \in \{\text{READ}, \text{STORE}\}$

As intruções do tipo (2) são chamadas de *instruções com argumentos*. Instruções com argumentos são tipicamente escritas na forma $a\ b$ (ao invés de (a, b)). Por exemplo, escrevemos STORE 12 ou invés de escrever (STORE, 12).

Definição 6.2.2 — Programa Assembly (PA). Um *programa assembly* é uma sequência finita $\Pi = \pi_1, \pi_2, \dots, \pi_n$ de *instruções*.

Ao invés escrevemos uma sequência de instruções separadas por vírgulas, é comum, como fazemos em computadores modernos, escrever programa assembly uma instrução por linha.

■ **Exemplo 6.1** Um exemplo de programa assembly (que não necessariamente computa algo útil):

```

ADD    33
READ   10
READ   ^10
HALF
STORE  12
HALT

```

SEMÂNTICA DE INSTRUÇÕES E EXECUÇÃO DE PROGRAMAS ASSEMBLY

Assim como fitas e cabeças de leitura são interpretações físicas do objeto matemático conhecido como Máquina de Turing, não fazendo parte da definição matemática em si, a interpretação do que “faz” uma instrução assembly, que normalmente chamamos de *semântica da linguagem* (por exemplo, a semântica da instrução HALF pode ser a seguinte: o valor contido em algum registrador padrão, que fica em determinado local do processador do computador, é dividido por 2) também é algo externo a definição matemática de um Programa Assemby. Se pensarmos um pouco a fundo, mesmo quando estamos lidando com modelos computacionais que são linguagens de alto nível, a semântica da linguagem em questão se refere, em última análise, a diferentes processos físicos ocorrendo na máquina relacionadas as diferentes instruções que a linguagem oferece.

Não iremos apresentar a semântica a fundo de cada uma das instruções da nossa definição de linguagem Assembly, assim como não vamos definir precisamente o que significa a “execução” de um PA, uma vez que estes conceitos são intuitivos para quem tem experiência com programação. O importante aqui é entender que o poder de *expressividade* de um PA é o mesmo que o de MTs. Na seção 2.6 do livro [PAP94] a semântica das instruções apresentadas aqui é vista em detalhes.

Nós sabemos que Programas assembly são capazes de resolver bem mais do que apenas

problemas de decisão (Máquinas de Turing também tem esta propriedade, como veremos no próximo capítulo). Entretanto, uma vez que estamos lidando, por hora, com problemas de decisão, será essencial definir o conceito de aceitação de linguagens por programas assembly.

Apesar de não estarmos nos preocupando com a semântica exata de PAs, vamos introduzir de alguns conceitos semânticos elementares para facilitar o entendimento do modelo. Primeiramente, precisamos pensar como um PA faz a leitura da string de entrada. Dado um PA, vamos assumir que o i -ésimo bit da string de entrada $x = x_1x_2\dots x_n$ que este programa toma é acessível usando-se a instrução `READ i`. Depois disso, a ideia geral é que a execução do programa consiste de uma série de instruções sendo *chamadas* (não necessariamente em ordem, uma vez instruções como `JUMP` servem para mudar o fluxo de execução do programa).

O último fragmento de semântica que precisamos esboçar aqui, é assumir que em nosso modelo, a aceitação ou rejeição de uma string de entrada é feita escrevendo-se um determinado bit para 0 ou 1 em algum “registrator” especial. Isso é feito chamando-se a instrução `LOAD i`. Um Programa Assembly para quando a instrução `HALT` é chamada.

Definição 6.2.3 A linguagem de um programa assembly Π , denotada $L(\Pi)$ é o conjunto de strings x de entrada satisfazendo a seguinte propriedade: *Se a execução do programa com a entrada x para e na última vez que a instrução `LOAD i` for chamada o valor de i for diferente de 0, então x é aceita.* Caso a última chamada à instrução `LOAD i` tenha o valor $i = 0$ ou nenhuma instrução do programa tipo `LOAD i` é chamada na execução do programa, a entrada é rejeitada.

Com isso, vamos enunciar o teorema a seguir, extremamente importante para entendermos a Tese de Church-Turing:

Teorema 6.2.1 Para todo Programa Assembly Π , existe uma MT M tal que $L(M) = L(\Pi)$.

Em outras palavras, se existe um programa assembly que resolve um problema, então existe uma Máquina de Turing que resolve o mesmo problema. Como já mencionamos, a prova deste teorema pode ser vista na seção 2.6 do livro [PAP94]. A outra direção do enunciado do teorema também é verdade, ou seja, para cada MT existe um PA equivalente que resolve o mesmo problema, o que significa que os dois modelos de computação são equivalentes.

Teorema 6.2.2 Para toda MT M , existe um Programa Assembly Π , tal que $L(\Pi) = L(M)$.

A demonstração do Teorema 6.2.2, vem do fato de que podemos escrever programas para “simular” MTs usando linguagens de alto nível (e portanto, também em Linguagem Assembly).

6.3 Máquinas de Turing e outros modelos de computação

Além de linguagens de programação modernas, uma série de outros modelos matemáticos são equivalentes a Máquinas de Turing. Alguns destes modelos foram propostos ainda na primeira metade do século XX, sendo os mais famosos o *cálculo λ* e as *funções μ-recursivas*. Estes modelos foram propostos com o objetivo puramente matemático de servir de definição de algoritmo, sem a intenção, a princípio, de ter correspondência com objetos físicos que possam ser de fato implementados. Com o avanço da ciência da computação, uma quantidade enorme de outros modelos matemáticos apareceram na literatura e provaram-se equivalentes a Máquinas de Turing.

Na frente prática, além da equivalência de MTs a computadores atuais, a pesquisa em áreas cujo objetivo é a construção de computadores usando substratos físicos “não tradicionais” também tem fornecido modelos matemáticos que são equivalentes a Máquinas de Turing.

Ainda é um pouco cedo para afirmar quais destes modelos advindos da tentativa de se usar substratos físicos não tradicionais refletem tecnologias que podem sair do papel. Vamos exemplificar isso usando duas áreas que tem sido mais ou menos proeminentes nos dias de hoje. Em menor escala, uma destas áreas é de computação molecular, mais precisamente, computação usando moléculas de DNA⁴. A outra área que usaremos como exemplo, e que é umas das mais ativas atualmente, é a área de computação quântica.

COMPUTAÇÃO QUÂNTICA

A ideia que sustenta a pesquisa em computação quântica é a seguinte: de acordo com as leis da mecânica quântica, a evolução no tempo de um conjunto de objetos de um sistema físico (estes objetos podem ser átomos, elétrons, fótons, etc) podem ser modelados com precisão arbitrária por uma abstração chamada de *círculo quântico*. A ideia básica é usar o estado de um objeto (ou conjunto de objetos) para registrar informação e fazer o processamento desta informação por meio da manipulação dos estados em que estes objetos possam se encontrar. Entretanto, como aqui os objetos usados são suficientemente pequenos, os tipos de manipulações possíveis (i.e., os tipos de transformações permitidas que levam um estado a outro) são regidas pelas leis da mecânica quântica.

Teorema 6.3.1 — Equivalência de MTs com outros modelos de computação. Os seguintes modelos matemáticos são equivalentes a Máquinas de Turing:

- (1) Variações de Máquinas de Turing (e.g., MT com múltiplas fitas, MT com uma fita infinita em apenas uma direção, MT cuja a entrada esteja em uma fita “read only” e as demais múltiplas fitas sejam “read-write”, MT com alfabetos que não sejam binários);
- (2) Cálculo λ , funções μ -recursivas, APs com 2 pilhas e outros modelos matemáticos;
- (3) Linguagens de programação modernas (e.g., C, C++, Java) e algoritmos em pseudo-código;
- (4) Modelos matemáticos de computação “não tradicional”, mas que sejam advindos de objetos físicos com implementação viável (e.g., diversos modelos de computação molecular);
- (5) Modelo de Circuitos Quânticos

O enunciado do Teorema 6.3.1 está um pouco vago, pois não definimos com precisão vários destes modelos matemáticos (e.g., Cálculo λ , funções μ -recursivas, Linguagem C, Linguagem Java, etc) e usamos vocabulário impreciso, como “outros modelos matemáticos” e “MT com outros alfabetos”⁵. O nosso objetivo central neste ponto não é apresentar os detalhes destas equivalências e sim reforçar que estas equivalências são **Teoremas Matemáticos**. Além disso, observamos o seguinte: no item (4) do enunciado do Teorema nós dissemos “implementação viável”. Isso tem conexão com algo que vamos discutir na próxima seção, que é a Tese de Church-Turing.

6.4 A Tese de Church-Turing e suas interpretações

A *Tese de Church-Turing (TCT)* é afirmação de que Máquinas de Turing “capturam o conceito de computação efetiva”. Há duas interpretações que normalmente são feitas desta tese. A primeira

⁴Há uma série de formalismos usados na área de computação molecular e computação com DNA. No contexto em que o objetivo é realizar *computação de propósito geral*, um dos modelos mais conhecidos é chamado de aTAM (da sigla em inglês “abstract tile assembly machine”).

⁵Note que, a rigor, MTs com alfabetos diferentes nunca vão aceitar as mesmas strings, pois por definição tais strings terão diferentes símbolos. Entretanto, a ideia aqui é que é possível codificar qualquer conjunto de símbolos usando apenas o alfabeto binário e estabelecer uma correspondência de um conjunto de strings quaisquer e um conjunto de string do alfabeto binário.

interpretação é que a TCT é uma definição matemática. A segunda intepretação é da TCT como uma afirmação sobre o mundo físico.

6.4.1 A TCT como definição matemática

Para entendermos a interpretação da TCT como sendo uma definição matemática, vamos usar uma analogia. Imagine que alguém faça a seguinte afirmação: *a definição de função contínua usando “epsilons” e “deltas”, como normalmente vemos em um curso de Cálculo, captura o conceito de continuidade de funções.*

Uma afirmação como esta acima pode ser debatida e algumas pessoas podem até discordar a respeito da afirmação, mas no fim ela é aceita por que ela é útil tem funcionado muito bem na prática desde a criação do Cálculo. O que acontece é que a ideia original de função contínua (funções que podemos “desenhar sem tirar a caneta do papel”) era subjetiva, e essa subjetividade impede que possamos fazer avanços matemáticos a respeito de funções contínuas. Uma vez que não parece existir alguma função contínua (como intuitivamente concebemos) que não possa ser expressa em termos desta definição usando epsilon e deltas, então os matemáticos **definiram** funções contínuas desta maneira. Portanto, funções que não podem ser expressas desta maneira **por definição não são contínuas** e funções que podem ser expressas desta maneira **por definição são contínuas**.

De maneira semelhante, a afirmação de que uma Máquina de Turing que sempre para é a **definição** de um algoritmo é vista como algo útil e que faz sentido na prática. Algumas pessoas poderiam debater se existe ou não existe um formalismo matemático capaz de representar algo que reconheçamos subjetivamente como um algoritmo, mas que não pode ser expresso na forma de Máquinas de Turing. Embora discussões como esta pareçam um pouco “anos 30”, ainda assim faz sentido para alguns debater esta questão pois trata-se de uma questão de definição matemática. O ponto chave é que, de maneira semelhante ao que fazemos com outras definições matemáticas (como a definição de funções contínuas, por exemplo), quando dizemos que se existe uma MT que sempre para para decidir um dado problema, então **por definição existe um algoritmo para o problema** e se não existe uma MT que sempre para para decidir um dado problema, então **por definição não existe um algoritmo para o problema**. Embora não se ulize tal expressão, esta interpretação não seria incorreto dizer que seria mais apropriado dizer *Definição* de Church-Turing, ao invés de Tese de Church-Turing.

6.4.2 A TCT como afirmação empiricamente verificável

Uma outra interpretação, é a de que a TCT é uma *afirmação empiricamente verificável*. Esta interpretação tem a vantagem de prover um critério objetivo para se refutar tal afirmação, caso ela venha a ser falsa. O critério que nos referimos é o mesmo usado para qualquer afirmação sobre o mundo físico⁶⁷: a afirmação deve ser descartada caso seja refutada experimentalmente. O que a Tese de Church-Turing afirma é o seguinte:

⁶⁷O termo informal “mundo físico” obviamente pode gerar discussões filosóficas de todo tipo, mas este não é o objetivo aqui, por mais interessante que isso possa ser. Sendo um pouco mais rigorosos, o que queremos dizer aqui é que estamos nos restringindo ao domínio do que pode ser verificado experimentalmente pelo método científico.

⁷Em particular, é importante remover algumas imprecisões filosóficas associadas a tal projeto. Uma confusão superficial, mas comum, é a seguinte: “Mas nós nem mesmo conhecemos as leis ‘finais’ da física para querer ter tal modelo!”. O ponto é que diferentes áreas da ciência, como química, geologia, ou ciência da computação estão interessadas em diferentes aspectos da realidade (o termo preciso é que estas áreas tem *epistemologias diferentes*). Embora, é claro, cientistas devem sempre estar antentos ao que há de sólido em outras ciências (incluindo a física) na medida do possível. A segunda confusão é que, em ciências empíricas, sempre que falamos em “todo e qualquer”, deve-se entender que estamos falando em “todo e qualquer, dentro do que pode ser aferido pelo que entendemos como método científico”.

TESE DE CHURCH-TURING: *Se um problema computacional pode ser resolvido por algum dispositivo fisicamente realizável, então ele pode ser resolvido por uma Máquina de Turing*

No Capítulo 7, veremos que existe um problema de decisão, chamado *problema da parada*, que não pode ser resolvido por Máquinas de Turing. Uma vez que uma consequência da TCT é que nenhum objeto no mundo físico seja capaz de resolver este problema, saberíamos precisamente o que tipo de evidência empírica precisaríamos para refutar a TCT: um aparato que resolva consistentemente o problema da parada. O consenso atual, que vem de uma série de direções (o que sabemos sobre as leis da física e mesmo sobre os fragmentos do que sabemos a respeito da direção que a física parece estar tomando) é que a existência de tal objeto parece ser bastante improvável.

No enunciado da TCT, quando nos referimos a um problema computacional, não nos referimos apenas a problemas de decisão. Neste contexto estamos nos referimos a algo extremamente amplo (essencialmente qualquer processo sistemático de transformação informação⁸). Por conta disto, a TCT tem uma implicação bastante significativa. Uma vez que podemos ver o estado de um objeto físico qualquer como a instanciação de alguma informação (i.e., a descrição do estado que o objeto se encontra é a informação em si), sabemos que a evolução no tempo de tal objeto, não importando quão complicado seja este objeto, pode ser simulado por uma Máquina de Turing.

CONEXÕES ENTRE COMPUTAÇÃO E FÍSICA

A Tese de Church-Turing é normalmente aceita por que, ao observarmos a natureza em seu nível mais fundamental e levarmos em consideração como objetos se comportam, quais estados que estes objetos podem estar, quais são seus graus de movimentos possíveis, e quais são os tipos de evolução que estes objetos podem sofrer no tempo, as restrições impostas pelas leis da mecânica quântica parecem sustentar a tese. Um ponto chave é que a descrição de um objeto pode ser aproximado com precisão arbitrária pelo modelo matemático conhecido com circuito quântico¹, e sabemos que estes modelos podem ser simulados por Máquinas de Turing (enunciamos este fato no Teorema 6.3.1). A citação abaixo expressa bem a vantagem que alguns cientistas vêem ao interpretar TCT como afirmação sobre a realidade física e por que a comunidade científica tende sustentar esta versão da tese.

Podemos ficar debatendo sem chegar a lugar nenhum sobre exatamente o que a Tese de Church-Turing quer dizer. Eu, pessoalmente, sempre preferi a versão da TCT em que ela é uma afirmação, empiricamente falsificável, a respeito dos tipos de problemas computacionais que podem ser resolvidos no mundo físico. Esta versão tem a enorme vantagem de tornar claro o que significa falsificá-la: uma revolução na física. — Scott Aaronson

¹Apesar do nome *circuito quântico*, este modelo não é exatamente um circuito no sentido em que estamos acostumados. Este modelo é um formalismo para descrever sistemas quânticos evoluindo no tempo.

CONTÍNUO VS DISCRETO?

A Tese de Church-Turing, como qualquer questão científica, é passível de debate. Existe uma área da computação, conhecida como *hipercomputação*, que é dedicada a estudar modelos que desafiem a TCT. Entretanto, a maioria das propostas que questionam a TCT são tipicamente variações da antiga ideia de computação analógica¹, uma ideia que parece esbarrar em alguns

⁸Se quisermos ser precisos, podemos ver um processo sistemático de transformação de informação como um mapeamento de uma string w para outra string $f(w)$. No Capítulo 7 veremos que problemas de decisão podem ser vistos como funções booleanas $f : \Sigma^* \rightarrow \{0, 1\}$ e que a ideia de problema computacional pode ser generalizada para funções $f : \Sigma^* \rightarrow \Sigma^*$ que mapeiam strings em outras strings.

obstáculos postos pela física teórica contemporânea. Em particular, o resultado mais importante nesta linha, demonstrado na década de 70, é chamado de *Limitante de Bekenstein* [Aar13]. Embora alguns parâmetros usados na mecânica quântica sejam contínuos, o Limitante de Bekenstein impõe um limite a quantidade de informação (que pode ser pensada em termos da quantidade de estados que podem ser observados em um sistema) que uma região finita do espaço pode conter.

¹Não nos referimos aqui a alguns dispositivos do nosso dia a dia que são ditos analógicos. Um dispositivo analógico, no sentido que nos referimos, seria capaz de realizar tarefas como, por exemplo, armazenar dados que requerem uma quantidade infinita de informação (e.g., armazenar o número π , com seus infinitos dígitos) e recuperar esta informação sem erros. No momento não há comprovação científica de seja possível realizar tais tarefas.

Além da TCT estar amparada pelo que sabemos de concreto sobre a mecânica quântica (e também por alguns resultados vindos de áreas da fronteira da física teórica), uma questão relevante que ampara a tese é a questão experimental. Embora seja comum que apareçam propostas de modelos que desafiam a TCT, até hoje todas as tentativas de implementação de algum modelo que desafie a tese falharam. A cada vez que isso ocorre, o consenso em torno da Tese de Church-Turing acaba sendo fortalecido, o que é normal acontecer com teses, princípios ou leis em qualquer área de investigação científica: cada vez que um experimento falha em refutar uma hipótese científica, a hipótese ganha mais força.

TURING E INTELIGÊNCIA ARTIFICIAL

Umas das consequências mais discutidas da Tese de Church-Turing é a afirmação de que cérebros humanos, sendo estes objetos físicos, podem ser simulados por Máquinas de Turing. Isso tende a gerar controvérsia, em particular quando interpretada de maneira incorreta. Portanto, para clarificar a conexão com inteligência artificial, é importante esclarecer o que a afirmação não diz:

(1) A afirmação não diz que já sabemos como fazer tal simulação, pois não sabemos exatamente como cérebros funcionam.

(2) A afirmação também não é a de que a simulação de um cérebro por uma Máquina de Turing é a melhor estratégia para se implementar inteligência artificial.

(3) A afirmação também não diz que a arquitetura específica de um cérebro é semelhante a arquitetura de computadores atuais (embora cérebros tenham evoluído para processar informação, processamento de informação pode ser feito usando-se muitas arquiteturas diferentes).

Estas questões acima são interessantes e dignas de pesquisa, mas não são relevantes aqui, pois a afirmação em questão é mais modesta: um cérebro, como qualquer objeto físico, pode ser simulado em princípio por uma Máquina de Turing.

Algo importante de se ressaltar é que a discussão da possibilidade da inteligência humana ser simulada por máquinas não é algo recente. Esta observação apareceu juntamente com nascimento computação e o próprio Alan Turing trabalhou nesta questão em seu famoso artigo em que o *Teste de Turing* é proposto.

Entretanto, é importante observar que o termo *inteligência* (assim como “consciência” ou “mente”, embora a TCT não faça menção tais termos) não é filosoficamente simples de se definir, sendo assim é natural que a TCT seja debatida neste âmbito. Para o leitor interessado no assunto, sugerimos o livro de Edward Feser [Fes19], que é um texto introdutório, acessível e com muita referência a respeito de filosofia da mente.

6.5 A Tese de Church-Turing estendida

Nesta seção vamos apresentar uma segunda afirmação que, ao contrário da TCT, **não** é consenso científico. Mas por que vamos perder tempo discutindo uma afirmação que possivelmente não é correta? O ponto é que entender esta segunda afirmação, conhecida como *Tese de Church-Turing Estendida*, é importante para compreendermos os desenvolvimentos recentes em teoria da computação, em particular, na área de computação quântica.

O seguinte teorema enuncia um fato importante com relação a equivalência de Máquinas de Turing a outros modelos de computação.

Teorema 6.5.1 Uma MT simula os modelos (1) a (4) do Teorema 6.3.1 com eficiência polinomial

Assim como no Teorema 6.3.1, enunciamos do Teorema 6.5.1 de maneira um pouco vaga, sem definir exatamente o que queremos dizer com *eficiência polinomial*. Entretanto, alunos familiarizados com análise de algoritmos entendem o que o enunciado do teorema quer dizer: não é possível definir algoritmo em qualquer um dos modelos (1), (2), (3) e (4) tal que número de passos necessários para a execução deste algoritmo seja exponencialmente menor do que o número de transições que a Máquina de Turing faria na simulação do algoritmo.

Observe que só incluímos os itens (1) a (4) e deixamos o item (5) de fora do enunciado do Teorema 6.5.1. O que acontece é que conjectura-se que não seja verdade que Máquinas de Turing simulem Circuitos Quânticos com eficiência polinomial. O modelo de Circuitos Quânticos é, até o momento, o único modelo com contrapartida em objetos físicos para o qual conjectura-se tal fato. O modelo de circuitos quânticos é a base da pesquisa em computação quântica.

A construção de computadores quânticos é possível em princípio, mas alguns pesquisadores questionam esta possibilidade. Este questionamento significa dizer que o modelo de circuitos quânticos não são modelos fisicamente realizáveis⁹. O que estes pesquisadores fazem é afirmar que não somente a TCT é sólida, mas que ela é mais sólida do que o consenso atual. Esta afirmação, conhecida como Tese de Church-Turing Estendida (TCTE), e que data da década de 60, afirma o seguinte: *todo problema computacional que possa ser resolvido de maneira eficiente no mundo físico, pode ser resolvido de maneira eficiente por uma Máquina de Turing*. Neste enunciado, a palavra eficiente quer dizer polinomial. Ao contrário do que acontece com a TCT, a TCTE não é consenso científico, pois tal afirmação sugeriria que o modelo de circuitos quânticos não é realista. Entretanto, o consenso científico é que o modelo de circuitos quânticos é simplesmente consequência das leis da mecânica quântica.

ALGORITMOS QUÂNTICOS

Atualmente, alguns problemas admitem algoritmos quânticos (i.e., algoritmos escritos na forma de circuitos quânticos) que os resolvam que são exponencialmente mais eficientes do que os melhores algoritmos clássicos que conhecemos. Quando nos referimos a algoritmos clássicos, queremos dizer Máquinas de Turing ou qualquer um dos modelos (1) a (4) do Teorema 6.3.1, por exemplo. Entretanto, ninguém foi capaz de provar matematicamente não existam algoritmos polinomiais clássicos para tais problemas, mas atualmente trabalha-se com a conjectura de que eles não existam e que o modelo de computação quântica é exponencialmente mais eficiente do que o modelo clássico para alguns problemas específicos. Esta conjectura é conhecida como a conjectura de que $P \neq BQP$.

Observe que mesmo que prove-se tal conjectura e conclua-se que o modelo de computação

⁹Computadores quânticos pequenos (i.e., contendo poucos *qubits*) já foram construídos. O que estes pesquisadores questionam é a possibilidade do modelo não ser escalável.

quântica é inherentemente mais eficiente que o modelo de MTs e, além disso, a construção de computadores quânticos seja realmente possível, como espera-se que seja, o que estes dois fatos juntos fazem é simplesmente refutar a TCTE. Por outro lado, a **TCT continua completamente intacta**, pois em termos do que se é possível computar (ou seja, ignorando questões de eficiência) computadores quânticos e clássicos são equivalentes. Em outras palavras, o conjunto de problemas que podem ser resolvidos por computadores quânticos é precisamente o conjunto das linguagens recursivas.

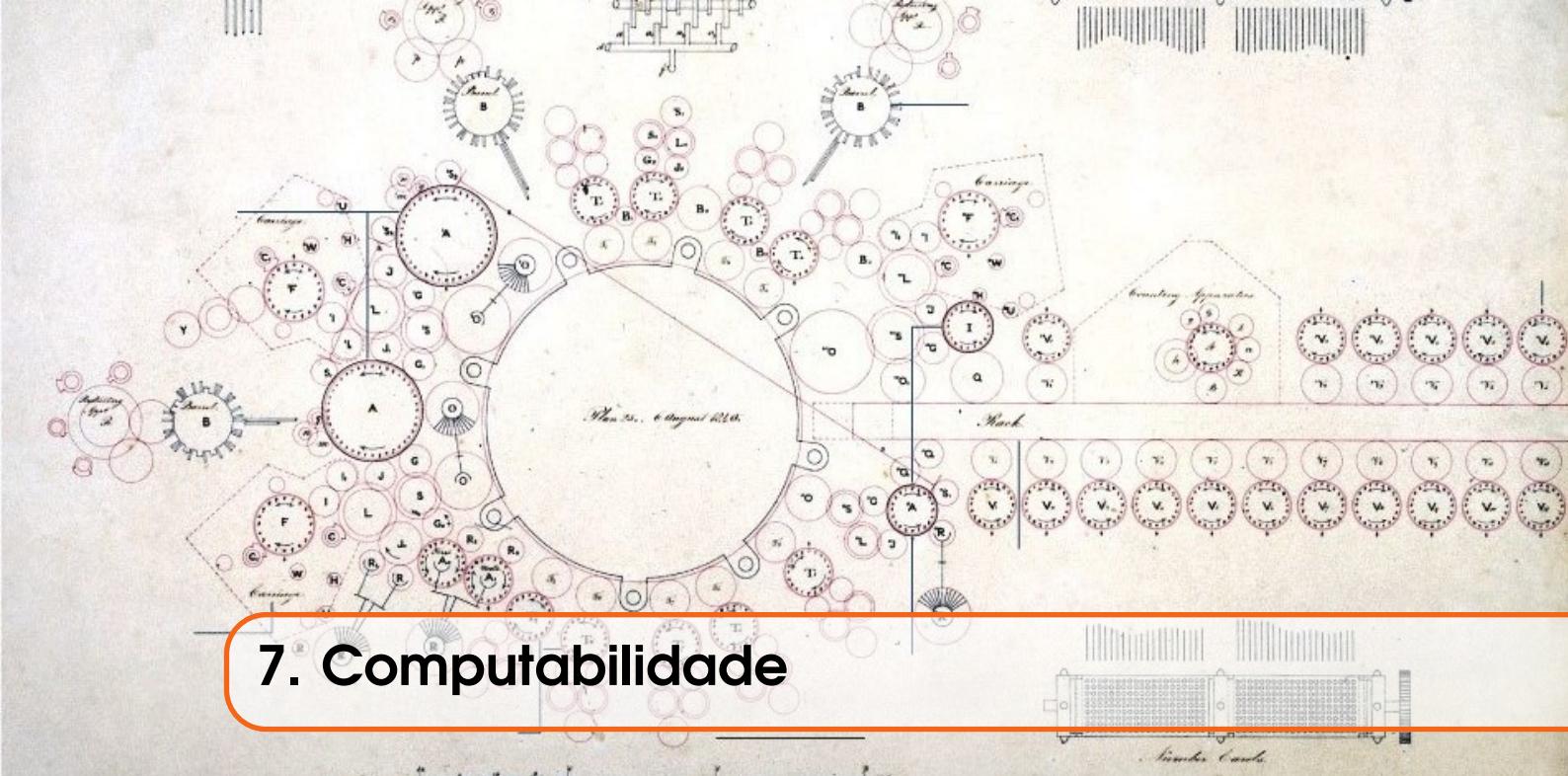
O que ocorre atualmente é que embora computadores quânticos sejam capazes de resolver precisamente os mesmos problemas que computadores clássicos, tem-se bastante interesse na construção de computadores quânticos, pois alguns problemas computacionais que eles poderiam resolver de maneira exponencialmente mais rápida são bastante importantes.

6.5.1 Exercícios

Exercício 6.1 O Teorema 6.3.1 afirma que uma MT com 3 fitas pode ser simulada por uma MT padrão com “eficiência polinomial”. Defina formalmente o que significa a afirmação: *Uma MT com 3 fitas pode ser simulada com eficiência polinomial*. Note que não estamos pedindo para você provar esta afirmação, estamos pedindo apenas para você a definir formalmente a afirmação. (Dica: use o conceito de computação com MTs, ou seja, a notação \vdash , e notação assintótica.) ■

Exercício 6.2 Descreva sucintamente o que é Tese de Church-Turing (TCT) vista como afirmação sobre o “mundo físico” e qual é a vantagem dela sobre a versão da TCT vista apenas como definição matemática. ■

Exercício 6.3 Prove que um AP com duas pilhas pode simular uma Máquina de Turing. ■



7. Computabilidade

Neste capítulo vamos estudar dois resultados centrais em Teoria da Computação. Os dois resultados estão presentes no artigo clássico, publicado por Alan Turing, publicado em 1936. O primeiro resultado é a prova de que existem problemas para os quais não existem algoritmos que os resolvam. O segundo é a existência da Máquina de Turing Universal, uma instância de uma Máquina de Turing, que é capaz de simular todas as possíveis Máquinas de Turing. A Máquina de Turing Universal pode ser vista como um modelo matemático que descreve o que entendemos por um computador.

7.1 Funções computáveis

Muitas vezes é conveniente entender que uma Máquina de Turing resolvendo um problema de decisão pode ser visto como a máquina computando uma função booleana $f : \{0, 1\}^* \rightarrow \{0, 1\}$. Mais precisamente, se $w \in \{0, 1\}^*$ fornecida como entrada é aceita pela máquina, podemos dizer que $f(w) = 1$, e, se w é rejeitada, então $f(w) = 0$. Entretanto, temos que ter um pouco de cuidado, pois algumas MTs podem eventualmente entrar em loop infinito com certas strings de entrada.

A ideia de MTs computando funções booleanas é natural no caso de MTs que sempre param, pois, para qualquer string binária de entrada x , ela termina sua execução aceitando ou rejeitando x , o que pode ser pensado como a máquina computando o valor 1 ou 0. Mas isso não acontece no caso de MTs que possam ficar computando indefinidamente. A notação que vamos introduzir a seguir será útil para lidar com este tipo de situação. Como de costume, estamos assumindo que $\Sigma = \{0, 1\}$.

Notação 7.1. Suponha que uma string $x \in \Sigma^*$ é fornecida como entrada para uma MT M . Dependendo do resultado da computação, escreveremos:

- $M(x) = 1$ quando M aceita x e para.
- $M(x) = 0$ quando M rejeita x e para.
- $M(x) = \nearrow$ quando M não para com a entrada x .

Observe que uma consequência da Notação 7.1 é que se M é um algoritmo, $\forall x \in \Sigma^*, M(x) \neq \nearrow$.

Definição 7.1.1 — Função booleana computável. Uma função $f : \Sigma^* \rightarrow \Sigma$ para a qual existe uma MT M tal que $\forall x \in \{0,1\}^*, M(x) = f(x)$, é denominada uma *função booleana computável*.

Em algumas situações vamos lidar com MTs que podem tomar como entrada uma string x e computar uma string resultante y como saída. O que queremos dizer com “string resultante” é a string que ficou na fita depois que a MT **aceitou** a string de entrada. O objetivo de introduzir a notação a seguir é tornar esta ideia precisa.

Notação 7.2. Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma Máquina de Turing. Se $q_0x \vdash_M^* y'q_Fy''$, tal que $q_F \in F$ e $y = y'y''$, então escrevemos $M(x) = y$.

Observe que a expressão $M(x) = y$ não faz sentido se M rejeita y ou se M não para. Também devemos tomar cuidado não confundir a notação acima com a Notação 7.1 no caso da string y conter apenas um bit. O contexto deve estar sempre claro no uso desta notação. Entretanto, observe que o Exercício 5.12 indica que tal ambiguidade não é um problema sério e, portanto, podemos escrever $M(x) = 1$ ou $M(x) = 0$ para nos referirmos a aceitação e rejeição de strings sem maiores consequências, desde que o contexto esteja claro. Com isso, observe também que *funções booleanas computáveis* são equivalentes a problemas decidíveis.

Definição 7.1.2 — Função computável. Uma função $f : \Sigma^* \rightarrow \Sigma^*$ para a qual existe uma MT M tal que $\forall x \in \Sigma^*, M(x) = f(x)$, é denominada uma *função computável*.

Note que uma *função booleana computável* é um caso particular de uma *função computável*.

7.2 Codificando objetos matemáticos em binário

Em geral, quando estamos pensamos em alto nível de abstração, um algoritmo pode tomar uma variedade de objetos matemáticos como entrada e retornar também diferentes tipos de objetos matemáticos. Por exemplo, podemos pensar em um algoritmo tomado vários grafos como entrada e retornando uma lista de números inteiros como saída. Nesta seção vamos lidar com alguns destes detalhes técnicos, uma vez que a entrada de uma Máquinas de Turing é apenas uma string. A saída pode ser apenas um bit ou, em certos contextos, uma string.

7.2.1 Notação para Máquinas de Turing tomando vários argumentos de entrada

Considere o algoritmo implementado em linguagem C que toma como entrada dois números e determina se um é múltiplo do outro. Vamos chamar este algoritmo de **MULTIPLA**. Se a entrada for, por exemplo, os números 5 e 10, a notação que usariam seria **MULTIPLA(5,10)** e diríam que a saída do algoritmo é SIM.

Adaptando esta ideia para MTs, uma ideia seria colocar os dois números em binário (ou seja, 101 e 1010) e concatená-los em uma string que estaria presente na fita de entrada da máquina. Usando a Notação 7.1, escreveríamos $M(1011010) = 1$, pois a MT começaria a computação com a string 1011010 em sua fita e a aceitaria, pois o número $N(1010)$ é múltiplo do número $N(101)$.

Mas isso realmente faz sentido? Dada a string de entrada 1011010, como a MT faz para adivinhar onde termina um número e onde começa o outro nesta string? A string 1011010 poderia ser a concatenação de um outro par de strings, como, por exemplo 10110 e 10. E agora?

Na realidade esta é uma dificuldade que é encontrada na prática em computadores de hoje em dia. Esta dificuldade é superada de diversas maneiras e é uma das razões pela qual usa-se sistemas de codificação (e.g., tabela ASCII, que nos permite definir símbolos de espaçamento, quebra de linha, etc) padronizados. No caso de MTs, nós também podemos usar estratégias parecidas.

Nós não vamos nos preocupar tanto com estes detalhes de baixo nível, pois isto não é realmente relevante e acabaria nos tirando o foco das questões realmente importantes envolvendo Máquinas de Turing. Ainda assim, como é bem comum lidarmos múltiplos argumentos de entrada concatenados em uma única string, é sempre bom levantar esta questão para que estejamos certos que as nossas definições estão corretas e que estamos trabalhando em terreno firme.

MÚLTIPLAS ENTRADAS: SOLUÇÃO SIMPLES

Uma outra possível maneira de lidar com este tipo de tecnicidade seria usar uma Máquina de Turing que tenha o alfabeto $\{0, 1, \#\}$. O Teorema 6.3.1 diz que o poder de computação de MTs com diferentes alfabetos é o mesmo das MTs que estamos usando (i.e., MT com alfabeto binário). Observe que, tendo um símbolo a mais, podemos usá-lo como separador. No exemplo do ínicio desta seção, poderíamos representar o par $(5, 10)$ usando a string $101\#1010$.

Notação 7.3 (Notação para MTs tomando múltiplos argumentos). *Se MT toma múltiplas strings de entrada, digamos, a uma n -tupla de strings $(x_1, x_2, x_3, \dots, x_n)$, dependendo da conveniência, usaremos tanto a notação $M(x_1, x_2, \dots, x_n)$ quanto a notação $M(x_1x_2x_3\dots x_n)$.*

O que a Notação 7.3 faz é essencialmente encapsular as rotinas de baixo nível que Máquinas de Turing tem que executar para lidar com várias strings de entrada.

7.2.2 Representando objetos matemáticos

Assim como objetos matemáticos são representados em baixo nível com símbolos 0 e 1 em computadores reais, teremos que representar os objetos sendo manipulados por nossas MT como strings binárias. Entretanto, precisamos tomar cuidado para não confundir o objeto matemático em si com sua representação binária.

Dado um objeto matemático S (este objeto pode ser, por exemplo, um número inteiro, um grafo, uma equação, uma expressão matemática, etc), usaremos a notação $\llcorner S \lrcorner$ para nos referir a string que codifica S .

Notação 7.4 (Codificação em binário). *Dado um objeto matemático S , a notação $\llcorner S \lrcorner$ se refere a string que representa a codificação de S em binário. A maneira exata de como codificar o objeto S depende do tipo de objeto em questão e deve sempre estar clara no contexto.*

Considere, por exemplo, um grafo $G = (V, E)$. O que queremos dizer com a notação 7.4 é que os objetos matemáticos G e $\llcorner G \lrcorner$ não significam a mesma coisa. O primeiro é um grafo, ou seja, um par de conjuntos, enquanto o segundo é uma string.

Podemos pensar em várias maneiras de se representar um grafo usando uma string e, em geral, nós não vamos nos preocupar com a maneira exata de se fazer isso. Entretanto, quando estivermos usando uma representação binária para algum objeto matemático, precisamos ter certeza que de que é possível realizar tal tarefa (por exemplo, se r é um número real, o objeto $\llcorner r \lrcorner$ não faz sentido algum). No caso de um dado grafo G , a maneira mais comum de representá-lo é concatenar os bits da matriz de adjacência de G linha por linha¹, como no exemplo a seguir.

■ **Exemplo 7.1** Seja $G = (V, E)$, onde $V = \{v_1, v_2, v_3\}$ e $E = \{v_1v_2, v_1v_3, v_2v_3\}$. Como a matriz de adjacência deste grafo é $\begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$, então a string que representa o grafo é $\llcorner G \lrcorner = 011101110$. ■

¹Caso estivéssemos lidando com um grafo com peso nas arestas a sua matriz de adjacência não seria binária, mas mesmo neste caso, com um pouco de trabalho não é difícil representar o grafo como uma string binária.

Com isso isto em mente, se quisermos que uma MT M tome como entrada um grafo G , escreveríamos $M(\lfloor G \rfloor)$ (ao invés de escrever $M(G)$, o que não faz sentido, pois a entrada de uma MT é uma strings e não um grafo).

O ponto mais importante que queremos levantar nesta seção é que Máquinas de Turing também poder ser representadas por strings binárias, afinal, MTs também são objetos matemáticos bem definidos, finitos e discretos. Usando novamente nossa analogia com computadores reais, um algoritmo implementado em alguma linguagem de programação (que sabemos que são equivalentes a MTs) armazenado na memória de um computador nada mais é do que uma sequência de bits na memória deste computador. Ou seja, um algoritmos pode ser vistos como strings. No caso de Máquinas de Turing, se uma MT M toma como entrada uma outra MT M' , a ideia é que a MT M' nada mais é do que uma sequência de 0's e 1's na fita da MT M .

Exercício 7.1 Mostre como representar uma MT qualquer usando uma string binária. (Dica: MTs são definidas por sua tabela de transições.) ■

Exercício 7.2 Seja M a Máquina de Turing da Figura 5.2.2. Apresente a string $\lfloor M \rfloor$. ■

7.2.3 Problemas de Decisão

Nesta seção, nós vamos definir uma série de problemas de decisão usando a notação que aprendemos para codificar objetos matemáticos.

Definição 7.2.1 — Primalidade. O problema de decidir se um número natural é primo pensado formalmente como a seguinte linguagem: $L_p = \{\lfloor n \rfloor \in \Sigma^* \mid n \text{ é um número primo}\}$.

Definição 7.2.2 — Quadrado Perfeito. O problema de decidir se um número natural é um quadrado perfeito é definido pela linguagem $L_{SQ} = \{\lfloor n \rfloor \in \Sigma^* \mid \exists x \in \mathbb{Z} \text{ tal que } n = x^2\}$. Em outras palavras, $L_{SQ} = \{\lfloor 0 \rfloor, \lfloor 1 \rfloor, \lfloor 4 \rfloor, \lfloor 9 \rfloor, \lfloor 16 \rfloor, \lfloor 25 \rfloor, \dots\}$.

As próximas três definições correspondem a problemas em teoria dos grafos. Relembreamos que um grafo $G = (V, E)$ é um par de conjuntos, sendo que V é chamado de conjunto de vértices de G e E é um conjunto de pares de vértices. Cada elemento de E é chamado de aresta de G . Neste livro vamos assumir familiaridade do aluno com conceitos básicos de teoria dos grafos.

Definição 7.2.3 — Conectividade de Grafos. O problema de decidir se um dado grafo é conexo é definido pela linguagem $L_c = \{\lfloor G \rfloor \in \Sigma^* \mid G \text{ é um grafo conexo}\}$.

Definição 7.2.4 — Grafos Eulerinos. O problema de decidir se um grafo é euleriano é definido pela linguagem $L_e = \{\lfloor G \rfloor \in \Sigma^* \mid G \text{ é um grafo euleriano}\}$.

Definição 7.2.5 — Grafos Hamiltonianos. O problema de decidir se um dado grafo é hamiltoniano é definido pela linguagem $L_h = \{\lfloor G \rfloor \in \Sigma^* \mid G \text{ é um grafo hamiltoniano}\}$.

Para o problema a seguir, relembreamos que se uma fórmula booleana escrita em *forma normal conjuntiva* é uma conjunção de disjunções ela é dita estar

Definição 7.2.6 — Satisfatibilidade de fórmulas booleanas (SAT). O problema de decidir se uma dada fórmula booleana escrita na *Forma Normal Conjuntiva (CNF)* é satisfazível é definido pela linguagem $L_{\text{SAT}} = \{\llcorner \phi \lrcorner \in \Sigma^* \mid \phi \text{ é uma fórmula booleana satisfazível escrita em CNF}\}$.

Neste contexto em que vemos linguagens como problemas, vamos as vezes usar a expressão *instância verdadeira* do problema L como sinônimo de uma string $w \in L$ e *instância falsa* do problema L como sinônimo de uma strings $w \notin L$.

■ **Exemplo 7.2** Como a fórmula $\phi_1 = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_3)$ é satisfazível (uma valoração que satizfaz ϕ é $x_1 = V$, $x_2 = V$ e $x_3 = F$), então dizemos que $\llcorner \phi_1 \lrcorner$ é uma instância verdadeira de L_{SAT} . Por outro lado, como $\phi_2 = (\bar{x}_1 \vee \bar{x}_2) \wedge (x_1) \wedge (x_2)$ não é satisfazível, dizemos que $\llcorner \phi_2 \lrcorner$ é uma instância falsa de L_{SAT} . ■

Convenção 7.1 (Literais em fórmulas em CNF). *Seja C_i uma cláusula de uma fórmula ϕ em CNF. Iremos assumir que em C_i não ocorrem literais repetidos. Iremos assumir que se o literal l em C_i é corresponde a variável x , então não existe em C_i o literal \bar{x} . De maneira semelhante, se $l = \bar{x}$, então não ocorre literal x .*

Exercício 7.3 Lembrando que o grafo K_4 é o grafo completo com 4 vértices, responda: $\llcorner K_4 \lrcorner$ é uma instância verdadeira ou falsa do problema L_E ? ■

Exercício 7.4 A string $\llcorner K_4 \lrcorner$ é uma instância verdadeira ou falsa do problema L_H ? ■

7.3 Máquinas de Turing, pseudo-códigos, generalidade e especificidade

Resolver o problema de teste de primalidade significa encontrar uma Máquina de Turing que decida a linguagem L_p . Obviamente, pela equivalência de MTs e linguagens de programação modernas (vimos isso no Teorema 6.3.1), não precisamos ir tão longe, pois basta mostrarmos um o pseudo-código de um algoritmo de primalidade, como o algoritmo a seguir.

Primo: (N)

- 1: **if** $N = 1$ **then**
- 2: Retorna Falso
- 3: **for** $i = 2; i \leq \sqrt{N}; i++$ **do**
- 4: **if** $N \text{ mod } i = 0$ **then**
- 5: Retorna Falso
- 6: Retorna Verdadeiro

Neste momento é natural nos questionarmos se, agora que sabemos que Máquinas de Turing são equivalentes a nossa noção intuitiva de algoritmo, vale mesmo a pena usarmos o formalismo matemático de Máquinas de Turing para nos referirmos a algoritmos. Em situações concretas, como no caso acima, claramente é bem mais conveniente apresentar um algoritmo em pseudo-código do que apresentar uma Máquina de Turing. Vamos usar a regra geral, descrita no quadro abaixo:

PSEUDO-CÓDIGOS OU MÁQUINAS DE TURING?

Sempre que estivermos pensando em problemas específicos, como testar se um grafo é conexo, testar se uma matriz é inversível, verificar se uma sequência de números está ordenada, etc, nós não iremos usar Máquinas de Turing. Ao invés disso iremos usar algoritmos escritos na

forma de pseudo-código.

Por outro lado, em situações em que estamos falando sobre algoritmos de maneira abstrata, como é comum em teoria da computação, Máquinas de Turing são a escolha adequada. Em teoria da computação é comum situações em que queremos provar afirmações do tipo “não existe nenhum algoritmo M com determinada propriedade” ou “para todo algoritmo M , determinado fato ocorre”. A vantagem de se usar Máquinas de Turing é que temos uma definição precisa e bastante simples de objeto matemático que condensa todo e qualquer algoritmo possível.

7.4 O problema da Parada

Os problemas vistos nos exemplos na Seção 7.2.3 podem ser resolvidos por uma variedade de algoritmos diferentes. Podemos nos questionar sobre a eficiência dos algoritmos que resolvem tais problemas, mas este não será o nosso foco agora. Agora, a questão é simplesmente saber se existe ou não existe um algoritmo para um determinado problema (em outras palavras, se uma determinada linguagem é recursiva o não).

Exercício 7.5 Apresente algoritmos para resolver os problemas de decisão da Seção 7.2.3. ■

O objetivo desta seção é apresentar um problema, conhecido como *Problema da Parada*, que não admite nenhum algoritmo que o resolva. O problema, visto de maneira intuitiva, é o seguinte: dada uma MT M arbitrária juntamente com uma string x arbitrária, queremos saber se M eventualmente finaliza a sua execução ou se M fica em loop infinito quando a string x é fornecida como entrada. Formalmente o problema é o seguinte:

Definição 7.4.1 — O problema da parada. O *problema da parada* é definido pela linguagem $L_H = \{\langle M \rangle x ; \text{ tal que } M \text{ é uma MT, } x \in \Sigma^* \text{ e } M(x) \neq \uparrow\}$.

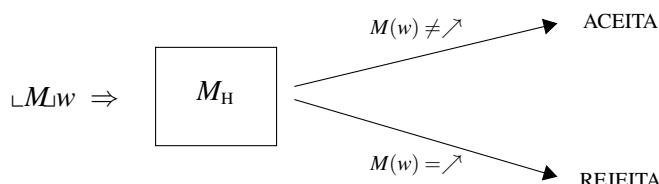
Resolver o problema da parada significaria fornecer uma MT M_H que decide L_H . Ou seja, uma MT M_H que tome $(\langle M \rangle, x)$ como entrada e que tenha o seguinte comportamento:

- Se $M(x) = 0$ ou $M(x) = 1$, então $M_H(\langle M \rangle, x) = 1$.
- Se $M(x) = \uparrow$, então $M_H(\langle M \rangle, x) = 0$.

Teorema 7.4.1 — Teorema da Parada. Não existe algoritmo que decide a linguagem L_H .

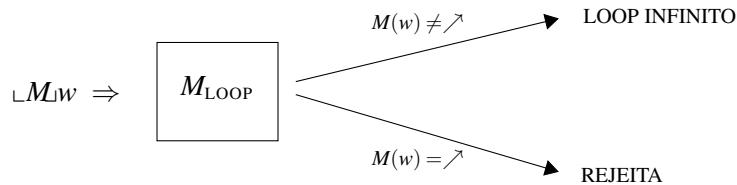
Prova: Suponha que exista uma Máquina de Turing M_H que decide L_H . Vamos mostrar que isso levará a uma contradição e, portanto, concluiremos que M_H não existe por redução ao absurdo.

A máquina M_H tem o seguinte comportamento quando a string $\langle M \rangle w$ é fornecida como entrada. Se $M(w) = \uparrow$, então a máquina M_H deve rejeitar a string de entrada. Por outro lado, se $M(w) \neq \uparrow$ (note que não importa se $M(w) = 0$ ou se $M(w) = 1$), então M_H deve aceitar a string de entrada. O diagrama abaixo ilustra o funcionamento de M_H :



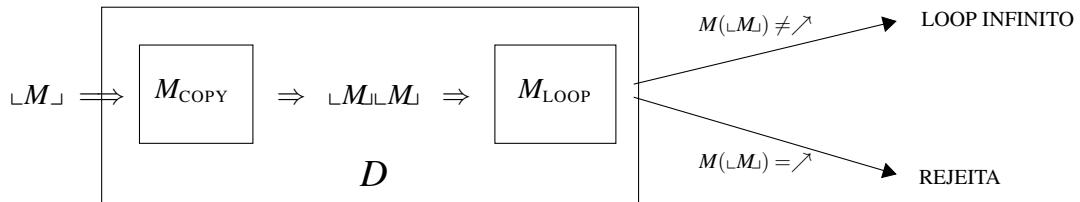
Dado que a máquina M_H existe, vamos agora concluir que a máquina M_{LOOP} , que vamos descrever a seguir, também existe. A máquina M_{LOOP} é essencialmente M_H com uma pequena modificação.

Dada uma string que M_H aceite, ao invés da máquina aceitar e parar, a máquina deve entrar em loop infinito. o funcionamento de M_{LOOP} é descrito abaixo.



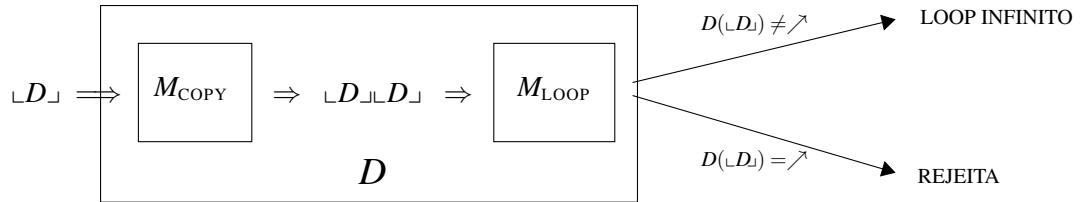
Antes de seguir em frente, devemos antentar ao tipo de argumentação que estamos usando. Estamos trabalhando com a existência de M_H , pois esta foi a nossa suposição inicial, mas será que, de fato, M_{LOOP} existe? Nossa argumento foi que podemos obtê-la fazendo uma modificação em M_H . Para termos certeza que o nosso argumento está correto, devemos ser capazes de mostrar, passo a passo, como obter M_{LOOP} a partir de M_H (o objetivo do Exercício 7.9 é provar formalmente que se M_H existe, então M_{LOOP} também existe).

Agora vamos construir uma MT D que é uma MT composta de duas MTs distintas. A primeira parte é uma máquina M_{COPY} , que duplica a string de entrada (a construção desta máquina é o objetivo do Exercício 5.8), ou seja, $\forall w \in \Sigma^*, M_{COPY}(w) = ww$, e a segunda parte consiste da MT M_{LOOP} , que descrevemos anteriormente. O funcionamento de D é descrito pela figura a seguir).



A construção formal de D é intuitiva, mas os estudantes que gostam de demonstrar teoremas de maneira extrememete rigorosa são encorajados a resolver o Exercício 7.10, cujo objetivo é mostrar que, de fato, se as máquinas M_{COPY} e M_{LOOP} existem, então D também existe.

O comportamento de D com a entrada $\llcorner M \llcorner$ é o seguinte: $D(\llcorner M \llcorner) = \nearrow \Leftrightarrow M(\llcorner M \llcorner) \neq \nearrow$. Considere agora a string $\llcorner D \llcorner$ (sabemos que esta string existe, pois D existe). Agora vejamos o que acontece quando fornecemos a string $\llcorner D \llcorner$ como entrada para a máquina D :



A conclusão que chegamos é que $D(\llcorner D \llcorner) \neq \nearrow \Leftrightarrow D(\llcorner D \llcorner) = \nearrow$, o que é uma contradição lógica. Com isso concluímos que M_H não existe. \square

7.5 A Máquina de Turing Universal

Considere uma MT \mathcal{U} que toma como entrada uma outra máquina M e uma string x e simule o comportamento de $M(x)$, ou seja, \mathcal{U} “executa” a MT M quando esta tem como argumento de entrada a string x . A ideia é que o resultado da computação de $\mathcal{U}(\llcorner M \llcorner, x)$ seja o mesmo resultado da computação de $M(x)$ (caso $M(x) = \nearrow$, a máquina \mathcal{U} deve ficar em “loop infinito”).

Adicionalmente, quando vemos nossas máquinas de Turing computando funções não booleanas, se $M(x) = y$, então $\mathcal{U}(\lfloor M \rfloor, x) = y$

A primeira pergunta que devemos fazer é se \mathcal{U} realmente existe. Afinal, dada uma tarefa, não podemos simplesmente supor que exista uma MT que a realize tal tarefa. A existência de tal máquina foi um dos resultados que Alan Turing provou em seu famoso artigo de 1936. Esta MT é conhecida como *Máquina de Turing Universal*.

Antes de enunciar o teorema da existência de \mathcal{U} , vamos refletir um pouco sobre o seguinte:

- Até agora estávamos vendo Máquinas de Turing como “software” e definimos algoritmos como sendo Máquinas de Turing que sempre param (MTs que ficam em loop infinito, por definição, não são algoritmos, mas elas podem ser pensadas como sendo programas de computador que ficam em loop infinito).
- No caso da Máquina de Turing Universal é bastante razoável pensarmos nela como um modelo matemático para um computador. Uma MT universal \mathcal{U} tem a capacidade de rodar qualquer outra MT M com qualquer possível entrada de dados x , e, ao final, retornar a saída de $M(x)$. A máquina \mathcal{U} pode ficar executando M indefinidamente se $M(x) = \uparrow$. Isso é essencialmente o que um computador faz. Claramente, essa visão de “software” e “computador” pode ser maleável, pois muitas vezes temos implementações de algoritmos feitas em hardware de propósito específico e, por outro lado, também temos softwares que funcionam como uma Máquina de Turing Universal. Alguns exemplos de softwares que podem ser vistos com MTs universais são, por exemplo, interpretadores de linguagens de programação ou softwares emuladores².
- Finalmente, note que existência de uma MT universal do ponto vista físico (i.e., a possibilidade de se implementar fisicamente uma MT Universal) é algo bastante poderoso. Quando o conceito foi concebido por Alan Turing em 1936 não existiam computadores e muito menos software. Entretanto, a indústria de software de hoje seria uma impossibilidade matemática se o objeto matemático \mathcal{U} não existisse sob a luz da Tese de Church-Turing, pois em tal situação não seria possível construir computadores capazes de rodar cada algoritmo possível e imaginável. Em tal cenário, para cada problema específico precisaríamos implementar o respectivo algoritmo que o resolve diretamente em hardware.

Teorema 7.5.1 Existe uma MT \mathcal{U} tal que \forall MT M e $\forall x \in \Sigma^*$, temos $\mathcal{U}(\lfloor M \rfloor, x) = M(x)$.

Idéia da Prova: Lembramos que para provarmos que uma certa MT existe, basta apresentarmos explicitamente a definição de tal máquina. A apresentação da definição em detalhes da 7-tupla \mathcal{U} é muito trabalhosa e é algo que está fora do escopo deste curso. O que vamos fazer aqui é apresentar a ideia geral. Vamos esboçar o funcionamento de uma MT \mathcal{U}_3 , que é uma MT com 3 fitas que realiza a tarefa que \mathcal{U} deve realizar. A nossa definição de permite que Máquinas de Turing tenham apenas uma fita, mas pelo Teorema 6.3.1, podemos concluir que se \mathcal{U}_3 existe, então a MT \mathcal{U} com as propriedades desejadas também existe.

A ideia é que mantenhamos $\lfloor M \rfloor$ na primeira fita de \mathcal{U}_3 . Vamos tratar esta fita como se ela fosse uma fita de entrada onde queremos manter intacta a descrição de M . A descrição de M é o programa que queremos que \mathcal{U}_3 rode. Ainda no início da computação, colocamos a string x na segunda fita de \mathcal{U}_3 . O que a máquina \mathcal{U}_3 vai fazer é simular passo a passo o que aconteceria no caso de x ser colocada colocada na fita (única) da máquina M . Na computação de $M(x)$, a cada passo, a fita de M conterá uma certa string. O conteúdo segunda fita de \mathcal{U}_3 será precisamente o conteúdo estaria presente na fita de M durante a computação de $M(x)$. A terceira fita de \mathcal{U}_3 será uma fita de memória auxiliar. Durante o processo de simulação da máquina M , vamos armazenar nesta

²Pense no seguinte: o seu emulador favorito de Super Nintendo pode ser visto como uma MT universal!

terceira fita dois números em binário. O primeiro número corresponde ao estado que a máquina M 正在被模拟时所处的状态。O segundo número é um índice que corresponde a posição da fita que a cabeça de leitura de M está posicionada.

Durante a computação, \mathcal{U}_3 vai atualizando a sua segunda fita para refletir precisamente como a MT M alteraria a sua fita única. Se eventualmente a M atingir um estado final a MT \mathcal{U}_3 identifica isso e vai para o seu estado final, e portanto aceita a entrada e parar. No caso em que M não tenha uma transição definida e esteja em um estado que não seja final (ou seja, M irá rejeitar a entrada), a MT \mathcal{U}_3 irá para um estado especial que é um estado que não tem nenhuma transição definida e que também não é final, e portanto \mathcal{U}_3 vai parar rejeitando a entrada. Caso M fique executando indefinidamente, a MT \mathcal{U}_3 simplesmente vai continuar simulando M indefinidamente também. \square

Relembrando que L_H é a linguagem da parada, temos o seguinte teorema:

Teorema 7.5.2 L_H é recursivamente enumerável.

Exercício 7.6 Foneça uma prova para o Teorema 7.5.2. ■

Uma consequência do Teorema 7.5.2 é que o conjunto \mathcal{R} está estritamente contido no conjunto \mathcal{RE} . Enunciamos isto no corolário a seguir:

Corolário 7.5.3 $\mathcal{R} \subsetneq \mathcal{RE}$.

Prova: Provamos no Exercício 5.7 que $\mathcal{R} \subseteq \mathcal{RE}$. Agora precisamos provar que esta inclusão é própria. Para tal, precisamos mostrar que existe pelo menos uma linguagem que esteja contida em \mathcal{RE} , mas que não esteja contida em \mathcal{R} . Pelo Exercício 7.6, $L_H \in \mathcal{RE}$. Pelo Teorema 7.4.1, $L_H \notin \mathcal{R}$. Portanto $\mathcal{R} \subsetneq \mathcal{RE}$.

Teorema 7.5.4 Existe L tal que $L \notin \mathcal{RE}$.

Exercício 7.7 Forneça uma prova para o Teorema 7.5.4. ■

MÁQUINAS, PROGRAMAS E DADOS

“Antes de Alan Turing a suposição era que as três categorias, máquina, programa e dados, eram entidades completamente separadas. A máquina era um objeto físico, era ‘hardware’. O programa era o planejamento da computação que iríamos executar. Os dados eram a entrada numérica. A máquina universal de Turing mostrou que essas distinções eram uma ilusão. Essa fluidez é essencial hoje em dia na prática: um programa é dado para um compilador [, por exemplo].” – Martin David

7.6 Máquinas de Turing não determinísticas (MTN)

Se quisermos fornecer uma definição para uma Máquina de Turing não Determinística, a primeira ideia que nos vem à mente é modificar a definição de Máquina de Turing para que a função de transição $\delta(q, X)$ retorne um conjunto de triplas $\{(q_1, Y_1, D_1), (q_2, Y_2, D_2), \dots, (q_k, Y_k, D_k)\}$. Esta definição seria parecida com o que fizemos, quando definimos NFAs como generalizações de DFAs no Capítulo 3. A definição que daremos aqui é um pouco diferente, mas é possível provar que o poder computacional destas duas definições é o mesmo.

Definição 7.6.1 — Máquina de Turing não determinística (MTN). Uma *Máquina de Turing não determinística* é uma 7-tupla $M_N = (Q, \Sigma, \Gamma, (\delta_0, \delta_1), q_0, B, F)$.

A definição dos componentes $Q, \Sigma, \Gamma, q_0, B, F$ desta tupla são iguais a definição de MTs. O par (δ_0, δ_1) consiste de duas funções de transições, também definidas exatamente como eram definidas em Máquinas de Turing determinísticas.

Do ponto de vista de definição matemática, a única diferença que MTNs tem em relação a MTs é que, ao invés de uma função de transição δ , MTNs tem um par de funções (δ_0, δ_1) . A questão agora é *interpretar* a definição de MTNs para que possamos definir como é o processo de computação não determinística neste caso. A ideia é que, a cada passo, a máquina tem a capacidade de advinhar qual das duas funções ela deve usar para fazer a transição. A linguagem de uma MTN N é o conjunto de toda string para a qual existe uma computação que a aceite, ou seja, toda string x tal que, a cada passo, existe uma escolha entre δ_0 e δ_1 que leve N a aceitar x .

De maneira semelhante a MTs veremos, o processo de computação de uma Máquina de Turing como uma sequência de descrições instantâneas.

Definição 7.6.2 — Descrição Instantânea (ID) de MTNs. Dada uma MTN $N = (Q, \Sigma, \Gamma, (\delta_0, \delta_1), q_0, B, F)$, uma *Descrição Instantânea* de N é uma string $\alpha q \beta$ tal que $\alpha, \beta \in \Gamma^*$ e $q \in Q$.

A interpretação do que a string $\alpha q \beta$ é a mesma que ocorria no caso de MTs determinísticas. O símbolo \vdash_N , definido a seguir, representa um passo computacional de uma MTN. A diferença em relação a MTs é que a partir de um ID I , a computação pode se dirigir a possivelmente dois IDs diferentes no próximo passo, dependendo de qual das duas funções de transição foi escolhida pela Máquina de Turing não Determinística.

Definição 7.6.3 — O símbolo \vdash . Seja $N = (Q, \Sigma, \Gamma, (\delta_0, \delta_1), q_0, B, F)$ uma Máquina de Turing não determinística e I um ID da máquina N .

- Se a aplicação da função δ_0 leva o ID I ao ID I_0 , então escrevemos $I \vdash_{N_0} I_0$.
- Se a aplicação da função δ_1 leva o ID I ao ID I_1 , então escrevemos $I \vdash_{N_1} I_1$.

Se $I \vdash_{N_0} I'$ ou $I \vdash_{N_1} I'$, então escrevemos $I \vdash_N I'$.

Definição 7.6.4 Dada uma MTN N , o símbolo \vdash_N^* é definido indutivamente:

Base: $I \vdash_N^* I$ para qualquer ID I de N .

Indução: $I \vdash_N^* J$ se $\exists K$ tal que $I \vdash_N K$ e $K \vdash_N^* J$.

Definição 7.6.5 — Linguagens aceitas por MTNs. Dada uma Máquina de Turing não Determinística $N = (Q, \Sigma, \Gamma, (\delta_0, \delta_1), q_0, B, F)$, a linguagem $L(N) = \{w \in \Sigma^*; q_0 w \vdash_N^* I_F\}$, tal que I_F é um ID final de $N\}$ é chamada de *linguagem de N* ou linguagem *aceita* por N .

Definição 7.6.6 — Árvore de Computações Possíveis de MTNs. Seja N uma NTM e x uma string. A árvore de computações possíveis de N com x é uma árvore cujos nós são IDs da computação de $N(x)$ definida da seguinte maneira. A raiz é o ID inicial da computação e cada nó I tem no máximo dois filhos, dependendo do seguinte:

- Se I é um ID final, então I não possui filhos.
- Se existe um único I' tal que $I \vdash_N I'$, então I' é único filho de I (esta situação ocorre no caso em que as duas funções δ_0 e δ_1 retornam o mesmo valor).
- Se existem IDs distintos I', I'' tal que $I \vdash_N I'$ e $I \vdash_N I''$, então I' e I'' são os dois filhos de I .

Uma observação importante é que uma árvore de computações possíveis de MTNs pode ter alguns ramos infinitamente longos³ nos casos em que ramos da computação fiquem em loop infinito. Entretanto, se uma MTN N aceita uma string x , existe pelo menos um ramo finito nesta árvore, e a folha no final deste ramo é um ID final.

A seguir enunciamos dois teoremas que mostram que o conjunto de linguagens decididas por MTNs é precisamente o conjunto das linguagens recursivas. Veremos na Parte III deste livro que MTNs, embora não sejam modelos realistas de computação, são ferramentas úteis para explorar questões relacionadas a existência de algoritmos eficientes para a resolução de certos problemas computacionais.

Teorema 7.6.1 Se N é uma MTN, então existe uma MT M tal que $L(M) = L(N)$.

Idéia da prova: Uma MT pode simular uma NTM.

Teorema 7.6.2 Se M é uma MT, então existe uma MTN N tal que $L(M) = L(N)$.

Idéia da prova: Seja δ a função de transição de M_D . Basta construir uma MTN em que $\delta_1 = \delta_2 = \delta$.

Exercício 7.8 Apresente uma MTN que escreve uma string binária de tamanho 5 na fita tal que cada uma das 2^5 strings binárias diferentes aparece a final de um ramo diferente de sua árvore de computações possíveis. ■

7.7 Exercícios

Exercício 7.9 Seja uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ que decide a linguagem L . Mostre como obter uma MT M' com o seguinte comportamento:

- (1) Se $M(x) = 1$, então $M'(x) = \nearrow$
- (2) Se $M(x) = 0$, então $M'(x) = 0$.

Exercício 7.10 Seja M_{COPY} a máquina fornecida no Exercício 5.8. Seja M_{LOOP} a máquina cujo funcionamento é explicado na prova do Teorema 7.4.1. Prove que se $M_{\text{COPY}} \sim M_{\text{LOOP}}$, então existe uma máquina D tal que $D(\llbracket D \rrbracket) \neq \nearrow \Leftrightarrow D(\llbracket D \rrbracket) = \nearrow$, o que é uma contradição lógica. ■

Exercício 7.11 Prove de maneira formal que L_H (a linguagem da parada) é recursivamente enumerável. ■

Exercício 7.12 Seja \mathcal{M} o conjunto de todas as máquinas de turing e seja \mathcal{L} o conjunto de todas as linguagens sobre Σ . Usando o argumento da diagonalização mostre que $|\mathcal{M}| < |\mathcal{L}|$ e conclua que existem linguagens que não são recursivamente enumeráveis. ■

³Um detalhe que devemos atentar é que árvores são casos particulares de grafos, e grafos são definidos como sendo objetos finitos. Portanto, a rigor, precisaríamos usar em nossa definição conceitos como árvores infinitas ou grafos infinitos (grafos infinitos e árvores infinitas também são objetos matemáticos bem definidos, embora não tão amplamente estudados como os seus equivalentes finitos), mas não vamos nos preocupar com isso, pois a noção intuitiva de uma árvore em que alguns ramos podem ser infinitamente longos é suficiente para os nossos propósitos.

Exercício 7.13 No Exercício 7.12 mostramos que linguagens que não são recursivamente enumeráveis *existem*. Neste exercício vamos mostrar explicitamente que uma dada linguagem não é recursivamente enumerável. Seja L_H a linguagem da parada. Prove que $\overline{L_H} \notin \mathcal{RE}$. ■

Exercício 7.14 Prove o seguinte problema é indecidível. Dada uma Máquina de Turing M , queremos decidir se $M(\varepsilon) \neq \nearrow$. ■

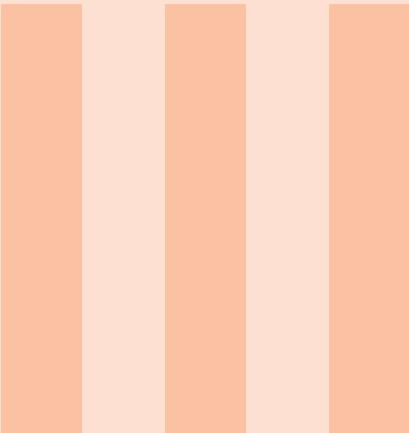
Exercício 7.15 Prove que $L = \{\llcorner M \lrcorner ; \exists x \in \Sigma^* \text{ tal que } M(x) \neq \nearrow\}$ é indecidível. ■

Exercício 7.16 Dado como entrada $\llcorner M \lrcorner$, onde M é uma MT, a pergunta que você deve responder é se existe uma MT M' com o seguinte comportamento:

- Se $M(\varepsilon) = 1$, então $M'(\llcorner M \lrcorner) = 1$;
- Se $M(\varepsilon) = 0$, então $M'(\llcorner M \lrcorner) = 0$;
- Se $M(\varepsilon) = \nearrow$, então $M'(\llcorner M \lrcorner) \neq \nearrow$ (ou seja, M' para, independente de aceitar ou rejeitar).

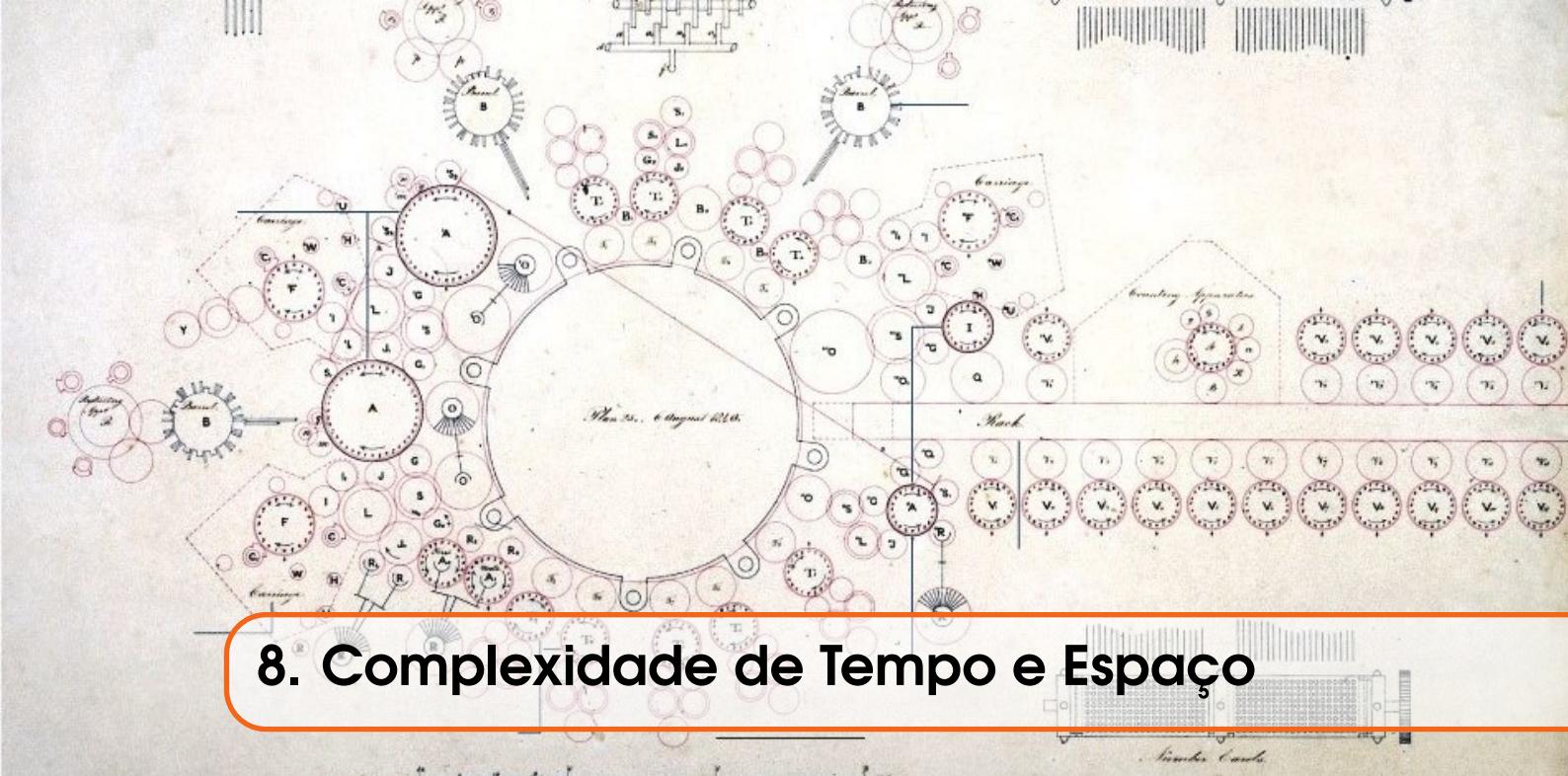
Em caso afirmativo, apresente a MT M' , e, em caso negativo, prove que M' não existe. ■

Exercício 7.17 Descreva como são as árvores de computações possíveis de MTNs tal que $\delta_1 = \delta_2$ apresentadas na “ideia da prova” do Teorema 7.6.2. ■



Parte 3: Complexidade Computacional

8	Complexidade de Tempo e Espaço ..	97
8.1	Complexidade de Tempo e de Espaço de Máquinas de Turing	
8.2	As classes P , NP e P-space	
8.3	Exercícios	
9	A classe NP	103
9.1	Decidir ou verificar?	
9.2	Certificados e verificação em tempo polinomial	
9.3	Exercícios	
10	NP-completude	109
10.1	NP-completude e o Teorema de Cook-Levin	
10.2	Lidando com problemas de busca e otimização	
10.3	Provando a NP -completude de problemas	
10.4	Exercícios	
	Bibliography	117
	Livros	



8. Complexidade de Tempo e Espaço

Intuitivamente, nós percebemos que alguns problemas computacionais parecem ser mais difíceis de serem resolvidos do que outros. Por exemplo, considere os seguintes problemas:

- (1) Dados dois números inteiros, calcular a soma dos dois números;
- (2) Dado um tabuleiro de xadrez em que as peças brancas têm a vez de jogar, determinar a jogada ótima para as peças brancas.

Resolver o primeiro problema parece ser bem mais fácil do que resolver o segundo problema. Se quisermos ser justos na comparação, podemos imaginar que os números que estamos somando têm 64 dígitos, assim estaríamos lidando com instâncias do problema de tamanho mais ou menos parecidas com a instância do problema no tabuleiro de xadrez (afinal, um tabuleiro de xadrez têm 64 posições). Ainda assim, com papel e caneta em dois minutos calculamos a soma dos dois números em questão. Por outro lado, para encontrar uma jogada ótima para as peças brancas¹ parece haver uma quantidade astronômica de possibilidades que teremos que levar em consideração. Este segundo problema parece ser difícil, pois para todas as possíveis ramificações de jogo advindas de todas as possíveis respostas do oponente, estamos tentando garantir que sempre deva existir uma próxima jogada ótima, e assim sucessivamente até o final do jogo.

Embora possa parecer “óbvio” que dados x e y , o problema de encontrar o número z , tal que $z = x + y$ é um problema simples (ao contrário de se realizar uma jogada ótima no xadrez), é importante observar que o número z procurado tem pelo menos 64 dígitos, portanto existem 10^{64} números com o tamanho da resposta procurada. Ou seja, o espaço de busca da solução para o valor que satisfaz a soma $x + y$ também é astronomicamente grande. Ainda assim, em poucos passos, sistematicamente nós conseguimos encontrar o valor z que estamos procurando.

Há algo ainda mais importante do que o fato de que somar números de especificamente 64 dígitos é mais fácil encontrar uma jogada ótima para as peças brancas em um tabuleiro 8×8 . No

¹Neste contexto, dizemos que uma jogada é ótima se existe garantidamente um xeque-mate a partir dela (mesmo que o xeque-mate seja, digamos, 40 jogadas adiante). Se quiséssemos ser mais precisos, precisaríamos levar em consideração que tal jogada pode não existir, portanto, um enunciado mais preciso do problema em questão seria “encontrar uma jogada ótima ou concluir que as peças pretas garantidamente podem empatar ou derrotar as peças brancas”.

caso mais geral, ou seja, o caso em que queremos somar dois números de n dígitos e o caso em que queremos encontrar uma jogada ótima em um tabuleiro de “xadrez generalizado” (uma versão do xadrez jogado em um tabuleiro $n \times n$), a dificuldade de se resolver o segundo problema, que já era astronomicamente maior, cresce vertiginosamente com o crescimento do valor de n .

O ponto chave é que o espaço de busca nos dois casos cresce exponencialmente com n , mas, para o primeiro problema, nós temos um algoritmo muito eficiente para encontrar a solução neste espaço de 10^n possíveis números de n dígitos. Nesta parte do curso, o nosso objetivo é tornar mais precisa a ideia intuitiva que temos de que alguns problemas são inherentemente mais difíceis de serem resolvidos do que outros.

PROBLEMAS INDECIDÍVEIS vs PROBLEMAS INTRATÁVEIS

Nesta parte do curso, os tipos de problemas que estaremos lidando são chamados de *problemas intratáveis*. Tais problemas não admitem algoritmos eficientes (como o caso do problema do xadrez generalizado) ou, na maior parte dos casos, conjectura-se não admitir algoritmos eficientes (este é o caso dos famosos problemas *NP-completos*). Ainda assim estaremos lidando com problemas decidíveis.

Podemos pensar que problemas indecidíveis, como o problema da parada, estão na categoria dos problemas “impossíveis” (e não meramente “difíceis”, como o problema do xadrez generalizado ou dos problemas *NP-completos*), pois simplesmente não existe algoritmos para tais problemas. Entretanto, é razoável pensar que instâncias grandes de problemas intratáveis, na prática, também podem ser impossíveis de serem resolvidas por limitações físicas de espaço e de tempo que a natureza nos impõe.

8.1 Complexidade de Tempo e de Espaço de Máquinas de Turing

Para tornar precisa a ideia da quantidade de trabalho que um problema exige para que possamos o solucionar, iremos definir o conceito de complexidade de tempo e complexidade de espaço de Máquinas de Turing. A complexidade de tempo de uma máquina, se refere ao número de transições que ela leva para resolver um dado problema e a complexidade de espaço se refere a quantidade de células da fita que ela usa na resolução deste problema.

Nesta parte do curso, será conveniente trabalhar com Máquinas de Turing com 3 fitas. O Teorema 6.5.1 nos diz que MTs tradicionais são equivalentes não apenas em termos de computabilidade (ou seja, problemas que estes dois modelos podem resolver são os mesmos), mas também em termos de eficiência² a MTs com 3 fitas, e portanto podemos trabalhar com O funcionamento destas máquinas é descrito a seguir.

(1) A primeira fita, chamada de *fita de entrada*, é a fita que armazena a string de entrada. Nesta fita permite-se apenas leitura (a fita é do tipo “read-only”);

(2) A segunda fita, chamada de *fita de trabalho*, é uma fita padrão em que se permite leitura e escrita. A ideia é que nesta fita em a que a computação efetivamente ocorre;

(3) A terceira fita, chamada de *fita de saída*, é a fita onde é armazenada a “resposta” do problema (esta fita também é chamada de fita de resposta). Nesta fita permite-se apenas escrita (a fita é do tipo “write-only”);

A principal vantagem de termos uma fita de saída é facilitar a nossa vida quando estivermos

²A rigor, existe uma diferença polinomial de eficiência entre estes dois modelos. Por exemplo, existem problemas que podem ser decididos em $O(n)$ passos usando uma MT com 3 fitas (ou seja, dada uma string de entrada de n bits, o número de transições que a MT de 3 fitas executa é $O(n)$), mas que requerem no mínimo $O(n^2)$ passos no modelo de MTs com apenas uma fita. Entretanto, como veremos adiante, o grau destes polinômios não são importantes para o tipo de questões que estaremos discutindo nesta parte do curso.

resolvendo problemas de não sejam de decisão (i.e., queremos obter uma string de Σ^* como saída, ao invés de apenas uma resposta SIM/NÃO). Neste caso vamos assumir que no final da computação a terceira fita contém a saída do algoritmo. Portanto, neste caso, quando formos escrever $M(x) = y$, queremos dizer que com a entrada x na fita de entrada, o algoritmo termina com y na fita de saída.

No caso de problemas de decisão, vamos assumir que a máquina escreve o símbolo 1 ou 0 na fita de saída antes de parar (dependendo do caso em que a máquina vai aceitar ou rejeitar a string). No Exercício 5.16 mostramos que, se uma Máquina de Turing sempre para, podemos assumir que a máquina escreve o bit de resultado 1 ou 0 na terceira fita antes de efetivamente parar a sua execução.

Definição 8.1.1 — Complexidade de tempo. A complexidade de tempo de uma Máquina de Turing M é uma função $t_M : \mathbb{N} \rightarrow \mathbb{N}$ tal que, para qualquer string de entrada de tamanho n , a máquina para depois de executar no máximo $t_M(n)$ transições.

■ **Exemplo 8.1** Seja M uma Máquina de Turing. Se para qualquer string de entrada de tamanho n , a máquina M sempre para depois de fazer, no máximo, $n^2 + 3n$ transições, então dizemos que a complexidade de tempo de M é $n^2 + 3n$. ■

Definição 8.1.2 — Complexidade de espaço. Dada uma MT M , a sua complexidade de espaço é uma função $s_M : \mathbb{N} \rightarrow \mathbb{N}$ tal que, para qualquer string de entrada de tamanho n , a máquina M para depois de usar no máximo $s_M(n)$ posições da fita de trabalho.

■ **Exemplo 8.2** Seja M uma Máquina de Turing. Se, para qualquer string de entrada de tamanho n , a máquina M sempre para usando no máximo, $\log n + 7$ posições da fita de trabalho, então a complexidade de espaço de M é $\log n + 7$. ■

Assim como estamos acostumado a fazer em análise de algoritmos, em muitos casos nós poderemos usar notação assintótica. No Exemplo 8.1, dizemos que a complexidade de tempo de M é $O(n^2)$. No Exemplo 8.2, dizemos que a complexidade de espaço de M é $O(\log n)$.

Notação 8.1. A notação $\text{poly}(n)$ será usada para se referir a uma função que seja assintoticamente limitada por um polinômio em n . Ou seja, se $f(n) = O(n^r)$ para algum $r \in \mathbb{N}$ constante e independente de n , então diremos que $f(n) = \text{poly}(n)$.

Definição 8.1.3 — Complexidade de tempo/espaço polinomial. Se uma MT M tem complexidade de tempo $\text{poly}(n)$, dizemos que M é *polinomial*. Se M tem complexidade de espaço $\text{poly}(n)$, dizemos que M é *de espaço polinomial*. Note que a função $\text{poly}(n)$ não necessariamente precisa ser um polinômio, pois basta que ela seja assintoticamente limitada por um polinômio.

Definição 8.1.4 — Complexidade de tempo/espaço em MTs não determinísticas. Uma Máquina de Turing não determinística é polinomial se dada uma entrada de tamanho n , todos os ramos da árvore de computações possíveis tem profundidade $\text{poly}(n)$.

Observe que se uma MTN é polinomial, então independente das possíveis escolhas não determinísticas que a MTN faz, ela sempre para depois de $\text{poly}(n)$ transições. Observe que uma consequência da Definição 8.1.4 é que, em particular, estaremos lidando apenas com MTNs que não possuem ramos infinitos em sua árvore de computações possíveis.

ESPAÇO E TEMPO: RECURSOS DISPONÍVEIS PARA SE FAZER COMPUTAÇÃO

Algo interessante de se observar é que tempo e espaço são recursos básicos que a natureza nos oferece quando pensamos em efetivamente realizar computação em um meio físico. Dependendo do substrato físico que estivermos utilizando para fazer computação, espaço e tempo podem ser traduzidos de diferentes maneiras.

Por exemplo, em um algoritmo rodando em um laptop, espaço significa memória RAM e tempo significa número de intruções executadas pelo processador. No caso de computação usando moléculas de DNA, espaço significa a quantidade de moléculas necessárias para se realizar a computação e tempo significa o número de vezes que as moléculas devem “interagir” (i.e., fazer pontes de hidrogênio). Independentemente do modelo de computação que estamos usando, em última análise, os recursos específicos utilizados parecem ter correspondência com os conceitos gerais de espaço e tempo que conhecemos em física.

8.2 As classes P, NP e P-space

Na seção anterior nós definimos o que é a complexidade de uma Máquina de Turing. Nesta seção, nós vamos usar esta definição como base para definir a complexidade inerente de se resolver determinados problemas de decisão.

Definição 8.2.1 — Decisão em tempo polinomial. Dizemos que uma linguagem L pode ser *decidida deterministicamente em tempo polinomial* se existe uma MT polinomial que decide L . Dizemos que uma linguagem L é *decidida não deterministicamente em tempo polinomial* se existe uma MT não determinística polinomial que decide L .

Definição 8.2.2 — Decisão em espaço polinomial. Uma linguagem L pode ser *decidida deterministicamente em espaço polinomial* se existe uma MT de espaço polinomial que decide L . Uma linguagem L é *decidida não deterministicamente em espaço polinomial* se existe uma MT não determinística que decide L em espaço polinomial.

Vamos agora classificar linguagens de acordo com a quantidade de recursos necessários para que possamos decidí-las. Tais conjuntos de linguagens são conhecidos como *classes* de linguagens (ou classes de problemas). De agora em diante, sempre que dissermos *classes de linguagens*, estamos nos referindo a conjuntos de linguagens.

Definição 8.2.3 — A classe P. O conjunto de todas as linguagens decidíveis deterministicamente em tempo polinomial é denotado por P.

Definição 8.2.4 — A classe NP. O conjunto de todas as linguagens decidíveis não deterministicamente em tempo polinomial é denotado por NP.

Definição 8.2.5 — A classe P-space. O conjunto de todas as linguagens decidíveis deterministicamente em espaço polinomial é denotado por P-space.

Definição 8.2.6 — A classe NP-space. O conjunto de todas as linguagens decidíveis não deterministicamente em espaço polinomial é denotado por NP-space.

O PROBLEMA P vs NP

Uma vez definidas as quatro classes acima, podemos começar a fazer o tipo de pergunta que temos feito desde o começo do curso. Estas classes são todas distintas?

Algo importante que devemos lembrar é que, muitas vezes, classes de linguagens definidas de maneiras diferentes podem eventualmente ser iguais. Por exemplo, no Capítulo 3 vimos que a classe das linguagens aceitas por DFAs era precisamente a mesma classe das linguagens aceitas por NFAs. Mas, por outro lado, em alguns modelos de computação, há diferença entre computação determinística e não determinística. Por exemplo, a classe das linguagens aceitas por PDAs não é a mesma classe das linguagens aceitas por DPDA.

Os Teoremas 7.6.1 e 7.6.2, vistos no Capítulo 5, enunciam que o conjunto de linguagens aceitas por MTs é exatamente o mesmo conjunto das linguagens aceitas por MTNs. Para provar tal resultado, o argumento que usamos é que uma MT pode simular uma MTN (o mesmo argumento vale para provar que MTs *decidem* as mesmas linguagens que podem ser decididas por MTNs). A pergunta chave neste capítulo é a seguinte: uma MT pode sempre simular de maneira *eficiente* uma dada MTN?

No capítulo seguinte veremos o problema SAT pode ser resolvido por Máquinas de Turing não determinísticas de tempo polinomial, algo que conjectura-se ser impossível de ser realizado por Máquinas de Turing determinísticas. O mesmo ocorre com vários outros problemas conhecidos como problemas NP-completos.

Exercício 8.1 Mostre que $P \subseteq NP$.

Solução: Seja $L \in P$. Pela Definição 8.2.3, existe uma MT polinomial $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ que decide L . Agora considere a Máquina de Turing não determinística $N = (Q, \Sigma, \Gamma, (\delta, \delta), q_0, B, F)$. A máquina N comporta-se exatamente da mesma maneira que M , portanto N também decide L em tempo polinomial. Logo $L \in NP$ e consequentemente $P \subseteq NP$. \square

Se quisermos provar que $P \neq NP$, teríamos que ser capazes de mostrar que a classe P está *estritamente* contida na classe NP , ou seja, devemos provar que existe pelo menos um problema que esteja em NP , mas que não esteja em P .

A conjectura normalmente aceita em ciência da computação é que $P \neq NP$. Em particular, vários problemas importantes, conhecidos como problemas NP-completos, são os problemas candidatos a pertencerem à classe $NP \setminus P$. O Problema da Satisfatibilidade, conhecido simplesmente como problema SAT (veja a Definição 7.2.6, no Capítulo 7), é um destes problemas. Mostrar que o problema SAT está na classe NP é uma tarefa simples (veremos isso no próximo capítulo). A parte difícil, e que é um dos maiores problemas matemáticos em aberto atualmente, é mostrar que o problema não pertence a classe P .

Qual é a dificuldade de se provar que o problema SAT (ou qualquer outro problema candidato a estar em $NP \setminus P$) não admite um algoritmo polinomial? Como já vimos em capítulos anteriores, em geral, não é uma tarefa fácil provar que determinado algoritmo não existe. Para demonstrarmos que $L_{SAT} \notin P$ teríamos que excluir logicamente a possibilidade de que todos os infinitos algoritmos polinomiais falham na tarefa de decidir L_{SAT} .

P-SPACE vs NP-SPACE?

O problema P vs NP é muito famoso, mas, por outro lado, por que nunca escutamos nada sobre o problema P-space vs NP-space. Qual é o motivo disso? O motivo é que este não é um problema em aberto. Não é tão difícil mostrar que P-space = NP-space.

Uma maneira diferente de se definir classes de complexidade, e que pode ser útil em algumas circunstâncias, é a seguinte.

Definição 8.2.7 — TIME($f(n)$). Dada uma função $f : \mathbb{N} \rightarrow \mathbb{N}$, a classe $\text{TIME}(f(n))$ é o conjunto de toda linguagem que pode ser decidida por uma Máquina de Turing com complexidade de tempo $c \cdot f(n)$, para alguma constante c .

■ **Exemplo 8.3** A classe $\text{TIME}(n^3)$ é a classe de toda linguagem que pode ser decidida por uma MT que execute $c \cdot |x|^3$ transições, para alguma constante $c > 0$, para qualquer string x fornecida como entrada para M . ■

■ **Exemplo 8.4** A classe $\text{TIME}(2^n)$, que é conjunto de toda linguagem que pode ser decidida por uma MT que execute $c \cdot 2^{|x|}$ transições, , para alguma constante $c > 0$, para qualquer string x fornecida como entrada para M . ■

Definição 8.2.8 — SPACE($f(n)$). Dada uma função $f : \mathbb{N} \rightarrow \mathbb{N}$, a classe $\text{SPACE}(f(n))$ é o conjunto de toda linguagem que pode ser decidida por uma Máquina de Turing com complexidade de espaço $c \cdot f(n)$, para alguma constante c .

8.3 Exercícios

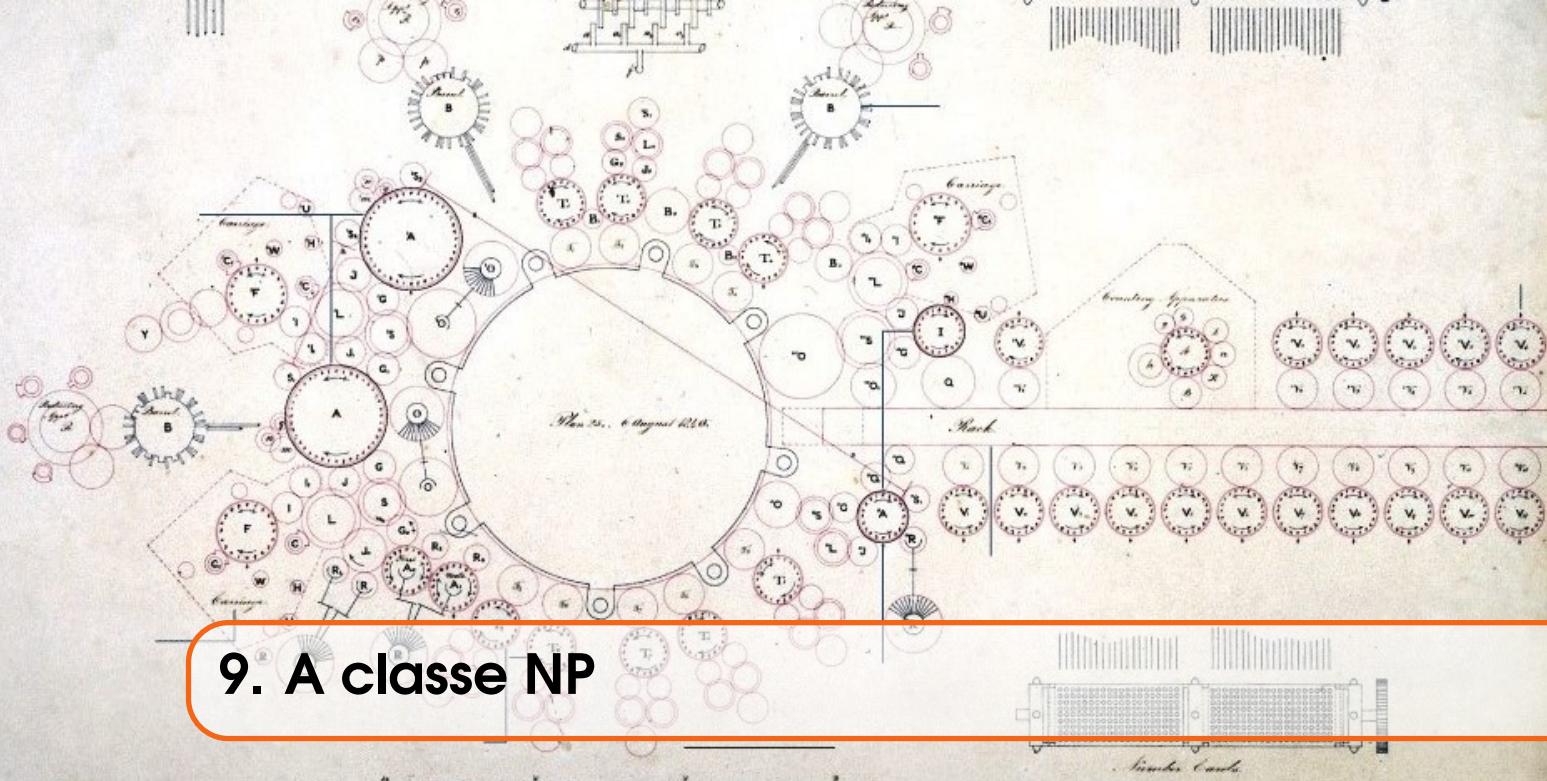
Exercício 8.2 Apresente uma definição para a classe P em termos da definição da classe $\text{TIME}(f(n))$. ■

Exercício 8.3 Caso estivéssemos utilizando uma Máquina de Turing usando apenas 1 fita, a classe P seria a mesma ou seria diferente? Justifique a sua resposta. Qual seria a desvantagem de se definir P-space usando MTs que possuam apenas 1 fita? ■

Exercício 8.4 Caso estivéssemos utilizando uma Máquina de Turing usando apenas 1 fita, teríamos que alterar a nossa definição de complexidade de espaço. Há alguma vantagem em se usar MTs com 3 fitas ao invés de MTs com apenas 1 fita quando estamos lidando com complexidade de espaço? ■

Exercício 8.5 Lembrando que \mathcal{R} é o conjunto das linguagens recursivas, prove que $\text{NP} \subseteq \mathcal{R}$. ■

Exercício 8.6 Forneça uma definição para $\text{NTIME}(f(n))$ de maneira semelhante a Definição 8.2.7, mas agora considerando Máquinas de Turing não Determinísticas. ■



9. A classe NP

O que é mais fácil? **Decidir** se existe uma solução para uma dada instância do problema Sudoku, ou meramente **verificar** se uma solução que já nos foi fornecida “de bandeja” está correta?

	2		5		1		9	
8			2		3			6
	3			6			7	
		1				6		
5	4						1	9
		2			7			
	9			3			8	
2			8		4			7
	1		9		7		6	

Problema

4	2	6	5	7	1	3	9	8
8	5	7	2	9	3	1	4	6
1	3	9	4	6	8	2	7	5
9	7	1	3	8	5	6	2	4
5	4	3	7	2	6	8	1	9
6	8	2	1	4	9	7	5	3
7	9	4	6	3	2	5	8	1
2	6	5	8	1	4	9	3	7
3	1	8	9	5	7	4	6	2

Solução

Figure 9.1: O Problema Sudoku.

Na Figura 9.1 à esquerda apresentamos uma instância do Problema Sudoku. O problema de decisão que estamos interessados é o seguinte: dado um grid 9×9 , queremos decidir se é possível preencher as posições faltantes tal que toda linha, toda a coluna, e cada um dos blocos 3×3 do grid tenha exatamente um dígito diferente. Alguns grids admitem solução (instâncias verdadeiras) e outros não admitem solução (instâncias falsas). Na Figura 9.1 à direita temos um exemplo de preenchimento do grid original. Observe que tal grid preenchido é uma prova de que a instância original é verdadeira (i.e., admite um preenchimento correto). Agora considere um problema de decisão diferente: dado um grid 9×9 preenchido (como o da figura a direita), queremos decidir se, de fato, o preenchimento está correto.

Qual dos dois problemas parece ser mais difícil? Decidir se existe uma solução para uma dada instância do problema Sudoku ou simplesmente verificar se uma solução que já nos foi fornecida está correta? Neste capítulo veremos que esta dicotomia *decidir vs verificar* está no coração de

todos os problemas da classe NP e está intimamente ligada ao famoso problema P vs NP.

9.1 Decidir ou verificar?

Nesta seção nós vamos formalizar a ideia intuitiva que temos da oposição entre decidir e verificar um problema. Para formalizar essa ideia, vamos usar o problema SAT. Primeiramente relembramos que uma fórmula com n variáveis x_1, \dots, x_n escrita em CNF é uma fórmula booleana que tem a forma $\phi = (l_{11} \vee l_{12} \vee \dots \vee l_{1k_1}) \wedge (l_{21} \vee l_{22} \vee \dots \vee l_{2k_2}) \wedge \dots \wedge (l_{m1} \vee l_{m2} \vee \dots \vee l_{mk_m})$, sendo que os literais l_{ij} podem ser tanto x_1, \dots, x_n , quanto $\bar{x}_1, \dots, \bar{x}_n$. A rigor, uma instância do problema SAT é uma string (as instâncias verdadeiras são strings $\llbracket \phi \rrbracket \in L_{\text{SAT}}$ e as instâncias falsas são strings $\llbracket \phi \rrbracket \notin L_{\text{SAT}}$), mas aqui seremos mais flexíveis em nossa notação.

Seja ϕ uma instância verdadeira do problema SAT. Agora imagine que alguém queira nos convencer que ϕ é, de fato, satisfazível, mas somos céticos em relação a esse fato. Vamos considerar agora dois cenários:

- (1) A pessoa que quer nos convencer simplesmente nos passa a fórmula booleana ϕ e afirma que a fórmula é satisfazível.
- (2) A pessoa que quer nos convencer nos passa a fórmula booleana ϕ juntamente com uma valoração $v = v_1, \dots, v_n$ que satisfaz ϕ (ou seja uma sequência de valores $v_i \in \{\text{V}, \text{F}\}$, tal que, fazendo a substituição $x_i = v_i$, a fórmula ϕ assume o valor verdade V).

Em qual destes dois cenários nós teríamos menos trabalho?

Aparentemente o cenário (2) é mais fácil, pois nós recebemos uma valoração “de bandeja” que atesta que a fórmula é satisfazível. Tudo o que nos resta fazer, com todo o nosso ceticismo, é substituir os valores v_1, \dots, v_n nos locais da fórmula onde apareçam as variáveis x_1, \dots, x_n e verificar se, de fato, todas as cláusulas de ϕ são satisfeitas. Mais precisamente, veremos que é simples apresentar um algoritmo polinomial que toma (ϕ, v) como entrada e verifica se a valoração v satisfaz ϕ . Na Seção 9.2 veremos o pseudo-código deste algoritmo.

Por outro lado, no cenário (1), teríamos muito mais trabalho para nos convencermos de que ϕ é satisfazível. Parece difícil escapar da ideia de apelar para a força bruta e testar cada uma das 2^n possíveis valorações até que você encontremos a valoração que satisfaça ϕ (eventualmente encontrariamos uma valoração, pois a premissa é que ϕ é satisfazível). Uma vez que nós encontremos uma valoração que passe em nosso teste, nós ficamos convencidos que ϕ é satisfazível.

HÁ COMO ESCAPAR DA FORÇA BRUTA?

Embora a ideia de testar todas as valorações possíveis pareça ingênua, os melhores algoritmos conhecidos para o problema SAT não escapam de ideias semelhantes. Essencialmente, no pior caso, todos os algoritmos conhecidos para o problema SAT usam estratégias que exploram um número exponencial de valorações até que uma seja encontrada ou, no caso da instâncias ser falsa, conclui que a fórmula não é satisfazível.

Algo que pode ter passado despercebido nesta discussão é que uma instância verdadeira do problema SAT têm um *certificado* que atesta ela é realmente verdadeira. Esse certificado é a valoração correta, fornecida “de presente” no cenário (2). Um valoração correta funciona como um “certificado de garantia” de que a fórmula é realmente satisfazível. Além disso, nós podemos *verificar* se o certificado que nos foi fornecido é válido fazendo pouco esforço computacional. Mais precisamente, tendo a fórmula e o certificado em mãos, existe um algoritmo de tempo polinomial que responde SIM se v satisfaz ϕ e NÃO se v não satisfaz ϕ .

Outro ponto fundamental é que uma instância falsa ϕ' , por definição, não possui uma valoração que possa ser usada como certificado. Com isso, o nosso algoritmo verificador irá sempre refutar

(ϕ', v) , independente da valoração v recebida. Em outras palavras, nós, sendo céticos, nunca poderemos ser convencidos incorretamente de que uma fórmula seja satisfazível quando esta fórmula não for realmente satisfazível.

GRAFOS HAMILTONIANOS

Podemos repetir o mesmo raciocínio que usamos na nossa discussão do problema SAT para vários outros problemas de decisão. Vamos considerar, por exemplo, o problema de testar se um grafo é hamiltoniano. De maneira semelhante ao caso do problema SAT, imagine agora que alguém queira nos convencer que um dado grafo G de n vértices é hamiltoniano.

Novamente, vamos pensar em dois cenários. Um cenário em que recebemos apenas G e outro cenário em que recebemos G juntamente com uma permutação v_1, \dots, v_n dos vértices do grafo atestando que G é hamiltoniano (ou seja, uma sequência de vértices tal que, podemos percorrer um ciclo hamiltoniano no grafo usando-se esta sequência de vértices como “guia”).

No caso em que recebemos o grafo juntamente com a permutação, precisaríamos de pouco esforço computacional para verificar se G é hamiltoniano: basta tomarmos v_1, \dots, v_n e verificar se, de fato, v_1, \dots, v_n pode ser usado como guia para percorremos um circuito hamiltoniano em G . Para tal, primeiramente verificamos se a sequência é uma permutação de $V(G)$ (uma permutação de $V(G)$ é uma sequência de vértices sem repetição e que todos os vértices de G aparecem na sequência). Em seguida, verificamos se podemos percorrer o grafo usando esta sequência de vértices como guia (ou seja, testamos se as arestas $v_i v_{i+1} \in E(G)$, $i = 1, 2, \dots, n-1$ estão presentes no grafo. Para finalizar, verificamos se existe a aresta para “fechar” o ciclo e voltarmos ao vértice inicial (ou seja, se $v_n v_1 \in E(G)$).

Observe que, novamente, temos a seguinte situação: se G não é hamiltoniano, então por definição não existe um certificado (ou seja, uma permutação) que ateste que G é hamiltoniano. Na próxima seção, o nosso objetivo será generalizar as ideias de certificados e verificação eficiente para problemas de decisão em geral.

9.2 Certificados e verificação em tempo polinomial

Apresentamos agora a definição formal de certificados e verificação polinomial:

Definição 9.2.1 — Certificados e Verificadores Polinomiais. Seja $L \subseteq \Sigma^*$ um problema de decisão. Dizemos que o problema L pode ser *verificado* em tempo polinomial se existe uma Máquina de Turing polinomial V_L , chamada de verificador de L , e existe um polinômio $\text{poly}(n)$ tal que o seguinte é satisfeito:

- (a) Para cada string $x \in L$, existe pelo menos uma string $c \in \Sigma^*$, chamada de *certificado de x* tal que $V_L(x, c) = 1$. Além disso, a string c é no máximo “polinomialmente grande” em função de $|x|$. Mais precisamente, $|c| = \text{poly}(|x|)$.
- (b) Por outro lado, se $x \notin L$, então $\forall c \in \Sigma^*$, $V_L(x, c) = 0$ (em outras palavras, se x é uma instância falsa do problema L , não importa qual string c passamos como candidata a ser o certificado de x , o verificador V_L vai sempre rejeitar a entrada).

9.2.1 Verificando o problema SAT em tempo polinomial

Para sermos mais concretos, vamos formalizar as ideias vistas na Seção 9.1 e explorar a ideia de verificação polinomial no caso do problema L_{SAT} . Para tal, devemos mostrar uma MT polinomial V que possa ser usada como verificador e um polinômio $\text{poly}(n)$ tal que o tamanho dos certificados das instâncias verdadeiras de tamanho n nunca são maiores $\text{poly}(n)$.

Vamos começar com os certificados. Seja uma instância verdadeira $\perp \phi \perp$ do problema L_{SAT} e

seja n o número de variáveis que aparece nesta fórmula (observe que cada variável x_i pode aparecer em ϕ na forma original ou na forma negada). Como já discutimos na Seção 9.1, um certificado para uma fórmula satisfazível pode ser uma valoração que a satisfaça. Mais precisamente, um certificado para $\llbracket \phi \rrbracket$ é a string de bits $v = v_1 v_2 \dots v_n$ tal que cada bit v_i indica o valor que a variável x_i deve receber para que a fórmula seja satisfeita. O valor V ou F que x_i deve receber depende do bit v_i ser 1 ou 0.

De maneira mais concreta, considere a fórmula satisfazível $\phi_1 = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_3)$ do Exemplo 7.2. Um certificado para $\llbracket \phi \rrbracket$ é a string 110 (pois a valoração $x_1 = V$, $x_2 = V$ e $x_3 = F$ satisfaz ϕ).

Exercício 9.1 Mostre strings v , que sugerimos que sejam usadas como certificado, satisfazem a condição $|v| = \text{poly}(|\llbracket \phi \rrbracket|)$. ■

Solução: O certificado tem tamanho linear no tamanho da instância, pois uma instância com n variáveis tem tamanho pelo menos n (o tamanho exato de $|\llbracket \phi \rrbracket|$ depende de apresentarmos o esquema exato de codificação que usamos para escrever a fórmula em binário, mas não é difícil ver que são necessários pelo menos n bits para uma fórmula com n variáveis) e o certificado v tem tamanho exatamente n , pois podemos usar um bit para cada variável que aparece na fórmula. Portanto a função $\text{poly}(n)$ pode ser a função linear $f(n) = n$.

Agora que já vimos que strings podem ser usadas como certificados para instâncias verdadeiras de L_{SAT} , vamos mostrar formalmente um algoritmo verificador para L_{SAT} . Como mencionamos no Capítulo 7, quando estamos lidando com problemas concretos, é muito mais conveniente apresentar o pseudo-código de um algoritmo ao invés de usar Máquinas de Turing. O verificador, apresentado no Algoritmo **Verificador_SAT**(ϕ, v), recebe uma fórmula booleana ϕ e uma valoração v e retorna SIM ou NÃO, dependendo do caso em que v satisfaça ou não satisfaça a fórmula ϕ .

Verificador_SAT: (ϕ, v)

```

1: for  $i = 1; i \leq m; i++ \text{ do}$                                 /* Para cada uma das  $m$  cláusulas */
2:    $C_i = \text{Falso}$                                          /* Assume inicialmente que  $C_i$  não vai ser satisfeita */
3:   for  $j = 1; j \leq k_i; j++ \text{ do}$                          /* Para cada literal da cláusula */
4:     IND = GetIndexVar( $l_{ij}$ )                                     /* Obtém índice da variável que aparece no literal */
5:     if LitSat( $l_{ij}, v_{\text{IND}}$ ) then                            /* Verifica se o bit  $v_{\text{IND}}$  da valoração  $v$  faz  $l_{ij} = V$  */
6:        $C_i = \text{True}$                                          /* Em caso afirmativo a cláusula foi satisfeita */
7:       Break                                                 /* Sai do loop interno para testar próxima cláusula */
8:     if  $C_i = \text{Falso}$  then                                /* Se todos literais falharam, a cláusula fica falsa */
9:       Return Falso                                         /* Portanto a  $v$  não satisfaz  $\phi$  */
10:  Return Verdadeiro                                       /* Passou no teste da linha 10 para toda cláusula */

```

No Algoritmo **Verificador_SAT**(ϕ, v), as m cláusulas da fórmula são ϕ de C_1, C_2, \dots, C_m , ou seja, ϕ é uma conjunção da forma $C_1 \wedge C_2 \wedge \dots \wedge C_m$, sendo que cada cláusula C_i é uma disjunção de literais da forma $C_i = (l_{i1} \vee l_{i2} \vee \dots \vee l_{ik_i})$, onde k_i é o número de literais da cláusula C_i . A função $\text{GetVar}(l_{ij})$ é uma função que retorna o índice da variável que aparece no literal l_{ij} , por exemplo, tanto no caso em que $l_{ij} = x_4$, quanto no caso em que $l_{ij} = \bar{x}_4$, a função retorna 4. A função $\text{LitSat}(l_{ij}, v_{\text{IND}})$ testa se o literal l_{ij} é satisfeito pela valoração em questão. Por exemplo, se $l_{ij} = x_4$, o literal é satisfeito quando $v_4 = 1$. Por outro lado, se $l_{ij} = \bar{x}_4$, o literal é satisfeito quando $v_4 = 0$.

Exercício 9.2 Mostre que $L_H = \{\llcorner G \lrcorner ; G \text{ é um grafo hamiltoniano}\}$ pode ser verificado em tempo polinomial. ■

9.2.2 Redefinindo a classe NP

O fato de uma linguagem poder ser verificada em tempo polinomial não necessariamente implica que a linguagem possa ser decidida em tempo polinomial. Veremos a seguir que uma das maneiras de enunciar a conjectura de que $P \neq NP$ é conjecturar que existem problemas que podem ser verificados em tempo polinomial, mas que não podem ser decididos em tempo polinomial. Em particular, o problema L_{SAT} é candidato a ser um destes problemas, assim como o problema do grafo hamiltoniano e o problema de decidir se um grid $n \times n$ do problema Sudoku admite preenchimento correto.

Como dissemos, a possibilidade de verificarmos uma linguagem L em tempo polinomial não implica necessariamente na possibilidade de decidirmos L em tempo polinomial. Por outro lado, a possibilidade decidirmos L em tempo polinomial implica na possibilidade verificarmos L em tempo polinomial.

Teorema 9.2.1 Seja $L \subseteq \Sigma^*$ se $L \in P$, então L pode ser verificada em tempo polinomial.

Ideia da prova: Se $L \in P$, então existe uma MT polinomial M que decide L . O que vamos fazer é usar a própria máquina M como verificador de L (possivelmente fazendo uma pequena alteração em M para que ela tome dois argumentos de entrada, pois verificadores tomam dois argumentos de entrada). E quais seriam os certificados da instâncias verdadeiras? Para toda string $x \in L$, o certificado do x é a string ε . O verificador se comporta exatamente com o algoritmo que decide L , simplesmente ignorando o certificado.

Teorema 9.2.2 Seja \mathcal{C} a classe de problemas que podem ser verificados em tempo polinomial. Então $\mathcal{C} = NP$.

Pelo Teorema 9.2.2, podemos definir a classe NP alternativamente da seguinte maneira.

Definição 9.2.2 — Classe NP (definição equivalente). Uma linguagem $L \subseteq \Sigma^*$ está em NP se existe um polinômio $p : \mathbb{N} \rightarrow \mathbb{N}$ e uma MT M com complexidade de tempo polinomial (essa MT é chamada de verificador de L) tal que $\forall x \in \Sigma^*$:

$$x \in L \Leftrightarrow \exists u \in \Sigma^{p(|x|)}; M(x, u) = 1$$

Para $x \in L$ e $u \in \Sigma^{p(|x|)}$ satisfazendo $M(x, u) = 1$, a string u é chamada de certificado de x (com relação à linguagem L e à MT M).

Note que a definição acima parece ser bastante diferente da definição original da classe NP (i.e., Definição 8.2.4), entretanto, ambas se referem exatamente o mesmo conjunto de linguagens.

9.3 Exercícios

Exercício 9.3 Seja uma linguagem $L \subseteq \Sigma^*$ e seja $\bar{L} = \Sigma^* \setminus L$. Dada uma classe de complexidade \mathcal{C} , definimos o complemento de \mathcal{C} da seguinte maneira: $co\text{-}\mathcal{C} = \{\bar{L}; L \in \mathcal{C}\}$. Com relação a esta definição, responda o seguinte:

- Seja L_E o problema de testar se um grafo não é euleriano. Prove que $L_E \in co\text{-P}$.
- Prove que $P = co\text{-P}$.

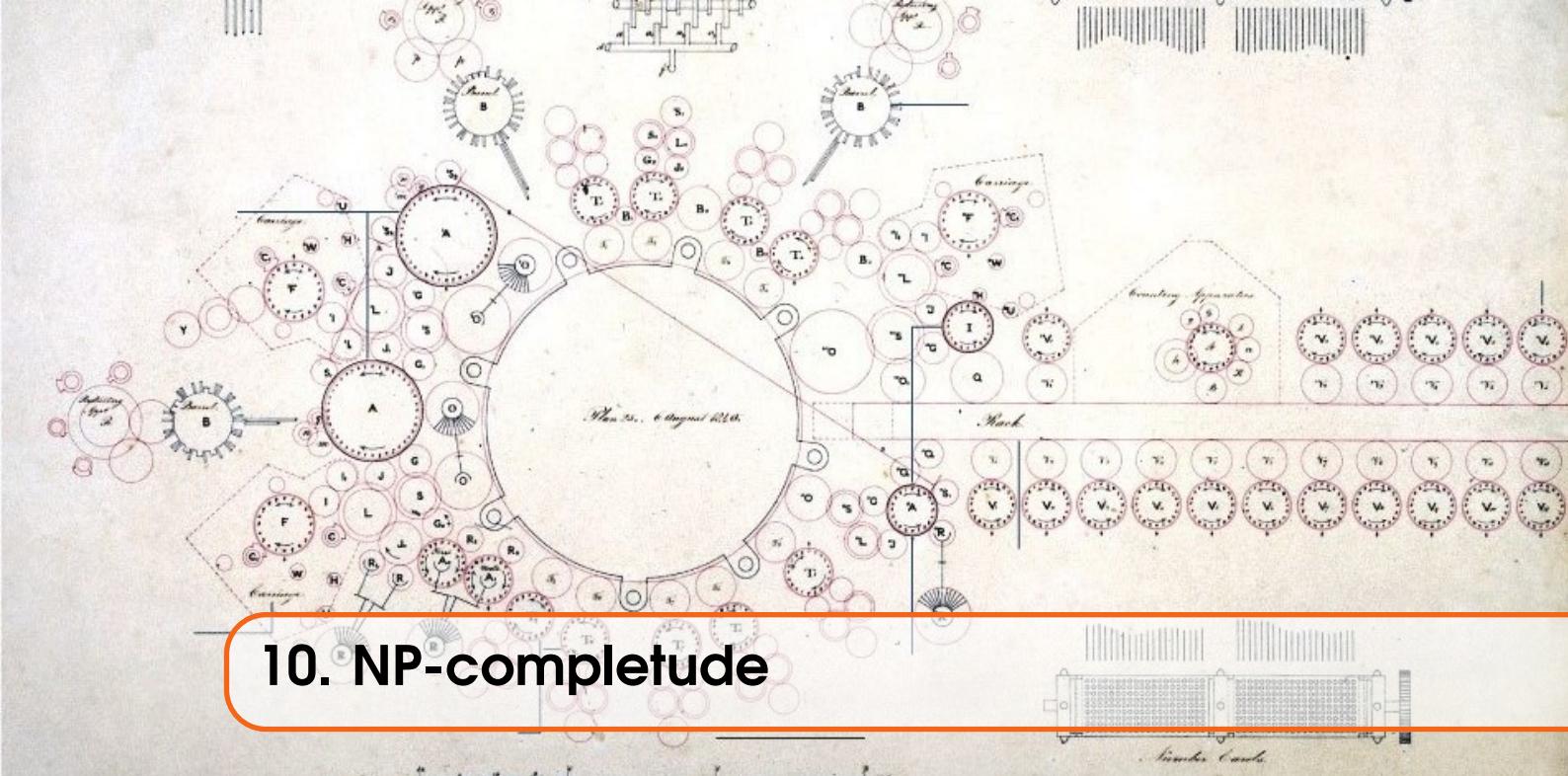
■

Exercício 9.4 Uma conjectura conhecida em complexidade computacional é que $NP \neq co\text{-NP}$. Por que não poderíamos provar que $NP = co\text{-NP}$ usando uma estratégia de prova semelhante a estratégia usada no exercício anterior?

■

Exercício 9.5 Na Definição 9.2.1, item (a), um dos requisitos é que o certificado tenha tamanho polinomial no tamanho da instância. Por que motivo faz sentido esta restrição?

■



10. NP-completude

Neste capítulo estudaremos mais a fundo os problemas da classe NP. Em particular, veremos que alguns destes problemas podem ser considerados os “mais difíceis” de toda classe. Mas como poderíamos tornar precisa a ideia de que um certo problema é o mais difícil da classe NP? O que vamos fazer é provar que se existe um algoritmo polinomial para tal problema, então todos os demais problemas da classe NP também admitem algoritmos polinomiais.

10.1 NP-completude e o Teorema de Cook-Levin

Nesta seção vamos apresentar um teorema que foi provado por Stephen Cook e Leonid Levin no início da década de 70. O teorema diz o seguinte: caso exista um algoritmo polinomial para o problema SAT, então todos os problemas da classe NP podem ser resolvidos em tempo polinomial.

Antes de apresentar o Teorema de Cook-Levin, vamos atacar a seguinte questão: Como poderia a existência de um algoritmo polinomial para um problema L_2 implicar a existência de um algoritmo polinomial para outro problema L_1 , sendo que estes dois problemas podem ser aparentemente muito diferentes? O primeiro passo para entender isso é entender a ideia de *redução polinomial*.

Suponha que existe um algoritmo polinomial M_2 para L_2 . A ideia central é mostrar que instâncias do problema L_1 podem ser vistas, também como instâncias de L_2 “disfarçadas”. Como isso é feito? Apresentando um algoritmo R que, tome uma instância x do problema L_1 como entrada e retorne uma instância y de L_2 como saída. Este algoritmo R é chamado de redução. O ponto chave é que R deve ter a seguinte propriedade: se x é uma instância verdadeira de L_1 , y também deve ser uma instância verdadeira de L_2 , e, por outro lado, se x é uma instância falsa de L_1 , y também deve ser uma instância falsa de L_2 .

O QUE ERA UMA INSTÂNCIA MESMO?

Lembrando que uma instância de um problema L é uma string qualquer $x \in \Sigma^*$ que codifica um objeto (por exemplo, se o problema em questão for um problema sobre grafos, tipicamente x é a codificação de um grafo). Se $x \in L$, dizemos que x é uma instância verdadeira de L e se $x \notin L$, dizemos que x é uma instância falsa de L .

Mas qualquer redução R basta? Não! O algoritmo R , que transforma instâncias de L_1 para L_2 , deve ser polinomial. Desta maneira podemos obter um algoritmo polinomial M_1 para resolver L_1 usando uma combinação de R com M_2 (ambos polinomiais). Para resolver L_1 , primeiramente transformamos o problema L_1 em L_2 usando R e, depois, usamos M_2 para resolver L_2 .

Definição 10.1.1 — Redução de tempo polinomial. Sejam L_1 e L_2 linguagens sobre Σ . Dizemos que L_1 é *polinomialmente redutível* à L_2 se existe uma MT polinomial R tal que $x \in L_1 \Leftrightarrow R(x) \in L_2$. Neste caso escrevemos $L_1 \leq_P L_2$. Para simplificar, muitas vezes diremos simplesmente que L_1 é *redutível* a L_2 (ao invés de dizer “polinomialmente” redutível).

■ **Exemplo 10.1** Neste momento ainda é um pouco cedo para vermos explicitamente como tais reduções funcionam. Mas exemplificar o conceito, algo que podemos adiantar é que com certo trabalho é possível demonstrar que L_{SAT} é redutível ao problema L_{H} (i.e., usando a notação da Definição 10.1.1, escrevemos $L_{\text{SAT}} \leq_P L_{\text{H}}$). ■

Concretamente, o que devemos fazer para provar que existe a redução que o Exemplo 10.1 faz referência, é apresentarmos um *algoritmo polinomial* que transforma fórmulas em grafos. Mais precisamente, o algoritmo transforma fórmulas satisfazíveis em grafos hamiltonianos e fórmulas não satisfazíveis em grafos não hamiltonianos.

Antes de entendermos como obter reduções entre problemas, vamos ver um teorema que nos esclarece por que um algoritmo polinomial para um problema L_2 e uma redução de L_1 para L_2 implica em um algoritmo polinomial para L_1 .

Teorema 10.1.1 Se $L_1 \leq_P L_2$ e $L_2 \in \mathbb{P}$, então $L_1 \in \mathbb{P}$.

Prova: Como $L_2 \in \mathbb{P}$, então existe uma MT polinomial M_2 que decide L_2 . Como $L_1 \leq_P L_2$, existe uma MT polinomial R tal que $x \in L_1 \Leftrightarrow R(x) \in L_2$. Consire o algoritmo polinomial M_1 que, dado $x \in \Sigma^*$, tem o seguinte comportamento: $M_1(x) = M_2(R(x))$. Portanto, $M_1(x) = 1$, se $x \in L_1$, e $M_1(x) = 0$, se $x \notin L_1$. Consequentemente M_1 decide L_1 e, assim, $L_1 \in \mathbb{P}$. □

Definição 10.1.2 — Problemas NP-difíceis. A classe NP-difícil é o conjunto de problemas de decisão L tal que $\forall L' \in \text{NP}, L' \leq_P L$. Se $L \in \text{NP-difícil}$, normalmente dizemos que L é *NP-difícil*.

Definição 10.1.3 — Problemas NP-completos. A classe NP-completo é o conjunto de problemas de decisão L tal que o seguinte é satisfeito:

- L é NP-difícil;
- $L \in \text{NP}$.

Dizemos também que L é *NP-completo* no caso em que $L \in \text{NP-completo}$.

Teorema 10.1.2 Seja $L \in \text{NP-completo}$. Se $L \in \mathbb{P}$, então $\mathbb{P} = \text{NP}$.

Prova: Suponha que o problema NP-completo L esteja contido na classe \mathbb{P} . Seja L' um problema qualquer de NP. Como L é NP-completo, então $L' \leq_P L$, e portanto, pelo Teorema 10.1.1, $L' \in \mathbb{P}$. Consequentemente $\text{NP} \subseteq \mathbb{P}$. Como $\mathbb{P} \subseteq \text{NP}$ (ver Exercício 8.1), então $\mathbb{P} = \text{NP}$. □

Teorema 10.1.3 — Teorema de Cook-Levin. L_{SAT} é NP-completo.

A prova do Teorema de Cook-Levin está fora do escopo deste curso.

Corolário 10.1.4 Se L_{SAT} admite um algoritmo polinomial, então é $P = NP$.

Uma vez que a maior parte dos pesquisadores da área de complexidade computacional conjectura que $P \neq NP$, uma consequência é que é bastante improvável que o problema SAT (e também qualquer problema NP-completo ou NP-difícil) admita um algoritmo polinomial.

Exercício 10.1 Mostre que a relação \leq_P é transitiva. ■

10.2 Lidando com problemas de busca e otimização

Muitos problemas que aparecem na prática não são problemas de decisão. Muitos destes problemas têm a forma de *problemas de busca* ou *problemas de otimização*. Por exemplo, considere o problema de fatorar um número¹, isto é, dado n , queremos retornar a sequência de fatores primos p_1, \dots, p_k , tal que o produto destes números seja n . Este tipo de problema é chamado de problema de busca.

Um outro problema famoso, conhecido como *Problema do Caixeiro Viajante* é o seguinte: dado um grafo com peso nas arestas representando as distâncias entre pares de cidades, encontrar o menor *tour* percorrendo cada cidade uma única vez. Este tipo de problema é conhecido como problema de otimização. Tais problemas são variações de problemas de busca em que a solução que estamos buscando satisfaz algum critério de maximização ou minimização.

A VERSÃO DE DECISÃO DE UM PROBLEMA

Na área de complexidade computacional, quando temos um problema de busca ou otimização, é bastante comum definirmos um problema de decisão semelhante ao problema original e trabalharmos com esta versão de decisão do problema. Qual é vantagem disso? Problemas de decisão, embora mais simples, comumente ainda assim “capturam” a dificuldade inerente dos problemas originais.

A vantagem que temos em trabalhar com problemas de decisão é que estes problemas podem ser classificados como pertencendo as classes P, NP e NP-completo. Por outro lado problemas de busca e otimização, embora possam ser descritos de maneira formal, não possuem definições simples práticas como definições em termos de linguagens.

■ **Exemplo 10.2 — O problema da clique máxima (MAXCLIQUE).** Dado um grafo G , encontrar a maior clique de G . Note que, do ponto de vista formal, para resolver este problema precisamos encontrar uma MT que tome como entrada $\llcorner G \lrcorner$ e retorne $\llcorner S \lrcorner$, tal que S é um conjunto de vértices $S \subseteq V(G)$ que induza a maior clique do grafo G . ■

O problema acima não é de decisão, mas poderíamos pensar em uma versão de decisão deste mesmo problema da seguinte maneira: dado um par (G, k) , determinar se G tem uma clique de tamanho pelo menos k . Formalmente, a versão de decisão do problema é seguinte:

Definição 10.2.1 — O problema da clique (CLIQUE). O problema de decisão, conhecido como problema CLIQUE, é definido pela linguagem $L_{\text{CL}} = \{\llcorner(G, k)\lrcorner \in \Sigma^* \mid G \text{ é um grafo que contém uma clique de tamanho pelo menos } k\}$.

¹O problema de fatoração é um problema computacional clássico e a dificuldade de resolução do mesmo é a base do protocolo de criptografia RSA.

Exercício 10.2 Mostre que se existe um algoritmo polinomial para o problema MAXCLIQUE, então existe um algoritmo polinomial para o problema de decisão CLIQUE. ■

Na seção 10.3, provaremos que o problema CLIQUE é NP-completo. Uma vez que um algoritmo polinomial para o problema de otimização MAXCLIQUE pode facilmente ser adaptado para resolver o problema de decisão CLIQUE (Exercício 10.2), conclui-se que se o problema de otimização puder ser resolvido em tempo polinomial, então $P = NP$. Em outras palavras, para mostrarmos que ambos os problemas são intratáveis, basta mostrarmos que a versão de decisão de problema é intratável.

Um outro problema de otimização semelhante ao problema MAXCLIQUE é o problema de encontrar o maior conjunto independente de um grafo. Neste problema, ao invés de estarmos procurando pelo maior subgrafo completo, estamos procurando pelo maior subgrafo que não contém nenhuma aresta. A versão de decisão deste problema é a seguinte:

Definição 10.2.2 — Problema do conjunto Independente (CI). O problema de decisão, conhecido como problema do conjunto independente, é definido pela linguagem $L_{CI} = \{\llcorner(G, k)\lrcorner \in \Sigma^* \mid G \text{ é um grafo que contém um conjunto independente de tamanho pelo menos } k\}$.

10.3 Provando a NP-completude de problemas

Na seção 10.1 vimos que o problema SAT parece ter um papel especial na classe NP. Uma maneira de interpretar o Teorema de Cook-Levin é que qualquer problema da classe NP pode ser visto como uma versão “disfarçada” do problema SAT, pois todos estes problemas podem ser reduzidos ao problema SAT. O que veremos agora é que outros problemas da classe NP também possuem esta mesma propriedade. Em outras palavras, diversos outros problemas também são NP-completos. Para que possamos explorar isto, o teorema a seguir será essencial.

Teorema 10.3.1 Suponha que L_1 é um problema NP-completo. Se $L_2 \in NP$ e $L_1 \leq_P L_2$, então L_2 também é um problema NP-completo.

Prova: Seja uma linguagem qualquer L de NP. Como L_1 é NP-completo, temos que $L \leq_P L_1$. Como a relação \leq_P é transitiva (veja Exercício 10.1), podemos usar o fato que $L \leq_P L_1$ e $L_1 \leq_P L_2$ para concluir que $L \leq P L_2$. Como $L_2 \in NP$, concluímos que L_2 é NP-completo. □

A demonstração de que o problema L_{SAT} é NP-completo é o resultado apresentado no famoso Teorema de Cook-Levin. A prova deste teorema requer certo trabalho. Entretanto, para que possamos mostrar a NP-completude de outros problemas, uma vez que temos o Teorema de Cook-Levin em mãos, não é tão complicado. Observe que o Teorema 10.3.1, diz que nós podemos usar um dado problema NP-completo para provar que um novo problema também é NP-completo. Como o Teorema de Cook-Levin nos diz que o problema L_{SAT} é NP-completo, nós podemos usá-lo para mostrar que outros problemas também são NP-completos. Mostramos um exemplo disto na Seção 10.3.1.

10.3.1 Provando que o problema do conjunto independente é NP-completo

Nesta seção vamos mostrar que $L_{SAT} \leq_P L_{CI}$. Para tal, precisamos mostrar uma redução, ou seja, um algoritmo polinomial, que toma como entrada uma instância $\llcorner\phi\lrcorner$ do problema SAT e retorna como saída a instância $\llcorner(G_\phi, k)\lrcorner$ do problema do conjunto independente satisfazendo o seguinte: se ϕ é satisfazível, então G_ϕ contém um conjunto independente de tamanho pelo menos k . Por outro lado, se ϕ não é satisfazível, então todos os conjuntos independentes de G_ϕ são menores que k .

Para evitar que a notação fique muito sobrecarregada, nós vamos simplesmente escrever ϕ e (G_ϕ, k) para nos referirmos as instâncias dos problemas em questão. Esquematicamente, a redução que procuramos é ilustrada pela Figura 10.1.

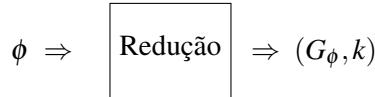


Figure 10.1: A redução toma uma fórmula ϕ e retorna (G_ϕ, k) , onde G_ϕ é um grafo e k um número natural tal que ϕ é satisfazível $\Leftrightarrow G_\phi$ contém um conjunto independente de tamanho k .

Seja $\phi = C_1 \wedge \dots \wedge C_m$ uma instância do problema SAT com n variáveis $x_i, i = 1, \dots, n$. Para cada $j = 1, \dots, m$, denotamos por $A(C_j)$ o conjunto dos literais que aparecem na cláusula C_j . A redução irá tomar como entrada a fórmula ϕ e produzir como saída a instância (G_ϕ, m) . O algoritmo é descrito em detalhes abaixo:

Redução(ϕ)

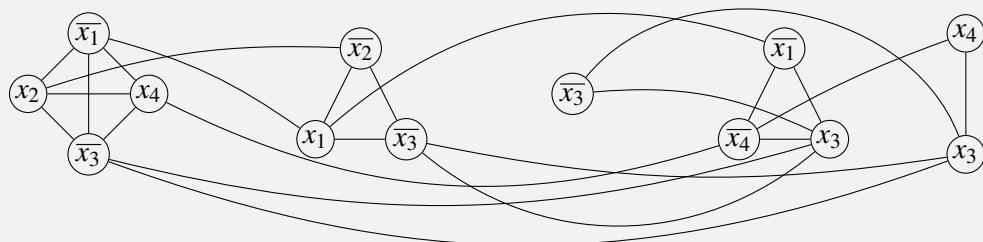
1. Crie um conjunto de vértices V de tamanho $\sum_i^m |A(C_j)|$.
2. Particione o conjunto V em subconjuntos V_1, \dots, V_m , tal que cada subconjunto V_j tem tamanho $|A(C_j)|$ e os rotule² cada vértice de v_j com um elemento diferente do conjunto A_j .
3. Adicione as seguintes arestas: para cada conjunto de vértices V_j , adicione todas as arestas possíveis entre pares de vértices de V_j (ou seja, cada V_j deve induzir uma clique no grafo).
4. Além das arestas presentes das cliques induzidas por V_j , adicione arestas ligando vértices rotulados com literais complementares, ou seja, adicione as arestas uv tal que u tenha rótulo x_i e v tenha rótulo \bar{x}_i , $i = 1, \dots, n$.
5. Retorne (G_ϕ, m)

UM EXEMPLO DA REDUÇÃO

Vamos mostrar um exemplo para tornar essa discussão mais concreta. Suponha que a entrada da redução é a fórmula abaixo:

$$\phi_1 = (\bar{x}_1 \vee x_2 \vee \bar{x}_3 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_3) \wedge (\bar{x}_1 \vee x_3 \vee \bar{x}_4) \wedge (x_3 \vee x_4)$$

Neste caso, a saída da redução é $(G_{\phi_1}, 5)$, sendo que G_{ϕ_1} é o grafo da figura abaixo.



Observe que cada cláusula da fórmula ϕ_1 corresponde a uma clique no grafo G_{ϕ_1} da figura acima. A primeira cláusula da fórmula, isto é, $(\bar{x}_1 \vee x_2 \vee \bar{x}_3 \vee x_4)$, corresponde a clique de

²Devemos ter cuidado para não confundir o vértice de um grafo com o rótulo de um vértice deste um grafo. Em um grafo, cada vértice é um elemento diferente, entretanto, vértices diferentes podem ainda assim ter rótulos iguais.

tamanho 4, mais à esquerda no desenho do grafo. Observe que os literais \bar{x}_1 , x_2 , \bar{x}_3 e x_4 da cláusula são os rótulos desta clique no grafo. Cada uma das demais cláusulas da fórmula corresponde a uma clique no grafo (as cliques tem tamanho 3, 1, 3 e 2, respectivamente). Além das arestas presentes nestas cliques, o grafo contém arestas conectando cada literal x_i ao seu complemento \bar{x}_i . Por exemplo, o literal \bar{x}_1 aparece na primeira cláusula e o literal x_1 na segunda cláusula, portanto há uma aresta ligando os vértices com estes rótulos nas cliques.

Importante: Lembramos que a redução precisa garantir que G_ϕ possui um conjunto independente de tamanho m se e somente se a fórmula $\phi = C_1 \wedge \dots \wedge C_m$ é satisfazível. Veremos agora que, de fato, isso é verdade.

A razão disso é que o grafo G_ϕ expressa as dependências entre as cláusulas de ϕ . No exemplo do quadro acima, note que a primeira cláusula contém o literal \bar{x}_1 e a segunda cláusula contém o literal x_1 . Uma vez que estes dois literais não podem ser satisfeitos ao mesmo tempo, há uma dependência entre estas duas cláusulas. A ideia central é que se existe uma valoração que satisfaz ϕ , essa valoração satisfaz (pelo menos) um literal em cada uma das m cláusulas, e os literais satisfeitos estão relacionados aos rótulos dos vértices do conjunto independente de tamanho m no grafo G_ϕ .

Para provarmos que (G_ϕ, m) é uma instância verdadeira se, e somente se, ϕ é satisfazível, vamos considerar um conjunto independente S de tamanho máximo no grafo G_ϕ . Uma vez que cada um dos vértices de S deve pertencer a uma clique diferente (afinal, quaisquer dois vértices dentro de uma mesma clique são adjacentes), S deve ter m vértices ou menos.

Caso 1: (G_ϕ, m) é uma instância verdadeira.

Neste caso temos que provar que ϕ é satisfazível. Como (G_ϕ, m) é uma instância verdadeira, temos que $|S| = m$. A única possibilidade em que isso ocorre é o caso em que S seja um conjunto $\{v_1, \dots, v_m\}$, com um vértice de cada clique diferente. Uma valoração que satisfaz ϕ pode ser obtida a partir dos rótulos dos vértices $\{v_1, \dots, v_m\}$.

Caso 2: (G_ϕ, m) é uma instância falsa.

Neste caso temos que provar que ϕ não é satisfazível. Suponha por absurdo que ϕ é satisfazível e, portanto, existe pelo menos um literal satisfeito em cada cláusula de ϕ . Seja S' o conjunto de vértices correspondentes aos literais satisfeitos em ϕ . Como (G_ϕ, m) é uma instância falsa, observe que o $|S'| < m$. Entretanto, o conjunto S' é independente (afinal, neste conjunto não pode haver nenhum par de vértices com literais x_i e \bar{x}_i) e tem tamanho m , o que contradiz o fato que S é um conjunto independente máximo.

10.4 Exercícios

Nos exercícios a seguir, as seguintes definições serão necessárias:

Definição 10.4.1 — Isomorfismo de Grafos. Dados dois grafos G_1 e G_2 , decidir se o grafo G_1 é isomorfo ao grafo G_2 .

Definição 10.4.2 — Problema do caixeiro viajante. Dado um grafo completo G com peso nas arestas, encontrar o caminho hamiltoniano de G com o menor custo.

Definição 10.4.3 — Problema da coloração mínima. Dado um grafo G determinar encontrar uma coloração de G com o menor número possível de cores.

Definição 10.4.4 — Problema da k-coloração. Dado um grafo (G, k) decidir se existe uma coloração de G com k ou menos cores.

Definição 10.4.5 — Problema da 3-Coração. Dado um grafo G decidir se existe uma coração de G com 3 ou menos cores.

Definição 10.4.6 — Problema da cobertura mínima por vértices. Dado um grafo G determinar encontrar a cobertura por vértices de G com o menor número possível de vértices.

Definição 10.4.7 — Problema do fluxo máximo. Dado (G, s, t) , tal que G é um grafo com pesos nas arestas e $s, t \in v(G)$, determinar o fluxo máximo de G partindo de s e chegando em t .

Definição 10.4.8 — Problema do emparelhamento bipartido. Dado um grafo bipartido G decidir se existe um emparelhamento perfeito de G .

Definição 10.4.9 — Problema 3-SAT. Dada uma fórmula ϕ em CNF tal que cada cláusula tenha no máximo 3 literais, decidir se ϕ é satisfazível.

Exercício 10.3 Para cada um dos problemas definidos anteriormente, faça o seguinte:

- Se o problema for de decisão, apresente uma definição formal para o problema usando uma linguagem.
- Se o problema for de otimização, apresente uma versão de decisão para o problema e em seguida apresente uma definição formal usando uma linguagem.
- Prove que a versão de decisão de cada um destes problemas está em **NP**.

Exercício 10.4 Seja L_{EB} o problema do Emparelhamento Bipartido e L_{FLUXO} a versão de decisão do problema de fluxo em grafos. Mostre que $L_{EB} \leq_P L_{FLUXO}$ e conclua que $L_{EB} \in P$ usando a versão de decisão do algoritmo de Ford-Fulkerson.

Exercício 10.5 Prove que o problema 3-SAT é NP-completo.

Exercício 10.6 Prove que o problema da 3-coloração é NP-completo.

Exercício 10.7 Prove que o problema da k-coloração é NP-completo.

Exercício 10.8 Prove que se existe um algoritmo polinomial para encontrar uma coração mínima para um grafo, então $P = NP$.

Exercício 10.9 Prove que o problema da clique (Definição 10.2.1) é NP-completo.

Exercício 10.10 Seja L_{CV} a versão de decisão do problema de cobertura por vértices que você obteve no Exercício 10.3. Mostre que $L_{CI} \leq_P L_{CV}$.

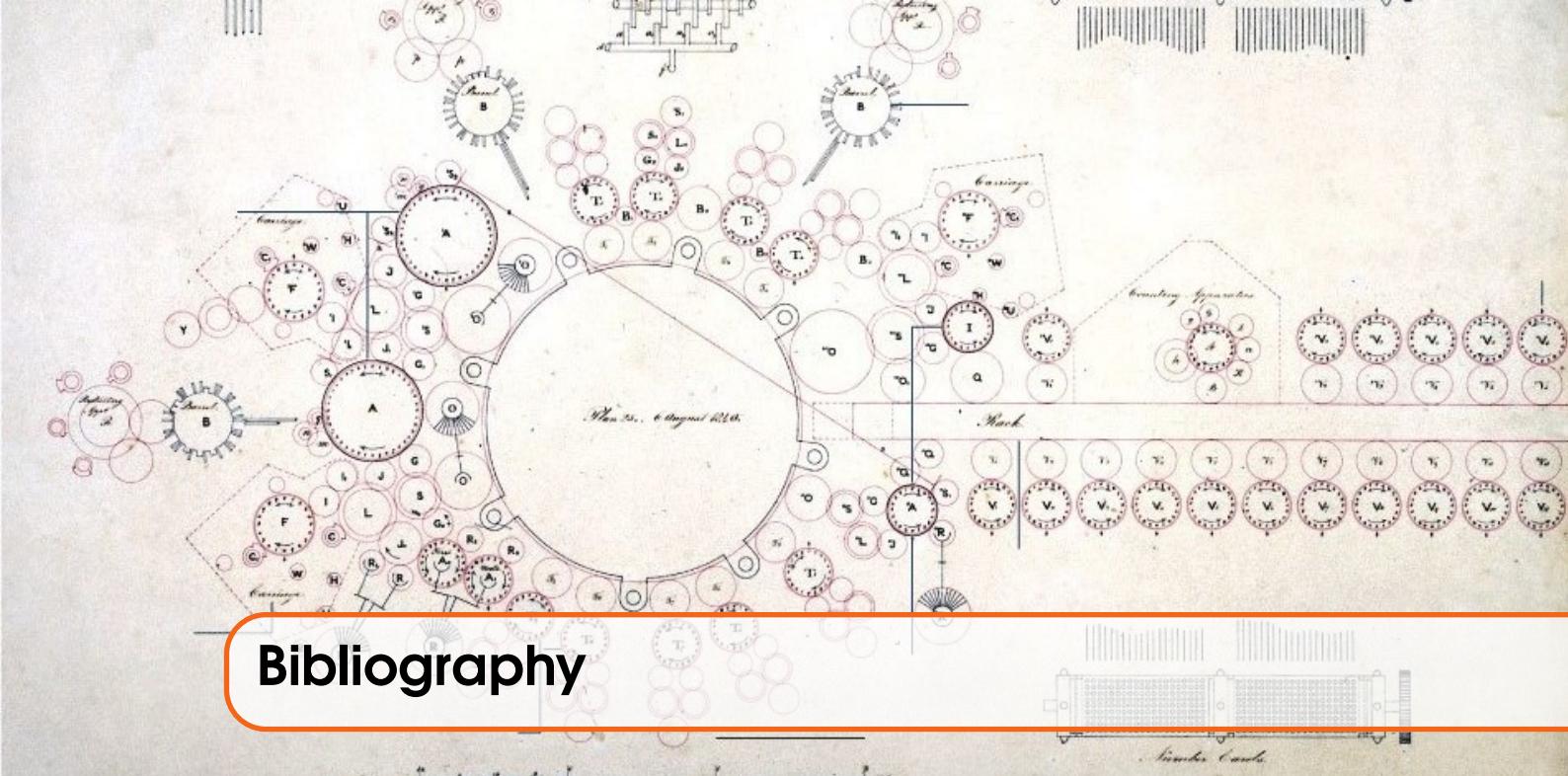
Exercício 10.11 Considere o problema \mathcal{NP} -completo, conhecido em inglês como “Subset Sum Problem”, definido abaixo:

- Uma instância do problema é par (S, k) onde S é um conjunto de números inteiros e k é

um número inteiro. Uma instância de (S, k) é verdadeira se e somente se S contém um subconjunto tal que a soma dos números neste subconjunto é k .

- (a) O que seria um certificado para uma instância verdadeira de Π ?
- (b) Apresente um algoritmo polinomial verificador para Π .

■



Bibliography

Livros

- [Aar13] Scott Aaronson. *Quantum Computing Since Democritus*. 1st edition. Cambridge, 2013 (cited on page 80).
- [Fes19] Edward Feser. *Filosofia da Mente - Um guia para iniciantes*. 1st edition. Edições Santo Tomás, Nov. 2019 (cited on page 80).
- [HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to the Theory of Computation*. 2nd edition. Addison Wesley, 2006 (cited on page 37).
- [Sip06] Michael Sipser. *Introduction to the Theory of Computation*. 2nd edition. Thomson Course Technology, 2006 (cited on page 16).
- [Sud05] Thomas A Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. 3rd edition. Addison Wesley, 2005 (cited on page 16).