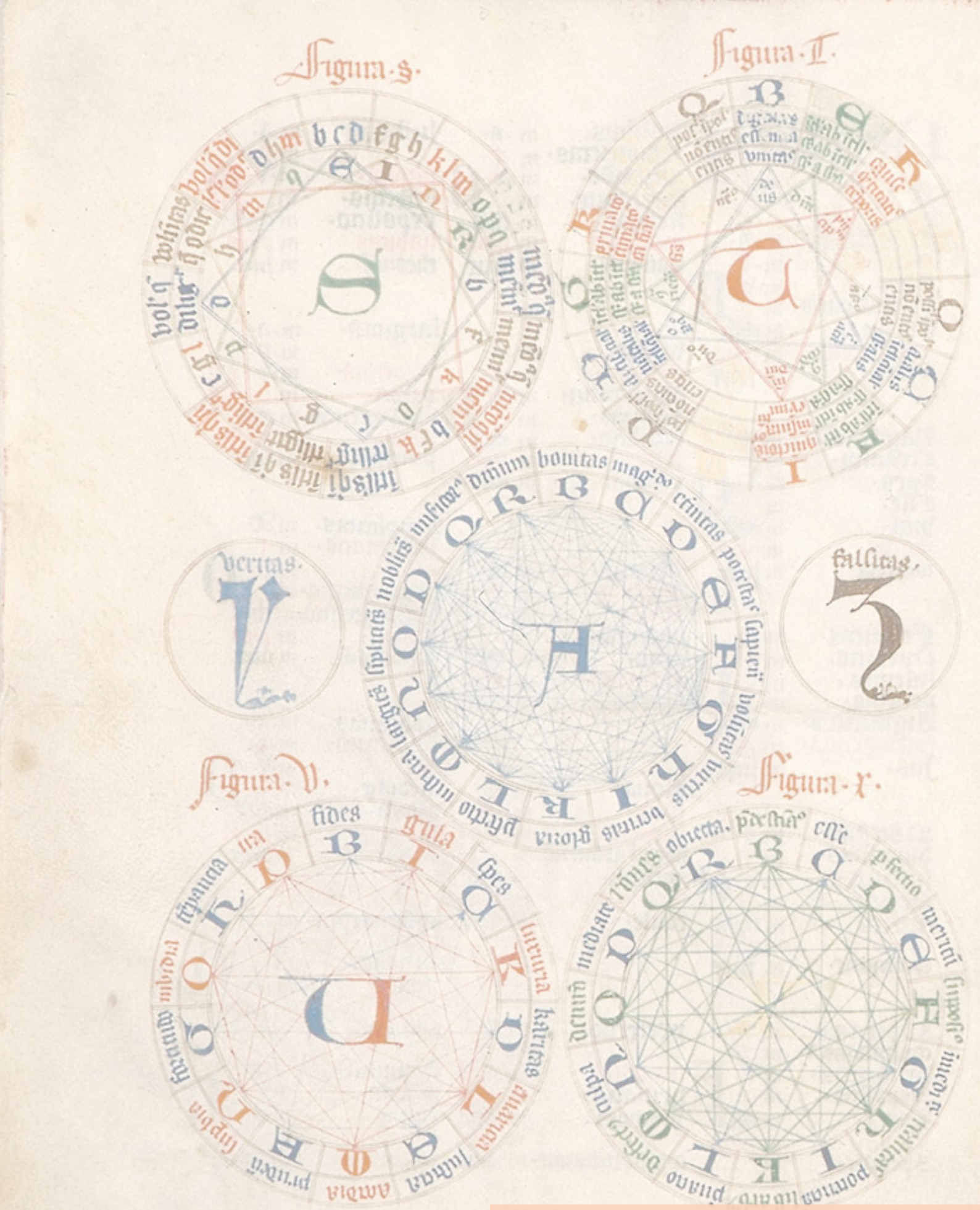
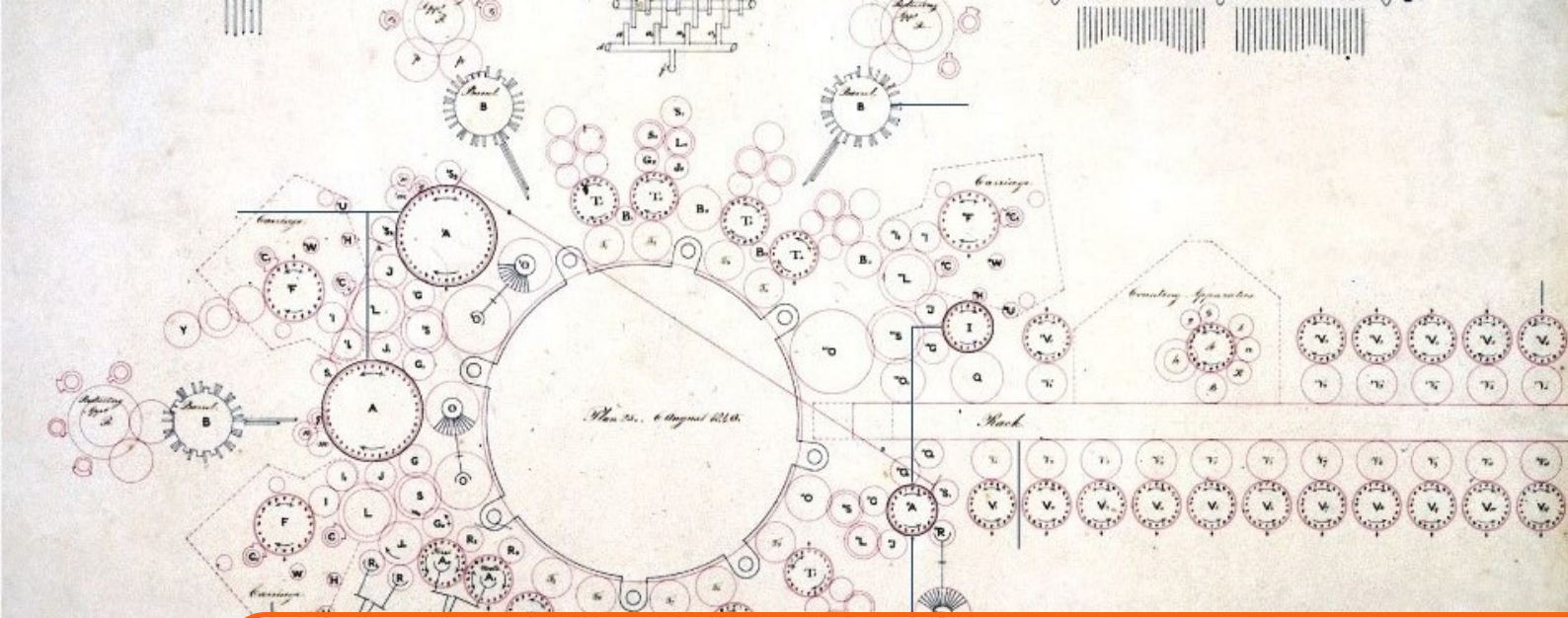


Autômatos, computabilidade e complexidade computacional

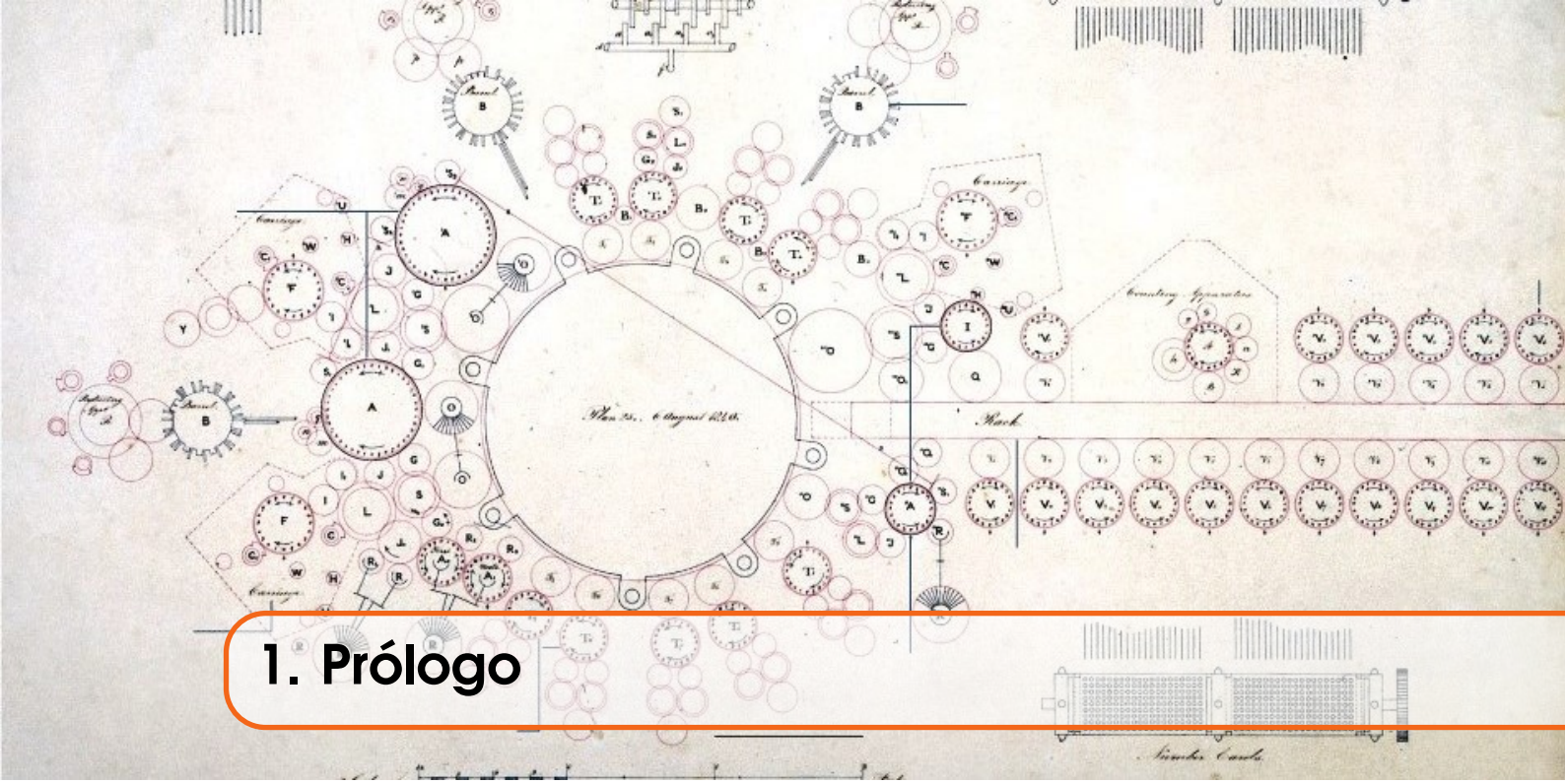
Murilo V. G. da Silva

AUTÔMATOS, COMPUTABILIDADE E COMPLEXIDADE COMPUTACIONAL
© MURILO VICENTE GONÇALVES DA SILVA 2017 – 2025

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

1.1 O que é computação?

A maioria de nós tem alguma ideia, mesmo que superficial, do que são algoritmos e computadores. Esses conceitos são tão corriqueiros que, normalmente, não paramos para pensar muito a fundo sobre eles. Contudo, como veremos neste livro, ao nos propormos a responder algumas perguntas fundamentais sobre computação, precisaremos ir além de uma compreensão intuitiva ou superficial do que são algoritmos e computadores.

Mas que perguntas fundamentais são essas? Podemos começar com a seguinte: será que existe algum problema computacional para o qual não há nenhum algoritmo *eficiente* (dentre os infinitos algoritmos concebíveis) que o resolva? Podemos ir um passo além e fazer uma pergunta ainda mais radical: será que existe algum problema para o qual não existe *nenhum* algoritmo (independentemente de eficiência) que o resolva? A área de estudos conhecida como *teoria da computação* estabelece as bases da ciência da computação ao apresentar definições matematicamente precisas para conceitos como algoritmos e computadores, e ao buscar respostas para perguntas fundamentais como essas.

Um dos primeiros cuidados que precisamos tomar é pensar em algoritmos e computadores sem nos restringirmos aos artefatos tecnológicos do nosso cotidiano, como linguagens de programação, desktops, laptops e smartphones. Esses artefatos são apenas casos particulares de conceitos muito mais gerais. A ideia é começar a enxergar a computação como uma ciência que transcende as tecnologias existentes. Mas seria possível falar de algoritmos e computadores de maneira abstrata, sem recorrer a nenhum artefato tecnológico específico?

Um bom ponto de partida para percebermos que isso é possível é a observação de que muitos algoritmos importantes foram descobertos muito antes do advento dos computadores modernos. O algoritmo de Euclides, por exemplo, é conhecido há cerca de 2300 anos[1]. Mas e quanto aos computadores? No caso dos algoritmos, parece concebível tratá-los de forma abstrata, mas seria também possível falar de computadores de maneira puramente abstrata, sem nos referirmos a

[1] Podemos citar exemplos ainda mais simples: diferentes algoritmos para operações aritméticas, como soma e multiplicação, também já eram conhecidos muito antes da existência dos computadores modernos.

nenhuma máquina específica? Podemos dar uma resposta curta e uma resposta longa. A curta é "sim". A longa é "sim, mas com algumas sutilezas".

A chave para entender por que é possível falar de computadores de modo puramente abstrato está em reconhecer que um computador é, essencialmente, uma máquina capaz de executar qualquer algoritmo concebível. Assim, compreender o que é um computador equivale a compreender a ideia abstrata do conjunto, potencialmente infinito, de todos os algoritmos possíveis. A sutileza está no fato de que, na medida em que algoritmos e computadores possam ser implementados fisicamente, há restrições sobre o que pode ser considerado um algoritmo possível.

> O CONCEITO DE INFORMAÇÃO EM DIFERENTES ÁREAS DA CIÊNCIA
>
> Com a popularização dos computadores na segunda metade do século XX, outro conceito intimamente relacionado à computação também se popularizou: o conceito de informação. Essa popularização frequentemente nos leva a associar computação e informação a artefatos tecnológicos específicos, algo que, como já mencionamos, queremos evitar neste livro.
>
> Para ilustrar o que queremos dizer, observe que, em situações nas quais descrevemos processos computacionais no mundo natural, como moléculas de DNA *armazenando informação* ou cérebros *processando informação*, não é incomum pensarmos que estamos apenas usando metáforas inspiradas por tecnologias contemporâneas. Embora isso de fato aconteça[†], não podemos deixar de perceber que informação e computação são conceitos que existem no mundo, seja na matemática ou na natureza, e que podem ser estudados como objetos em si, independentes de qualquer contingência tecnológica.
>
> O conceito de *informação* também já era estudado antes da popularização dos computadores digitais. A noção começou a ganhar espaço no vocabulário científico no final do século XIX, quando físicos se depararam com ideias como entropia e termodinâmica. Já no século XX, uma série de descobertas científicas — como o surgimento da mecânica quântica (teoria na qual a ideia de *informação* quântica é central) e a descoberta da estrutura do DNA (intrinsecamente ligada à *informação* genética) — consolidaram a intuição moderna de que a informação é um conceito fundamental e que permeia o mundo em diferentes níveis de análise.
>
> [†]Pode haver uma contingência histórica que nos leva a criar tais associações. Assim como hoje usamos informação como metáfora para descrever fenômenos naturais, no século XIX era comum descrever o universo como um grande mecanismo, em analogia às engrenagens de um relógio e às leis da mecânica clássica.

1.2 Algoritmos, problemas e computadores

Além de algoritmos e computadores, há também um outro conceito que normalmente usamos de maneira intuitiva e que iremos tornar preciso neste livro: o conceito de *problema computacional*. Por exemplo, considere o problema de testar se um dado número é primo ou o problema de verificar se existe um caminho ligando dois vértices em um grafo. Em teoria da computação, definiremos exatamente o que é, de *maneira genérica*, um problema computacional.

Uma vez que temos em mãos definições matematicamente precisas para algoritmos e para problemas computacionais, a pergunta que fizemos na seção anterior surge naturalmente: será que existe algum problema para o qual não existe nenhum algoritmo que o resolva? É importante enfatizar que não estamos falando de problemas para os quais, hoje, ainda não conhecemos soluções algorítmicas, mas que, eventualmente, poderão ser descobertas no futuro. A pergunta aqui é mais profunda: existem problemas que, por princípio, *não podem ser resolvidos por nenhum algoritmo*? No Capítulo 6 veremos que a resposta é *sim*. O ponto central é que não vamos apenas afirmar isso, mas prová-lo, obtendo *certeza matemática* dessa limitação fundamental.

REFLETINDO UM POUCO: LIMITE TECNOLÓGICO OU PRINCÍPIO FUNDAMENTAL?

A existência de problemas insolúveis refletiria apenas uma limitação do que atualmente entendemos por computadores? Ou seja: será que tais problemas seriam insolúveis por computadores apenas *na nossa concepção corrente* de computação? Não poderiam, talvez, ser resolvidos no futuro por computadores "exóticos", baseados em leis da física ainda desconhecidas? Ou a insolubilidade de certos problemas é um princípio mais profundo e universal?

O posição mais comum entre os teóricos da computação, expresso pela *Tese de Church-Turing* [Aar13; HMU06; Sip06], é que qualquer problema computacional solúvel pode ser resolvido, *em princípio*, por computadores como os que concebemos hoje. Esta tese, como qualquer proposição científica, está aberta a debate. Entretanto, a maioria das propostas de modelos alternativos de computação que tentam refutá-la, embora matematicamente bem definidas, não correspondem ao que intuitivamente entendemos por processos computacionais sistemáticos, além de parecerem fisicamente inviáveis [Aar13][†]. O Capítulo 7 deste livro é dedicado à discussão dessa tese.

Apesar disso, permanece aberta a possibilidade de construirmos computadores *fundamentalmente mais eficientes* que os atuais. A pesquisa em computação quântica, por exemplo, investiga precisamente a viabilidade de máquinas *exponencialmente* mais rápidas na resolução de *alguns* tipos de problemas.

[†]Um exemplo comum envolve modelos que dependem de "oráculos mágicos", capazes de executar passos computacionais sem explicação. Outro exemplo são modelos que pressupõem computação *verdadeiramente analógica*, algo que contradiz alguns princípios aceitos da física contemporânea.

Embora a teoria da computação seja um campo matemático, e não físico, é importante notar que os objetos abstratos que estudamos, como algoritmos, informação e computadores, se manifestam, de alguma forma, no mundo físico[2]. O seu laptop rodando Windows é talvez o exemplo mais óbvio de um processo computacional ocorrendo em um meio físico. Mas ele não é o único. O cálculo de logaritmos com uma máquina mecânica de engrenagens no século XIX, o cérebro de um primata processando sinais elétricos sensoriais, moléculas de DNA executando rotinas biológicas ou partículas subatômicas manipulando informação quântica em um computador quântico são todos exemplos de computação em diferentes substratos físicos. (Figura 1.1).

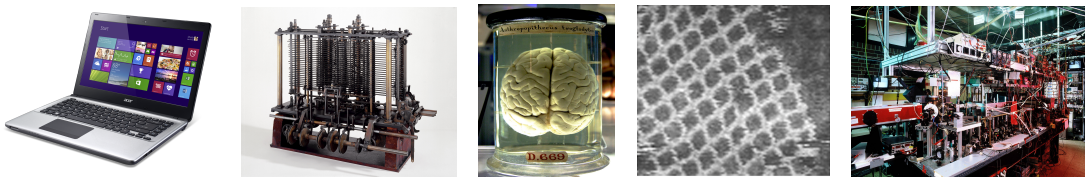


Figura 1.1: Computação ocorrendo em diferentes meios físicos: circuitos eletrônicos, engrenagens mecânicas, estruturas biológicas, moléculas de DNA e partículas subatômicas.

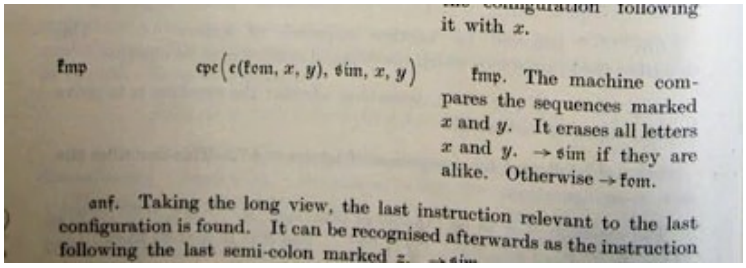
O objetivo da teoria da computação é prover modelos matemáticos para processos computacionais que sejam *independentes do substrato físico* em que a computação ocorre. Queremos um modelo que, ao mesmo tempo, seja abstrato, abrangente e realista o suficiente para capturar *qualquer computação possível*, em qualquer meio físico. Se tivermos tal modelo, bastará estudá-lo para compreender o que pode e o que não pode ser *efetivamente computado*.

[2]O ponto de contato entre a natureza matemática da teoria da computação e as questões empíricas é sutil. Sem entrar em discussões filosóficas mais profundas, queremos apenas destacar que há uma correspondência entre os modelos matemáticos da computação e os substratos físicos capazes de realizá-los.

Esse é um objetivo ambicioso: buscamos um modelo que descreva *toda e qualquer manipulação sistemática de dados*. Surpreendentemente, um modelo com essas propriedades foi concebido na década de 1930: a *Máquina de Turing*[3]. A tese científica que sustenta que Máquinas de Turing possuem tal nível de generalidade é chamada *Tese de Church-Turing*[4].

> COMPUTAÇÃO: INSTANCIANDO OBJETOS ABSTRATOS EM OBJETOS FÍSICOS
>
> Em um computador objetos abstratos, como informação e algoritmos, são instanciados em objetos físicos. Um aspecto fundamental a compreender é que o conjunto de relações matemáticas entre esses objetos abstratos corresponde aos graus de liberdade de movimento desses sistemas físicos.

Os computadores que usamos no dia a dia são exemplos concretos desses modelos. Sabemos rigorosamente como funcionam, e, portanto, temos descrições matemáticas exatas deles. Veremos que esses modelos são equivalentes a certas Máquinas de Turing conhecidas como *Máquinas de Turing Universais*. A característica essencial dessas máquinas é que podem *simular*[5] qualquer outra Máquina de Turing e, consequentemente, qualquer processo computacional que possa ocorrer em algum substrato físico. Isso significa que, pelo menos em princípio, não existem processos de computação fisicamente realizáveis que resolvam problemas impossíveis para nossos computadores atuais, desde que estes disponham de tempo e memória suficientes. A definição matemática da Máquina de Turing Universal é, portanto, o modelo formal do que chamamos de *computador*. O fato de existir uma Máquina de Turing Universal, um dos resultados centrais do artigo de Alan Turing (Figura 1.2 mostra um trecho do artigo), é o que chamamos de *universalidade* da computação.



it with x.

fmp $\quad cpc\big(e(\mathfrak{fom}, x, y), \mathfrak{sim}, x, y\big)$ fmp. The machine compares the sequences marked x and y. It erases all letters x and y. $\rightarrow \mathfrak{sim}$ if they are alike. Otherwise $\rightarrow \mathfrak{fom}$.

anf. Taking the long view, the last instruction relevant to the last configuration is found. It can be recognised afterwards as the instruction following the last semi-colon marked z. $\rightarrow \mathfrak{sim}$

Figura 1.2: Trecho do artigo de 1936 de Alan Turing, *On Computable Numbers, with an Application to the Entscheidungsproblem*, onde foram estabelecidas as bases da ciência da computação.

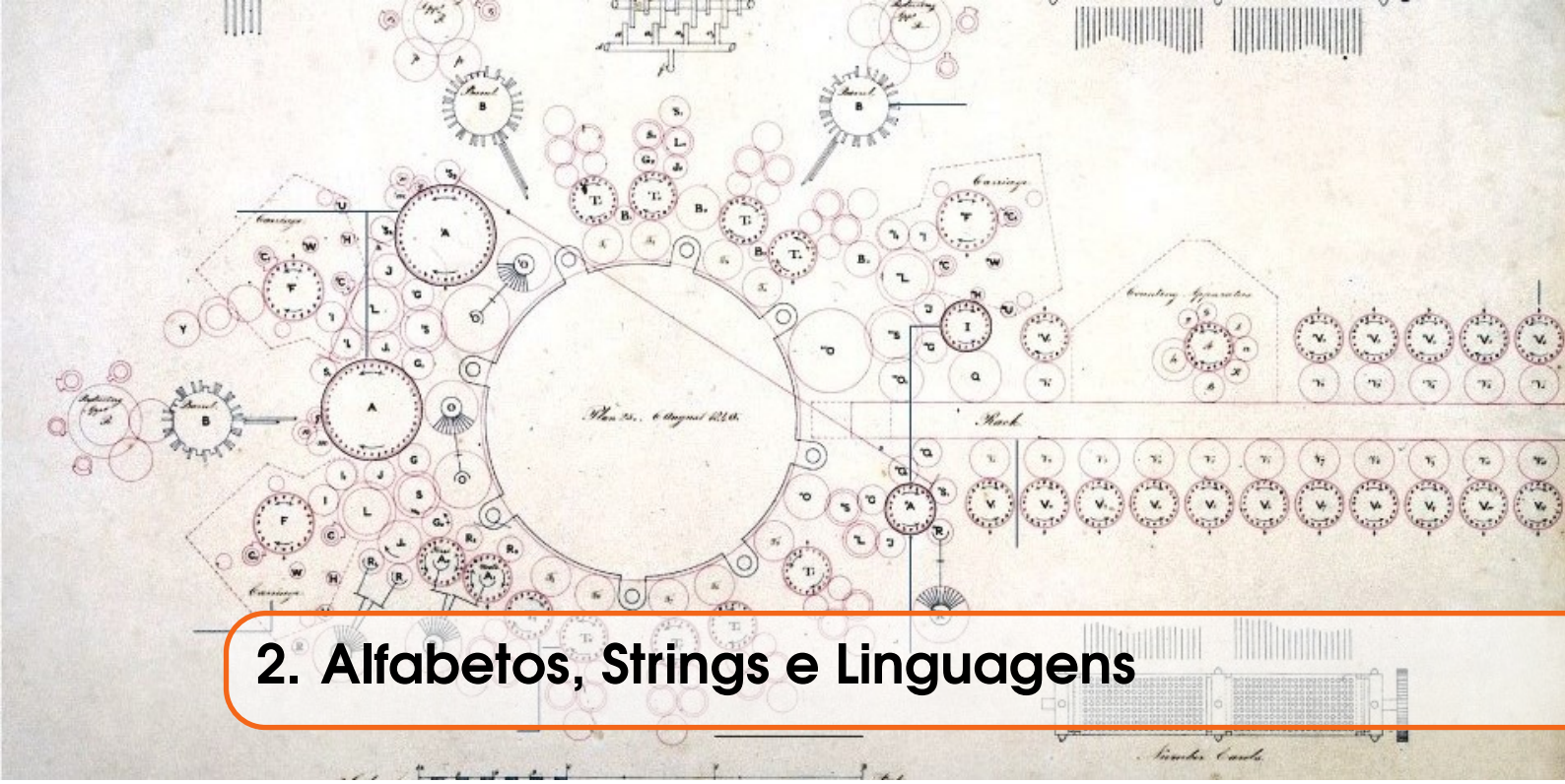
> A TESE DE CHURCH-TURING E A UNIVERSALIDADE DA COMPUTAÇÃO
>
> *"A lição que tomamos da universalidade da computação é que podemos construir um objeto físico, que chamamos de computador, capaz de simular qualquer outro processo físico; e o conjunto de todos os possíveis movimentos desse computador, definido pelo conjunto de todos os programas concebíveis, está em* ***correspondência de um para um*** *com o conjunto de todos os possíveis movimentos de qualquer outro sistema físico concebível."* — David Deutsch

[3] Embora pareça ousado, empreendimentos análogos existem em outras ciências. Na física, por exemplo, certas equações pretendem descrever "todo e qualquer movimento" ou "todo e qualquer estado possível de um sistema". De modo semelhante, a ciência da computação busca modelos matemáticos que descrevam "toda e qualquer manipulação sistemática de informação".

[4] Há, de fato, duas interpretações da Tese de Church-Turing. Uma a trata como uma definição matemática; a outra, como uma afirmação empírica sobre a realidade física. Nesta última versão, a que nos referimos aqui.

[5] Aqui, "simular" tem um significado matematicamente preciso, que expressa uma correspondência de um para um entre a computação ocorrendo nos dois modelos.

Parte 1: Teoria de linguagens e autômatos



2. Alfabetos, Strings e Linguagens

2.1 Alfabetos e strings

Algoritmos, computadores e problemas computacionais são objetos matemáticos complexos. Nosso 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 e não vazio 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 de conjuntos. 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 conjunto de símbolos Σ_C de caracteres da tabela ASCII[1], que contém o conjunto de todos caracteres válidos em um programa escrito na linguagem C no padrão ANSI C.

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.

[1] O conjunto de símbolos que contém caracteres básicos como $0, ..., 9, a, ..., z, A, ..., Z, +, -, *, \&, \#, \%, >, <, ...$, etc.

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

■ **Exemplo 2.3** A sequência *aaba* é uma string sobre o alfabeto $X = \{a, b, c, d\}$ do Exemplo 2.1. ■

■ **Exemplo 2.4** A sequência 00101 é uma string sobre o alfabeto binário. ■

■ **Exemplo 2.5** Um programa escrito em linguagem C pode ser visto matematicamente como uma string sobre o alfabeto Σ_C mencionado anteriormente. ■

UM PROGRAMA PODE SER VISTO COMO UMA LONGA STRING

Observe que um programa em Linguagem C é uma sequência de símbolos ANSI. Esta sequência de símbolos inclui símbolos especiais que normalmente não são mostrados na tela do editor de texto (i.e., caracteres que indicam quebra de linha, indentação, espaçamento, etc que instruem o editor de texto a como dispor o código na tela do computador).

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 símbolos, 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 uma string 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 2.6** As strings *ab*, *bb* e *bbc* são algumas das substrings da string *abbbbc*. ■

■ **Exemplo 2.7** 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 concatenção não é uma operação comutativa. A partir do Exemplo 2.7, 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 os livros [Sud05] e [LP98], 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 linguangens formais*.

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

Notação 2.2. *Seja Σ um alfabeto e s um símbolo de Σ. Embora símbolos e strings sejam objetos matemáticos diferentes, nos referiremos a s tanto como um símbolo de Σ quanto a uma string de tamanho 1 sobre Σ.*

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

Observe que, de acordo com nossa definição, $|\varepsilon| = 0$. 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 $w^{\mathrm{R}} = 0111$). Segue a definição indutiva[2]:

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 $w = \varepsilon$, então $w^{\mathrm{R}} = \varepsilon$.
Caso Geral: Seja w uma string com pelo menos um símbolo. Se a string w tem a forma $w = xs$, tal que x é uma string e s é uma string de tamanho 1, então $w^{\mathrm{R}} = sx^{\mathrm{R}}$.

Notação 2.3. *Seja Σ um alfabeto e $s \in \Sigma$. Diremos que uma string sobre este alfabeto é da forma s^n se a string é formada por n símbolos s 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 2.9** A string 11111 pode ser escrita como 1^5. ■

■ **Exemplo 2.10** Considere a string $w = 10$. Com isso $w^6 = 101010101010$. ■

Note que, em particular, $x^0 = \varepsilon$ para qualquer string x.

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 $\Sigma^0 = \{\varepsilon\}$.

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 $i \in \mathbb{N}$, note que Σ^i é um conjunto finito. A seguir vamos definir o conjunto infinito de todas as strings sobre Σ, ou seja, $\Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$. 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

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

[2]Uma 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.

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 (Resolvido) Dado um alfabeto Σ, a Definição 2.1.8 estabelece o conjunto Σ^* de todas as strings possíveis com símbolos de Σ. Apresente uma definição alternativa para Σ^* (ou seja, uma definição diferente da Definição 2.1.8, mas que resulte no mesmo conjunto de strings) que seja recursiva.

Solução:

Dado um alfabeto Σ, conjunto Σ^* é definido da seguinte forma:
Base: $\varepsilon \in \Sigma^*$
Indução: Se $w \in \Sigma^*$, então $\forall a \in \Sigma$ temos que $wa \in \Sigma^*$. ■

Exercício 2.4 Nesta seção, fornecemos uma definição informal para a concatenação de uma string x com ela mesmo n vezes, denotada x^n. Forneça uma definição formal indutiva para x^n. ■

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

Exercício 2.6 (Opcional) Seja w uma string sobre um alfabeto qualquer e k um inteiro não negativo. Mostre que $(w^i)^{\mathrm{R}} = (w^{\mathrm{R}})^i$. ■

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, $L = \{aba, acccc, b\}$ é uma linguagem sobre o alfabeto $\Sigma = \{a,b,c\}$.

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*.

Nos Exemplos 2.11, 2.12, 2.13 e 2.14 apresentamos algumas linguagens binárias que serão bastante comuns neste curso:

■ **Exemplo 2.11** $L_{01} = \{\varepsilon, 01, 0011, 000111, ...\} = \{w \in \{0,1\}^* : w \text{ é da forma } 0^n1^n, n \geq 0\}$. ■

■ **Exemplo 2.12** $L_{\mathrm{PAL}} = \{\varepsilon, 0, 1, 00, 11, 010, 101, 000, ...\} = \{w \in \{0,1\}^* : w \text{ é um palíndromo}\}$. ■

■ **Exemplo 2.13** $L_{\text{P}} = \{10, 11, 101, 111, 1011, ...\} = \{w \in \{0,1\}^* : w \text{ é um número primo escrito em binário}\}$. ■

■ **Exemplo 2.14** $L_{\text{SQ}} = \{0, 1, 100, 1001, 10000, ...\} = \{w \in \{0,1\}^* : w \text{ é um número quadrado perfeito escrito em binário}\}$. ■

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 2.15** Seja $\Sigma_{\text{P}} = \{a, b, c, ..., z\}$ 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 sinais gráficos como acentos, hífens, pontuação, etc). Poderíamos chamar este conjunto de strings de "linguagem portuguesa". Por exemplo, observe que $bola \in L$, $caneta \in L$, $inconstitucional \in L$.

■ **Exemplo 2.16** Seja $\Sigma_{\text{P}} = \{a, b, c, ..., z, \sqcup\}$. ■

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 a língua portuguesa, sendo que os exemplos vistos acima são úteis apenas para ilustrar a relação que há entre o conceito matemático de linguagem e o conceito intuitivo de linguagem natural. 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:

$L_{\text{C}} = \{w \in \Sigma_{\text{C}}^* : \text{“}w \text{ é um programa válido em linguagem C”}\}$

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*. Mais adiante neste livro 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 interpretamos 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 2.4. *O número natural representado pela string binária w é denotado por $N(w)$.*

Por exemplo, se $w_1 = 101$ e $w_2 = 1111$, então $N(w_1) = 5$ e $N(w_2) = 15$. Note que N não é uma bijeção, pois várias strings podem estar associadas à um mesmo número. Por exemplo, $N(01) = N(1) = 1$.

Convenção: $N(\varepsilon) = 0$.

Uma das vantagens da Notação 2.4, é tornar a nossa escrita mais concisa na definição de algumas linguagens. Por exemplo, a linguagem dos números primos e a linguagens do números múltiplos de 3 podem ser denotadas, respectivamente, por $L_{\text{P}} = \{w : N(w) \text{ é um número primo}\}$ e $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$.

2.2.1 Linguagens e problemas computacionais

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 2.13. 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}.

> TESTANDO SE UM OBJETO MATEMÁTICO POSSUI CERTA PROPRIEDADE
>
> 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 usanddo uma linguagem.

Exercício 2.7 Seja Σ um alfabeto arbitrário. É verdade que os conjuntos Σ^*, $\varnothing$, $\{\varepsilon\}$ e Σ^4 são todos linguagens sobre Σ? Justifique. ■

Exercício 2.8 Seja $\Sigma = \{0,1\}$. Forneça uma definição formal para a linguagem sobre Σ 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

Nesta seção, apresentaremos três operações fundamentais sobre linguagens, denominadas *união*, *concatenação* e *fecho*.

Definição 2.2.2 — União de Linguagens. Dadas duas linguagens L e M, a *união de L com M*, denotada $L+M$ é a linguagem obtida pela união de L com M.

Definição 2.2.3 — Concatenação de Linguagens. 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 2.17** Considere as linguagens $L = \{00, 0\}$ e $L' = \{1, 111\}$. A concatenação destas duas linguagens é $LL' = \{001, 00111, 01, 0111\}$. ■

■ **Exemplo 2.18** Seja $L = \{a, aa, aaa, ...\}$ e $M = \{\varepsilon, x, xx, xxx, ...\}$. Para computarmos quais são as strings de LM, começamos com a primeira[3] string de L e a concatenamos com cada uma das strings de M, obtendo a, ax, axx, $axxx$,..., etc. Adicionalmente, tomamos a segunda string de L e também concatenamos com cada uma das strings de M, obtendo aa, aax, $aaxx$, $aaxxx$,..., etc. Seguimos fazendo isso com cada uma das (infinitas) strings de L. De maneira mais precisa, temos que $LM = \{a^i x^k \ : \ i \geq 1 \text{ e } k \geq 0\}$ ■

Exercício 2.10 (Resolvido) Seja $L = \{a, aa, aaa\}$. Qual a linguagem LLL?

Solução: Primeiramente, observe que $LL = \{aa, aaa, aaaa, aaaaa, aaaaaa\}$. Para obter a linguagem LLL precisamos concatenar L com LL. Portanto, $LLL = \{aaa, aaaa, aaaaa, aaaaaa, aaaaaa, aaaaaaaa, aaaaaaaaa\}$. ■

Exercício 2.11 (Resolvido) Seja $\Sigma = \{0,1\}$ e sejam L_3 e R as linguagens sobre Σ definidas da seguinte forma: $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$ e $R = \{0\}$. Qual é a linguagem L_3R?

Solução: As strings contidas na linguagem da concatenção 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: $L_3R = \{w : N(w) \text{ é um múltiplo de } 6\}$. ■

[3] A rigor não existe ordem nos elementos de um conjunto, mas para simplificar a explicação, nos referimos aqui a a como a primeira string de L, aa como segunda string de L, e assim por diante.

Notação 2.5. *Usamos a notação L^i, para $i \geq 2$, para denotar a concatenação de uma linguagem L consigo mesmo $i-1$ vezes, isto é, $L^i = \underbrace{LL...L}_{i \text{ vezes}}$. Para o caso $i = 0$ e $i = 1$ a notação se refere, respectivamente, aos conjuntos $L^0 = \{\varepsilon\}$ e $L^1 = L$.*

Exercício 2.12 (Resolvido) Seja $\Sigma = \{0,1\}$. Considere a linguagem $R = \{0\}$ sobre Σ. Qual é a linguagem $\Sigma^* R$? E qual é a linguagem $\Sigma^* R^2$?

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

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

■

A operação de concatenação de uma linguagem com ela mesma pode ser feita infinitamente, como definido a seguir:

Definição 2.2.4 — Fecho de uma Linguagem. O *Fecho de uma linguagem 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, $L^* = \bigcup_{i \geq 0} L^i$.

Terminologia 2.2. *O Fecho de uma linguagem L também é conhecido como Fechamento de L ou Fecho de Kleene de L.*

■ **Exemplo 2.19** Seja $L = \{000, 111\}$. Então $L^* = \{\varepsilon, 000, 111, 000000, 000111, 111000, 111111, 000000000, 000000111, 000111000, ...\}$. ■

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, note que 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.

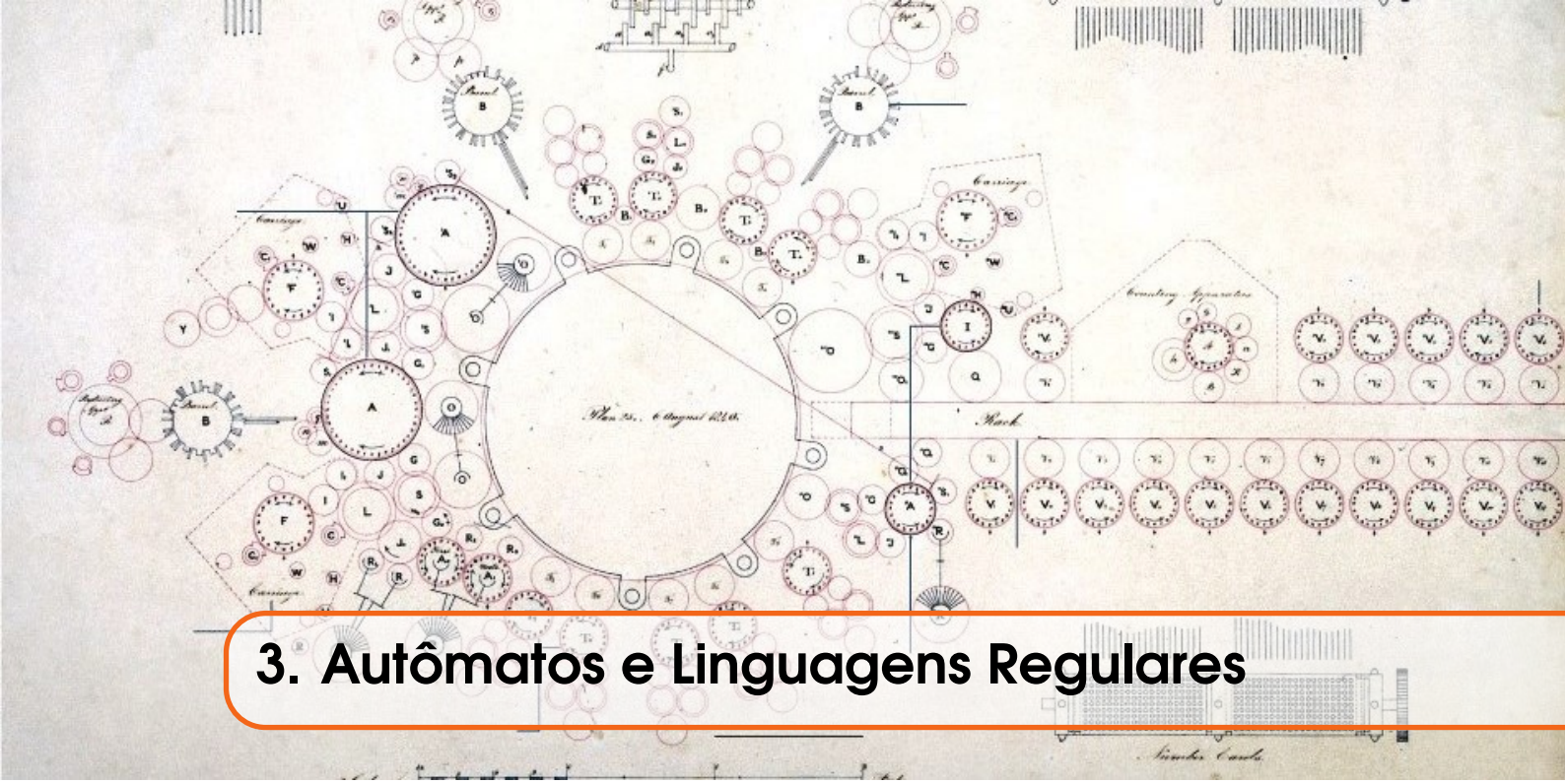
Exercício 2.13 (Resolvido) Sejam $L = \{1\}$ e $R = \{0\}$ linguagens. Qual é a linguagem LR^*?

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

Exercício 2.14 Nas questões abaixo, A é sempre um alfabeto e L é sempre uma linguagem.

- 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 $A^* = L^*$.
- Seja L uma linguagem não vazia. Qual é a linguagem resultante da concatenação de L com o conjunto vazio?
- Para todo L, $\varepsilon \in L^*$? (note que L pode ser até mesmo o conjunto vazio)

■



3. Autômatos e Linguagens Regulares

Um dos objetivos deste 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 neste Capítulo 3 é a definição de *autômatos finitos*, um modelo matemático com poder de expressão menor do que o das Máquinas de Turing. Por menor poder de expressão, queremos dizer que apenas um subconjunto de todos os possíveis algoritmos pode ser expresso na forma de um autômato finito. O estudo desse modelo será fundamental para nos familiarizarmos com os conceitos matemáticos que serão utilizados na compreensão das Máquinas de Turing.

3.1 Autômatos Finitos Determinísticos (AFDs)

Considere uma máquina que vende chocolates que custam 6 reais. A Figura 3.1 ilustra tal máquina juntamente com um diagrama que descreve seu mecanismo de funcionamento interno.

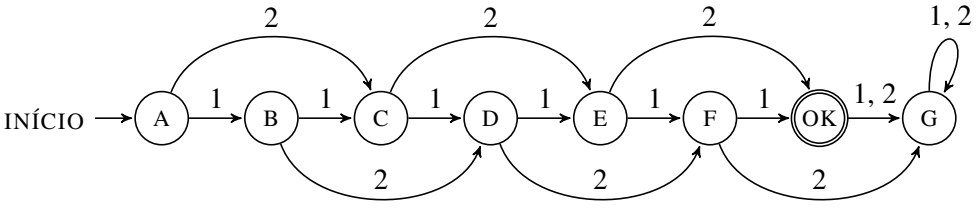


Figura 3.1: Uma máquina que vende chocolates de R$ 6 que podem ser comprados com moedas de R$ 1 e R$ 2. Ao lado um diagrama simplificado descrevendo o seu mecanismo de funcionamento interno. Por exemplo, se inserirmos duas moedas de R$ 2 e duas moedas de R$ 1, a máquina, atingirá o estado OK, indicando que a máquina "aceitou a entrada". Observe as sequências de moedas que fazem a máquina atingir o estado OK são exatemente aquelas cuja soma dos valores das moedas seja R$ 6.

Diagramas como o do lado direito da Figura 3.1 são conhecidos como *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 cujos símbolos são moedas), que podem ser de 1 ou 2 reais. A máquina libera o chocolate se as moedas somarem exatamente 6 reais. Há uma indicação de INÍCIO no estado A, pois este é o estado que a máquina se encontra antes de todo processo começar. 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, ao final da sequência de moedas o estado da máquina OK. Caso a máquina termine a computação em qualquer outro estado, entendemos que ela simplesmente devolve todo dinheiro sem liberar nenhum chocolate. Embora este seja um exemplo de máquina simples demais para ser útil na prática, ele servirá para entendermos como funcionam autômatos.

REFLETINDO UM POUCO: MODELOS MATEMÁTICOS E OBJETOS FÍSICOS

No Capítulo 1 dissemos que podemos pensar em computadores como "instanciações de objetos abstratos e seus possíveis relacionamentos matemáticos em objetos físicos e seus conjuntos de possíveis graus de liberdade de movimento". Note que a Figura 3.1 apresenta exatamente isso, pois:

(1) A máquina de chocolates é um objeto físico. Podemos imaginá-la como tendo engrenagens internas, de forma que, a cada instante, a máquina se encontra em um determinado estado. Observe que, embora essa máquina não seja um computador de propósito geral, ela é a instanciação física de um algoritmo.

(2) O diagrama é o modelo matemático. Em particular, trata-se de um objeto matemático definido por um conjunto de estados, um conjunto de símbolos que aparecem nos rótulos das transições (o conjunto 1, 2) e, além desses conjuntos, por certas relações matemáticas que associam estados e símbolos, sendo essas relações indicadas pelas setas do diagrama. A definição exata dessas relações ficará clara no decorrer desta seção.

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

3.1.1 Modelando matematicamente autômatos

Nesta seção veremos uma definição precisa para diagramas como os vistos na seção anterior. Na discussão anterior, utilizamos esses diagramas de maneira intuitiva. Entretanto, para tratá-los formalmente como objetos matemáticos, é necessário apresentar uma definição rigorosa.

Definição 3.1.1 — Autômato Finito Determinístico (AFD). Uma Autômato Finito Determinístico 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* $\delta : Q \times \Sigma \to Q$;

$q_0 \in Q$ é o *estado inicial*;

$F \subseteq Q$ é o conjunto de *estados de aceitação*.

Terminologia 3.1. *Na maior parte dos casos, diremos simplesmente autômatos ou AFDs para nos referirmos aos autômatos finitos determinísticos. Observamos também que em alguns livros, utiliza-se o acrônimo DFA, advindo da expressão em inglês "deterministic finite automaton". O conjunto de estados de aceitação de um AFD também é chamado de conjunto de estados finais.*

Algo importante a observar é que a Definição 3.1.1 se refere a um autômato genérico, ou seja, os conjuntos e a relação que compõem a 5-tupla não estão especificados. Quando formos modelar matematicamente um algoritmo em particular, como o algoritmo da máquina de chocolates da Figura 3.1, precisaremos apresentar uma *instância específica* dessa definição. O Exemplo 3.1, apresentado a seguir, ilustra isso.

■ **Exemplo 3.1 — Modelo matemático da máquina de vender chocolates.** Uma definição matemática para o algoritmo implementado pela máquina da Figura 3.1 é o autômato finito determinístico $C = (Q_c, \Sigma_c, \delta_c, \mathrm{A}, F_c)$, sendo que os componentes da 5-tupla são:

$Q_c = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{OK}, \mathrm{G}\}$;

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

$\delta_c : Q_c \times \Sigma_c \to Q_c$ é a função definida caso a caso a seguir:

$\delta_c(\mathrm{A}, 1) = \mathrm{B}$, $\delta_c(\mathrm{A}, 2) = \mathrm{C}$, $\delta_c(\mathrm{B}, 1) = \mathrm{C}$, $\delta_c(\mathrm{B}, 2) = \mathrm{D}$, $\delta_c(\mathrm{C}, 1) = \mathrm{D}$, $\delta_c(\mathrm{C}, 2) = \mathrm{E}$,
$\delta_c(\mathrm{D}, 1) = \mathrm{E}$, $\delta_c(\mathrm{D}, 2) = \mathrm{F}$, $\delta_c(\mathrm{E}, 1) = \mathrm{F}$, $\delta_c(\mathrm{E}, 2) = \mathrm{OK}$, $\delta_c(\mathrm{F}, 1) = \mathrm{OK}$, $\delta_c(\mathrm{F}, 2) = \mathrm{G}$,
$\delta_c(\mathrm{OK}, 1) = \mathrm{G}$, $\delta_c(\mathrm{OK}, 2) = \mathrm{G}$, $\delta_c(\mathrm{G}, 1) = \mathrm{G}$, $\delta_c(\mathrm{G}, 2) = \mathrm{G}$.

O estado A é o estado inicial;

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

Definições genéricas e definições específicas

Uma habilidade que é muito importante para estudantes de teoria da computação é a distinção entre um conceito genérico e uma instância particular de tal conceito. Na Definição 3.1.1 temos a definição para um AFD em geral e no no Exemplo 3.1, a 5-tupla $C = (Q_c, \Sigma_c, \delta_c, \mathrm{A}, F_c)$ é a definição do AFD que *especificamente* correspondente a máquina de vender chocolates.

Vamos usar a seguinte analogia: pense nas funções $f(x) = x^2$ e $g(x) = \log x$. Temos aqui dois casos *específicos* e matematicamente bem definidos de funções. Entretanto, sabemos que é perfeitamente possível sermos mais *gerais* e fornecermos uma definição para o conceito de *função*[†] tal que as funções $f(x)$ e $g(x)$ anteriores são casos particulares (ou instanciações) 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*. Em nossos estudos, é bastante importante desenvolver a flexibilidade mental necessária para compreender essa distinção entre o caso geral de uma definição e seus casos particulares.

Diagrama de estados

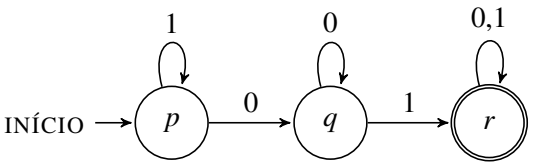
No caso de uma instância particular de um AFD, é possível fornecer uma representação gráfica do autômato na forma de um diagrama de estados, com setas para indicar o comportamento da função de transição, como fizemos na Figura 3.1. Por convenção, indicamos os estados finais por dois círculos concêntricos e o estado inicial pela indicação INÍCIO. Um ponto importante a observar é que não é possível desenhar um diagrama para um AFD "em geral", mas apenas para instâncias particulares. Usando a analogia apresentada anteriormente, não podemos desenhar o gráfico do conceito geral de uma função, mas apenas o gráfico de casos particulares, como $f(x) = x^2$ ou $g(x) = \log x$.

[†]Curiosamente, uma função é definida como um caso particular de um objeto matemático ainda mais geral, conhecido como *relação*.

■ **Exemplo 3.2 — Detectando substrings.** Considere o problema de detectar se uma dada string binária de entrada contém a substring 01. Esse problema pode ser resolvido pelo autômato $A = (Q, \Sigma, \delta, p, \{r\})$, tal que $Q = \{p, q, r\}$, $\Sigma = \{0, 1\}$ e a função δ é definida abaixo:

$$\delta(p,1) = p,\ \ \delta(p,0) = q,\ \ \delta(q,0) = q,\ \ \delta(q,1) = r,\ \ \delta(r,0) = r,\ \ \delta(r,1) = r.$$

O diagrama do autômato A é o seguinte:

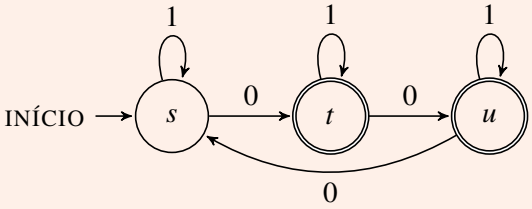


■

Exercício 3.1 (Resolvido) Desenhe o diagrama do autômato $A = (\{s,t,u\}, \{0,1\}, \delta, t, \{t,u\})$, tal que a função δ é definida abaixo:

$$\delta(s,0) = t, \qquad \delta(s,1) = s, \qquad \delta(t,0) = u,$$
$$\delta(t,1) = t, \qquad \delta(u,0) = s, \qquad \delta(u,1) = u.$$

Solução:



■

A função δ do Exemplo 3.2 é um tipo de função definida ponto a ponto (ou seja, ela não possui uma regra geral que descreva seu comportamento, como ocorre com as funções que estamos acostumados a estudar em Cálculo, por exemplo, $f(x) = x^2$). Em Teoria da Computação, funções que não podem ser facilmente descritas por regras gerais são bastante comuns. Para facilitar sua representação, descreveremos δ por meio de uma tabela, como a apresentada no exemplo a seguir:

	0	**1**
→ ***p***	q	p
q	q	r
****r***	r	r

Table 3.1: Tabela de transições da função δ do AFD do Exercício 3.2.

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 de aceitação. 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.

3.1.2 Aceitação e rejeição de strings: ideia intuitiva

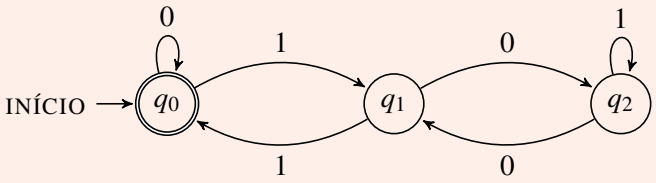
A ideia central é enxergar o AFD como um objeto que recebe uma string $w = w_1 w_2 \dots w_n$ como entrada, processa os símbolos de w, um a um, e produz uma saída. Durante o processamento de w, cada passo consiste em ler um símbolo w_i, mudar de estado e descartar w_i. Assim, o AFD descarta, passo a passo, os símbolos de w enquanto muda de estado. 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, após a última transição de estados, o AFD atinja um determinado estado q. A ideia é que, se q for um estado de aceitação, diremos que o AFD *aceitou* a string w. Por outro lado, se q não for um estado de aceitação, diremos que o AFD *rejeitou* a string w.

Exercício 3.2 (Resolvido) Forneça um AFD com alfabeto binário que aceite as strings w tal que $N(w)$ é um múltiplo de 3.

Solução: O seguinte autômato resolve o problema:



■

Exercício 3.3 Seja $\Sigma = \{a, b, c\}$. Seja $L = \{w \in \Sigma^* : w \text{ é da forma } xbbya, \text{ sendo que } x, y \in \Sigma^*\}$. Construa um AFD que aceite a string w fornecida como entrada se, e somente se, $w \in L$. ■

Exercício 3.4 Seja $\Sigma = \{0, 1\}$. Seja $L = \{w \in \Sigma^* : |w| \leq 3\}$. Construa um AFD que aceite a string w fornecida como entrada se, e somente se, $w \in L$. ■

Exercício 3.5 Seja $\Sigma = \{0, 1\}$. Seja $L = \{w \in \Sigma^* : w \text{ contém a substring } 011\}$. Construa um AFD que aceite a string w fornecida como entrada se, e somente se, $w \in L$. ■

O que precisamos fazer agora é transformar essa ideia intuitiva de que um AFD aceita algumas strings e rejeita outras em definições matemáticas precisas.

3.1.3 Aceitação e rejeição de strings: definição formal

Voltando à nossa máquina de vender chocolates, considere o seguinte: se a máquina estiver no estado B e receber três moedas de 1 real, em que estado ela vai parar? A resposta é que a máquina atingirá o estado E. O mais importante aqui é atentar para a pergunta que fizemos: *dado um estado e uma sequência de moedas, qual é o estado resultante?*

O que vamos fazer agora é formalizar essa ideia de que, dado um estado q e uma sequência de símbolos w, obtemos um estado resultante q'. O objeto matemático que modela essa ideia é uma função $\hat{\delta} : Q \times \Sigma^* \to Q$, cuja definição é obtida a partir da função δ original do autômato. Para o exemplo da máquina de vender chocolates, $\hat{\delta}_c$ deve ser definida de modo que $\hat{\delta}_c(\text{B}, 111) = \text{E}$.

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

Caso base: $w = \varepsilon$.
$\hat{\delta}(q, \varepsilon) = q$

Passo indutivo: $|w| > 0$.
Seja w uma string da forma $w = xa$, em 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. Essa definição 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 $\hat{\delta}(q_0, w) \in F$, dizemos que D *aceita* a string w. Caso contrário, isto é, se $\hat{\delta}(q_0, w) \notin F$, dizemos que D *rejeita* a string w.

Podemos generalizar a ideia de aceitação de uma string para a 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. Nesse caso, dizemos que L é a linguagem de D. Essa noção é formalizada a seguir.

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

Seja L uma linguagem. Se um AFD D satisfaz $L(D) = L$, dizemos que D *decide* a linguagem L. Dizemos também que D *aceita*[1] a linguagem L. Estamos agora em condições de definir formalmente o conceito de linguagem regular:

Definição 3.1.5 — Liguagem Regular. Uma linguagem L é dita *regular* se existe um AFD D satisfazendo $L = L(D)$.

Exercício 3.6 Seja $\Sigma = \{0, 1\}$. Construa um AFD que aceite a string fornecida como entrada se, e somente se, a string pertence a linguagem L, para cada um dos casos abaixo:

(a) $L = \Sigma^*$.
(b) $L = \{w : w \text{ é terminada em } 00\}$.
(c) $L = \{w : w \text{ tenha um número ímpar de 1's}\}$.
(d) $L = \{w : w \text{ tem ao mesmo tempo um número par de 0's e um número par de 1's}\}$
(e) $L = \{w : \text{o número de 0's em } w \text{ é divisível por 3 e o número de 1's é divisível por 5}\}$.
(f) $L = \{w : \text{se } w' \text{ é uma substring de } w \text{ com } |w'| = 5, \text{ então } w' \text{ tem pelo menos dois 0's}\}$. ■

Exercício 3.7 Considere um AFD $D = (Q, \Sigma, \delta, q_0, F)$. 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 3.8 Seja L uma linguagem sobre o alfabeto Σ. O complemento de L, denotado por $\overline{L}$, é definido por $\overline{L} = \Sigma^* \setminus L$. Prove que, se L é regular, então $\overline{L}$ também é regular. ■

[1] Alguns textos utilizam também a expressão "L é reconhecida por D".

Exercício 3.9 Considere o seguinte AFD $D = (Q, \Sigma, \delta, q_0, F)$, tal que

- $Q = q_0, q_1, q_2, \ldots, q_6$.
- $\Sigma = 0, 1, 2, \ldots, 6$.
- $F = q_0$.
- A função de transição δ é definida por $\delta(q_j, i) = q_k$, onde $k = (j + i) \bmod 7$.

Qual é a linguagem aceita por D?

Exercício 3.10 Seja $D = (Q, \Sigma, \delta, q_0, F)$ um AFD. Escreva um programa em pseudocódigo que receba como entrada uma string $w \in \Sigma^*$ e responda SIM se $w \in L(D)$ e NÃO caso contrário.

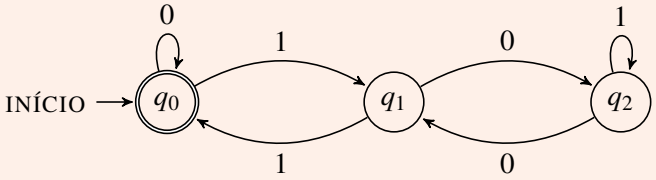
Exercício 3.11 Considere os seguintes AFDs: $D_Q = (Q, \Sigma, \delta_Q, q_0, F_Q)$ e $D_P = (P, \Sigma, \delta_P, p_0, F_P)$, sendo $Q = \{q_0, q_1, \ldots, q_k\}$ e $P = \{p_0, p_1, \ldots, p_h\}$.

Construiremos agora um terceiro AFD, denotado por D_A, a partir dos dois AFDs anteriores. A ideia é que, dada qualquer string w, o AFD D_A simule simultaneamente a computação de D_Q com w e a de 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, nessa mesma transição, o AFD D_A estará no estado (q_1, p_5). A definição formal dos componentes de $D_A = (A, \Sigma, \delta_A, a_0, F_A)$ é dada a seguir:

- $A = Q \times P$ (ou seja, para cada $q_i \in Q$ e $p_j \in P$, existe um estado $(q_i, p_j) \in A$).
- $a_0 = (q_0, p_0)$.
- $F_A = \{$"conjunto de todos os elementos (q_i, p_j) tais que $q_i \in F_Q$ e $p_j \in F_P$"$\}$.
- Definição de δ_A: para todo $q_i, q_j \in Q$, 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?

Exercício 3.12 (Opcional) No Exercício 3.2, mostramos que o autômato abaixo aceita a linguagem $L_3 = \{w : N(w)$ é um múltiplo de $3\}$.



Prove formalmente que essa solução está correta. Como dica, use indução para mostrar que $w \in L_3$ se, e somente se, $\hat{\delta}(q_0, w) = q_0$.

Exercício 3.13 (Opcional) Mostre que, para todo inteiro $k \geq 1$, existe um AFD D_k que testa se um número é divisível por k, isto é, que aceita a linguagem $L_k = \{w : N(w)$ é divisível por $k\}$.

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, existe exatamente uma transição definida para (q,a). Vamos apresentar agora um modelo de autômato que pode ter várias escolhas possíveis de transições a serem executadas a partir de um par (q,a). Esse autômato possuirá uma habilidade "mágica" de *adivinhar* qual é a transição correta que deve ser escolhida. O diagrama a seguir ilustra um exemplo de autômato com essas propriedades:

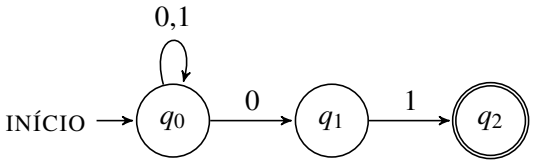


Figura 3.2: Um autômato finito não determinístico.

Imediatamente notamos uma diferença entre o autômato da Figura 3.2 e os autômatos vistos na Seção 3.1. Se o autômato da Figura 3.2 estiver no estado q_0 e o símbolo lido for 0, existem duas opções que podem ser escolhidas pelo autômato: (1) continuar no estado q_0; ou (2) mudar para o estado q_1. A questão-chave é entender quais strings são aceitas e quais são rejeitadas pelo autômato.

Por exemplo, a string 001 pode ser aceita caso o autômato processe o primeiro bit da string realizando a transição $\textcircled{q_0} \to \textcircled{q_0}$, processe o segundo bit realizando a transição $\textcircled{q_0} \to \textcircled{q_1}$ e finalize a computação processando o bit 1 com a transição $\textcircled{q_1} \to \textcircled{q_2}$. Por outro lado, essa mesma string pode ser rejeitada caso o autômato processe os três bits da string sempre escolhendo a transição $\textcircled{q_0} \to \textcircled{q_0}$.

A ideia aqui é que o autômato tem a capacidade de adivinhar quais transições deve realizar para que a string de entrada seja aceita. Isto é, a linguagem do autômato consistirá de todas as strings para as quais exista uma sequência de escolhas que leve o autômato à aceitação. Ainda assim, note que nem toda string pertence à linguagem do autômato, pois, mesmo com a habilidade de adivinhar quais escolhas fazer, pode ocorrer que simplesmente não exista nenhuma sequência de escolhas de transições que leve o autômato a aceitar determinadas strings. Por exemplo, o autômato da Figura 3.2 não consegue aceitar a string 000, independentemente das escolhas feitas durante seu processamento. Portanto, a string 000 não pertence à linguagem do autômato.

Se prestarmos atenção no autômato em questão, veremos que ele tem a capacidade de aceitar exatamente as strings terminadas em 01. Dada uma string da forma $x01$, em que x é uma substring qualquer, o autômato pode processar toda a substring x usando a transição $\textcircled{q_0} \to \textcircled{q_0}$. Quando restarem apenas os dois símbolos finais, o autômato utiliza a transição $\textcircled{q_0} \to \textcircled{q_1}$ e, em seguida, a transição $\textcircled{q_1} \to \textcircled{q_2}$. Observe que, se a string não termina em 01, o autômato nunca conseguirá terminar o processamento no estado de aceitação.

Observe que a habilidade de o autômato poder escolher entre duas transições possíveis não é a única diferença que esse novo modelo apresenta em relação aos autômatos determinísticos da seção anterior. Outra situação que pode ocorrer nesse modelo é aquela em que o autômato não possui nenhuma transição definida para um determinado par (estado, símbolo). Por exemplo, se o autômato da Figura 3.2 estiver no estado q_1 e o próximo símbolo a ser lido for 0, o autômato não terá nenhuma transição que possa realizar. Nesse caso, diremos que o autômato *morre*.

NÃO DETERMINISMO?

Claramente, do ponto de vista prático, um autômato "mágico", que adivinha as transições que deve realizar, não parece ser um modelo realista de computação que corresponda a algo implementável na prática. Ainda assim, o estudo desse modelo matemático será bastante útil.

Em Teoria da Computação, esse tipo de situação ocorre com frequência, e modelos "'não realistas" acabam sendo relevantes porque, em última análise, são *ferramentas matemáticas úteis*, inclusive para nos ajudar a compreender modelos realistas de computação. No Capítulo 9, veremos que certas Máquinas de Turing não determinísticas são usadas para definir um conjunto de linguagens conhecido como NP, que faz parte do famoso (e concreto) problema P *vs* NP. Em um curso mais aprofundado de Complexidade Computacional, a definição de modelos 'irreais" de computação, que ainda assim são matematicamente úteis para lidar com problemas ou modelos concretos de computação, ocorre com bastante frequência.

3.2.1 Definição formal para autômatos finitos não determiníticos

Autômatos finitos não determinísticos têm uma definição formal muito parecida com a 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 haver zero, um ou mais estados possíveis de serem atingidos por uma transição rotulada por a. Mais precisamente, a função de transição tem a forma $\delta(q,a) = S$, em que S é um conjunto qualquer de estados. Por exemplo, a função δ do autômato da Figura 3.2, quando aplicada ao par $(q_0,0)$, deve produzir o conjunto q_0,q_1, ou seja, $\delta(q_0,0) = q_0,q_1$. Note que, como os elementos da imagem de δ são conjuntos, o contradomínio da função δ é o conjunto de todos os subconjuntos de Q, ou seja, o conjunto potência $\mathscr{P}(Q)$.

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* $\delta : Q \times \Sigma \to \mathscr{P}(Q)$;

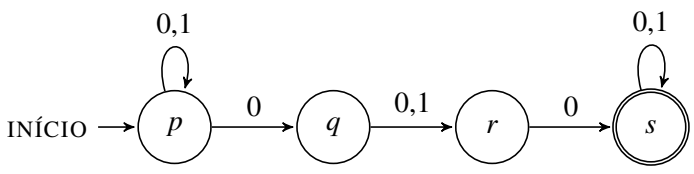
$q_0 \in Q$ é o *estado inicial*;

$F \subseteq Q$ é o conjunto de *estados finais*.

■ **Exemplo 3.3** 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:

$\delta(q_0,0) = \{q_0,q_1\}$, $\delta(q_0,1) = \{q_0\}$, $\delta(q_1,1) = \{q_2\}$, $\delta(q_1,0) = \delta(q_2,0) = \delta(q_2,1) = \emptyset$. ■

Assim como os AFDs podem ser representados por tabelas de transições, o mesmo ocorre com os AFNs. A única diferença é que cada célula da tabela correspondente à aplicação da função de transição δ contém um conjunto de estados. A seguir um exemplo de AFD e sua respectiva tabela de transições.



	0	**1**
→ ***p***	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\varnothing$
****s***	$\{s\}$	$\{s\}$

Uma maneira bastante útil de descrever as possíveis escolhas que um AFN A faz durante a computação com uma string de entrada w é por meio de uma árvore, chamada de *árvore de computações possíveis de A com w*. Para definirmos formalmente este conceito, precisamos antes apresentar a seguinte definição auxiliar que é a base da definição: Dado um AFN A, q um um estado de A e w é uma string sobre o alfabeto de A, uma (A,q,w)-*árvore* é uma árvore enraizada em q, definida recursivamente da seguinte maneira:

Base: Se $w = \varepsilon$, então a árvore contém apenas o nó raíz q

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

Definição 3.2.2 — Árvore de computações possíveis. Seja $A = (Q, \Sigma, \delta, q_0, F)$ um AFN e $w \in \Sigma^*$. Uma *árvore de computações possíveis de A com w* é uma (A, q_0, w)-árvore.

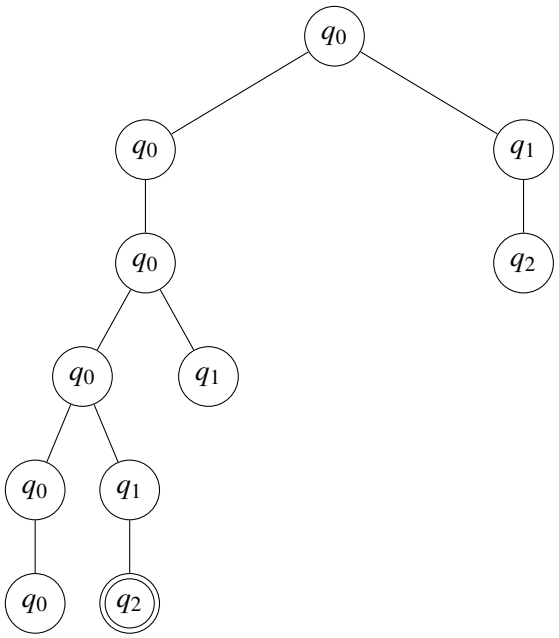
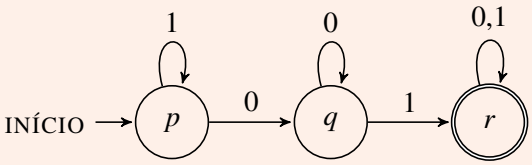


Figura 3.3: A árvore de computações possíveis do AFN N da Figura 3.2 com a string 01001.

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 em que o AFN pode estar após i transições, 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 pelo menos um estado de aceitação. No exemplo da Figura 3.3, o nível 5 da árvore contém o estado q_2. O fato de esse nível conter apenas um nó correspondente a um estado de aceitação significa que o AFN tem exatamente uma computação possível que aceita a string 01001. Essa computação é definida pela sequência de estados do caminho que liga a raiz ao estado final em questão.

Exercício 3.14 Desenhe a árvore de computações possíveis para o AFN do Exemplo 3.3 com a string de entrada 111010. ■

Exercício 3.15 (Resolvido) Apresente uma definição formal para um AFN do diagrama abaixo:



Solução: Observe que o diagrama acima é idêntico ao diagrama do AFD apresentado no Exemplo 3.2, cuja tabela de transições é a seguinte:

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

Entretanto, neste exercício, estamos considerando que o diagrama acima corresponde a um AFN, e não a um AFD. No caso de AFNs cujo diagrama seja idêntico ao de um AFD, ocorre o seguinte: se, no AFD, a função de transição satisfaz $\delta(x,y) = z$, então, no AFN, a função de transição correspondente satisfaz $\delta(x,y) = \{z\}$. Com isso, a definição formal do AFN é dada pela seguinte tabela de transições:

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

■

Note que, apenas olhando o diagrama do Exercício 3.15, não é possível dizer se se trata de um autômato determinístico ou não determinístico. Este é um exemplo em que um desenho pode ser ambíguo e, não por acaso, normalmente não consideramos desenhos como objetos matemáticos formais. Por outro lado, note que, na solução do Exercício 3.15, a definição formal do AFN não apresenta nenhuma ambiguidade, pois fica claro que se trata de um autômato não determinístico: a imagem da função de transição contém conjuntos de estados, em vez de estados.

Exercício 3.16 Desenhe a árvore de computações possíveis para o AFN do Exercício 3.15 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 do que a definição correspondente para AFDs. A ideia básica é a seguinte: dado um estado q e uma string x, queremos determinar o conjunto de estados em que o autômato pode estar após o processamento de x, supondo que a computação tenha começado no estado q.

Definição 3.2.3 Dado um AFN $N = (Q, \Sigma, \delta, q_0, F)$, definimos $\hat{\delta} : Q \times \Sigma^* \to \mathscr{P}(Q)$:

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 $\hat{\delta}(q, x) = \{p_1, p_2, ..., p_k\}$. Então
$\hat{\delta}(q, w) = \bigcup_{i=1}^{k} \delta(p_i, a)$

Exercício 3.17 Calcule $\hat{\delta}(q_0, 00101)$ do AFN do Exemplo 3.3. ∎

Definição 3.2.4 — Linguagem de um AFN. Seja $N = (Q, \Sigma, \delta, q_0, F)$ um AFN. A linguagem de N, denotada por $L(N)$, é definida por $L(N) = \{w \in \Sigma^* \ : \ \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$.

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

3.2.3 AFDs são casos particulares de AFNs

Embora seja intuitivo que AFDs são casos particulares de AFNs, vamos formalizar esta ideia.

Teorema 3.2.1 Seja $D = (Q, \Sigma, \delta, q_0, F)$ um AFD. Então existe um AFN N tal que $L(D) = L(N)$.

Prova: A ideia é generalizar o que fizemos no Exercício Resolvido 3.15. 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 definida da seguinte for: se $\delta(x, y) = z$, então defina $\delta'(x, y) = \{z\}$. □

Terminologia: Dado um AFD D, o AFN *equivalente* a D é o AFN obtido pelo procedimento acima.

Definição 3.2.5 — Árvore de computações possíveis para AFDs. Seja $D = (Q, \Sigma, \delta, q_0, F)$ um AFD e $w \in \Sigma^*$. A *árvore de computações possíveis de D com w* é árvore de computações possíveis do AFN equivalente a D.

Exercício 3.18 Qual é o formato de uma árvore de computações possíveis de um AFD com uma string qualquer? ∎

Exercício 3.19 Considere o seguinte caso particular de um AFN: o autômato pode ter várias transições saindo de um mesmo estado com o mesmo rótulo, mas nunca pode morrer. Em outras palavras, trata-se de um AFN cuja função de transição tem a forma $\delta : Q \times \Sigma \to (\mathscr{P}(Q) \setminus \varnothing)$. Prove que, dado um AFN qualquer N, é sempre possível construir um AFN N' que nunca morre e que aceita a mesma linguagem que N. ∎

3.3 Equivalência entre AFDs e AFNs

Um aspecto com o qual lidaremos com frequência 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 que generalizam os AFDs. Nesses casos, dizemos que esses outros modelos são mais poderosos (ou mais expressivos) do que o modelo de AFD.

A Expressividade de um Modelo de Computação

A ideia de expressividade de um modelo de computação é central na Teoria da Computação. Embora os AFDs sejam capazes de realizar tarefas interessantes, como, por exemplo, testar a divisibilidade (veja o Exercício 3.13), veremos, no Capítulo 4, que existem tarefas algorítmicas que eles não conseguem realizar. Em particular, provaremos que não existe um AFD D tal que $L(D)$ seja a linguagem dos números primos escritos em binário. Em outras palavras, os AFDs não são capazes de testar a primalidade.

Por outro lado, sabemos que é possível (e relativamente simples) escrever um algoritmo em pseudocódigo (ou em uma linguagem de programação, como C) para testar a primalidade de um número. Concluímos, assim, que o formalismo dos AFDs é mais *fraco* do que o dos pseudocódigos, pois toda computação realizada por um AFD pode ser expressa em pseudocódigo (veja o Exercício 3.10), mas nem toda computação expressa em pseudocódigo pode ser representada por um AFD.

O que está no pano de fundo desta discussão é a seguinte questão: sempre que estudamos um modelo de computação, procuramos compreender como ele se compara aos demais em termos de sua capacidade de expressar algoritmos. Como nesta seção estamos estudando os AFNs, queremos saber se eles são capazes de expressar algoritmos para realizar tarefas que os AFDs não conseguem realizar.

No Teorema 3.2.1, mostramos que toda linguagem aceita por um AFD também pode ser aceita por um AFN. Isso significa que os AFNs são, pelo menos, tão poderosos quanto os AFDs. Esse resultado é natural, pois os AFNs constituem uma generalização dos AFDs. A pergunta que surge, então, é: os AFNs são *estritamente* mais poderosos do que os AFDs? Veremos nesta seção que a resposta é **não**. Em outras palavras, mostraremos que, se uma linguagem pode ser aceita por um AFN, então existe um AFD que aceita a mesma linguagem. Isso pode parecer surpreendente, pois os AFNs, em algumas situações, possuem a capacidade de "adivinhar" qual transição devem realizar, uma capacidade de que os AFDs não dispõem.

3.3.1 Algoritmo de construção de conjuntos

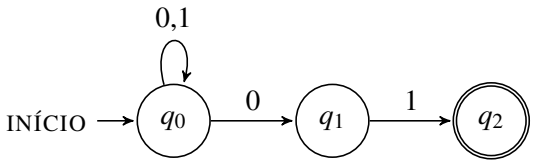
O Algoritmo AFD-EQUIVALENTE, apresentado a seguir, recebe um AFN $N = (Q_N, \Sigma, \delta_N, q_0, F_N)$ como entrada e retorna um AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ tal que $L(N) = L(D)$. A ideia central do algoritmo é construir um AFD D capaz de "simular" N.

Considere uma string de entrada $w = w_1 \cdots w_i \cdots w_n$. Quando o AFN N está prestes a processar o i-ésimo símbolo de w, os possíveis estados em que ele pode se encontrar são exatamente os nós do i-ésimo nível da árvore de computações possíveis de N com a entrada w. Seja P_i o conjunto desses estados e seja P_{i+1} o conjunto dos estados do nível seguinte da árvore. O algoritmo constrói um AFD cujos estados são subconjuntos de Q_N. Em particular, haverá uma transição do estado P_i para o estado P_{i+1} rotulada por w_i, isto é, $\delta_D(P_i, w_i) = P_{i+1}$.

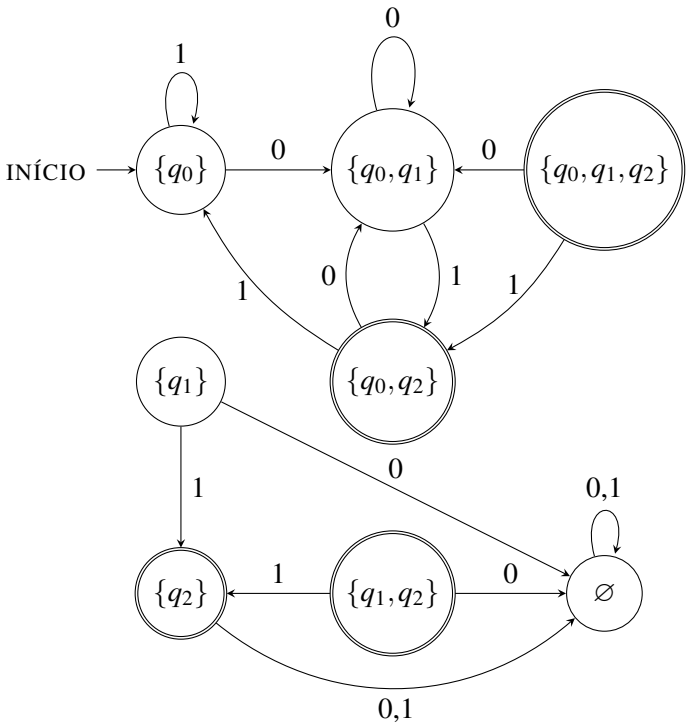
Para construir a função de transição δ_D, o algoritmo considera todos os pares (P, a), em que $P \subseteq Q_N$ e $a \in \Sigma$. Para cada um desses pares, calcula o conjunto de estados alcançáveis a partir de algum estado de P por meio da leitura do símbolo a, definindo, assim, o valor de $\delta_D(P, a)$. A seguir a formalização do algoritmo.

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

Considere o AFN abaixo (observe quee este AFN é o mesmo que vimos na Figura 3.2).



Ao aplicarmos o Algoritmo AFD-EQUIVALENTE ao AFN acima, obtemos como resultado o AFD de oito estados apresentado abaixo, juntamente com sua respectiva tabela de transições.



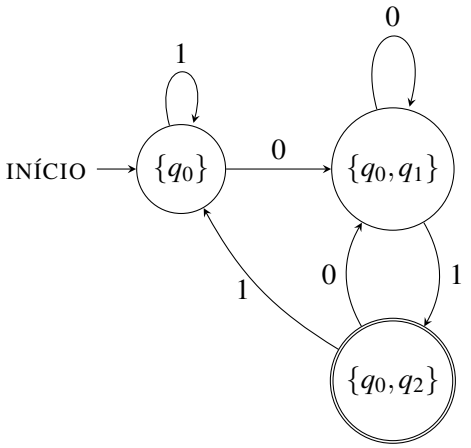
	0	**1**
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow \{q_0\}$	$\{q_0,q_1\}$	$\{q_0\}$
$\{q_1\}$	$\varnothing$	$\{q_2\}$
$^*\{q_2\}$	$\varnothing$	$\varnothing$
$\{q_0,q_1\}$	$\{q_0,q_1\}$	$\{q_0,q_2\}$
$^*\{q_0,q_2\}$	$\{q_0,q_1\}$	$\{q_0\}$
$^*\{q_1,q_2\}$	$\varnothing$	$\{q_2\}$
$^*\{q_0,q_1,q_2\}$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

Primeiramente, observe que o AFD acima possui dois "pedaços". Embora isso possa parecer estranho (e até mesmo indesejável, uma vez que alguns estados nunca são alcançados a partir do estado inicial), do ponto de vista formal, trata-se de um AFD perfeitamente válido. Uma segunda característica do autômato acima é que cada estado do AFD é um conjunto (isto é, cada elemento da primeira coluna da tabela de transições é um conjunto). Isso também não representa nenhum problema, pois os estados de um AFD podem ser objetos de qualquer natureza, desde que a imagem da função δ seja composta por elementos do mesmo tipo que os estados do autômato (note que os elementos da segunda e da terceira colunas da tabela de transições também são conjuntos). O que não devemos fazer é confundir essa situação com a dos AFNs, nos quais os elementos da imagem da função δ têm natureza diferente da dos estados do autômato.

3.3.2 Algoritmo de construção de conjuntos: versão melhorada

O algoritmo apresentado na seção anterior funciona corretamente e produz um AFD que aceita a mesma linguagem que o AFN de entrada. Entretanto, como já mencionamos, é possível notar que o AFD possui muitos estados desnecessários, que podem ser eliminados. Precisamos considerar apenas os estados *alcançáveis* a partir do estado inicial $\{q_0\}$.

Observando o diagrama do AFD apresentado no final da seção anterior, notamos que, a partir do estado inicial q_0, atingimos os estados $\{q_0\}$ e $\{q_0,q_1\}$, dependendo do símbolo lido. A partir de $\{q_0,q_1\}$, atingimos $\{q_0,q_2\}$ quando o símbolo lido é 1. Por sua vez, a partir de $\{q_0,q_2\}$, independentemente do símbolo lido, os únicos estados alcançáveis são $\{q_0\}$ e $\{q_0,q_1\}$, que já foram mencionados. Portanto, como a computação sempre começa no estado $\{q_0\}$, os únicos estados alcançáveis para qualquer string de entrada são $\{q_0\}$, $\{q_0,q_1\}$ e $\{q_0,q_2\}$. Com isso, podemos eliminar os estados desnecessários e simplificar o autômato, conforme abaixo.



	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 o formalismo neste ponto, mas, se quiséssemos fornecer uma definição formal para o conceito de estado alcançável, poderíamos recorrer à ideia de um vértice alcançável por uma busca em largura em um grafo direcionado. Pense no AFD obtido como saída do Algoritmo AFD-EQUIVALENTE como um grafo direcionado cujos vértices são os estados e cujas arestas ligam o vértice p ao vértice q sempre que existir algum símbolo a tal que $\delta(p,a) = q$. Com isso, os *estados alcançáveis* são exatamente os vértices que podem ser alcançados por algum caminho direcionado a partir do vértice correspondente ao estado inicial.

Convenção: A partir de agora, sempre que formos executar o Algoritmo de construção de conjuntos, vamos executar passo a passo, mantendo apenas os estados alcançáveis. O AFD obtido desta forma será chamado de AFD *equivalente* ao AFN fornecido como entrada para algoritmo.

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

Exercício 3.20 (Opcional) Prove o Teorema 3.3.1. Dica: prove por indução que $\forall w \in \Sigma^*$, $\hat{\delta}_D(q_0,w) \in F_N \Leftrightarrow \hat{\delta}_N(\{q_0\},w) \cap F_N$. ∎

A seguir enunciamos a equivalência entre AFDs e AFNs enquanto modelos de computação.

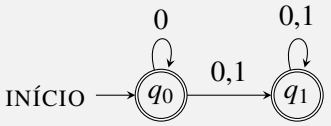
Teorema 3.3.2 Uma linguagem L é aceita por um AFD se e somente se L é aceita por um AFN.

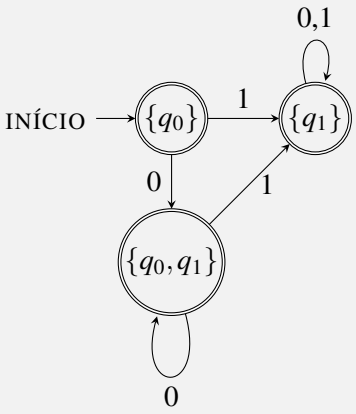
Prova: Consequência do Teorema 3.3.1 Teorema 3.2.1.

MINIMIZAÇÃO DE AUTÔMATOS

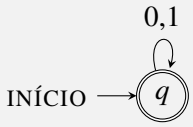
A versão melhorada do algoritmo serve para eliminar estados inúteis gerados pela versão inicial. Entretanto, isso não significa que o AFD produzido pelo algoritmo seja mínimo, isto é, que ele possua o menor número possível de estados dentre todos os AFDs que reconhecem a mesma linguagem do AFN de entrada.

No exemplo usado anteriormente, o AFD produzido pelo algoritmo é, por coincidência, mínimo. Entretanto, isso não ocorre em geral. Considere o AFN apresentado à esquerda na figura abaixo. Ao executarmos o algoritmo, obtemos o AFD apresentado à direita. Note que todos os estados desse AFD são alcançáveis.





Entretanto, a linguagem aceita por esse AFD é $\{0,1\}^*$, e ela pode ser aceita pelo AFD a seguir, que possui apenas um estado:



Nosso foco aqui não será entrar nos pormenores de AFNs e AFDs, pois estamos interessados apenas em compreender que tais autômatos constituem modelos de computação e podem ser utilizados para representar determinados algoritmos. Os leitores interessados em um estudo mais aprofundado podem consultar os livros de Hopcroft, Motwani e Ullman [HMU06] e de Lewis e Papadimitriou [LP98], que apresentam algoritmos para a minimização de AFDs.

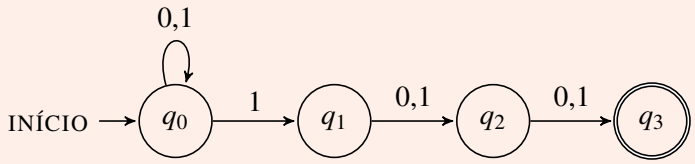
Exercícios:

Exercício 3.21 Considere o o AFN dado pela tabela abaixo:

	0	1
$\to p$	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\{r\}$
r	$\{s\}$	$\varnothing$
$*s$	$\{s\}$	$\{s\}$

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

Exercício 3.22 Qual a linguagem aceita pelo AFN abaixo?



■

Exercício 3.23 Forneça um AFD equivalente ao AFN do Exercício 3.22. ■

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 [HMU06]. 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 $\{0,1,2,3,4,5,6,7,8,9,\cdot,+,-\}$. 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:

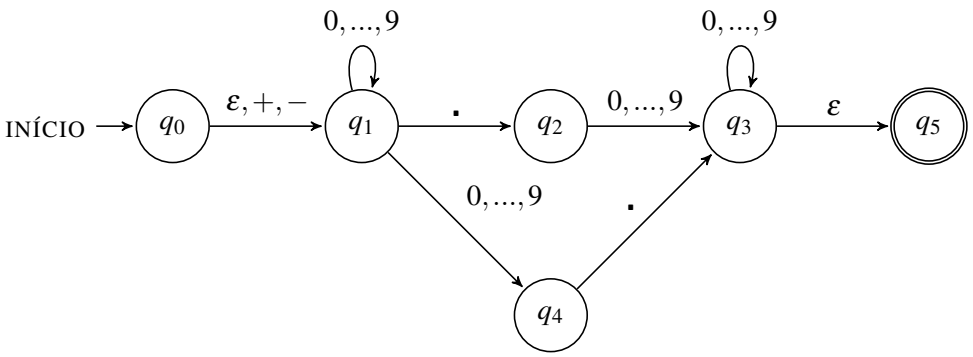


Figura 3.4: Um ε-AFN que aceita números decimais.

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 $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathscr{P}(Q)$, sendo que $\varepsilon \notin \Sigma$.

■ **Exemplo 3.4** O ε-AFN do exemplo anterior é $E = (\{q_0, ..., q_5\}, \{., +, -, 0, ..., 9\}, \delta, q_0, \{q_5\})$ tal que a tabela de transições de δ é a seguinte:

	ε	*sinal*	$\cdot$	dígito
q_0	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$
q_1	$\varnothing$	$\varnothing$	$\{q_2\}$	$\{q_1, q_4\}$
q_2	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_3	$\{q_5\}$	$\varnothing$	$\varnothing$	$\{q_3\}$
q_4	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$
$^*q_5$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

■

Observe que no Exemplo 3.4, se quiséssemos ser rigorosos, a tabela de transições teria que ter 14 colunas. Mais precisamente, 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 ε-*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 trasições 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: $q \in \varepsilon$-Fecho(q)

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

O ε-Fecho de um estado q também é simplesmente chamado de *fecho* de 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 transição estendida de um AFN, com o cuidado extra de adicionar os estados que podem ser atingidos por trasições ε.

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

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

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

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

Então

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

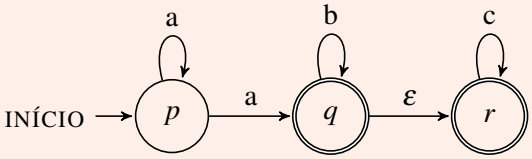
O próximo passo é definir o que é a linguagem de um ε-AFN.

Definição 3.4.4 — Linguagem de ε-AFNs. Seja $E = (Q, \Sigma, \delta, q_0, F)$ um ε-AFN. Definimos $L(E) = \{w \ : \ \hat{\delta}(q_0, w) \cap F \neq \varnothing\}$ 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 3.24 (Resolvido) 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 1 Construindo um AFD a partir de um ε-AFN

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

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

Convenção: De maneira semelhante ao Algoritmo AFD-EQUIVALENTE, a partir de agora sempre que formos construir um AFD equivalente a um dado ε-AFN, vamos construir o AFD passo a passo, mantendo apenas os estados alcançáveis.

Teorema 3.5.1 Se o AFD $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ é obtido pelo Algoritmo 1 quando toma como entrada o ε-AFN $E = (Q_N, \Sigma, \delta_N, q_0, F_N)$, então $L(D) = L(N)$.

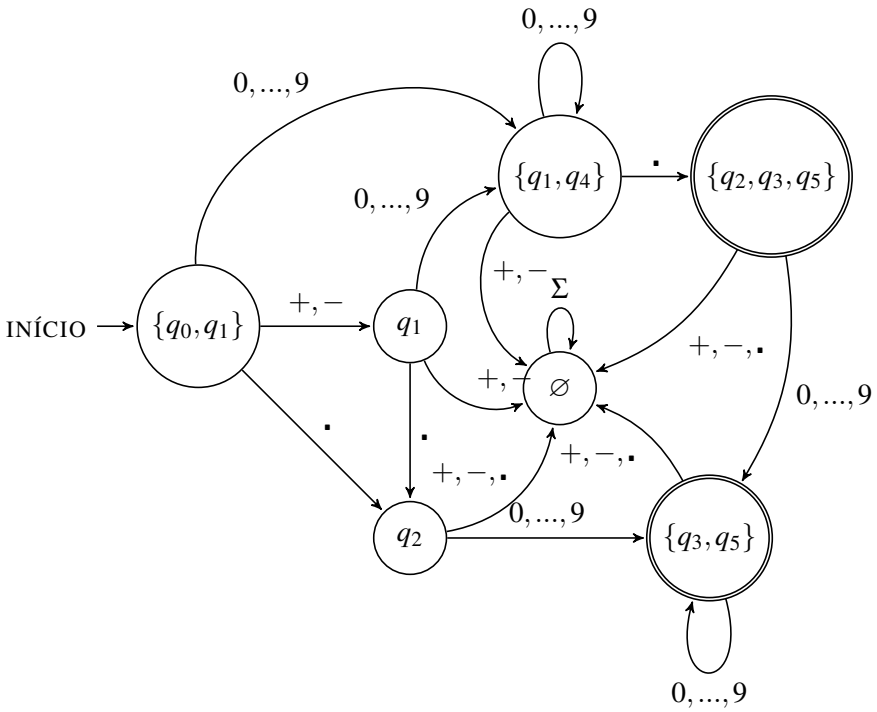
Teorema 3.5.2 Uma linguagem é aceita por um AFD se e somente se é aceita por um ε-AFN.

Vamos construir um AFD equivalente ao ε-AFN da Figura 3.4 usando o Algoritmo 1. A tabela

do AFD que o algoritmo retorna é o seguinte:

	$+,-$	$0,\ldots,9$	.
$\to \{q_0,q_1\}$	$\{q_1\}$	$\{q_1,q_4\}$	$\{q_2\}$
$\{q_1\}$	$\varnothing$	$\{q_1,q_4\}$	$\{q_2\}$
$\{q_1,q_4\}$	$\varnothing$	$\{q_1,q_4\}$	$\{q_2,q_3,q_5\}$
$\{q_2\}$	$\varnothing$	$\{q_3,q_5\}$	$\varnothing$
$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
$*\{q_2,q_3,q_5\}$	$\varnothing$	$\{q_3,q_5\}$	$\varnothing$
$*\{q_3,q_5\}$	$\varnothing$	$\{q_3,q_5\}$	$\varnothing$

O diagrama do AFD obtido pelo Algoritmo 1 é o seguinte:



Exercícios:

Exercício 3.25 Considere o ε-AFN dado pela tabela abaixo:

	ε	a	b	c
$\to p$	$\varnothing$	$\{p\}$	$\{q\}$	$\{r\}$
q	$\{p\}$	$\{q\}$	$\{r\}$	$\varnothing$
$*r$	$\{q\}$	$\{r\}$	$\varnothing$	$\{p\}$

(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 3.26 Repita a questão 1 para o ε-AFN dado pela tabela abaixo:

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

■

Exercício 3.27 Apresente um AFD que decida a mesma linguagem do ε-AFN do Exercício Resolvido 3.24 ■

3.6 Expressões Regulares (ERs)

Uma *Expressão Regular* é uma expressão matemática que representa uma linguagem.

CONSTRUINDO UMA LINGUAGEM PASSO A PASSO

A ideia chave aqui é entender que uma expressão regular é uma fórmula que indica como construir linguagens mais complicadas a partir de linguagens triviais. Por exemplo, considere a linguagem $L =$ *"strings que começam com 0 e tem em seguida um número arbitrário de 1's ou começam com 1 e tem em seguida um números arbitrário de 0's"*. Vamos construir passo a passo L seguindo as seguintes instruções:

- Comece com as linguagens $\{0\}$ e $\{1\}$ e siga os seguintes passos:

  Passo 1: A partir de $\{0\}$, obtenha o fecho $\{0\}^* = \{\varepsilon, 0, 00, 000, ...\}$.

  Passo 2: A partir de $\{1\}$, obtenha o fecho $\{1\}^* = \{\varepsilon, 1, 11, 111, ...\}$.

  Passo 3: A partir de $\{0\}$ e $\{1\}^*$, obtenha a concatenação $\{0\} \cdot \{1\}^* = \{0, 01, 011, ...\}$.

  Passo 4: A partir de $\{1\}$ e $\{0\}^*$, obtenha a concatenação $\{1\} \cdot \{0\}^* = \{1, 10, 100, ...\}$.

  Passo 5: A partir de $\{1\}\{0\}^*$ e $\{0\}\{1\}^*$, obtenha a união $\{0, 01, 011, ...\} \cup \{1, 10, 100, ...\}$.

Observe que a linguagem obtida ao final do Passo 5 é exatamente L. Obtivemos esta linguagem começando com as linguagens triviais $\{0\}$ e $\{1\}$ e executando uma sequência de operações que podem ser de união, concatenação e fecho. A ideia é que uma fórmula do $10^* + 01^*$, chamada expressão regular, indica quais operações devem ser feitas:

A expressão 0^* se refere ao Passo 1, indicando como obter a linguagem $\{0\}^*$.

A expressão 1^* se refere ao Passo 2, indicando como obter a linguagem $\{1\}^*$.

A expressão 01^* se refere ao Passo 3, indicando como obter a linguagem $\{0\}\{1\}^*$.

A expressão 10^* se refere ao Passo 4, indicando como obter a linguagem $\{1\}\{0\}^*$.

A expressão $10^* + 01^*$ se refere ao Passo 5, indicando como obter a linguagem L.

O nosso objetivo agora é definir indutivamente quais expressões matemáticas são consideradas expressões regulares válidas. Algo bastante importante em nossa definição é que no mesmo momento tivermos estabelecido que uma certa expressão regular R é válida, também vamos estabelecer exatamente qual linguagem $L(R)$ corresponde a tal expressão.

Definição 3.6.1 — Expressões Regulares (ER).
Seja Σ um alfabeto qualquer. Uma expressão matemática consistindo de símbolos de Σ, parêntesis e '$+$' e '$*$' é uma expressão regular se tem forma descrita recursivamente a seguir:

Base:
(1) As expressões $\underline{\varepsilon}$ e $\underline{\varnothing}$ são expressões regulares que correspondem, respectivamente, às linguagens $L(\underline{\varepsilon}) = \{\varepsilon\}$ e $L(\underline{\varnothing}) = \varnothing$.
(2) Para todo símbolo a de Σ, a expressão $\underline{a}$ é uma expressão regular correspondente a linguagem $L(\underline{a}) = \{a\}$.

Passo indutivo:
(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, $L((E)) = L(E)$.

Terminologia 3.2. *As expressões regulares definidas na base da definição são chamadas de expressões regulares elementares. As expressões regulares definidas no passo indutivo são chamadas de expressões regulares compostas.*

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 $\underline{a+bc}$ é equivalente a $\underline{a+(bc)}$;
- O operador de menor precedência é o operador $+$;
- Como de costume, usamos parentesis para alterar a precedência de operadores.

Exercício 3.28 (Resolvido) A expressão regular para a linguagem L das "strings contendo 0's e 1's alternados" é $\underline{(01)^* + (10)^* + 0(10)^* + 1(01)^*}$. Justifique isto passo a passo.

Solução: Vamos ser bastante cuidadosos e resolver passo a passo.

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 1, concluímos que $\underline{1}$ 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 $\underline{01}$ também é uma expressão válida. A linguagem que esta expressão representa é $L(\underline{01}) = \{0\} \cdot \{1\} = \{01\}$.

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 $L(\underline{(01)^*}) = \{\varepsilon, 01, 0101, 010101, ...\}$.

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

Passo 6: Pelos Passos 4 e 5, $\underline{(01)^*}$ e $\underline{(10)^*}$ são expressões válidas que representam as linguagens $\{\varepsilon, 01, 0101, 010101, ...\}$ e $\{\varepsilon, 10, 1010, 101010, ...\}$, respectivamente. Pela Definição 3.6.1 (Indução, item 1), $\underline{(01)^* + (10)^*}$ é uma expressão válida e $L(\underline{(01)^* + (10)^*}) = \{\varepsilon, 01, 0101, 010101, ...\} \cup \{\varepsilon, 10, 1010, 101010, ...\} = \{\varepsilon, 01, 10, 0101, 1010, 010101, 1010101, ...\}$.

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 $\{0, 010, 01010, 0101010, ...\}$. Usando argumentos semelhantes, podemos concluir que $\underline{1(01)^*}$ é uma expressão regular válida e $L(\underline{1(01)^*}) = \{1, 101, 10101, 1010101, ...\}$. Com isso, podemos aplicar a Definição 3.6.1 (Indução, item 1) nas expressões $\underline{0(10)^*}$ e $\underline{1(01)^*}$ para concluir que $\underline{0(10)^* + 1(01)^*}$ é uma expressão regular válida representando a linguagem $\{0, 010, 01010, 0101010, ...\} \cup \{1, 101, 10101, 1010101, ...\}$.

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

Exercício 3.29 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 3.30 Seja $\Sigma = \{a, b\}$. Apresente a expressão regular para a linguagem das strings sobre Σ que tem tamanho ímpar. ■

Algo importante a ser mencionado é que, embora seja verdade que dada uma expressão regular, existe exatamente uma linguagem que corresponde a tal expressão, o oposto não é verdade. Isto é, dada uma linguagem L, não é verdade que existe exatamente uma expressão regular para L. De fato, dada uma linguagem L pode existir mais de uma expressão regular R tal que $L(R) = L$ (ou mesmo pode não existir nenhuma expressão regular para L). A exata relação entre expressões regulares e linguagens é estabelecida pelo seguinte teorema, que é central em teoria de linguagens regulares:

Teorema 3.6.1 Uma linguagem L é regular se e somente se existe uma expressão regular R tal que $L = L(R)$.

Para demonstrar o Teorema 3.6.1, precisamos demonstrar que as afirmações (1) e (2) abaixo são verdadeiras:

(1) Se L é uma linguagem regular, então existe uma expressão regular R tal que $L = L(R)$.
(2) Se R é uma expressão regular qualquer, então $L(R)$ é uma linguagem regular.

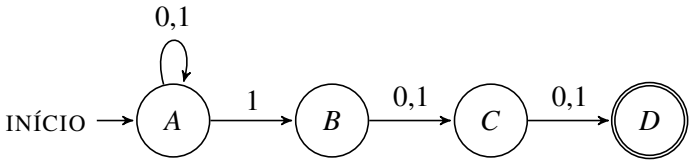
Apresentamos as provas das Afirmações (1) e (2) nas Seções 3.6.1 e 3.6.2, repectivamente.

3.6.1 Expressões regulares para linguagens de autômatos

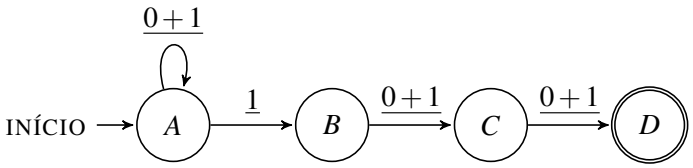
Nesta seção vamos demonstrar a seguinte afirmação: *se L é uma linguagem regular, então existe uma expressão regular R tal que $L = L(R)$*. Por definição, uma linguagem regular é uma linguagem aceita por um AFD. Entretanto, já vimos que AFDs e ε-AFNs aceitam exatamente o mesmo conjunto de linguagens (ver Teorema 3.5.2) e, portanto, para demonstrarmos a afirmação acima, basta que provemos o seguinte teorema:

Teorema 3.6.2 Seja $L(E)$ a linguagem de um $\varepsilon-$AFN E qualquer. Então existe uma expressão regular R tal que $L(R) = L(E)$.

Para provar o teorema acima, o que faremos é apresentar um algoritmo que, dado como entrada um $\varepsilon-$AFN E qualquer é capaz de produzir como saída uma expressão regular R tal que $L(R) = L(E)$. Para que possamos entender como funciona tal algoritmo, vamos, incialmente, considerar um caso simples. Seja E o autômato não determinístico abaixo:

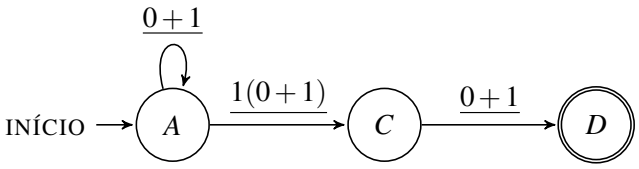


A partir de E, vamos apresentar um *autômato generalizado* E_0. Um autômato generalizado é uma variação dos autômatos que já conhecemos, mas que tem expressões regulares ao invés símbolos em suas transições. O autômato generalizado E_0 é apresentado abaixo:



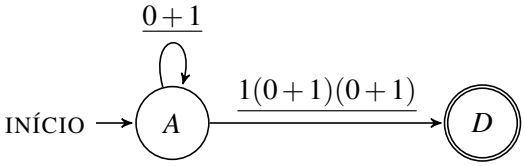
Observe que agora temos expressões regulares como rótulos das transições. Por exemplo, observe o conjunto de símbolos $\{0,1\}$ que figuravam como rótulo da transição do estado C para o estado D foram trocados pela expressão regular $\underline{0+1}$. A ideia é que a expressão regular corresponda às possíveis strings que façam o autômato sair C e ir para D (no caso as strings 0 e 1, ambas de tamanho 1). De maneira mais precisa, observe que $L(\underline{0+1}) = \{0,1\}$.

A ideia chave agora é produzir sequência de autômatos generalizados $E_0, E_1, E_2, \ldots$ de forma que todos sejam equivalentes, mas que tenham cada vez menos estados. Por exemplo, considere o autômato generalizado E_1 abaixo:



Observe que obtemos E_1 a partir de E_0 da seguinte forma: removemos o estado B e todas as transições que envolviam este estado (ou seja, Ⓐ $\to$ Ⓑ $A \to B$ e Ⓑ $\to$ Ⓒ) e incluímos a transição Ⓑ $\to$ Ⓒ com uma nova expressão como rótulo: $\underline{1(0+1)}$. A ideia é que a expressão $\underline{1(0+1)}$ é a concatenação das expressões $\underline{1}$ e $\underline{(0+1)}$. A ideia fundamental é que o conjunto de strings que faziam E_1 sair de A e chegar em C é representado pela expressão $\underline{1(0+1)}$, que agora figura no rótulo de Ⓑ $\to$ Ⓒ.

Agora o próximo passo é remover o estado C de E_1 e obter o autômato generalizado E_2 abaixo:

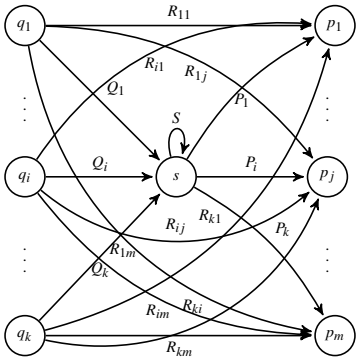


No caso do autômato acima, fica claro que as strings aceitas, isto é, as que fazem o autômato sair de A e atingir D são da forma xy, onde $x \in \{0,1\}^*$ e $y \in L(1(0+1)(0+1))$. A ideia é que o autômato processe x fazendo transições do tipo Ⓐ → Ⓐ e, quando faltarem os três últimos símbolos da string de entrada, use a transição Ⓐ → Ⓑ para processar a string y restante.

Simplificação de autômatos: caso geral

No exemplo visto anteriormente, as atualizações que foram feitas para obter um autômato generalizado E_{i+1} a partir de um autômato E_i são razoavelmente simples. O caso geral pode ser mais complicado. Ao removermos um estado s de E_i, precisamos observar todo par de estados (q_i, p_j) tal que ambas $p_i \to s$ e $s \to p_j$ sejam transições para poder obter autômato resultante E_{i+1}.

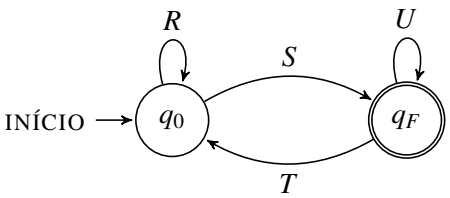
De acordo com a figura abaixo, suponha que os rótulos das transições $p_i \to s$ e $s \to p_j$ no autômato E_i são, respectivamente, Q_1 e P_1. Suponha também que o rótulo da transição $p_i \to q_j$ seja R_{ij} e da transição $s \to s$ seja S (lembrando que se alguma transição não existe no autômato, podemos pensar que ela tem rótulo $\varnothing$). Com isso, ao remover o estado s o rótulo da transição de $p_i \to q_j$ no autômato E_{i+1} passa a ser $R_{ij} + Q_i S^* P_j$.



Usando a ideia acima para fazer as atualizações das transições, vamos ver o algoritmo. Entretanto, precisamos antes tomar cuidado como o estado inicial e os estados finais do autômato. Vamos começar apresentando o algoritmo para o caso que o autômato em questão tenha apenas um estado final. Vamos considerar dois casos: no primeiro caso, o estado inicial não é o estado final e no segundo caso o estado inicial e o estado final coincidem.

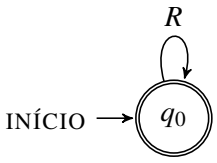
Caso 1: Algoritmo para obter ER a partir de um ε-AFN $(Q, \Sigma, \delta, q_0, \{q_F\})$, com $q_0 \neq q_F$:

1. Troque os rótulos das transições de E por expressões regulares equivalentes de forma que agora E seja um autômato generalizado
2. A cada passo remova um estado $q \in Q \setminus \{q_0, q_F\}$ e atualize as transições e as espressões regulares dos rótulos do autômato
3. Quando restar apenas o estado inicial q_0 e o estado final q_F, como o da figura abaixo, a expressão regular final é $(R + SU^*T)^*SU^*$



Caso 2: Algoritmo para obter ER a partir de ε-AFN (Q,Σ,δ,q_0,F) com $|F|=1$ e $q_0 \in F$:

1. Troque os rótulos das transições de E por expressões regulares equivalentes de forma que agora E seja um autômato generalizado
2. A cada passo remova um estado $q \in Q \setminus \{q_0\}$ e atualize as transições e as espressões regulares dos rótulos do autômato
3. Quando restar apenas o estado inicial (e final) q_0, como o da figura abaixo, a expressão regular final é R^*.



A ideia agora é usar o algoritmos vistos para os dois casos anteriores e apresentar um algoritmo que funciona para qualquer autômato.

Caso geral: Algoritmo para obter ER a partir de ε-AFN $E=(Q,\Sigma,\delta,q_0,F)$ qualquer

Suponha que $F=\{f_1,f_2,...,f_k\}$. Agora tome uma string $w \in L(E)$ e seja f_i o estado atingido pelo autômato quando w é aceita, tomando i o menor possível[2]. A ideia chave é que o autômato $E_i=(Q,\Sigma,\delta,q_0,\{q_i\})$ tem a seguinte propriedade:

(1) E_i também aceita a string w;

(2) E_i não aceita nenhuma string que não seja aceita por E.

Usando essa ideia, o algoritmo para obter a expressão regular de E é o seguinte:

- A partir de E, crie k autômatos $E_1,E_2,...,E_k$, onde E_i é o o autômato $(Q,\Sigma,\delta,q_0,\{q_i\})$
- Encontre as expressões regulares $R_1,R_2,...,R_k$ para cada autômato E_1 usando os algoritmos para os casos especiais de autômatos com apenas um estado final
- A partir daí, e ER para E é $R_1+R_2+...+R_k$

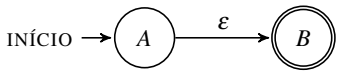
3.6.2 Construindo autômatos para linguagens de expressões regulares

Nesta seção, nosso objetivo é demonstrar a seguinte afirmação: *Se R é uma expressão regular qualquer, então $L(R)$ é uma linguagem regular.* Isto é enunciado de maneira precisa abaixo:

Teorema 3.6.3 Seja $L(R)$ é a linguagem representada por uma expressão regular R qualquer. Então existe um $\varepsilon-$AFD E tal que $L(E)=L(R)$.

Para provarmos o teorema, vamos mostar que o ε-AFN com alfabeto Σ que aceita a linguagem de R, de mesmo alfabeto, pode ser construído recursivamente a partir da própria definição recursiva para as expressões regulares (Definição 3.6.1).

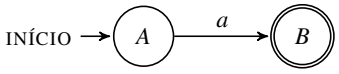
- Se R é uma expressão regular elementar, então ε-AFD tem uma das três formas abaixo:



Se $R=\underline{\varepsilon}$, então o autômato acima aceita a linguagem $L(\underline{\varepsilon})=\{\varepsilon\}$.

[2]Note que como o autômato não é determinístico, podem haver vários estados de aceitação possível, mas precisamos de apenas um deles e, arbitrariamente, tomamos o estado f_i para o menor índice i possível.

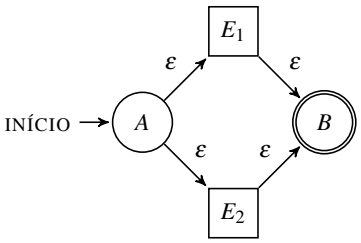
Se $R = \underline{\emptyset}$, então o autômato acima aceita a linguagem $L(\underline{\emptyset}) = \emptyset$.



Se $R = \underline{a}$, para algum $a \in \Sigma$, então o autômato acima aceita $L(\underline{a}) = \{a\}$.

- Se R é uma expressão regular composta, então $\varepsilon-$AFD pode ser construído recursivamente usando a seguinte estratégia:

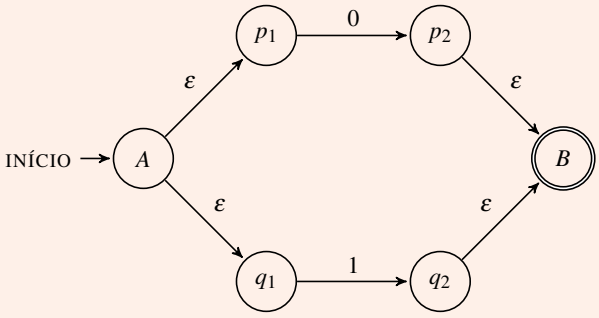
Caso 1. Suponha que e a expressão regular tem a forma composta $R = R_1 + R_2$. Suponha por indução que os $\varepsilon-$AFDs E_1 e E_2 aceitam, respectivamente, $L(R_1)$ e $L(R_2)$. O diagrama abaixo descreve a forma para o $\varepsilon-$AFD E que aceita a expressão composta R:



Observe o seguinte na construção acima: Os nós E_1 e E_2 representam autômatos para as linguagens $L(R_1)$ e $L(R_2)$. A transição $A \to R_1$ deve ser entendida como saindo do estado A e tendo destino o estado inicial do autômato E_1 e a transição $E_1 \to B$ deve ser entendida como saindo do estado final de E_1 e tendo como destino B. A mesma ideia vale para as transições $A \to R_2$ e $E_2 \to B$. Note também que nesta construção os estados que originalmente eram iniciais em E_1 e E_2 não são mais iniciais no autômato composto resultante.

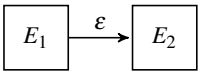
Exercício 3.31 (Resolvido) Apresente um autômato para aceitar a linguagem da expressão regular $R = \underline{0+1}$ sabendo que os dois autômatos E_1 e E_2 abaixo aceitam as linguagens das expressões $R_1 = \underline{0}$ e $R_2 = \underline{1}$, respectivamente:

Solução: Usando a construção para o Caso 1, obtemos abaixo o autômato E que aceita a linguagem da expressão $\underline{0+1}$:



■

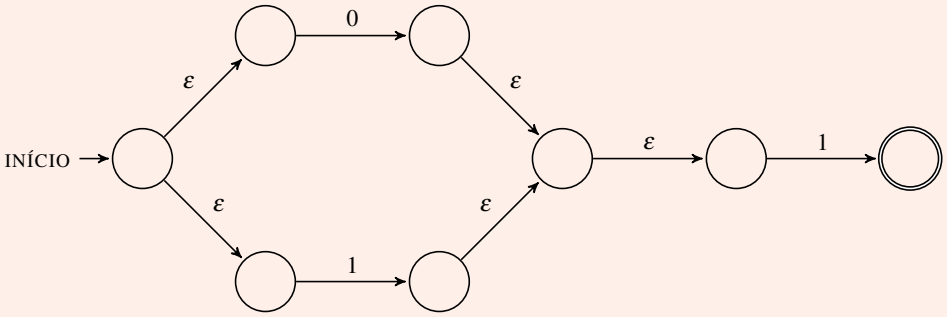
Caso 2. Suponha que e a expressão regular tem a forma composta $R = R_1R_2$. Suponha por indução que os ε-AFDs E_1 e E_2 aceitam, respectivamente, $L(R_1)$ e $L(R_2)$. O diagrama abaixo descreve a forma para o ε-AFN E que aceita a expressão composta R:



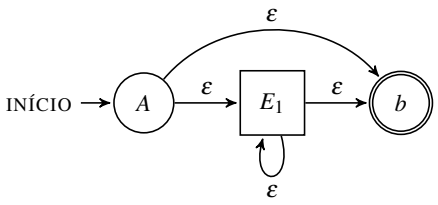
Observe o seguinte no esquema da construção acima: Os nós E_1 e E_2 representam autômatos para as linguagens $L(R_1)$ e $L(R_2)$. A transição $E_1 \to E_2$ deve ser entendida como saindo do final do autômato E_1 e tendo destino o estado inicial do autômato E_2. Note também que nesta construção o estado que originalmente era inicial em E_2 não é mais inicial no autômato composto resultante.

Exercício 3.32 (Resolvido) Apresente um autômato para aceitar a linguagem de $R = \underline{(0+1)1}$.

Solução: Sabendo que os autômatos E e E_1 apresentados no Exercício Resolvido 3.31 aceitam as linguagens das expressões $R_1 = \underline{(0+1)}$ e $R_2 = \underline{1}$ obtemos o autômato E para R:



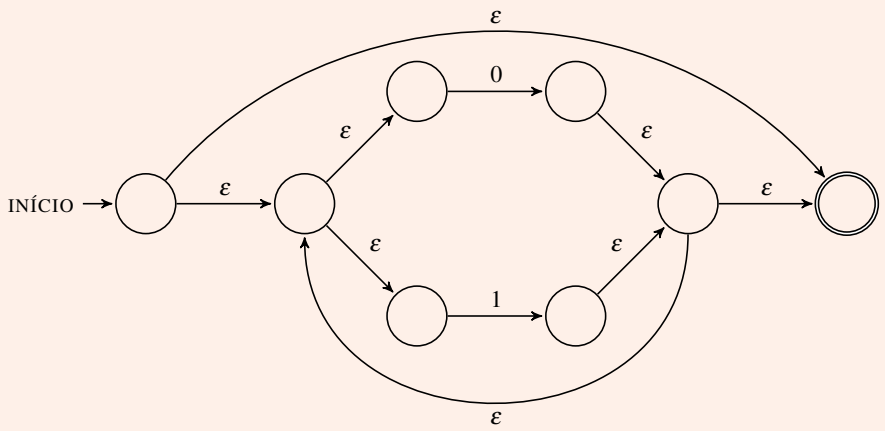
Caso 3. Suponha que e a expressão regular tem a forma composta $R = R_1^*$. Suponha por indução que o ε-AFN E_1 aceita $L(R_1)$. O diagrama abaixo descreve a forma para o ε-AFN E que aceita a expressão composta R:



Observe o seguinte no esquema da construção acima: O nó E_1 representa o autômato para a linguagem $L(R_1)$. A transição $E_1 \to E_1$ deve ser entendida como saindo do final do autômato E_1 e tendo destino o estado inicial do autômato E_1. A transição $A \to E_1$ sai de A e chega no estado inicial de E_1 e a transição $E_1 \to B$ sai do estado final de E_1 e chega em B. Note também que nesta construção o estado que originalmente era inicial em E_1 não é mais inicial no autômato composto resultante.

Exercício 3.33 (Resolvido) Apresente um autômato para aceitar a linguagem de $R = \underline{(0+1)^*}$.

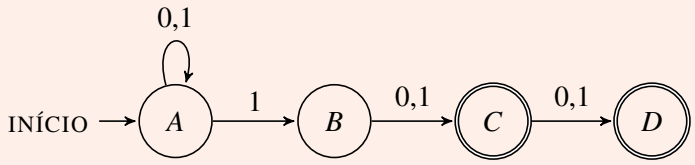
Solução: Sabendo que o autômato E apresentado na solução do Exercício Resolvido 3.31 aceita a linguagem da expressão $\underline{(0+1)}$, obtemos o seguinte autômato composto para R:



■

Exercícios

Exercício 3.34 Obtenha uma expressão regular que represente a linguagem do seguinte ε-AFN:



■

Exercício 3.35 Obtenha um ε-AFN cuja linguagem seja a mesma representada pela expressão regular $\underline{(0+1)^*1(0+1)}$. ■

Exercício 3.36 Considere os alfabetos $\Sigma_1 = \{a,b,c\}$ e $\Sigma_2 = \{0,1\}$. Forneça expressões regulares para as seguintes linguagens:

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

■

Exercício 3.37 Apresente uma expressão regular para strings binárias cujo tamanho é um múltiplo de 3 e a quantidade de símbolos 1 da string é ímpar. ■

Exercício 3.38 Dado o AFD abaixo, encontre uma expressão regular equivalente. Você deve

mostrar o desenvolvimento passo a passo de solução.

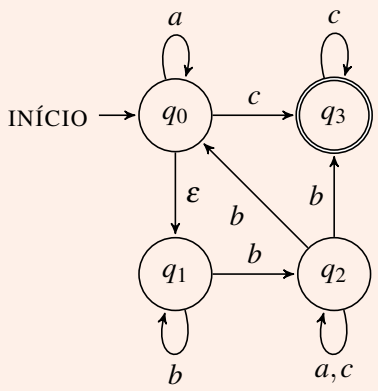
	0	1
$\rightarrow^* p$	s	p
q	p	s
r	r	q
s	q	r

■

Exercício 3.39 Forneça um ε-AFN que aceite a mesma linguagem da expressão regular $(00+11)^*+(111)^*0$. ■

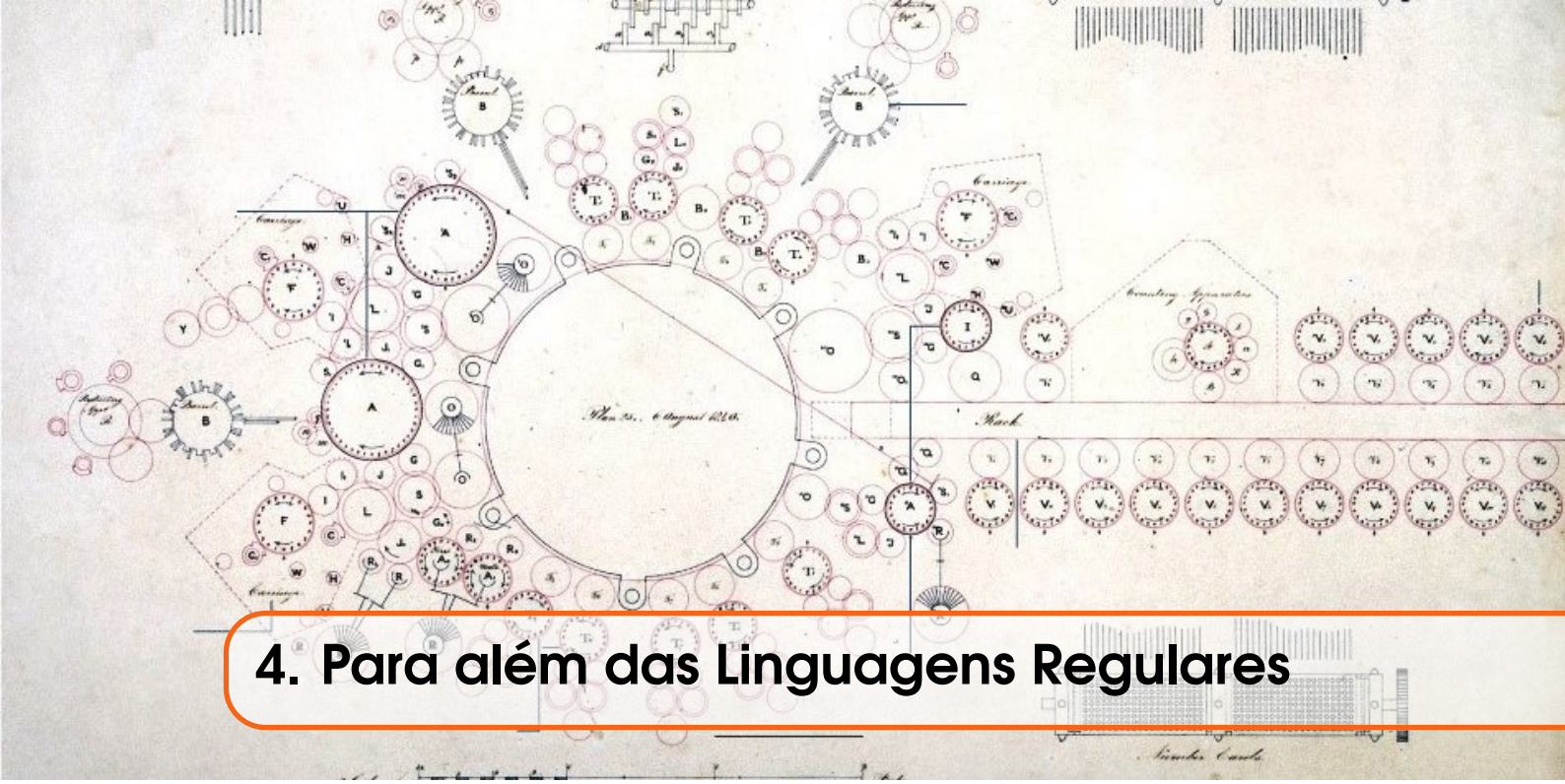
Exercício 3.40 Construa um ε-AFN que aceite a mesma linguagem da expressão regular $00(0+1)^*$. ■

Exercício 3.41 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. ■



4. Para além das Linguagens Regulares

Existem linguagens não regulares? Nas seções anteriores nós já antecipamos que a resposta é sim. Veremos neste capítulo que algumas linguagens extremamente simples como, por exemplo, a linguagem das strings binárias palíndromas não são regulares.

O ponto mais importante a ser apreendido aqui é o seguinte: Se, por um lado, mostrar que uma dada linguagem é regular é simples – basta explicitamente apresentarmos um AFD a aceite – por outro lado, para mostrarmos que uma dada linguagem 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.

Na Seção 4.1 vamos apresentar uma ferramenta matemática, conhecida como *Lema do Bombeamento*, que será extremamente útil para provar a inexistência de autômatos para aceitar certas linguagens.

4.1 O Lema do Bombeamento para Linguagens Regulares

O seguinte lema estabelece uma propriedade importante de strings de linguagens regulares:

Lema 4.1.1 — Lema do Bombeamento para Linguagens Regulares (LBLR). Seja L uma linguagem regular. Então existe uma constante inteira $t \geq 1$ tal que para toda string $w \in L$ satisfazendo $|w| \geq t$, existem strings $x, y, z \in \Sigma^*$ satizfazendo o seguinte:

$w = xyz$ e as condições abaixo são satisfeitas:

(1) $y \neq \varepsilon$ (2) $|xy| \leq t$ (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 para Linguagens Regulares (LBLR) ou, de maneira mais curta, Lema do Bombeamento (LB). Na Seção 4.1.1, veremos como fazer uso do LB para mostrar que uma dada linguagem não é regular. Antes disso, vamos ver a demonstração do lema.

Prova do Lema 4.1.1: Como L é regular, existe um AFD D que aceita L. Seja Q o conjunto de estados do autômato e seja $t = |Q|$.

Considere uma string $w \in L$ tal que $|w| \geq t$ fornecida como entrada para o autômato. A computação começa no estado inicial e, a cada passo, o autômato visita um dos estados de Q, finalizando em um estado de aceitação do autômato. Observe que após o autômato ter consumido t símbolos de w, a quantidade de transições executadas é t e, portanto, o número de estados visitados é $t+1$ estados (lembrando que o estado inicial deve ser contado). Como $t+1 > |Q|$, o autômato terá visitado pelo menos um estado duas vezes. Considere a sequência S dos estados visitados

$$S = \underbrace{q_0, ..., q_{i-1}}_{S_1}, \underbrace{q_i, ..., q_j}_{S_2}, \underbrace{q_{j+1}, ..., q_t}_{S_3}$$

de forma que q_i e q_j correspondem a primeira ocorrência de repetição de estados em S. Ou seja, sendo $S = S_1, S_2, S_3$, conforme o esquema acima, na subsequência S_1 aparecem apenas estados distintos e na subsequência S_2 os únicos estados iguais são q_i e q_j. Nesta sequência q_0 é o estado inicial e q_t deve ser um estado final do autômato.

Agora faça $w = xyz$ tal que

- x é o prefixo de w até o i-ésimo símbolo (i.e., x faz o autômato visitar os estados de S_1);
- y é a substring entre as posições $i+1$ e j (corresponde às visitas aos estados de S_2);
- z é o sufixo restante.

Vamos mostrar agora que as condições *(1)*, *(2)* e *(3)* do enunciado do lema são satisfeitas. Como $i < j$, então $y \neq \varepsilon$ e, portanto, a condição *(1)* é satisfeita. Como $j \leq t$, então $|xy| \leq t$ e, portanto, a condição *(2)* é satisfeita.

Agora considere a string xy^kz, para um k qualquer. Seja S' a sequência de estados visitados pelo autômato na computação de xy^kz. Como $q_i = q_j$, a sequência de estados nesta computação é

$$S' = S_1, \underbrace{S_2, S_2, ..., S_2}_{\text{repetido } k \text{ vezes}}, S_3 \ .$$

Como o último estado da sequência S' é q_t, um estado de aceitação, então xy^kz é aceita pelo autômato. Portanto a condição *(3)*, descrita abaixo, também é satisfeita:

(3) $\forall k \geq 0$, $xy^kz \in L$.

Com isso, a prova do lema está finalizada. □

4.1.1 Usando o Lema do Bombeamento

Nesta seção vamos entender como podemos usar o LB como ferramenta para demonstrarmos que uma dada liguagem 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 inexistêcia do AFD fica "encapsulada" dentro da demonstração do LB.

A seguir vamos ao nosso primeiro exemplo de demonstração que uma linguagem não é regular:

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

Prova: Suponha que L_{01} é regular. Portanto, usando o LB, sabemos que existe $t \in \mathbb{N}$, tal que se tomaramos uma string w de L de tamanho maior ou igual a t, deve existir $x, y, z \in \Sigma^*$ tal que w pode ser escrita como $w = xyz$ e as três afirmações a seguir são verdadeiras:

(1) $y \neq \varepsilon$ *(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. Logo, todos os símbolos 1 da string w estão em z (embora não **necessariamente** z contenha apenas símbolos 1).

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. □

Vamos utilizar agora o LB em uma linguagem um pouco mais interessante, que é a linguagem dos números primos escritos em unário:

Teorema 4.1.3 A linguagem $L_{uP} = \{1^p : p \text{ é um número primo }\}$ não é regular.

Prova: Suponha que L_{uP} é regular. Então o LB nos diz que existe $t \in \mathbb{N}$, tal que se tomaramos 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) $y \neq \varepsilon$ *(2)* $|xy| \leq t$ *(3)* $\forall k \geq 0, xy^k z \in L_{uP}$.

Considere a string $w = 1^p$ para algum primo $p \geq t$ (note que deve existir um primo $p > t$, pois existem infinitos primos). Note que w é uma string para a qual podemos aplicar o LB, pois $w \in L_{uP}$ 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 $xy^{p+1} z \in L_{uP}$.

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

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

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 $xy^{p+1} z \in L_{uP}$. □

O Teorema 4.1.3 mostra que o problema de reconhecer se um determinado número é primo não pode ser resolvido por um algoritmo (ou uma máquina) cujo funcionamento possa ser descrito por um autômato finito. Entretanto, observe que, nesse resultado, estamos supondo que os números sejam representados em unário. Também é possível provar um resultado mais "natural", no qual os números primos são representados por strings binárias, conforme o teorema a seguir. A prova desse teorema é bastante sofisticada e é apresentada na sequência como MATERIAL OPCIONAL.

Teorema 4.1.4 A linguagem $L_P = \{w : N(w) \text{ é um número primo}\}$ não é regular.

MATERIAL OPCIONAL: PROVA DO TEOREMA 4.1.4.

Suponha que $L_P = \{w : N(w) \text{ é primo}\}$ seja regular e seja t a constante dada pela generalização do Lema do Bombeamento apresentada no Exercício 4.2. Pelo *Teorema de Dirichlet*[a] sobre progressões aritméticas, existem infinitos primos da forma $n2^{t+1}+1$. Escolha um desses primos e denote-o por p. A representação binária de p termina em 10^t1, isto é, termina com um símbolo 1, seguido de exatamente t zeros, seguido de outro símbolo 1.

Agora considere a string $w = u0^t v$, onde o bloco 0^t corresponde exatamente aos t zeros consecutivos. Como o bloco possui comprimento t, a versão mais geral do Lema do Bombeamento garante a existência de uma decomposição $0^t = xyz$, em que $y = 0^s$, para algum inteiro $s > 0$, e tal que $uxy^kzv \in L_P$, para todo $k \geq 0$. Em outras palavras, aumentar o número de zeros nesse bloco produz sempre outro número primo.

Seja $q_k = N(uxy^kzv)$. Cada zero adicional inserido nesse bloco desloca o sufixo final uma posição para a direita. Logo, $q_k = (p-1)2^{(k-1)s}+1$. Como o lema afirma que todas essas palavras pertencem à linguagem, concluímos que q_k é primo para todo inteiro $k \geq 1$. Escolhemos agora $k = p$. Pelo *Pequeno Teorema de Fermat*[b], $2^{p-1} \equiv 1 \pmod{p}$. Elevando ambos os lados à potência s, $2^{(p-1)s} \equiv 1 \pmod{p}$. Como $k-1 = p-1$, segue que $2^{(k-1)s} \equiv 1 \pmod{p}$. Portanto, $q_k = (p-1)2^{(k-1)s}+1 \equiv -2^{(k-1)s}+1 \equiv 0 \pmod{p}$. Logo, p divide q_k. Além disso, $q_k > (p-1)+1 = p$, de modo que q_k possui um divisor próprio e, portanto, não é primo. Isso é uma contradição e, portanto, L_P não é regular.

[a] *Teorema de Dirichlet: Sejam $a,d \in \mathbb{N}$ com $\gcd(a,d)=1$. Então existem infinitos primos congruentes a a módulo d.*
[b] *Pequeno Teorema de Fermat: Seja p um primo. Então, $\forall a \in \mathbb{Z}$, se p não divide a, então $a^{p-1} \equiv 1 \pmod{p}$.*

Exercício 4.1 Prove que cada uma das linguagens a seguir não são regulares.

(a) A linguagem L_{WR} das strings binárias da forma ww^R.
(b) A linguagem L_{WW} das strings binárias da forma ww.
(c) A linguagem L_{EQ} das strings binárias que tem a mesma quantidade de 0's e 1's.
(d) A linguagem $L_{DB} = \{w : \text{o número de 0's em } w \text{ é o dobro do número de 1's}\}$.
(e) A linguagem $L_{+0} = \{0^i1^j \ : \ i > j\}$.
(f) A linguagem $L_{+1} = \{0^i1^j \ : \ i < j\}$.
(g) A linguagem $L_{SQ} = \{1^s : s \text{ é um quadrado perfeito}\}$. ■

Exercício 4.2 (Opcional) Prove que a seguinte versão mais geral do Lema do Bombeamento também é verdadeira:

Lema: Seja L uma linguagem regular. Então existe uma constante t tal que $\forall w \in L$, com $|w| \geq t$, o seguinte é verdadeiro:

$\exists u,x,y,z,v \in \Sigma^*$ tal que $w = xw'z = uxyzv$ e as três condições abaixo são satisfeitas:

(1) $y \neq \varepsilon$ (2) $|xy| \leq t$ (3) $\forall k \geq 0, ux(w')^kzv \in L$ ■

Exercício 4.3 Considere o alfabeto $\{0,1,M\}$ e a seguinte linguagem sobre este alfabeto: $L_{WMR} = \{wMw^R \ : \text{tal que } w \in \{0,1\}^*\}$. Use o Lema do Bombeamento para mostrar que L_{WMR} não é uma linguagem regular. ■

Exercício 4.4 Considere o alfabeto $\{0,1,M\}$ e a seguinte linguagem sobre este alfabeto: $L_{0M1} = \{0^iM1^i \ : \text{tal que } i \geq 1\}$. Use o Lema do Bombeamento para mostrar que L_{0M1} não é regular. ■

4.2 Autômatos com Pilha (AP)

No capítulo 3 nós apresentamos a definição de AFDs e definimos conjunto das linguagens regulares como sendo exatamente o conjunto de linguagens aceitas por AFDs. Depois disso, fomos incrementando as habilidades dos nossos autômatos. Primeiro incluímos a habilidade do autômato "advinhar" qual transição fazer e, em seguida, adicionamos a possibilidade do autômato fazer transições ε. Entretanto, o conjunto de linguagens aceitas por tais autômatos continua sendo as linguagens regulares.

Nesta seção vamos apresentar um cenário diferente: Vamos dar aos ε-AFNs uma habilidade que os tornarão mais poderosos, isto é, os tornarão capazes de aceitar linguagens que não são regulares. A habilidade que vamos dar ao autômato é a possibilidade de armazenar informação e recuperar informação em uma memória. Entretanto, o tipo de memória que o autômato terá é bastante restrito: uma pilha de dados (veja Figura 4.1). O ponto chave aqui é que, embora o autômato tenha acesso a um tipo de memória bastante restrito, este fato será suficiente para aumentar o seu poder de computação.

Antes de qualquer coisa, é importante observar que este novo modelo é capaz de realizar no mínimo as tarefas que um ε-AFN já realizava, ou seja, reconhecer linguagens regulares, pois podemos fazer computação com o autômato simplesmente ignorando a pilha de dados. Obviamente, já que autômato tem a pilha de dados, o mais interessante será fazer uso dela. Na seção seguinte veremos exatamente como isso vai funcionar.

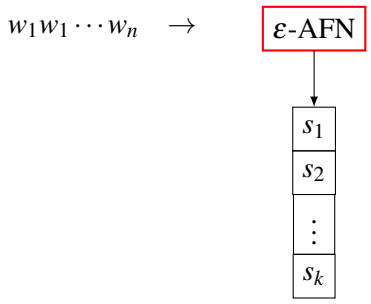


Figura 4.1: Esquema conceitual de autômatos com pilha.

4.2.1 O modelo matemático para autômatos com pilha

Nesta seção apresentaremos a definição matemática para um autômato com pilha. Para tal, precisamos esclarecer antes como este autômato funciona. Note primeiamente que durante a computação, um ε-AFN executa duas ações em cada um de seus passos:

(1) Processa um ou zero símbolos da string de entrada;
(2) Faz uma transição de estado.

Relembre que a quantidade de símbolos processados na ação (1) depende da transição efetuada (um símbolo nas transições comuns e zero símbolos nas transições ε). Quanto a ação (2), relembre também que a transição pode ser fazer com o autômato fique no mesmo estado que já se encontra. Autômatos com pilha, por sua vez, executam quatro ações a cada passo:

(1) Consome um ou zero símbolos da string de entrada;
(2) Desempilha o símbolo do topo da pilha;
(3) Faz uma transição de estado;
(4) Empilha uma quantidade finita de símbolos na pilha.

No caso dos ε-AFNs, as ações executadas a cada passo da dependem de um par (*estado atual*, *símbolo consumido*) e por isso, tem o domínio da função de transição igual a $Q \times \Sigma \cup \{\varepsilon\}$. No caso dos APs[1], as ações executadas a cada passo da computação do AP é determinado por uma tripla com a seguinte forma: (*estado atual, símbolo consumido, símbolo desempilhado*). Por este motivo, o domínio da função de transição δ do autômato com pilha deve ser $Q \times \Sigma \cup \{\varepsilon\} \times \Gamma$. A cada passo, um autômato com pilha, além de mudar de estado, também empilha símbolos na pilha de dados (formalmente, empilham uma string de símbolos). Mais precisamente, como estamos lidando com um modelo de computação não determinístico, dado (*estado, símbolo da entrada, símbolo da pilha*) a função δ devolve um conjunto de possíveis ações a serem executadas pelo autômato, onde cada elemento é do tipo (*novo estado, string empilhada*).

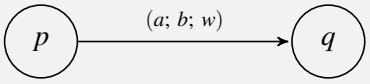
Definição 4.2.1 — Autômatos com Pilha (AP). Um autômato com pilha P é uma 7-tupla $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ tal que:

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 da pilha*, de forma que $Z_0 \notin \Sigma$.
$\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \to \{(q_1, \gamma_1), (q_2, \gamma_2), ..., (q_k, \gamma_k)\}$, tal que $q_i \in Q$ e $\gamma_i \in \Gamma^*$.

Na definição acima, o símbolo Z_0 é, por convenção, o único símbolo presente na pilha no início da computação, marcando o "fundo" da pilha. Assim como fazemos com os demais autômatos vistos anteriormente, vamos usar diagramas para representar APs. Nos diagramas, o rótulo de uma transição do tipo $p \to q$ contém, além do símbolo de Σ processado na transição, o símbolo de Γ e a string que deve ser empilhada quando a transição é efetuada.

RÓTULOS NOS DIAGRAMAS DE AUTÔMATOS COM PILHA

Os rótulos nas transições de um diagrama de um AP tem a forma $(a; b; w)$, como abaixo:



Neste caso, a e b são símbolos e w uma string. Neste caso, a transição é executada se o próximo símbolo da string de entrada for a e se o topo da pilha for b. Após a execução da transição empilha-se a string w. A sequência da símbolos de w é empilhada de trás para frente. Por exemplo, se $w = w_1 w_2 ... w_k$, o símbolo w_k é empilhado por primeiro, depois o símbolo w_{k-1}, e assim por diante, até que o símbolo w_1 é empilhado e fica no topo da pilha do autômato.

A seguir vamos apresentar um diagrama de um AP para a linguagem L_{RR} das strings binárias da forma ww^R, ou seja, palíndromos de tamanho par. Observamos que esta linguagem não é regular (a demonstração disto é o objetivo do Exercício 4.1). O AP P_{RR} da Figura 4.2 decide L_{RR}.

A 7-tupla que define o autômato é $P_{\text{RR}} = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$, onde a função δ é definida da seguinte forma: Para todo $a \in \{0,1\}$ e para todo $B \in \{0,1,Z_0\}$, a função está definida nos seguintes casos:

- $\delta(q_0, a, B) = \{(q_0, aB)\}$;
- $\delta(q_0, \varepsilon, B) = \{(q_1, B)\}$;
- $\delta(q_1, a, a) = \{(q_1, \varepsilon)\}$;
- $\delta(q_1, \varepsilon, Z_0) = \{(q_2, Z_0)\}$.

O funcionamento do autômato é o seguinte: A primeira metade da string de entrada é processada

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

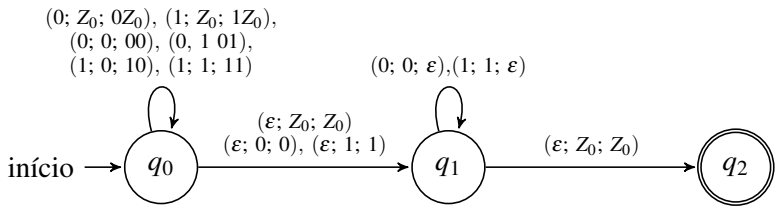


Figura 4.2: Diagrama do autômato com pilha para a linguagem dos palíndromos de tamanho par.

executando-se a transição do tipo $(q_0) \to (q_0)$. Observe que esta transição está definida para todas as possibilidades de símbolos a da entrada e B no topo da pilha (no Diagrama, trata-se do "laço" sobre o estado q_0). Quando o autômato chega na metade da string, sendo não determinístico, ele "advinha" que deve fazer a transição ε e mudar para o estado q_1. Depois disso, o autômato processa a segunda metade da string fazendo as trasições do tipo $(q_1) \to (q_1)$ desde que o símbolo lido na entrada seja idêntico ao símbolo no topo da pilha. Caso a string de entrada seja um palíndromo de tamanho par, isso funciona até que todo símbolo da segunda metade da string de entrada seja consumido e pilha seja toda esvaziada, exceto pelo símbolo especial Z_0, que marca o fundo da pilha. Para finalizar a computação, o autômato faz a transição do tipo $(q_0) \to (q_1)$.

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 um símbolo da pilha por padrão. Mas o que acontece quando algum AP tenta fazer uma trasição, mas 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 e a string é rejeitada.

Como já observamos, o modelo da Definição 4.2.1 não é determinítico, pois ele é uma generalização de um modelo que já não era determinístico. Uma pergunta importante é se este não determinismo faz diferença, ou seja, se seria possível apresentar uma versão determinística de um AP que aceite as mesmas linguagens que APs não determinísticos. Na Seção 4.2.4 daremos uma definição para APs determinísticos e veremos que estes **não** aceitam as mesmas linguagens aceitas por APs não determinísticos, embora ainda aceitem um conjunto maior de linguagens do que as linguagens regulares.

Exercício 4.5 Apresente a definição formal e também o diagrama de um autômato com pilha que decida a linguagem $L_3 = \{w : N(w) \text{ é um múltiplo de } 3\}$. ■

Exercício 4.6 Mostre que toda linguagem regular é aceita por um autômato com pilha. ■

Exercício 4.7 Apresente a definição formal e o diagrmaa de um AP que aceite a linguagem $L = \{0^n 1^n \ : \ n \geq 1\}$. ■

Exercício 4.8 Apresente o diagrama de um AP que aceite a linguagem $L = \{0^i 1^j \ : \ i > j\}$. ■

Exercício 4.9 Considere o alfabeto Σ que contém os símbolos de "abertura" e "fechamento" de parêntesis, ou seja $\Sigma = \{(,)\}$. Apresente o diagrama de um AP que aceite a linguagem das strings de parêntesis balanceados sobre σ. Por exemplo, a string $(()(()))$ deve ser aceita, enquanto a string $())($ não deve ser aceita. ■

4.2.2 Definição formal de computação com autômatos com pilha

A computação de um autômato com pilha a computação começa com o autômato no estado inicial q_0 com a string de entrada ainda sem ter sido processada e com a pilha do autômato contendo apenas o símbolo Z_0. A partir daí, a cada passo, a "situação" que a computação se encontra, será chamada de *configuração* da computação. A ideia é que a configuração da computação sempre depende do seguinte:

(1) O estado q em que o AP se encontra;
(2) A string w de símbolos ainda não processada pelo AP;
(3) A string γ de símbolos que está armazenada na pilha de dados.

A tripla (q, w, γ) que define uma configuração do autômato pode ser vista como uma "fotografia" do autômato, capturando a situação da computação naquele momento do tempo[2]. Formalizamos isso a seguir.

Definição 4.2.2 — Configuração de um AP. 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 uma configuração de P.

O símbolo $\vdash_P$, usado para representar um passo que leva a computação de certa configuração para uma outra configuração, é definido da seguinte maneira:

Definição 4.2.3 — Um passo computacional. Seja um AP $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$. Escrevemos $(q, aw, X\beta) \vdash_P (p, w, \alpha\beta)$ se $(p, \alpha) \in \delta(q, a, X)$, onde w é uma string qualquer de Σ^* e β é uma string qualquer de Γ^*.

■ **Exemplo 4.1** Sabemos que o AP P_{RR} da Figura 4.2 com a entrada 010010 no seu primeiro passo computacional sai da configuração $(q_0,\ 010010,\ Z_0)$ e atinge a configuração $(q_0,\ 10010,\ 0Z_0)$. Usando a notação da Definição 4.2.3, podemos escrever $(q_0,\ 010010,\ Z_0) \vdash_{P_{\text{RR}}} (q_0,\ 10010,\ 0Z_0)$. ■

Dadas configurações C_i e C_j de P, a expressão $C_i \vdash_P C_j$ é lida "C_i *produz* C_j com um passo computacional". A seguir vamos definir o símbolo $\vdash_P^*$ para generalizar esta ideia para um número arbitrário de passos:

Definição 4.2.4 — Sequência de passos computacionais. O símbolo $\vdash_P^*$, usado para representar uma sequência de passos que leva a computação de certa configuração C_i para uma outra configuração C_j é definido recursivamente:

Base: Se C_i é uma configuração de um AP, então $C_i \vdash_P^* C_i$.

Indução: $C_i \vdash_P^* C_j$ se $\exists C_k$ tal que $C_i \vdash_P C_k$ e $C_k \vdash_P^* C_j$.

A expressão $C_i \vdash_P^* C_j$ é lida "C_i *produz* C_j".

■ **Exemplo 4.2** Sabemos que o AP P_{RR} da Figura 4.2 aceita a string 010010, pois trata-se de um palíndromo de tamanho par. A computação passo a passo é a seguinte: $(q_0,\ 010010,\ Z_0) \vdash_{P_{\text{RR}}} (q_0,\ 10010,\ 0Z_0) \vdash_{P_{\text{RR}}} (q_0,\ 0010,\ 10Z_0) \vdash_{P_{\text{RR}}} (q_0,\ 010,\ 010Z_0) \vdash_{P_{\text{RR}}} (q_1,\ 010,\ 010Z_0) \vdash_{P_{\text{RR}}} (q_1,\ 10,\ 10Z_0) \vdash_{P_{\text{RR}}} (q_1,\ 0,\ 0Z_0) \vdash_{P_{\text{RR}}} (q_1,\ \varepsilon,\ Z_0) \vdash_{P_{\text{RR}}} (q_2,\ \varepsilon,\ Z_0)$. Usando a notação da Definição 4.2.4 podemos escrever: escrever $(q_0, 010010, Z_0) \vdash_{P_{\text{RR}}}^* (q_2, \varepsilon, Z_0)$. ■

[2] A tripla contendo estes três elementos também é chamada em alguns textos de *descrição instantânea* do autômato.

Vimos no Exemplo 4.2 que o $(q_0, 010010, Z_0) \vdash^*_{P_{\text{RR}}} (q_2, \varepsilon, Z_0)$, o que significa que P_{RR} aceita 010010. A ideia agora é usar a notação $\vdash^*$ para formalizar a ideia de aceitação de uma string w qualquer por um AP P:

Definição 4.2.5 — Aceitação e rejeição de strings. Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP. Dado $w \in \Sigma^*$, dizemos que P *aceita* w se $(q_0, w, Z_0) \vdash^*_P (q, \varepsilon, \alpha)$ para $q \in F$ e $\alpha \in \Gamma^*$ qualquer. Caso contrário, dizemos que P *rejeita* w.

Definição 4.2.6 — Árvore de computações possíveis de APs. Dado um AP P e uma string w. A árvore de computações possíveis de $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ com a string $w \in \Sigma^*$ é definida indutivamente:

Base: Inclua na árvore o nó raíz correspondente à configuração $C_0 = (q_0, w, Z_0)$.

Indução: Seja C_i um nó da árvore. Se $C_i \vdash_P C_j$, então inclua o nó C_j como filho de C_i na árvore.

O autômato P_{RR} aceita a string 1111. A seguir apresentamos a árvore de computações possíveis:

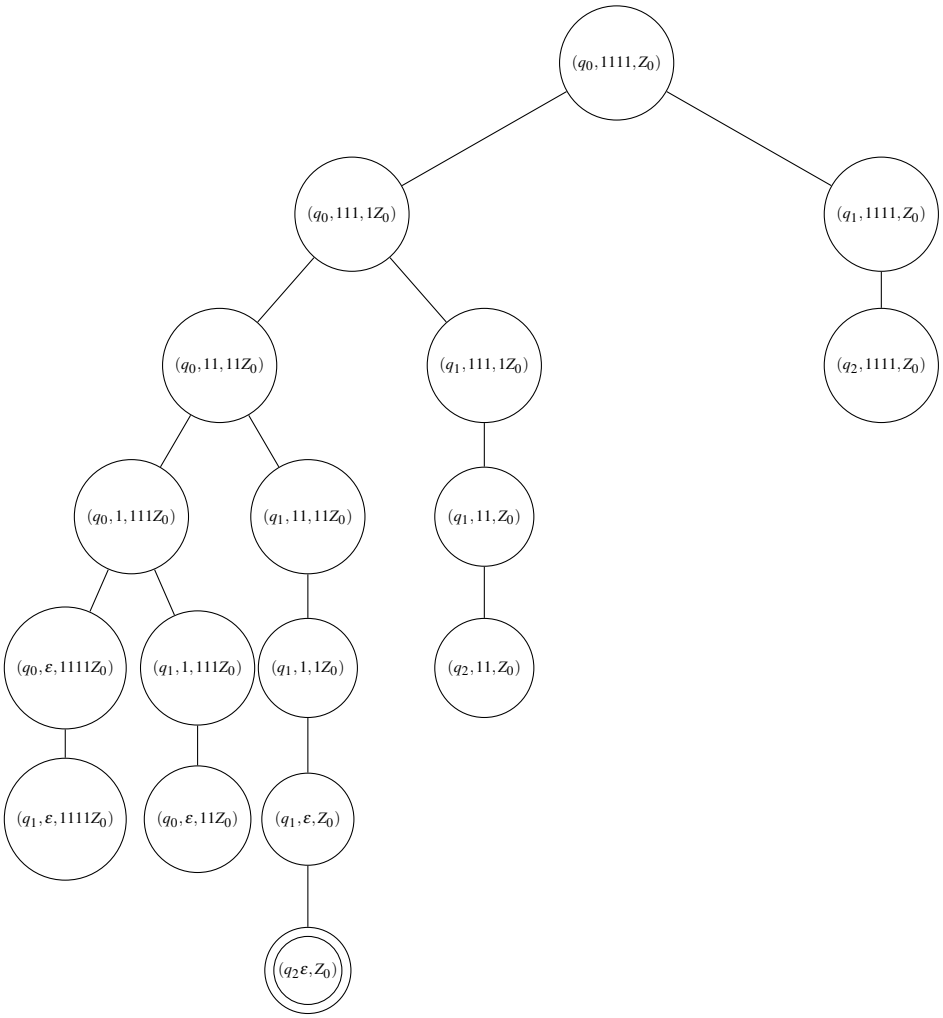


Figura 4.3: A árvore de computações possíveis do AP P_{RR} da Figura 4.2 com a string 1111. O nó marcado com dois círculos concêntricos é um nó que corresponde a uma configuração em que o AP aceita a string de entrada.

A linguagem de um AP é conjunto de strings que ele aceita, conforme a definição a seguir.

Definição 4.2.7 — Linguagens de Autômatos com pilha. Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ um AP. Definimos a linguagem de P como sendo $L(P) = \{w \in \Sigma^* : (q_0, w, Z_0) \vdash_P^* (q, \varepsilon, \alpha)$ para $q \in F$ e $\alpha \in \Gamma^*$ qualquer$\}$.

Definição 4.2.8 — Liguagem Livre de Contexto. Dada uma linguagem L, se existe um AP P tal que $L = L(P)$, então L é dita uma *linguagem livre de contexto*.

Exercícios:

Exercício 4.10 O conjunto das linguagens regulares está estritamente contido no conjunto das linguagens livres de contexto? Justifique sua resposta. ■

Exercício 4.11 Para responder as questões abaixo assuma que o alfabeto é $\Sigma = \{a, b, c\}$.
(a) Forneça um AP P tal que $L(P) = \{a^m b^n c^n : m, n \geq 0\}$.
(b) Forneça um AP P tal que $L(P) = \{a^n b^n c^m : m, n \geq 0\}$.
■

Exercício 4.12 (Opcional) Na Seção 4.1 aprendemos a usar o *Lema do Bombeamento para Linguagens Regulares* para provar que certas linguagens não são regulares. Para o caso de Linguagens Livre de Contexto também existe um *Lema do Bombeamento para Linguagens Livre de Contexto (LBLLC)* (veja os livros [HMU06] e [Sip06], por exemplo). Use o LBLLC para provar que $L = \{a^n b^n c^n; n \geq 0\}$ não é livre de contexto. ■

Exercício 4.13 A interseção de duas linguagens regulares também é uma liguagem 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 do Exercício 4.12 não é livre de contexto. ■

4.2.3 Strings que fazem o autômato esvaziar a pilha

Relembramos 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 trasição com a pilha vazia. O que vamos fazer agora é mudar completamente como entendemos a computação feita pelo AP. Vamos projetar APs de maneira que eles morram exatamente quando terminarem de ler a string de entrada.

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

■ **Exemplo 4.3** Como o AP P_{RR} da Figura 4.2 nunca esvazia a pilha, então $N(P_{RR}) = \varnothing$. ■

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 no sentido em que a máquina pode ser usada para fazer a distinção entre as strings que pertencem ou não pertencem a $N(P)$. A partir de agora, vamos formalizar esta ideia e dizer que as strings de $N(P)$ são as strings que P *esvazia a pilha*. Observe que o fato de que P aceite por pilha vazia uma dada string w, não implica necessariamente que P aceite esta mesma string w. Entretanto, os seguintes teoremas mostram há uma certa equivalência entre estas duas maneiras de se olhar para os autômatos com pilha.

Teorema 4.2.1 Seja P uma AP qualquer. Então existe um AP P' tal que $L(P') = N(P)$.

Teorema 4.2.2 Seja P uma AP qualquer. Então existe um AP P' tal que $N(P') = L(P)$.

4.2.4 Autômatos com pilha determinísticos (APDs)

Vamos finalizar esta seção sobre APs seguindo a direção oposta que vínhamos fazendo desde o Capítulo 3. Até agora vínhamos, pouco a pouco, *generalizando* o nosso modelo de computação. Agora, vamos dar "um passo para trás" e definir um modelo de computação um pouco mais restrito. Chamaremos este modelo de *Autômato com Pilha Determinístico (APD)*.

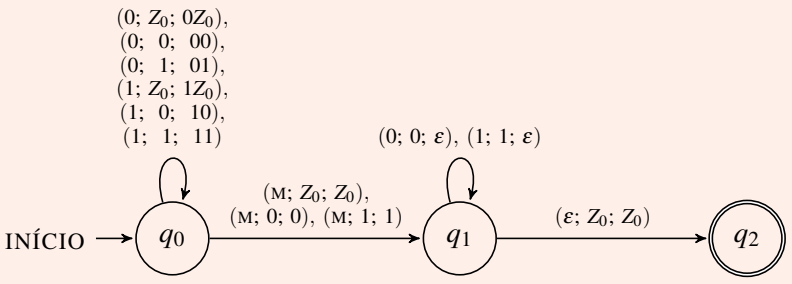
Definição 4.2.10 — Autômatos com Pilha Determinísticos (APDs). Um *Autômato com Pilha Determinístico (APD)* é um Autômato com Pilha $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ cuja função de transição δ tem as seguintes restrições:

1. Para quaisquer, $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$ e $X \in \Gamma$, temos $|\delta(q,a,X)| \leq 1$,
2. Se existe $q \in Q$, $a \in \Sigma$ e $X \in \Gamma$ tal que $\delta(q,a,X) \neq \varnothing$, então $\delta(q,\varepsilon,X) = \varnothing$.

Exercício 4.14 Mostre que toda linguagem regular pode ser decidida por um APD. ■

Exercício 4.15 (Resolvido) Considere a linguagem não regular do Exercício 4.4, isto é a linguagem $L_{\text{WMR}} = \{wMw^R \text{ : tal que } w \in \{0,1\}^* \}$ definida sobre o alfabeto $\{0,1,M\}$. Mostre um APD que decida esta linguagem.

Solução:



■

Teorema 4.2.3 O conjunto de linguagens decididas por APs é estritamente maior do que o conjunto das linguagens regulares.

Prova: A demonstração é consequência direta dos Exercícios 4.4, 4.14 e 4.15. □

Teorema 4.2.4 Toda linguagem decidida por um APD também é decidida por um AP.

Prova: Como APDs são casos particulares de APs, trivialmente toda linguagem decidida por um APD também é decidida por um AP. □

Uma pergunta que surge naturalmente agora é se toda linguagem decidida por AP também pode ser decidida por um APD. O teorema a seguir enuncia que isto não é verdadeiro.

Teorema 4.2.5 Não existe APD que decida a linguagem L_{RR}.

4.3 Gramáticas Livre de Contexto

Na Seção 3.6 nós vimos que podemos usar expressões regulares para reprentar linguagens. Mais precisamente, vimos que as linguagens representáveis por expressões regulares 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 $X \to W$, onde $X \in V$ e W é uma string de $(V \cup T)^*$.
- S é a variável inicial, onde $S \in V$.

Para simplificar, muitas vezes diremos apenas "gramáticas", ao invés de gramáticas livres de contexto e apenas "regras" ao invés de regras de produção.

■ **Exemplo 4.4** Um exemplo de gramática livre de contexto é a 4-tupla $G_{\text{PAL}} = (\{P\}, \{0,1\}, A, P)$ tal que os elementos do conjunto A são as seis regras de produção listadas abaixo:

$P \to \varepsilon$
$P \to 0$
$P \to 1$
$P \to 0P0$
$P \to 1P1$

■

Na Seção 4.3.2 veremos como gramáticas, como a que acabamos de ver no Exemplo 4.4, são formalmente associadas à linguagens. Por enquanto vamos apenas nos certificar que entendemos exatamente que tipo de objeto matemático é uma gramática. Vamos a mais um exemplo:

■ **Exemplo 4.5** A 4-tupla a seguir é uma gramática: $G_{\text{EX}} = (\{I,E\},\{a,b,0,1,+,*,(,)\},A_{\text{EX}},E)$, tal que as regras de A_{EX} são:

$E \to I$
$E \to E+E$
$E \to E*E$
$E \to (E)$
$I \to a$
$I \to b$
$I \to Ia$
$I \to Ib$
$I \to I0$
$I \to I1$ ■

Para simplificar, em vez de escrever a lista completa de regras de um dado conjunto de regras, vamos utilizar uma notação mais compacta. No caso do Exemplo 4.5, 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:

$E \to I|E+E|E*E|(E)$
$I \to a|b|Ia|Ib|I0|I1$

A ideia é que, se para uma variável A existem k regras regras $X \to \alpha_1$, $A \to \alpha_2$, $\cdots$, $X \to \alpha_k$, escrevemos tudo em uma linha da forma $A \to \alpha_1|\alpha_2|\cdots|\alpha_k$. De maneira semelhante, o conjunto de regras da gramática G_{PAL} do Exemplo 4.4 é escrito $P \to \varepsilon|0|1|0P0|1P1$.

4.3.2 Derivações de uma gramática

Dada uma gramática G, veremos agora qual é a linguagem $L(G)$ que esta gramática representa. 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 — Produção de string (um passo). 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 $A \to \gamma$ uma regra de P. Então dizemos que a string $\alpha\gamma\beta$ pode ser *produzida em um passo* da string $\alpha A\beta$ na gramática G. Neste caso escrevemos $\alpha A\beta \Rightarrow_{\text{G}} \alpha\gamma\beta$. Quando não houver ambiguidade, escreveremos apenas: $\alpha A\beta \Rightarrow \alpha\gamma\beta$.

Terminologia: As strings $\alpha\gamma\beta$ e $\alpha A\beta$ da Definição 4.3.2 são chamadas de *termos sentenciais*. Note que ambas são strings sobre o alfabeto $V \cup T$. As strings cujos símbolos pertencem apenas ao conjunto T são chamadas de *strings terminais*.

Note que uma produção só pode ser obtida de strings que são termos sentenciais que possuam pelo menos um símbolo de V, isto é, deve haver pelo menos uma variável "disponível" para aplicação de uma regra.

A partir de agora, estaremos interessados em aplicações sucessivas de regras de produção, como no exemplo a seguir:

■ **Exemplo 4.6** A string $a*(a+b00)$ pode ser produzida na gramática G_{EX} pela seguinte sequência de aplicações de regras:

$E \Rightarrow E*E \Rightarrow I*E \Rightarrow a*E \Rightarrow a*(E) \Rightarrow a*(E+E) \Rightarrow a*(I+E) \Rightarrow a*(a+E) \Rightarrow a*(a+I) \Rightarrow a*(a+I0) \Rightarrow a*(aI00) \Rightarrow a*(a+b00)$ ■

Definição 4.3.3 — Produção de string (múltiplos passos). Seja $G = (V,T,P,S)$ e sejam $\alpha,\beta,\gamma \in (V \cup T)^*$ strings. Dizemos que a string γ pode ser *produzida em múltiplos passos* a partir de α, escrito $\alpha \Rightarrow_G^* \gamma$, se o seguinte vale:

BASE: Se $\gamma = \alpha$, então vale $\alpha \Rightarrow_G^* \gamma$.
INDUÇÃO: Suponha que $\gamma \neq \alpha$ e o seguinte é verdadeiro $\alpha \Rightarrow_G^* \beta$ e $\beta \Rightarrow_G \gamma$. Então $\alpha \Rightarrow_G^* \gamma$.

Usando a notação da Definição 4.6, a partir da sequência de passos do Exemplo 4.6, sabemos que é válido escrever $E \Rightarrow_G^* a*(a+b00)$.

Definição 4.3.4 — A Linguagem de uma Gramática. Dada uma gramática $G = (V,T,P,S)$, a linguagem de G é $L(G) = \{w \in T^* \ : \ S \Rightarrow_G^* w\}$. Neste caso dizemos também que a linguagem $L(G)$ é *gerada* pela gramática G.

A seguir o teorema mais importante desta seção relacionando linguagens aceitas por autômatos com pilha, i.e., linguagens livres de contexto e linguagens geradas por gramáticas:

Teorema 4.3.1 Uma linguagem L é livre de contexto $\Leftrightarrow$ existe uma gramática G tal que $L(G)$.

A demonstração do Teorema 4.3.1 pode ser vista em [HMU06] ou [Sip06]). A ideia chave na demonstração é argumentar que $w \in L(G) \Leftrightarrow$ existe uma AP P tal que $w \in N(P)$.

Exercícios:

Exercício 4.16 Forneça um GLC que gere a linguagem $L = \{0^n 1^n \ : \ n \geq 1\}$. ■

Exercício 4.17 Forneça um GLC que gere a linguagem $L = \{0^i 1^j \ : \ i > j\}$. ■

Exercício 4.18 Considere o alfabeto Σ que contém os símbolos de "abertura" e "fechamento" de parêntesis, ou seja $\Sigma = \{(,)\}$. Forneça um GLC que gere a linguagem das strings de parêntesis balanceados sobre σ. Por exemplo, a string $(()(()))$ deve ser derivável, enquanto a string $())($ não deve ser derivável. ■

Exercício 4.19 Para responder as questões abaixo assuma que o alfabeto é $\Sigma = \{a,b,c\}$.
(a) Forneça uma gramática G tal que $L(G) = \{a^m b^n c^n : m,n \geq 0\}$.
(b) Forneça uma gramática G tal que $L(G) = \{a^n b^n c^m : m,n \geq 0\}$.
■

Exercício 4.20 Seja $\Sigma = \{a,b,c,d\}$. Forneça uma GLC para a seguinte linguagem:
$L = \{a^n b^n c^m d^m \ : \ n > 0, m > 0\} \cup \{a^n b^m c^m d^n ; n > 0, m > 0\}$. ■

4.3.3 Ambiguidade de gramáticas

Vamos começar esta seção observando o seguinte: Podemos produzir uma string em uma gramática de mais de uma forma. O exemplo a seguir esclarece isso:

■ **Exemplo 4.7** A string $a*b$ pode ser produzida na gramática G_{EX} usando diferentes sequências de aplicações de regras. Apresentamos abaixo duas delas:

(i) $E \Rightarrow E+E \Rightarrow I+E \Rightarrow a+E \Rightarrow a+I \Rightarrow a+b$
(ii) $E \Rightarrow E+E \Rightarrow E+I \Rightarrow I+I \Rightarrow a+I \Rightarrow a+b$ ■

As duas diferentes sequências de passos do Exemplo 4.8 na produção da string $a+b$ são ditas chamadas duas *derivações* diferentes. A seguir a definição formal:

Definição 4.3.5 — Derivação. Seja $G=(V,T,P,S)$ uma gramática. Uma *derivação* é uma sequência de passos $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_k$ tal que $S=\alpha_1$ e cada $\alpha_i \Rightarrow \alpha_{i+1}$ é uma produção válida na gramática G.

Note que, usando a notação para *produção* de strings (Def. 4.3.3), quando escrevemos $E \Rightarrow_{\text{G}} \alpha$, queremos dizer apenas que *existe pelo menos uma derivação* da string α, sem nenhuma preocupação sobre a quantidade derivações possíveis. O conceito de *ambiguidade* de gramáticas está relacionado à multiplicidade de possíveis derivações, mas isso ainda não é o cerne da questão. Para entendermos ambiguidade de gramáticas, precisamos intruduzir a ideia de *árvore sintática* de uma derivação.

Definição 4.3.6 — Árvores sintáticas. Seja $G=(V,T,P,S)$ uma gramática livre de contexto e seja $w=w_1w_2...w_k$ uma string de $L(G)$. Seja $\mathscr{D}$ uma derivação w em G. A *árvore sintática da derivação* $\mathscr{D}$ é uma árvore com raíz S com a seguintes relações entre os nós:

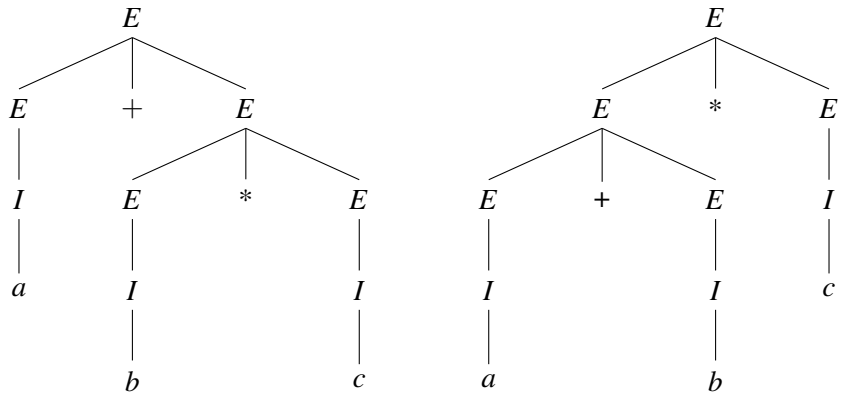
- Se o passo $\alpha A\beta \Rightarrow \alpha\gamma\beta$ aparece na derivação $\mathscr{D}$ e $\gamma=\gamma_1\gamma_2...\gamma_l$, então o nó A é pai dos nós $\gamma_1,\gamma_2,...,\gamma_l$, da esquerda para a direita.

O conjunto das árvores sintáticas de todas as possíveis derivações de w é chamado de conjunto das *árvores sintáticas da string* w .

■ **Exemplo 4.8** Condidere abaixo duas possíveis derivações da string $a+b*c$:

(i) $E \Rightarrow E+E \Rightarrow E+E*E \Rightarrow E+E*I \Rightarrow E+E*c \Rightarrow I+E*c \Rightarrow a+E*c \Rightarrow a+I*c \Rightarrow a+b*c$
(ii) $E \Rightarrow E*E \Rightarrow E+E*E \Rightarrow E+E*I \Rightarrow E+E*c \Rightarrow I+E*c \Rightarrow a+E*c \Rightarrow a+I*c \Rightarrow a+b*c$

Abaixo à esquerda a árvore sinática da derivação *(i)* e à derivação de *(ii)*.



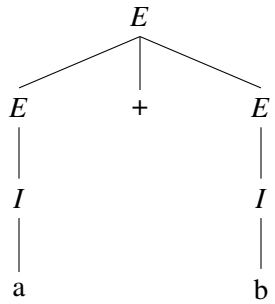
Como *(i)* e *(ii)* são duas possíveis derivações de $a+b*c$, então as duas árvores acima estão contidas no conjunto das árvores sintáticas de $a+b*c$. De fato, embora existam outras derivações possíveis para a string, todas elas acabam tendo uma das duas árvores acima (veja Exercício 4.22).

■

■ **Exemplo 4.9** No exemplo 4.8, vimos as duas derivações abaixo para a string $a * b$:

(i) $E \Rightarrow E+E \Rightarrow I+E \Rightarrow a+E \Rightarrow a+I \Rightarrow a+b$
(ii) $E \Rightarrow E+E \Rightarrow E+I \Rightarrow I+I \Rightarrow a+I \Rightarrow a+b$

Ambas derivações *(i)* e *(ii)* tem a mesma árvore de análise sintática, apresentada abaixo:



A árvore acima é a única árvore sintática de $a+b$. Note que, embora existam mais derivações possíveis para a string, além das derivações vistas em *(i)* e *(ii)*, todas elas acabem tendo exatamente a árvore sintática acima (veja Exercício 4.22). ■

Definição 4.3.7 — Ambiguidade de Gramáticas. Uma gramática G é dita ambígua se existe mais de uma árvore sintática para alguma string $w \in L(G)$.

A partir da Definição 4.3.7, podemos concluir que a gramática G_{EX}, vista anteriormente é ambígua, pois a string $a+b*c$ possui duas árvores sintáticas diferentes.

Como mencionamos antes, embora a multiplicidade de derivações não seja sinônimo de ambiguidade de gramáticas, existe uma relação entre os dois conceitos. Como veremos abaixo, se restringirmos um pouco os tipos possíveis de derivações para uma dada string, podemos estabelecer uma relação entre a multiplicidade de derivações de uma string e a ambiguidade de uma gramática. A ideia chave será permitir, para uma dada string α, apenas derivações em que a variável escolhida para aplicar a regra de produção seja àquela que aparece primeiro em α, ou seja, a variável "mais à esquerda" na string. Tal derivação é dita uma derivação *mais à esquerda*. De maneira análoga, podemos definir derivações *mais à direita*.

Teorema 4.3.2 Seja G uma gramática. Toda string de $L(G)$ tem exatamente uma derivação mais à esquerda se e somente se G não é ambígua.

Teorema 4.3.3 Seja G uma gramática. Toda string de $L(G)$ tem exatamente uma derivação mais à direita se e somente se G não é ambígua.

OBSERVAÇÕES IMPORTANTES:

- Reforçamos que 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 sintática.
- Entretanto, 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 (ver Teorema 4.3.2).

Considere uma liguagem L tal que exista uma gramática livre de contexto G tal que $L(G) = L$. Podemos nos perguntar se sempre é possível obter G de forma que G não seja ambígua. A resposta é não, pois pois existem linguagens que são ditas *inerentemente ambíguas*. Mais precisamente, isso quer dizer que existem linguagens L que podem ser geradas por gramáticas, porém toda gramática G tal que $L(G) = L$, tem-se que G é ambígua. Um exemplo de tal linguagem é $L = \{a^n b^n c^m d^m \ : \ n > 0, m > 0\} \cup \{a^n b^m c^m d^n ; n > 0, m > 0\}$ (relembramos que no Exercício 4.20 pedimos para o aluno apresentar uma gramática que gere esta linguagem), conforme o teorema enunciado abaixo:

Teorema 4.3.4 Seja $L = \{a^n b^n c^m d^m \ : \ n > 0, m > 0\} \cup \{a^n b^m c^m d^n ; n > 0, m > 0\}$. Então toda gramática G tal que $L = L(G)$ é ambígua.

A demonstração do Teorema 4.3.4 pode ser vista em [HMU06] ou [Sip06]).

Exercícios:

Exercício 4.21 Verifique que a derivação de G_{EX} do Exemplo 4.6 é mais a esquerda ■

Exercício 4.22 Apresente todas a árvores sintáticas para as strings de terminais $a+b$ e $a+b*c$ da gramática G_{EX}. ■

4.3.4 Ambiguidade de gramáticas e autômatos com pilha determinísticos

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 autômatos com pilha determinísticos. Além disso, vimos que algumas gramáticas são inerentemente 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 4.3.5 Se $L = L(P)$ para algum APD P, então existe uma gramática livre de contexto não ambígua G tal que $L(G) = L$.

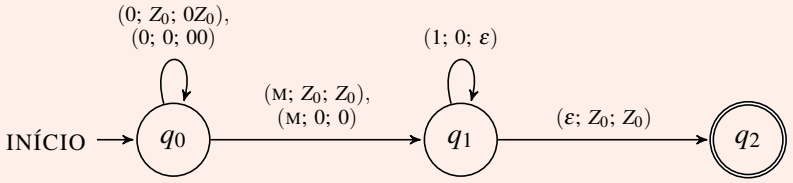
Teorema 4.3.6 Existe uma gramática livre de contexto não ambígua G tal que não existe nenhum APD que aceite a linguagem $L(G)$.

Exercícios

Exercício 4.23 Seja $A = \{a, b, c\}$ e seja L a linguagem $\{a^n bc^n \mid n \geq 1\}$ sobre o alfabeto A. Forneça um AP que aceite a linguagem L. ■

Exercício 4.24 Apresente um AP para a linguagem das strings palíndromas sobre o alfabeto $\Sigma = \{0, 1\}$. ■

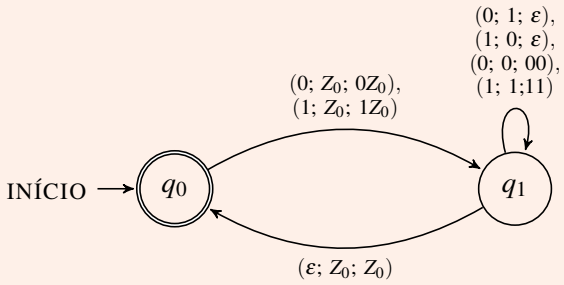
Exercício 4.25 A respeito do autômato com pilha determinístico P abaixo, que aceita uma linguagem $L(P)$ sobre o alfabeto $\{0,1,\text{M}\}$, responda as seguintes perguntas:



(a) Considere as duas sequências de configurações S_1 ou S_2 abaixo. Qual delas é a sequência correta para a computação da string 00M00 por P?
$S_1 = (q_0, 00\text{M}00, Z_0) \vdash (q_0, 0\text{M}00, 0Z_0) \vdash (q_0, \text{M}00, 00Z_0) \vdash (q_1, 00, 00Z_0)$.
$S_2 = (q_0, \ 00\text{M}00, Z_0) \vdash (q_0, 0\text{M}00, 0Z_0) \vdash (q_0, \text{M}00, 00Z_0) \vdash (q_1, 00, 00Z_0) \vdash (q_2, 00, 00Z_0)$.
(b) Apresente a sequência de configurações para a string 0M1
(c) Qual é a linguagem $L(P)$ aceita pelo autômato P?

■

Exercício 4.26 A respeito do autômato com pilha determinístico P abaixo, que aceita uma linguagem $L(P)$ sobre o alfabeto $\{0,1\}$, responda as seguintes perguntas:



(a) Apresente a sequência de configurações para a string 101 e para a string 0101
(b) Qual é a linguagem $L(P)$ aceita pelo autômato P?

■

Exercício 4.27 Observe que a linguagem da expressão regular $\underline{0^*1(0+1)^*}$ é a mesma linguagem da *gramática regular* $G = (V,T,P,S)$ com as regras abaixo:

$S \to A1B$
$A \to 0A \mid \varepsilon$
$B \to 0B \mid 1B \mid \varepsilon$

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

Exercício 4.28 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 $x_1, x_2, x_3, ...$, operadores binários $\wedge, \vee, \Rightarrow, \Leftrightarrow$, operador unário $\neg$ e abertura e fechamento de parênteses. ■

Exercício 4.29 Considere a gramática $G_{\text{PROG}} = (V, T, P, \text{[BLOCO]})$, onde os conjuntos de váriáveis, terminais e regras são:

- $V = \{$ [BLOCO], [IF-THEN], [IF-THEN-ELSE], [ATRIB], [COND] $\}$
- $T = \{$ `a`, $\cdots$, `z`, `0`, $\cdots$, `9`, `if`, `then`, `else`, `=`, `==` $\}$
- As regras são:

[BLOCO] $\rightarrow$ [IF-THEN] | [IF-THEN-ELSE] | [ATRIB]

[IF-THEN] $\rightarrow$ `if` [COND] `then` [BLOCO]

[IF-THEN-ELSE] $\rightarrow$ `if` [COND] `then` [BLOCO] `else` [BLOCO]

[ATRIB] $\rightarrow$ [VAR] `=` [NUM]

[COND] $\rightarrow$ [VAR] `==` [NUM]

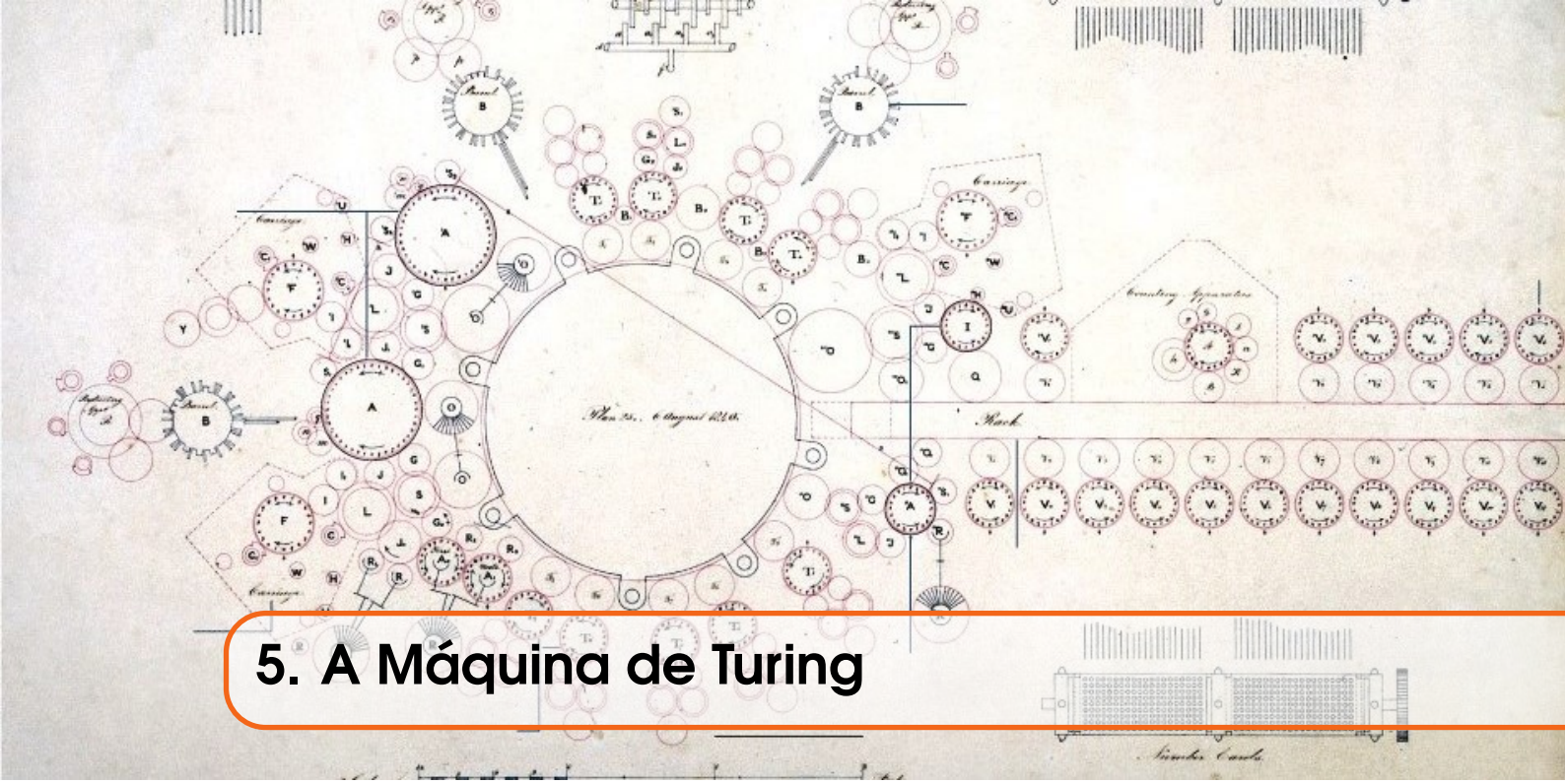
[VAR] $\rightarrow$ `a` | $\cdots$ | `z`

[NUM] $\rightarrow$ `0` | $\cdots$ | `9`

Mostre que a gramática G é ambígua.

■

Parte 2: Máquinas de Turing e Computabilidade



5. A Máquina de Turing

Neste capítulo, apresentaremos o modelo matemático proposto por Alan Turing em 1936, conhecido como Máquina de Turing. Esse modelo matemático é a base tanto para a definição formal de algoritmo quanto para a definição formal do que entendemos por computador.

5.1 Algoritmos e Problemas computacionais

Antes de falarmos de Máquinas de Turing, vamos formalizar algo com que já estamos acostumados, que é a ideia de que certos problemas computacionais podem ser vistos como linguagens. Para citar alguns exemplos básicos, vimos que o problema da divisibilidade por 3, o problema de testar palíndromos e o problema de testar primalidade podem ser modelados, respectivamente, pelas linguagens $L_3 = \{w \in \{0,1\}^* : N(w) \text{ é um múltiplo de } 3\}$, $L_{\text{PAL}} = \{w \in \{0,1\}^* : w = w^R\}$ e $L_{\text{P}} = \{w \in \{0,1\}^* : N(w) \text{ é um número primo}\}$. A seguir, vamos formalizar a correspondência entre linguagens e um tipo particular de problema, conhecido como *problema de decisão*.

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

Além de problemas de decisão, outros tipos de problemas também são relevantes para a teoria da computação. Entretanto, os problemas de decisão serão o foco de grande parte de nossas discussões neste livro. Assim, quando não houver ambiguidade, chamaremos um problema de decisão simplesmente de *problema*.

REFLETINDO UM POUCO: PROBLEMAS SÃO SEMPRE LINGUAGENS?

Como já mencionamos, em teoria da computação muitas vezes nos restringimos a estudar apenas problemas de decisão, que podem ser vistos como perguntas para as quais a resposta é SIM ou NÃO. Observe que isso é equivalente a dizer que a resposta que procuramos contém apenas um bit de informação.

Sabemos perfeitamente que nem todo problema computacional é um problema de decisão. Considere, por exemplo, o problema elementar de multiplicar dois números naturais a e b. A resposta para esse problema é o produto $a \cdot b$. Problemas desse tipo são conhecidos como *problemas de busca*, isto é, dada uma entrada, o algoritmo deve "buscar" uma solução que satisfaça certas propriedades. Note que, nesse caso, podemos interpretar a resposta desejada como uma string com mais de um bit, mais precisamente, a string w tal que $N(w) = a \cdot b$.

Nos capítulos subsequentes lidaremos com problemas de busca, bem como com *problemas de otimização* (um caso particular de problemas de busca em que, havendo mais de uma solução, desejamos aquela que minimiza ou maximiza algum valor). Entretanto, é importante destacar que, mesmo no cenário restrito aos problemas de decisão, já seremos capazes de explorar os fundamentos e os limites da computação.

Encontrar uma solução para um problema de decisão L significa apresentar um algoritmo que possa ser usado para determinar sistematicamente, em uma quantidade finita de passos, se uma dada string pertence ou não à linguagem L. Por exemplo, no Capítulo 3, vimos que o problema da divisibilidade por 3 pode ser resolvido usando um AFD. Nesse caso, para resolver o problema da divisibilidade por 3, não precisamos de toda a expressividade de uma linguagem de programação, nem mesmo da expressividade de um autômato com pilha, pois um AFD é suficiente para fornecer uma descrição algorítmica da solução do problema em questão.

No Capítulo 4, vimos que, ao permitir que autômatos utilizem uma memória, no caso uma pilha de dados, podemos expressar algoritmos que AFDs não são capazes de representar, como, por exemplo, algoritmos para testar se uma string é palíndroma. Entretanto, existem diversos problemas razoavelmente simples, como o problema de testar primalidade, que não podem ser resolvidos por autômatos com pilha. Não é difícil, porém, escrever um algoritmo que resolva o problema da primalidade em linguagem C, por exemplo. Na próxima seção veremos a definição de 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 segundo a qual as Máquinas de Turing não são apenas equivalentes a qualquer linguagem de programação existente, mas constituem um modelo matemático apropriado para representar rigorosamente o que entendemos intuitivamente por algoritmo.

CONTEXTO HISTÓRICO: AUTÔMATOS E MÁQUINAS DE TURING

Neste texto, fomos introduzindo modelos de computação 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 esses modelos nessa sequência é interessante do ponto de vista pedagógico, mas não reflete a sequência histórica em que eles apareceram na literatura científica. Curiosamente, Alan Turing propôs seu modelo na década de 1930, enquanto os modelos de computação vistos nos capítulos anteriores apareceram na literatura por volta das décadas de 1950 e 1960.

O modelo de Máquina de Turing é semelhante a um AFD acrescido de uma fita de dados. A Figura 5.1 ilustra, de forma abstrata, o funcionamento de um AFD, de um AP e de uma Máquina de Turing. Embora, conceitualmente, uma Máquina de Turing seja bastante simples, precisamos esclarecer alguns detalhes de "baixo nível" do modelo. Por exemplo, no modelo com que vamos

trabalhar aqui, por conveniência, assumiremos que a computação já se inicia com a string x posicionada na fita de memória (em vez de ser fornecida externamente, como ocorre com AFDs e APs). Na Seção 5.2, veremos tudo isso em detalhes.

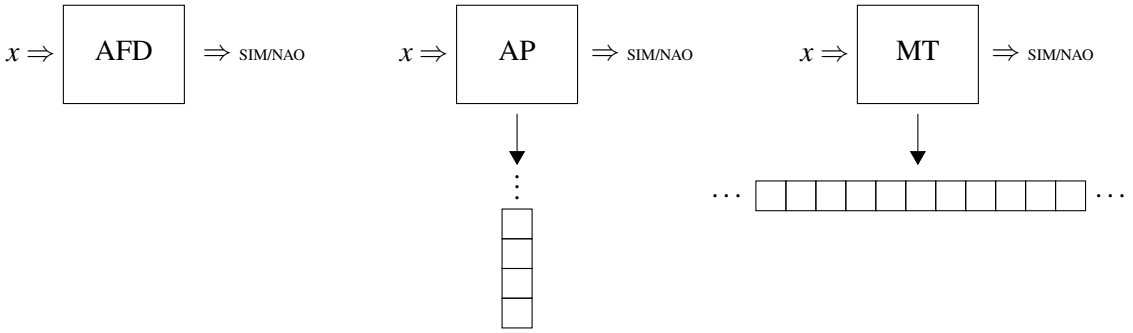


Figura 5.1: Comparação entre um AFD, um AP e uma Máquina de Turing. Os AFDs são máqinas de estados e APs são máquinas de estados com acesso a uma memória em forma de pilha de dados. A Máquina de Turing é essencialmente uma máquina de estados com uma memória em forma de *fita* de dados. A máquina pode acessar diferentes posições desta fita e ler/escrever um símbolos em tais posições.

5.2 Definição da Máquina de Turing

Primeiramente, como já mencionamos, vamos assumir que a string de entrada x, no início da computação, encontra-se localizada na fita de dados[1]. Isso simplifica um pouco a nossa definição da função de transição δ da máquina, como veremos a seguir. A Figura 5.2 ilustra isso.

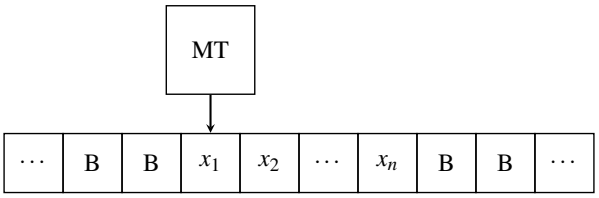


Figura 5.2: Uma máquina de Turing com a string $x = x_1x_2\cdots x_n$ na fita. Todas as infinitas células da fita à esquerda de x_1 e à direita de x_n contém o símbolo especial B, indicando que a célula está em branco.

A seguir apresentamos uma comparação de máquinas de Turing com Autômatos com pilha mostrando como tais máquinas funcionam a cada passo da computação:

Autômatos com pilha: o domínio da função δ de APs é uma tripla (estado, símbolo, símbolo), pois a cada momento a máquina se encontra em um dado estado lendo símbolos de dois lugares diferentes: string de entrada e topo da pilha;

Máquinas de Turing: Como a string de entrada se encontra já na fita de dados, o domínio da função δ de MTs é apenas um par (estado, símbolo), pois a cada momento a máquina estará em um determinado estado lendo uma posição específica da fita de dados.

[1] Na literatura, podemos encontrar algumas variações do modelo de Máquina de Turing diferentes do definido aqui. O importante é que todas essas variações acabam tendo o mesmo poder computacional do modelo apresentado aqui.

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. Quando não especificado, $\Sigma = \{0,1\}$
- Γ é o alfabeto da fita, tal que $\Sigma \subseteq \Gamma$.
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times D$ 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.

Figura 5.3: Alan Turing

A terminologia que usaremos para descrever os passos computacionais de uma máquina de Turing é o seguinte: 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', sobreescrevendo 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 $x = x_1 x_2 ... x_n$, 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 5.4 exemplifica isso.

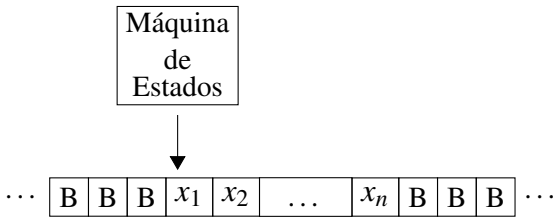


Figura 5.4: Máquina de Turing escaneando o primeiro símbolo da string $x = x_1 x_2 ... x_n$ armazenada na fita.

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 "amarrando" 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 da física 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 respresentar algoritmos.

Dissemos que MTs são "semelhantes" a AFDs (ao invés dizer "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 não[2].

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 3.19). 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.

[2]Isso 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).

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 5.5, apresenta um exemplo de um diagrama de uma MT M_{01} com estados $Q = \{q_0, ..., q_4\}$, $\Sigma = \{0,1\}$ e alfabeto $\Gamma = \{0,1,X,Y,\mathrm{B}\}$. Antes de entendermos o funcionamento da máquina, apresentamos a seguir em 5.1 a notação utilizada no diagrama.

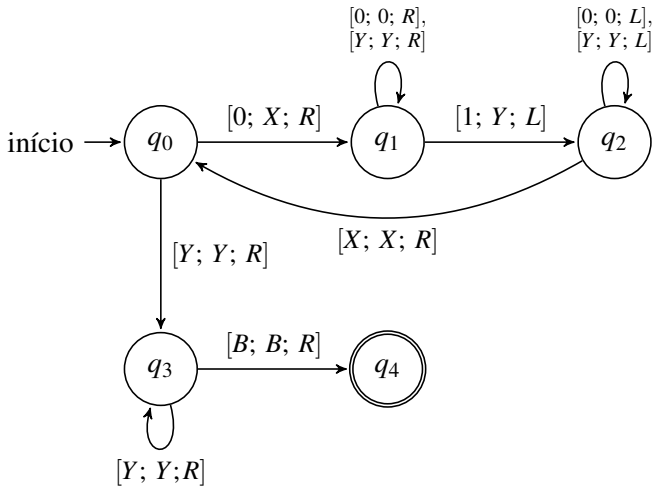


Figura 5.5: Diagrama de estados de uma Máquina de Turing.

Notação 5.1. *No diagrama de uma Máquina de Turing, uma transição de q_i para q_j com rótulo $[A;\ B;\ d]$ 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 não se move neste passo da computação.*

Observe que neste diagrama há uma transição ligando q_0 à q_1 com o rótulo $[0;\ X;\ R]$. Isto significa, de acordo com a Notação 5.1 que se a máquina estiver no estado q_0 com a cabeça de leitura sobre 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. O problema que a máquina de Turing M_{01} resolve é decidir se uma string pertence a linguagem $L_{01} = \{0^n 1^n \mid n \geq 1\}$. A seguir, apresentamos a ideia que a máquina usa para resolver este problema:

> *Estado q_0:* Neste estado a máquina lê um símbolo 0, "marca" a posição com um símbolo X e muda para o estado q_1;
>
> *Estado q_1:* Neste estado a máquina move o cabeçote para direita até encontrar um símbolo 1. Quando encontra o símbolo 1, marca a posição com Y e muda para o estado q_2;
>
> *Estado q_2:* Neste estado a máquina retorna até o último símbolo X que havia marcado, move o cabeçote para a direita do símbolo X e recomeça o processo no estado q_0. Observe que em q_0, quando não haver mais símbolos 0's na fita para marcar, a máquina muda q_3;
>
> *Estado q_3:* Neste estado a máquina vai movendo o cabeçote à direita para verificar se não restou algum símbolo 1 na fita. Se isso for verdade, muda para o estado de aceitação q_4.

Embora ainda não tenhamos definido exatamente o que significa uma MT "aceitar" ou "rejeitar" uma string (isto será feito na Seção 5.2.3), o aluno deve imaginar que estas definições são semelhantes às definições em outros modelos de computação. De qualquer forma, observe

que as únicas strings que fazem a MT da Figura 5.5 atingir seu estado final são as strings que pertencem a linguagem $L_{01} = \{0^n 1^n \mid n \geq 1\}$. Se a string não tiver o formato correto, a máquina irá encontrar uma transição não definida e irá parar em um estado de que não é final. Algo relevante na maneira que definimos Máquinas de Turing é que uma vez que a máquina atinge o estado final, ela **obrigatoriamente** para sua execução (afinal, nunca há transições definidas no estado final).

Para finalizar esta seção, observamos que o exemplo da MT da Figura 5.5, embora simples e didático, tem um pequeno inconveniente: A linguagem $L = \{0^n 1^n : n \geq 1\}$ é Livre de Contexto. Em outras palavras, não precisaríamos de todo poder de uma Máquina de Turing para decidir esta linguagem. Para um exemplo de linguagem que não é livre de contexto, considere $L_{abc} = \{a^n b^n c^n : n \geq 1\}$ sobre o alfabeto $\Sigma = \{a,b,c\}$ (ver Exercício Opcional 4.12). Para essa linguagem é possível projetar um MT não muito mais complicada que a máquina da Figura 5.5. No Exercício Resolvido 5.7 a máquina para L_{abc} é apresentada.

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 configurações.

Definição 5.2.2 — Configuração. Dada uma Máquina de Turing $M = (Q,\Sigma,\Gamma,\delta, q_0,B,F)$, uma *configuração* de M é uma tripla (α,q,β) tal que $\alpha,\beta \in \Gamma^*$ e $q \in Q$.

Seja $M = (Q,\Sigma,\Gamma,\delta, q_0,B,F)$ uma Máquina de Turing, $X_1...X_n \in \Gamma^*$ e $q \in Q$. A configuração[3] $(X_1X_2...X_{i-1},q,X_iX_{i+1}...X_n)$ indica que M, depois de fornecida alguma certa string de entrada e a computação ter ocorrido algum certo número de passos, encontra-se no estado q, com a string $x = X_1...X_n$ em sua fita 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.

A ideia agora é definir o símbolo $\vdash_M$ que, de maneira semelhante ao caso dos APs, representa um *passo computacional* de uma MT M. Entretanto, precisamos apresentar antes três casos particulares, que são os passos computacionais à esquerda, à direita ao centro:

Definição 5.2.3 — Passo computacional à esquerda. Seja $M = (Q,\Sigma,\Gamma,\delta,q_0,B,F)$ uma Máquina de Turing e $(X_1X_2...X_{i-1},q,X_iX_{i+1}...X_n)$ uma configuração de M. Se $\delta(q,X_i) = (p,Y,L)$, então escrevemos

$$(X_1X_2...X_{i-1},q,X_iX_{i+1}...X_n) \vdash_M^l (X_1X_2...X_{i-2},p,X_{i-1}YX_{i+1}...X_n).$$

Exceto nos dois casos: (1) $i = 1$; e (2) $i = n$ e $Y = B$. Nestes casos temos:

(1) Se $i = 1$, então escreveremos $(\varepsilon,q,X_1...X_n) \vdash_M^l (\varepsilon,p,BYX_2...X_n)$
(2) Se $i = n$ e $Y = B$, então escreveremos $(X_1X_2...X_{n-1},q,X_n) \vdash_M^l (X_1X_2...X_{n-2},p,X_{n-1})$.

Note que a definição acima reflete um passo da computação da MT com o movimento do cabeçote da máquina para a esquerda. Precisamos definir também os casos em que é válido sair de uma dada configuração para alguma outra nos casos em que $\delta(q,X_i) = (p,Y,R)$ e $\delta(q,X_i) = (p,Y,S)$. Este é o objetivo do exercício a seguir.

[3] Em alguns livros usa-se o termo "descrição instantânea", ao invés de configuração.

Exercício 5.1 Apresente definições para os símbolos $\vdash_M^r$ e $\vdash_M^s$ de forma que eles se refiram a passos computacionais à esquerda e estático, para os casos em que $\delta(q,X_i) = (p,Y,R)$ e $\delta(q,X_i) = (p,Y,S)$, respectivamente. ■

Definição 5.2.4 — Passo computacional de uma Máquina de Turing. Dada uma MT M, um *passo computacional* de M, denotado $\vdash_M$, se refere a qualquer um dos três casos de passos computacionais $\vdash_M^l$, $\vdash_M^r$ ou $\vdash_M^s$.

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

Definição 5.2.5 Dada uma MT M, o símbolo $\vdash_M^*$ é definido indutivamente:

Base: $I \vdash_M^* I$ para qualquer configuração 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 5.2.6 — Configurações iniciais e finais. Seja $M = (Q,\Sigma,\Gamma,\delta,q_0,B,F)$ e $w \in \Sigma^*$. A configuração (ε,q_0,w) é chamada de *configuração inicial de M com w*. Se $p \in F$, então qualquer configuração da forma (α,p,β), tal que α,β são strings de Γ^* quaisquer, é chamada de *configuração final de M*.

Exercício 5.2 Seja M a MT apresentada na Seção 5.2.2 e considere as strings $w_1 = 000111$ e $w_2 = 011$. Responda as seguintes questões:

(a) Qual é a configuração inicial de M com w_1?
(b) Apresente a sequência de configurações definida obtida pelos passos computacionais M quando a string w_1 é fornecida como entrada. Faça o mesmo para M com a string w_2.
(c) Existe uma configuração final de M com w_1?
(d) Existe uma configuração final de M com w_2?
(e) Existe mais do que uma configuração final de M com w_1? ■

O próximo passo é definir o conceito de aceitação de strings por Máquinas de Turing.

Definição 5.2.7 — Strings aceitas e strings não 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 uma configuração final de M. Caso contrário, dizemos que M *não aceita* w.

Definição 5.2.8 — 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 uma configuração final de M e máquina para ao atingir a configuração I_N.

STRINGS NÃO ACEITAS E STRINGS REJEITADAS

Considere a computação de uma MT M quando uma string w é fornecida como entrada. Observe que se w é rejeitada por M, também podemos dizer que w não é aceita por M. Entretanto, se w não é aceita por M, não podemos concluir imediatamente que w é rejeitada por M, pois pode ocorrer da MT continuar executando indefinidamente sem nunca parar.

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

- Ao contrário de AFDs e APs, na computação com Máqinas de Turing não existe o conceito de que a computação termina quando a máquina termina de ler a string de entrada. Pode ocorrer de que algumas máquinas possam parar (seja aceitando ou rejeitando certa string) antes de terem lido todos os símbolos da string de entrada. Pode também ocorrer que algumas máquinas continuem computando indefinidamente, ou seja, pode ocorrer que estas máquinas fiquem em *loop infinito* (por exemplo, veja o Exercício 5.3).

- 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. Mas enquanto a máquina não antige um estado final, existe sempre duas possibilidades: a máquina pode parar, se a transição não estiver definida neste dado estado, ou a máquina pode executar a transição, fazendo mais um passo computacional, e isso potencialmente pode seguir indefinidamente.

- A nossa busca por uma definição matemática formal para um algoritmo é essencialmente uma busca por uma definição genérica do o 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.

Exercício 5.3 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 5.2.9 — Linguagens de Máquinas de Turing. Dada uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, a linguagem $L(M) = \{w \in \Sigma^* : (\varepsilon, q_0, w) \vdash_M^* C_F$, tal que C_F é uma configuração final de $M\}$ é chamada de *linguagem de M*.

Definição 5.2.10 — Linguagens decididas por Máquinas de Turing. Seja L uma linguagem. Se existe uma *Máquina de Turing que sempre para M* tal que $L(M) = L$, dizemos que *M decide L*.

Definição 5.2.11 — Linguagens aceitas por Máquinas de Turing. Seja L uma linguagem. Se existe uma *Máquina de Turing M* tal que $L(M) = L$, dizemos que *M aceita L*.

Terminologia: Uma linguagem que pode ser decidida por uma MT também é dita uma liguagem *Recursiva*, ou *Decidível*. Uma linguagem que pode ser aceita por uma MT também dita uma linguagem *Recursivamente Enumerável*.

Notação: O conjunto das linguagens recursivas é denotado $\mathscr{R}$. O conjunto das linguagens recursivamente enumeráveis é denotado $\mathscr{RE}$.

Exercício 5.4 Mostre que $\mathscr{R} \subseteq \mathscr{RE}$. ■

Uma pergunta que que podemos fazer agora é se existe alguma linguagem que esteja em $\mathscr{RE}$, mas que não esteja em $\mathscr{R}$? Uma outra pergunta próxima a esta é a seguinte: existe alguma

linguagem que não esteja contida nem mesmo em $\mathscr{RE}$? No Capítulo 6 veremos que a resposta para ambas as perguntas é sim.

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.

Observe que de acordo com nossas definições, o conjunto das Linguagens Recursivas é o conjunto dos problemas decidíveis por Algoritmos.

Exercício 5.5 (Opcional) Considere a linguagem $L_{01} = \{0^n 1^n : n \geq 1\}$. Seja M_{01} a máquina de Turing da Figura 5.5. Prove que M_{01} decide L. ■

Exercício 5.6 Com relação a MT M_{01} da Figura 5.5, responda o seguinte:

(a) Apresente uma MT M'_{01} que tenha o seguinte comportamento: Se $x \notin L(M_{01})$, então M'_{01} deve rejeitar x e se $x \in L(M_{01})$, M'_{01} deve entrar em *loop infinito*.

(b) Apresente uma MT M'_{01} que tenha o seguinte comportamento: Se $x \in L(M_{01})$, então M'_{01} deve aceitar x. Se $x \notin L$, M'_{01} deve entrar em *loop infinito*.

(c) Considere a máquina M_{01}' do item (b). É verdade que $L(M_{01}) = L(M'_{01})$?

(d) Ainda a respeito da máquina M'_{01} do item (b), é verdade que M'_{01} aceita $L(M_{01})$? É verdade também que M'_{01} decide $L(M_{01})$?

■

Máquinas de Turing e Algoritmos

No Capítulo 7 veremos que o poder computacional de uma Máquina de Turing é equivalente ao poder computacional de um programa escrito por qualquer linguagem de programação como as que usamos cotidianamente, 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.

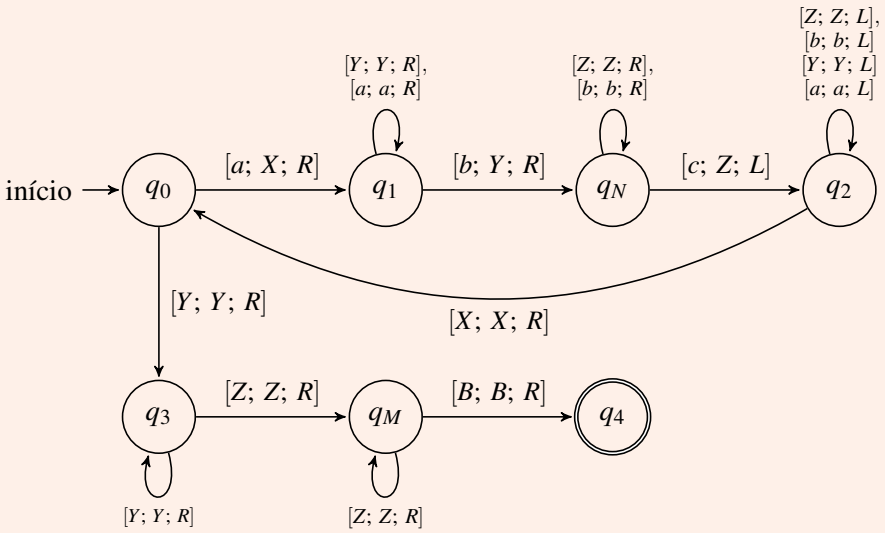
Algo importante que devemos nos antentar é que, quando usamos Máquinas de Turing na definição de algoritmos, não estamos dizendo a Máquina de Turing é o formalismo mais conveniente para se escrever algoritmos (caso não esteja convencido disto, pegue um algoritmo escrito em uma linguagem de alto nível,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

A vantagem de trabalharmos com Máquinas de Turing 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 equivalentes instâncias particulares de uma 7-tupla da Definição 5.2.1.

Exercícios:

Exercício 5.7 (Resolvido) Projete uma MT que decida se a string de entrada pertence a linguagem $L_{abc} = \{a^n b^n c^n : n \geq 1\}$, sobre o alfabeto $\Sigma = \{a, b, c\}$.

Solução: Segue a máquina:



O funcionamento da máquina acima é bastante semelhante ao funcionamento da MT da Figura 5.5.

■

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 Observe que a M_{COPY} a MT da Questão 5.8 não é o tipo máquina que normalmente vínhamos trabalhando nos capítulos anteriores do livro. Mais precisamente, M_{COPY} não é uma máquina projetada par se resolver um problema de decisão. Ainda assim, observe que a definição matemática do conjunto $L(M_{\text{COPY}})$ continua sendo aplicável. Com isso, responda qual é linguagem $L(M_{\text{COPY}})$ da máquina M_{COPY} da Questão 5.8. ■

Exercício 5.10 Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma MT que decida uma dada linguagem L qualquer. Mostre que é possível construir uma MT $M' = (Q', \Sigma, \Gamma', \delta', q_0', B, F')$ tal que:

- M' também decide L
- Quando a máquina M' para, ela deixa na fita apenas um bit 1 ou 0, dependendo do caso de aceitar ou rejeitar a string. De maneira mais precisa, a máquina M' se comporta da seguinte maneira:
  - Se M' aceita w, então $(\varepsilon, q_0', w) \vdash_M^* (\varepsilon, q_F, 1)$, onde $q_F \in F'$
  - Se M' rejeita w, então $(\varepsilon, q_0', w) \vdash_M^* (\varepsilon, q_N, 0)$, onde $q_N \notin F'$

■

5.4 Variações de Máquinas de Turing

A maneira que definimos Máquinas de Turing neste livro é uma entre muitas possíveis maneiras de definir de Máquinas de Turing. Por exemplo, em alguns livros ao invés da máquina possuir apenas uma fita, a máquina pode possuir duas fitas. Uma pergunta natural é a seguinte: O conjunto dos problemas decididos por Máquinas de Turing com duas fitas também é o conjunto das linguagens recursivas? A resposta é sim, máquinas com duas fitas decidem o mesmo conjunto de linguagens (de fato, o mesmo vale para k fitas, para qualquer constante k). A definição formal de uma *Máquina de Turing com k fitas* é uma 7-tupla $(Q,\Sigma,\Gamma,\delta,q_0,F)$, semelhante a uma MT com uma fita, de forma que a única diferença é que δ é uma função $\delta : (Q \setminus F) \times \Gamma^k \to Q \times \Gamma^k \times D^k$.

Na literatura também existem definições de Máquinas de Turing que parecem ser mais restritivas que a definição vista neste livro. Por exemplo, em algumas definições a fita é infinita em apenas uma direção, em outras o alfabeto da fita é restrito apenas a $\{0,1,B\}$. O conjunto das linguagens aceitas por estas máquinas continua sendo o conjunto das linguagens recursivas. Os Teoremas 5.4.1 e 5.4.2 enunciam a equivalência entre modelos de diferentes de máquinas de Turing.

Teorema 5.4.1 Se uma MT M com alfabeto da fita Γ decide uma linguagem L, então existe uma MT M' com alfabeto da fita $\Gamma' = \{0,1,B\}$ que decide L.

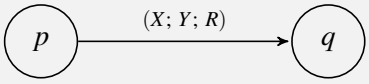
Teorema 5.4.2 Se uma MT M com k fitas decide uma linguagem L, então existe uma MT M' com uma fita que decide L.

A seguir o esboço das provas dos Teoremas 5.4.1 e 5.4.2 .

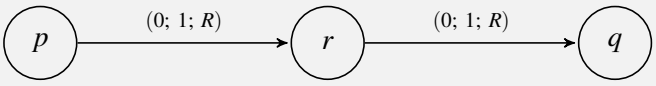
ESBOÇO DA PROVA DO TEOREMA 5.4.1

A partir de uma máquina M com alfabeto da fita Γ que resolva certo problema, podemos construir uma máquina M', normalmente com mais estados, com alfabeto $\{0,1,B\}$ para o mesmo. A ideia é primeiramente criar um sistema de codificação para os símbolos de Γ.

Suponha que os símbolos $X,Y \in \Gamma$ são codificados em binário como 00 e 11, respectivamente. Suponha que a máquina M tinha uma transição $\delta(p,X) = (q,Y,R)$, conforme abaixo:



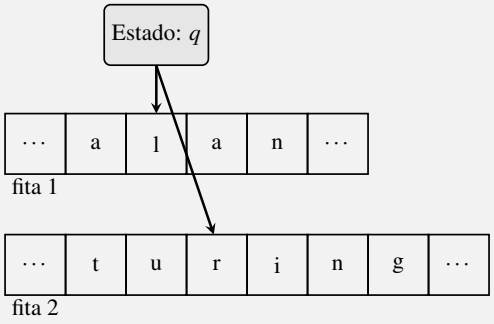
Com isso, a máquina M' teria um duas transições consecutivas substituindo 00 por 11 na fita e movendo o cabeçote duas vezes para a direita, conforme abaixo:



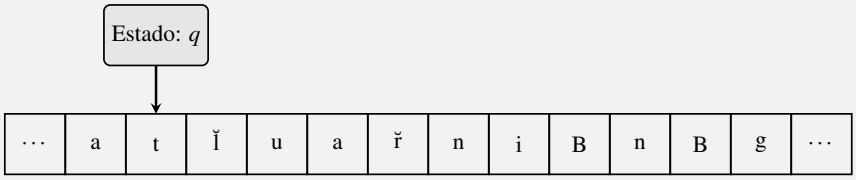
Note que mesmos símbolos 0 e 1 de Γ precisam ser codificados como strings binárias. Portanto, mesmo que o alfabeto Σ das duas máquinas seja o mesmo, dependendo de como vamos formalizar os detalhes de demonstração completa, pode ser necessário que a máquina M' primeiramente converta a string de entrada na versão codificada dela mesma, para depois começar a computação propriamente dita.

ESBOÇO DA PROVA DO TEOREMA 5.4.2

A estratégia para simular uma máquina de duas fitas em uma máquina de uma fita é as seguinte. Suponha que uma máquina M de duas fitas que tenha alfabeto $\Gamma = \{a, b, c, ..., z, \mathrm{B}\}$ esteja em uma configuração ilustrada pelo desenho abaixo:



A simulação da computação de M pode ser feita em na fita (única) de M', com alfabeto $\Gamma' = \{a, b, ..., z, \mathrm{B}, \breve{a}, \breve{b}, ..., \breve{z}, \breve{\mathrm{B}}\}$, de forma que a configuração de M' correspondente seja:



A ideia chave é que o conteúdo das duas fitas de M seja armazenado de maneira intercalada na fita única de M' e que os símbolos adicionais do alfabeto de M' sejam usados para indicar a posição de cada um dos cabeçotes de leitura de M.

Além de MTs com mais fitas ou com diferentes alfabetos, existem outras variações possíveis, como restringir as máquinas tenham exatamente um estado de aceitação e um estado de rejeição, máquinas com uma fita infinita em apenas uma direção, e muitas outras. Todos estas variações tem exatamente o mesmo poder de computação de MTs como definidas neste livro.

Exercício 5.11 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 5.12 Considere o seguinte problema: Dados dois números naturais cuja representação em binário são $a_1a_2\cdots a_n$ e $b_1b_2\cdots b_n$, calcular a soma dos dois números[a].

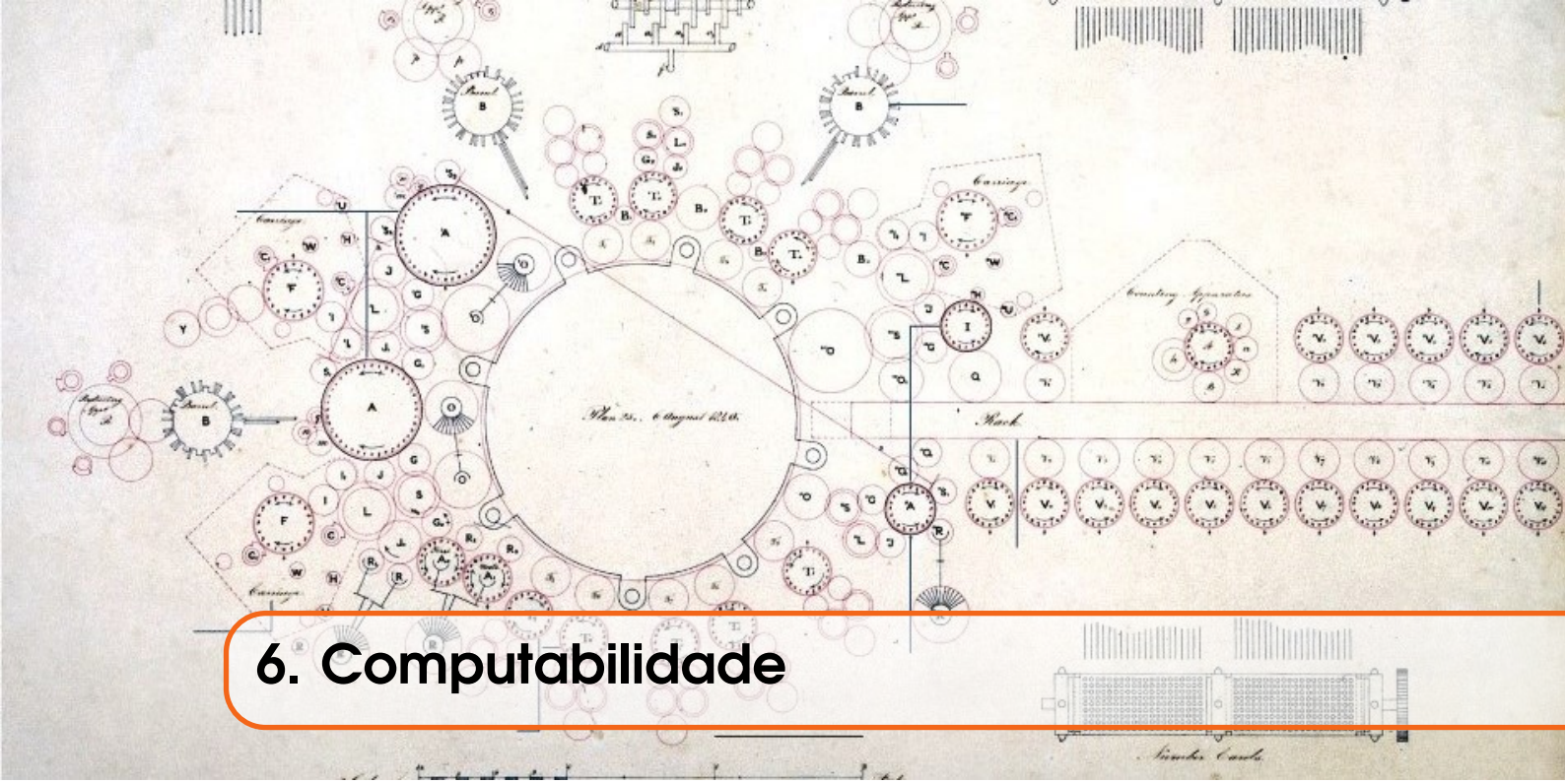
(a) Apresente uma MT com duas fitas com alfabeto $\Sigma = \{0, 1, \#\}$ e $\Gamma = \{0, 1, \#, \mathrm{B}\}$ que recebe na primeira fita $a_1a_2\cdots a_n\#b_1b_2\cdots b_n$, calcula a soma termina a execução com o resultado na segunda fita.

(b) Mostre que existe uma MT com uma fita que recebe $a_1a_2\cdots a_n\#b_1b_2\cdots b_n$, calcula a soma termina a execução contendo apenas o resultado da soma na fita.

■

[a] Observe que este não é um problema de decisão.

Exercício 5.13 (Opcional) Considere o problema do Exercício 5.12. Apresente uma MT com uma fita e alfabeto $\Sigma = \{0, 1\}$ que recebe $a_1a_2\cdots a_nb_1b_2\cdots b_n$, como entrada, calcula a soma termina a execução contendo apenas o resultado da soma na fita. ■



6. Computabilidade

Neste capítulo vamos estudar dois teoremas centrais em Teoria da Computação. Os dois teoremas estão presentes em um artigo publicado por Alan Turing em 1936. O primeiro teorema é a prova de que existem problemas que não podem ser resolvidos por algoritmos. O segundo é a existência de uma 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, que é uma máquina capaz de executar todos os possíveis programas de computador.

6.1 Detalhes de "baixo nível" em Máquinas de Turing

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 grafo ou uma árvore como entrada e retorna uma lista de números inteiros como saída. Como no caso de Máquinas de Turing, tanto a entrada quanto a saída são strings, precisamos sempre ter em mente que os objetos matemáticos que estivermos lidando, são representados por strings. Nesta seção vamos esclarecer alguns detalhes "técnicos" relativos a tais representações.

6.1.1 Notação para Máquinas de Turing tomando vários argumentos de entrada

Alguns algoritmos que vamos construir tomas vários argumentos de entrada. Uma vez que a entrada de uma máquina é apenas uma string, como fazemos saber quando termina um argumento e começa o próximo? A ideia é a mesma que usamos em computadores reais, que é usar algum esquema de codificação para os dados de entrada. O quadro MÚLTIPLAS ENTRADAS: SOLUÇÃO SIMPLES, mais adiante no texto, discute isso brevemente. Por simplicidade, quando estamos lidando com múltiplos argumentos de entrada, usamos a seguinte notação:

Notação 6.1 (Notação para MTs tomando múltiplos argumentos)**.** *Se MT toma múltiplas strings de entrada, digamos, a uma n-tupla de strings* $(x_1, x_2, x_3, ..., x_n)$*, dependendo da conveniência, usaremos tanto a notação* $M(x_1, x_2, ..., x_n)$ *quanto a notação* $M(x_1 x_2 x_3 .... x_n)$.

O que a Notação 6.1 faz é essencialmente encapsular as rotinas de baixo nível que Máquinas de Turing tem que executar para lidar com várias strings de entrada.

> MÚLTIPLAS ENTRADAS: SOLUÇÃO SIMPLES
>
> Supondo que estamos entrando com entradas que são strings binárias, uma possível maneira de lidar com este tipo de tecnicalidade seria usar uma Máquina de Turing que tenha o alfabeto $\Sigma = \{0,1,\#\}$ e $\Gamma = \{0,1,\#,B\}$. O Teorema 5.4.1 nos diz que o poder de computação de MTs com diferentes alfabetos Γ para a fita é o mesmo de MTs com alfabeto $\Gamma = \{0,1,B\}$. Podemos usar uma ideia semelhante para ver que a mesma ideia se aplica para o alfabeto de entrada Σ da máquina[a]. Assim podemos usar o símbolo # como separador para a entrada.
>
> 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.
>
> [a] O Teorema 5.4.1 refere-se apenas aos alfabetos da fita da máquina, mas o princípio é exatamente o mesmo: é necessário utilizar um esquema de codificação que represente os símbolos de um alfabeto não binário em termos de símbolos de um alfabeto binário. Entretanto, note que, estritamente falando, uma máquina com alfabeto de entrada binário nunca aceitará exatamente as mesmas strings aceitas por uma máquina que utilize outro alfabeto. Ainda assim, existe uma correspondência biunívoca entre as strings binárias aceitas e as strings originais.

6.1.2 Representando objetos matemáticos em binário

Assim como objetos matemáticos são representados, em baixo nível, pelos símbolos 0 e 1 em computadores reais, também representaremos os objetos manipulados por nossas Máquinas de Turing como strings binárias. Entretanto, é importante tomar cuidado para não confundir o objeto matemático em si com sua representação binária. Dado um objeto matemático S, usaremos a notação $\llcorner S \lrcorner$ para nos referir a string que codifica S.

Notação 6.2 (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[1] $G = (V,E)$. O que queremos dizer com a notação 6.2 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 $\llcorner r \lrcorner$ pode não fazer sentido). 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 linha[2], como no exemplo a seguir.

■ **Exemplo 6.1** 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 $\llcorner G \lrcorner = 011101110$. ■

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

[1] O apêndice deste livro apresenta a defição exata e alguns conceitos básicos a respeito de grafos para àqueles que não estão familiarizados com o assunto.

[2] Caso estivéssemos lidando com um grafo com peso nas arestas a sua matriz de adjacência não seria binária, mas mesmo neste caso não é difícil conceber uma representação binária para o mesmo.

A representação de objetos matemáticos em forma de strings binárias é importante, mas o ponto mais importante para nós 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 6.1 (Resolvido) Mostre como representar uma Máquina de Turing qualquer usando uma string binária.

Solução: Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, \mathrm{B}, F)$ uma Máquina de Turing. Lembramos que, em nossas máquinas de Turing, adotamos por convenção $\Sigma = \{0,1\}$. Podemos associar o número inteiro i ao estado $q_i \in Q$, para $i = 1,2,...,|Q|$. Também podemos estabelecer (arbitrariamente) uma ordenação $(A_1, A_2, \ldots, A_{|\Gamma|})$ para os elementos A_j de Γ e associar ao símbolo A_j o inteiro j, $i = 1,2,...,|\Gamma|$. O mesmo procedimento pode ser feito para cada um dos três elementos D_i do conjunto de direções $D = \{R, L, S\}$.

Seja k a quantidade de pares (q_i, A_j) tais que δ é definida no ponto (q_i, A_j). Antes de estabelecer a codificação de M em binário, vamos definir uma codificação "intermediária" de M como uma string $w_\#$ sobre o alfabeto $\{0,1,\#\}$. A string $w_\#$ é a concatenação de $k+2$ substrings $w_\mathrm{H}, w_1, w_2, \ldots, w_k, w_\mathrm{F}$, definidas a seguir:

- $w_\mathrm{H} = t_Q \# t_\Gamma \#$ é um cabeçalho, onde t_Q e t_Γ são, respectivamente, as representações binárias dos tamanhos $|Q|$ e $|\Gamma|$.
- Cada w_i é uma string binária correspondente a um ponto em que a função δ é definida. Por exemplo, se $\delta(q_m, A_n) = (q_u, A_v, D_w)$, então w_i é a concatenação de cinco strings binárias, representando os números m, n, u, v e w. Podemos assumir que todas as strings w_i têm o mesmo tamanho, determinado pelo maior valor entre $|Q|$, $|\Gamma|$ e $|D|$.
- w_F contém $|F|$ strings binárias, cada uma correspondente a um dos estados finais da máquina. O tamanho das codificações binárias é o mesmo utilizado nas strings w_i.

A figura abaixo ilustra os segmentos que compõem a string $w_\#$:

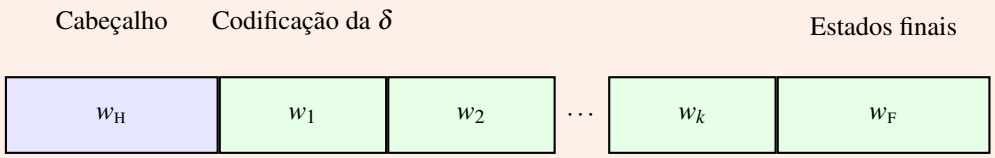


Figura 6.1: Codificação intermediária de uma Máquina de Turing como string $w_\#$. O cabeçalho w_H usa símbolos de $\{0,1,\#\}$ e as demais strings $w_1, ..., w_k, w_\mathrm{F}$ são binárias. Na codificação final $\llcorner M \lrcorner$ a string w_H é substituída pela string binária w_b usando a regra $0 \to 00$, $1 \to 11$ e $\# \to 01$.

Feito isso, basta converter o cabeçalho $w_\mathrm{H} = b_1 b_2 \ldots b_{|w_\mathrm{H}|}$ em uma string binária w_b de tamanho $2 \cdot |w_\mathrm{H}|$, de modo que cada bit 0 de w_H seja substituído por 00, cada bit 1 por 11 e cada símbolo # por 01. Assim, estabelecemos que a codificação binária de M é a string $\llcorner M \lrcorner = w_b w_1 w_2 \cdots w_k w_\mathrm{F}$.

■

Exercício 6.2 Seja M_{01} a Máquina de Turing da Figura 5.5. Apresente a string $\llcorner M_{01} \lrcorner$. ■

6.2 Funções computáveis

Uma Máquina de Turing que sempre para resolvendo um problema de decisão pode ser vista como uma máquina computando uma função booleana $f_M : \{0,1\}^* \to \{0,1\}$. A ideia é que para uma string binária w aceita pela máquina, temos que $f_M(w) = 1$ e, no caso de uma string w rejeitada pela máquina, temos que $f_M(w) = 0$. A seguir formalizamos isso:

Definição 6.2.1 — Função booleana computável. Seja $f : \{0,1\}^* \to \{0,1\}$ uma função booleana. Suponha que existe um algoritmo M que tem o seguinte comportamento:

- Se $f(x) = 1$, então M aceita x;
- Se $f(x) = 0$, então M rejeita x

Neste caso dizemos que f é uma *função booleana computável*.

Agora considere a seguinte notação:

Notação 6.3. *Suponha que uma string binária x é fornecida como entrada para uma Máquina de Turing 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.*
- $M(x) =$ PARA *quando M para com x, independente de aceitar ou rejeitar.*
- $M(x) =$ LOOP *quando M não para com a entrada x.*

Observe que, de acordo com a Notação 6.3, temos que:

(i) Se M é um algoritmo, então $\forall x \in \Sigma^*$, temos que $M(x) =$ PARA.
(ii) Se f é uma função booleana computatável, então existe um algoritmo M tal que $\forall x \in \Sigma^*$ ocorre que $M(x) = f(x)$.

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 **parou**. O objetivo de introduzir a notação a seguir é tornar esta ideia precisa.

Notação 6.4. *Seja $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ uma Máquina de Turing e $x \in \Sigma^*$. Suponha que $y = y'y''$ e que $(\varepsilon, q_0, x) \vdash_M^* (y', q_P, y'')$, onde $q_P \in Q$ e a máquina M para após antingir a configuração (y', q_P, y''). Então escrevemos $M(x) = y$.*

Observe que a expressão $M(x) = y$ pode gerar alguma ambiguidade e devemos tomar cuidado para não confundir a notação acima com a Notação 6.3 no caso da string y conter apenas um bit. Por exemplo: É possível que M aceite x parando com a string $w = 1$ na fita. De qualquer forma, isso não será um problema se deixarmos sempre claro o que queremos dizer com a expressão "$M(x)$". Além disso, conforme visto no Exercício 5.10, quando estamos lidando com problemas de decisão, podemos é sempre possível escrevermos $M(x) = 1$ ou $M(x) = 0$ sem ambiguidade nenhuma.

Definição 6.2.2 — Função computável. Uma função $f : \Sigma^* \to \Sigma^*$ 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*.

■ **Exemplo 6.2** A função para soma de números binários $f(\llcorner x \lrcorner, \llcorner y \lrcorner) = \llcorner x + y \lrcorner$ é uma função computável, pois existe uma Máquina de Turing que computa a função (ver Exercício 5.13.)

■

6.3 O problema da Parada

O objetivo desta se 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 6.3.1 — O problema da parada. O *problema da parada* é definido pela linguagem $L_{\text{H}} = \{\llcorner M \lrcorner\, x \ : \text{ tal que } M \text{ é uma MT}, x \in \Sigma^* \text{ e } M(x) = \text{PARA}\}$.

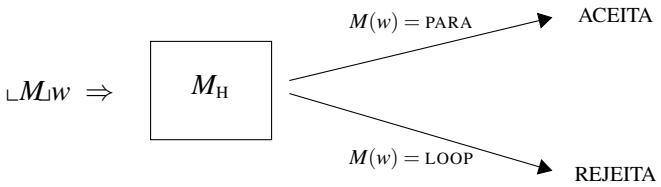
Resolver o problema da parada significaria fornecer uma MT M_{H} que decida L_{H}. Ou seja, uma MT M_{H} que tome $(\llcorner M \lrcorner, x)$ como entrada e que tenha o seguinte comportamento:

- Se $M(x) = \text{PARA}$, então $M_{\text{H}}(\llcorner M \lrcorner, x) = 1$.
- Se $M(x) = \text{LOOP}$, então $M_{\text{H}}(\llcorner M \lrcorner, x) = 0$.

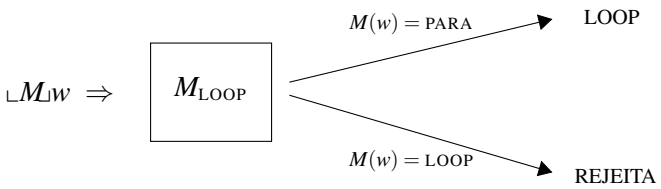
Teorema 6.3.1 — Teorema da Parada. Não existe algoritmo que decida a linguagem L_{H}.

Prova: Suponha que exista uma Máquina de Turing M_{H} que decida 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 $\llcorner M \lrcorner\, w$ é fornecida como entrada. Se $M(w) = \text{LOOP}$, então a máquina M_{H} deve rejeitar a string de entrada. Por outro lado, se $M(w) = \text{PARA}$, então M_{H} deve aceitar a string de entrada. O diagrama esquemático a seguir ilustra o funcionamento de M_{H}:

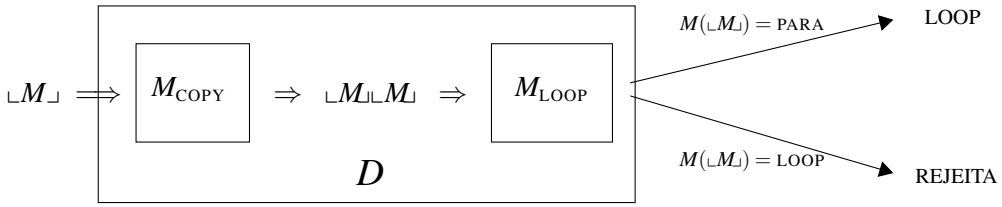


Dado que a máqina M_{H} existe, vamos agora concluir que a máquina M_{LOOP}, que vamos descrever a seguir, também existe. A máquina M_{LOOP} é essencialmente M_{H} com uma pequena modificação. Dada uma string que M_{H} aceite, ao invés da máquina aceitar e parar, a máquina deve entrar em loop infinito. o funcionamento de M_{LOOP} é ilustrado pelo diagrama esquemático abaixo.



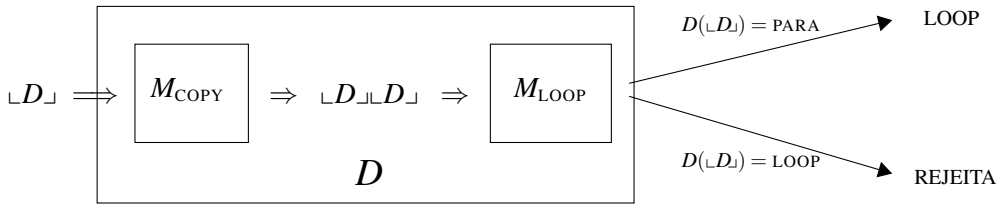
Antes de seguir em frente, devemos antentar ao tipo de argumentação que estamos usando. Estamos trabalhando com a existência de M_{H}, pois esta foi a nossa suposição inicial, mas será que, de fato, M_{LOOP} existe? Nosso argumento foi que podemos obtê-la fazendo uma modificação em M_{H}. Para termos 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 6.7 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, $\forall w \in \Sigma^*$, $M_{\text{COPY}}(w) = ww$, e a segunda parte consiste da MT M_{LOOP}, que descrevemos anteriormente. O funcionamento de D é ilustrado abaixo.



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 6.8, 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 $\llcorner M \lrcorner$ é o seguinte: $D(\llcorner M \lrcorner) = \text{LOOP} \Leftrightarrow M(\llcorner M \lrcorner) = \text{PARA}$. Considere agora a string $\llcorner D \lrcorner$ (sabemos que esta string existe, pois D existe). Agora vejamos o que acontece quando fornecemos a string $\llcorner D \lrcorner$ como entrada para a máquina D:



A conclusão que chegamos é que $D(\llcorner D \lrcorner) = \text{PARA} \Leftrightarrow D(\llcorner D \lrcorner) = \text{LOOP}$, o que é uma contradição lógica. Com isso concluímos que M_H não existe. □

6.4 A Máquina de Turing Universal

Considere uma Máquina de Turing $\mathscr{U}$ que recebe como entrada outra máquina M e uma string x, e simula o comportamento de $M(x)$. Em outras palavras, $\mathscr{U}$ executa a máquina M quando M tem como entrada a string x. A ideia é que o resultado da computação de $\mathscr{U}(\llcorner M \lrcorner, x)$ seja o mesmo resultado da computação de $M(x)$, incluindo o caso em que $M(x) = \textsc{loop}$, situação em que $\mathscr{U}$ também deve entrar em um loop infinito. Além disso, quando lidamos com Máquinas de Turing que computam funções não booleanas, isto é, quando $M(x) = y$, então $\mathscr{U}(\llcorner M \lrcorner, x) = y$.

A primeira questão que devemos considerar é se $\mathscr{U}$ realmente existe. Não podemos simplesmente assumir que, dada uma tarefa, exista uma Máquina de Turing capaz de realizá-la. A existência de tal máquina foi demonstrada por Alan Turing em seu famoso artigo de 1936. Essa máquina é conhecida como *Máquina de Turing Universal*.

Antes de enunciarmos o teorema que estabelece a existência de $\mathscr{U}$, vale a pena refletir sobre alguns pontos.

- Até este momento, consideramos as Máquinas de Turing como representações de "software". Definimos algoritmos como Máquinas de Turing que sempre param. Máquinas que entram em loop infinito, portanto, não são algoritmos, embora possam ser interpretadas como programas que nunca terminam sua execução.

- No caso da Máquina de Turing Universal, é natural interpretá-la como um modelo matemático de um computador. Uma Máquina de Turing Universal $\mathscr{U}$ tem a capacidade de executar qualquer outra MT M com qualquer entrada x, e produzir a saída correspondente $M(x)$. A máquina $\mathscr{U}$ continuará executando indefinidamente se $M(x) = \textsc{loop}$. Essa é essencialmente a função de um computador. Essa distinção entre software" e computador" é, contudo, flexível: existem implementações de algoritmos feitas diretamente em hardware de propósito específico, assim como há softwares que se comportam como uma Máquina de Turing Universal. Exemplos típicos de softwares com esse papel são os interpretadores de linguagens de programação e os emuladores[3].

- A existência de uma MT universal do ponto de vista físico, ou seja, a possibilidade de implementá-la em um dispositivo real, tem implicações profundas. Quando o conceito foi formulado por Alan Turing em 1936, não existiam computadores nem a noção moderna de software. A indústria de software atual, contudo, só é possível porque o objeto matemático $\mathscr{U}$ existe, à luz da Tese de Church-Turing. Caso contrário, não poderíamos construir computadores capazes de executar qualquer algoritmo possível; seria necessário projetar um hardware específico para cada problema a ser resolvido.

Teorema 6.4.1 Existe uma MT $\mathscr{U}$ tal que $\forall$ MT M e $\forall x \in \Sigma^*$, temos $\mathscr{U}(\llcorner M \lrcorner, x) = M(x)$.

Idéia da Prova: Para provarmos que uma certa MT existe, precisamos apresentar explicitamente 7-tupla $\mathscr{U}$. A definição em detalhes da 7-tupla $\mathscr{U}$ é muito trabalhosa e é algo que está fora do escopo deste curso, mas ideia geral não é complicada e vamos apresentar aqui. O que vamos fazer é esboçar o funcionamento de uma MT $\mathscr{U}_3$, que é uma MT com 3 fitas que realiza a tarefa que desejamos. Pelo Teorema 5.4.2, concluímos que a máquina $\mathscr{U}$ deve existir também.

A ideia é que mantenhamos $\llcorner M \lrcorner$ na primeira fita de $\mathscr{U}_3$. 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 $\mathscr{U}_3$ rode. Ainda no início da computação, colocamos a string x na segunda fita de $\mathscr{U}_3$. O que a máquina $\mathscr{U}_3$ 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 $\mathscr{U}_3$ será precisamente o

[3] Por exemplo, um emulador de Super Nintendo pode ser interpretado como uma Máquina de Turing Universal.

conteúdo estaria presente na fita de M durante a computação de $M(x)$ com o cabeçote desta fita estando localizada exatamente na posição que M estaria. A terceira fita de $\mathscr{U}_3$ será uma fita de memória auxiliar. Durante o processo de simulação da máquina M, nesta terceira fita é armazenado um número em binário que corresponde ao estado que a máquina M se encontra.

Durante a computação, $\mathscr{U}_3$ 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 $\mathscr{U}_3$ identifica isso e vai para o seu estado final, e portanto aceita a entrada e para. 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 $\mathscr{U}_3$ irá para um estado especial que é um estado que não tem nenhuma transição defindа e que também não é final, e portanto $\mathscr{U}_3$ vai parar rejeitando a entrada. Caso M fique executando indefinidamente, a MT $\mathscr{U}_3$ simplesmente vai continuar simulando M indefinidamente também. □

Relembrando que L_{H} é a linguagem da parada, temos o seguinte teorema:

Teorema 6.4.2 L_{H} é recursivamente enumerável.

Uma consequência do Teorema 6.4.2 é que o conjunto $\mathscr{R}$ está estritamente contido no conjunto $\mathscr{RE}$. Enunciamos isto no corolário a seguir:

Corolário 6.4.3 $\mathscr{R} \subsetneq \mathscr{RE}$.

Prova: Provamos no Exercício 5.4 que $\mathscr{R} \subseteq \mathscr{RE}$. Agora precisamos provar que esta inclusão é própria. Para tal, precisamos mostrar que existe pelo menos uma linguagem que esteja contida em $\mathscr{RE}$, mas que não esteja contida em $\mathscr{R}$. Pelo Exercício 6.3, $L_{\mathrm{H}} \in \mathscr{RE}$. Pelo Teorema 6.3.1, $L_{\mathrm{H}} \notin \mathscr{R}$. Portanto $\mathscr{R} \subsetneq \mathscr{RE}$.

Teorema 6.4.4 Existe L tal que $L \notin \mathscr{RE}$.

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 esas distinções eram uma ilusão. Essa fluidez é essencial hoje em dia na prática: [Por exemplo,] um programa é dado para um compilador.". – Martin David

Exercícios:

Exercício 6.3 Foneça uma prova para o Teorema 6.4.2. ■

Exercício 6.4 Forneça uma prova para o Teorema 6.4.4. ■

Exercício 6.5 Dadas duas máquinas de Turing M_1 e M_2, suponha que para qualquer string x, exatamente uma das duas máquinas para com x. Mostre que existe uma MT M_3 tal que $M_3(\llcorner M_1 \lrcorner, \llcorner M_2 \lrcorner, x) = M_i(x)$, $i \in \{1,2\}$, sendo M_i a máquina que para com x.

(Dica: use a mesma estratégia usada na prova do Teorema 6.4.1.) ■

6.5 Máquinas de Turing, pseudo-códigos, generalidade e especifidade

Considere a tarefa de resolver o problema, já amplamente discutido neste livro, de verificar se um determinado número é primo. Solucionar esse problema significa, como já sabemos, encontrar uma Máquina de Turing que decida a linguagem L_P. Entretanto, veremos que para provar a existência de uma Máquina de Turing que realize essa tarefa não será necessário apresentá-la explicitamente. A razão disso é que, no início do próximo capítulo, apresentaremos um teorema que estabelece a equivalência entre Máquinas de Turing e linguagens de programação[4]. Uma consequência direta dessa equivalência é que, para demonstrar a existência de uma Máquina de Turing que resolva um determinado problema, basta apresentar o pseudocódigo de um algoritmo que o resolva. Assim, se quisermos mostrar que existe uma Máquina de Turing capaz de decidir a primalidade de um número, basta descrever um algoritmo como o seguinte:

Primo: (N)
1: **if** $N = 1$ **then**
2: Devolva FALSO
3: **for** $i = 2; i < N; i$++ **do**
4: **if** $N \bmod i = 0$ **then**
5: Devolva FALSO
6: Devolva VERDADEIRO

Neste ponto, é natural nos perguntarmos se, uma vez que sabemos que as Máquinas de Turing são equivalentes à nossa noção intuitiva de algoritmo, ainda vale a pena estudar esse formalismo matemático e utilizá-lo para representar algoritmos. Em situações práticas, como no exemplo anterior, é claramente mais conveniente descrever um algoritmo em pseudocódigo do que apresentá-lo por meio de uma Máquina de Turing. No entanto, as Máquinas de Turing continuam sendo extremamente úteis. A razão para isso é explicada no quadro abaixo, que apresenta uma regra geral para determinar quando devemos recorrer às Máquinas de Turing e quando é mais apropriado usar pseudocódigo para representar algoritmos.

> PSEUDOCÓDIGOS OU MÁQUINAS DE TURING?
>
> Sempre que estivermos lidando com problemas específicos, como verificar se um grafo é conexo, testar se uma matriz é inversível ou determinar se uma sequência de números está ordenada, não utilizaremos Máquinas de Turing. Em vez disso, recorreremos a algoritmos descritos em forma de pseudocódigo.
>
> Por outro lado, em situações em que tratamos de algoritmos de maneira abstrata, como é comum em teoria da computação, as Máquinas de Turing são a escolha adequada. Nessa área, é frequente querermos demonstrar afirmações do tipo "não existe nenhum algoritmo M com determinada propriedade" ou "para todo algoritmo M, certo fato ocorre". A vantagem de usar Máquinas de Turing é que dispomos de uma definição precisa e simples de um objeto matemático capaz de representar qualquer algoritmo possível.

[4] O teorema enuncia que Máquinas de Turing e programas escritos em Assembly são equivalentes. Como todo programa de alto nível é convertido em Assembly para ser executado, vale também a equivalência entre Máquinas de Turing e programas em linguagens de alto nível.

6.6 Máquinas de Turing não determinísticas (MTN)

Neste seção vamos apresentar a versão não determinística das Máquinas de Turing. Uma primeira ideia que nos vem a mente é modificar a definição de Máquina de Turing para que a função de transição $\delta(p,X)$ retorne um conjunto de triplas $\{(q_1,Y_1,D_1),\ (q_2,Y_2,D_2),\ldots,\ (q_k,Y_k,D_k)\}$, de forma a termos uma definição análoga à outras máquinas não determinísticas que vimos em capítulos anteriores. A definição que daremos aqui é um pouco diferente, mas é possível provar que o poder computacional destas duas definições é o mesmo.

Definição 6.6.1 — Máquina de Turing não determinística (MTN). Uma *Máquina de Turing não determinística* é uma 7-tupla $N = (Q,\Sigma,\Gamma,(\delta_0,\delta_1),q_0,B,F)$, de forma que cada um dos componentes Q,Σ,Γ,q_0,B,F desta tupla tem o mesmo sigficado do caso de máquinas de Turing determinísticas. A diferença é que N não tem apenas uma função de transição, mas um par (δ_0,δ_1) de funções de transição.

A questão agora é entender a semântica da definição, isto é, porque ela é um modelo de computação não determinística. A ideia é que, a cada passo, a máquina tam 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, o processo de computação de uma Máquina de Turing não determinísica é dado por uma sequência de configurações (α,q,β), entendidas de maneira análoga ao caso determinístico.

Definição 6.6.2 — Configuração de uma MTN. Dada uma MTN $N = (Q,\Sigma,\Gamma,(\delta_0,\delta_1),q_0,B,F)$, uma *Configuração* de N é uma tripla (α,q,β) tal que $\alpha,\beta \in \Gamma^*$ e $q \in Q$.

Terminologia: Uma configuração (α,q,β) de uma MTN N é dita *final*, se q é um estado final de N.

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 uma configuração C, a computação pode se dirigir a possivelmente duas configurações diferentes no próximo passo, dependendo de qual das duas funções de transição foi escolhida na execução da Máquina de Turing não Determinística.

Definição 6.6.3 — Passo computacional não determinístico $\vdash_N$. Seja $N = (Q,\Sigma,\Gamma,(\delta_0,\delta_1),q_0,B,F)$ uma Máquina de Turing não determinística e C uma configuração da máqina N.

- Se a aplicação da função δ_0 faz máquina sair da configuração C e atingir a configuração C_0, então escrevemos $C \vdash_{N_0} C_0$.
- Se a aplicação da função δ_0 faz máquina sair da configuração C e atingir a configuração C_1, então escrevemos $C \vdash_{N_0} C_1$.

Escrevemos $C \vdash_N C'$ tanto para o caso de $C \vdash_{N_0} C'$ ou para o caso de $C \vdash_{N_1} C'$.

Definição 6.6.4 Dada uma MTN N, o símbolo $\vdash_N^*$ é definido indutivamente:

Base: $C_i \vdash_N^* C_i$ para qualquer configuração C_i de N.

Indução: $C_i \vdash_N^* C_j$ se $\exists C_k$ tal que $C_i \vdash_N C_k$ e $C_k \vdash_N^* C_j$.

Definição 6.6.5 — 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 $L(N) = \{w \in \Sigma^*; (\varepsilon,q_0,w) \vdash_N^* C_F$, tal que C_F é uma configuração final de $N\}$ é chamada de *linguagem de* N.

Definição 6.6.6 — 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 configurações de N, definida da seguinte maneira: A raiz é a configuração inicial da computação x por N e cada nó C é uma folha ou tem exatamente dois filhos, dependendo do seguinte:

- Se C é uma configuração final, então C é uma folha.
- Se existem configurações C', C'' tal que $C \vdash_N C'$ e $C \vdash_N C''$, então C' e C'' são os dois filhos de C.

A Figura 6.2 ilustra um exemplo de árvore de computações possíveis de uma NTM.

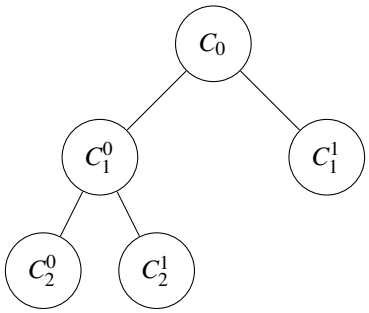


Figura 6.2: Um exemplo de árvore de computações possíveis de uma NTM com configuração inicial C_0. Neste caso existem configurações C_1^0 e C_1^1 tal que $C_0 \vdash_N C_1^0$ e $C \vdash_N C_1^1$. Além disso, existem configurações C_2^0 e C_2^1 tal que $C_1^0 \vdash_N C_2^0$ e $C_1^0 \vdash_N C_2^1$. As três configurações C_2^0, C_2^1 e C_1^1 são configurações em que a máquina para (não necessariamente finais).

Agora vamos fazer algumas observações importantes a respeito de árvores de computações possíveis. A primeira é que nada impede que uma árvore de computações possíveis tenha nós repetidos. A segunda é que podem existir casos em que tais árvores tenham alguns ramos infinitamente longos[5] 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 uma configuração final.

Teorema 6.6.1 Se L é aceita por uma MTN N, então $\forall x \in L$, a árvore de computações possíveis de N com x tem pelo menos um ramo finito.

Prova: Seja $x \in L$ e q_0 o estado inicial de N. Como N aceita x, $(\varepsilon, q_0, x) \vdash_N^* C_F$, tal que C_F é uma configuração final de N. Então a sequência de configurações saindo de (ε, q_0, x) e chegando a C_F é um ramo finito da árvore de computações possíveis.

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.

[5] Um 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.

Teorema 6.6.2 Se N é uma MTN, então existe uma MT M tal que $L(M) = L(N)$.

Idéia da prova: Podemos projetar uma MT M que simula a execução de uma NTM N com a entrada x construindo, nível por nível, sua árvore de computações possíveis (i.e., fazendo um busca em largura na árvore de computações possíveis). Caso seja atingido uma configuração final, M aceita a entrada. Para fazer esta simulação, M inicialmente escreve na fita a configuração incial $\llcorner(\varepsilon, q_0, x)\lrcorner$ de N (sendo q_0 o estado inicial de N). A partir daí, a máquina começa a sua rotina principal, que é a seguinte: Para cada configuração $\llcorner C\lrcorner$ que esteja escrita na fita faz o seguinte:

(1) Se a configuração C é final, para e aceita;

(2) Se C não é uma configuração final, computa e escreve na fita as configurações $\llcorner C_0\lrcorner$ e $\llcorner C_1\lrcorner$ que podem ser obtidas de C e, em seguida, apaga $\llcorner C\lrcorner$ da fita.

Note que pelo Teorema 6.6.1, para toda string x aceita por N, a árvore de computações possíveis tem um ramo finito atingindo uma configuração final de N. □

Teorema 6.6.3 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. Basta tomar N com os mesmos componentes da 7-tupla de M, exceto que o par (δ_1, δ_2) de funções de transição de N deve ser $\delta_1 = \delta_2 = \delta$. □

Exercícios

Exercício 6.6 Apresente uma MTN que escreve uma string binárias 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 possíves. ■

Exercício 6.7 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 $M'(x) = \text{LOOP}$
(2) Se $M(x) = 0$, então $M'(x) = 0$.
■

Exercício 6.8 Seja M_{COPY} a máquina fornecida no Exercício 5.8. Seja M_{LOOP} a máquina cujo funcionamento é explicado na prova do Teorema 6.3.1. Prove que se M_{COPY} M_{LOOP}, então existe uma máquina D tal que $D(\llcorner D\lrcorner) = \text{PARA} \Leftrightarrow D(\llcorner D\lrcorner) = \text{LOOP}$, o que é uma contradição lógica. ■

Exercício 6.9 Prove de maneira formal que L_{H} (a linguagem da parada) é recursivamente enumerável. ■

Exercício 6.10 Seja $\mathscr{M}$ o conjunto de todas as máquinas de turing e seja $\mathscr{L}$ o conjunto de todas as linguagens sobre Σ. Usando o argumento da diagonalização mostre que $|\mathscr{M}| < |\mathscr{L}|$ e conclua que existem linguagens que não são recursivamente enumeráveis. ■

Exercício 6.11 No Exercício 6.10 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 $\overline{L_{\text{H}}} \notin \mathscr{RE}$. ■

Exercício 6.12 Prove o seguinte problema é indecidível. Dada uma Máquina de Turing M, queremos decidir se $M(\varepsilon) = \text{PARA}$. ■

Exercício 6.13 Prove que $L = \{\llcorner M \lrcorner \, ; \exists x \in \Sigma^* \text{ tal que } M(x) = \text{PARA}\}$ é indecidível. ■

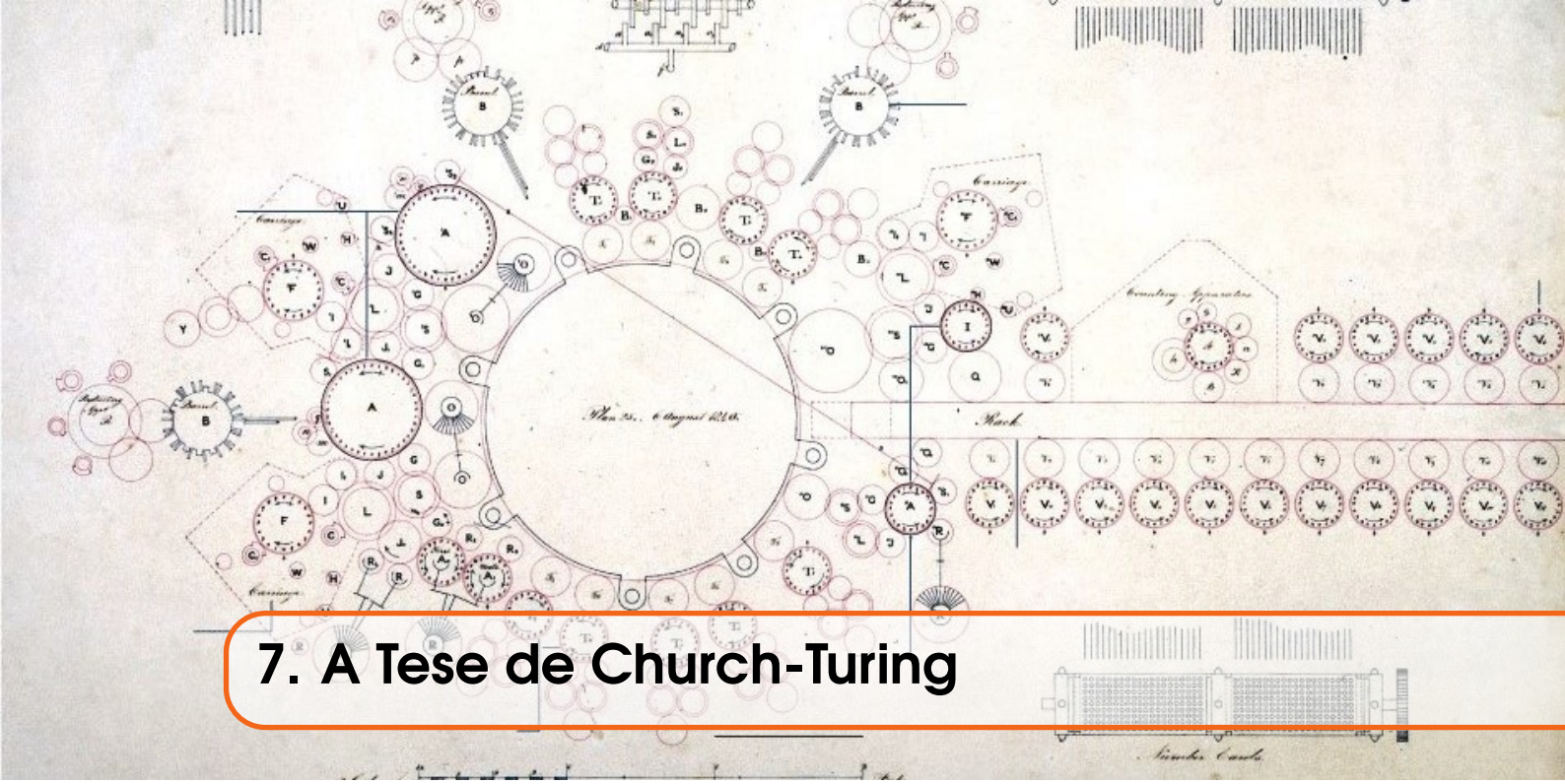
Exercício 6.14 Prove que $L = \{\llcorner M \lrcorner \, ; \exists x \in \Sigma^* \text{ tal que } M(x) = \text{LOOP}\}$ é indecidível. ■

Exercício 6.15 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 $M(\varepsilon) = \text{LOOP}$, então $M'(\llcorner M \lrcorner) \neq \text{LOOP}$ (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 6.16 Descreva como são as árvores de computações possíveis de MTNs tal que $\delta_1 = \delta_2$ apresentadas na "ideia da prova" do Teorema 6.6.3. ■



7. A Tese de Church-Turing

Neste capítulo, discutiremos duas teses distintas que são normalmente chamadas pelo mesmo nome: a *Tese de Church-Turing*. A primeira é a afirmação de que a Máquina de Turing constitui uma *definição matemática* que captura a noção intuitiva do que entendemos por processo computacional. A segunda é a afirmação de que qualquer processo computacional *fisicamente realizável* pode ser modelado matematicamente por uma Máquina de Turing.

7.1 Perspectiva histórica

No início do século XX, alguns matemáticos observaram que o processo de demonstrar um teorema assemelhava-se a um procedimento mecânico: Dada uma afirmação, o que fazemos é aplicar 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, alguns matemáticos começaram a especular se o raciocínio matemático poderia ser completamente definido em termos de um processo mecânico.

Isso motivou o matemático David Hilbert a propor, em 1928, um problema à comunidade matemática: encontrar um algoritmo que, ao receber como entrada uma afirmação matemática, responda SIM, se a afirmação é verdadeira, ou NÃO, se a afirmação é falsa[1]. Note que, da forma como Hilbert propôs o problema, ele sequer considerava a possibilidade de que tal algoritmo pudesse não existir.

Resolver o problema proposto por Hilbert, conhecido como *Entscheidungsproblem* ("problema da decisão", em alemão), era uma tarefa ambiciosa, pois o objetivo era encontrar um algoritmo extremamente poderoso, capaz de automatizar todo o processo de demonstração matemática. Qualquer pessoa seguindo tal algoritmo seria, em princípio, capaz de provar qualquer teorema[2]. Em 1931, Kurt Gödel demonstrou que o projeto de Hilbert esta fadado ao fracasso, provando os famosos

[1] Ou seja, SIM se a afirmação é um *teorema* ou NÃO, se é uma uma afirmação falsa. Observamos que o que chamamos aqui de afirmação matemática pode ser formalmente definida usando algum sistema lógico formal, usando certos axiomas e certas regras de inferência definidas a priori.

[2] O matemático Gottfried Leibniz também já havia refletido sobre esse mesmo problema no século XVII; no entanto, àquela época, ainda não se dispunha do ferramental matemático necessário para formular a questão com precisão.

teoremas da incompletude, que afirmam que nem todo teorema verdadeiro pode ser provado dentro de um sistema axiomático consistente e suficientemente expressivo[3]. Ainda assim, a esperança de Hilbert permanecia válida em um cenário mais restrito: talvez existisse um algoritmo que, ao menos, decidisse se uma afirmação era ou não demonstrável dentro de um sistema formal fixado.

Em 1936 Alonzo Church e, logo em seguida, Alan Turing mostraram que mesmo essa expectativa era inalcançável, ou seja, mesmo no âmbito delimitado por Gödel, o algoritmo proposto por Hilbert não poderia existir. Um ponto fundamental que devemos destacar é que, para demonstrar a inexistência de um algoritmo, o primeiro passo necessário era tornar preciso o que se entende por *algoritmo*. Alonzo Church utilizou um formalismo matemático chamado *cálculo* λ, enquanto Alan Turing desenvolveu um outro modelo, conhecido hoje como Máquina de Turing. Ambos os modelos são equivalentes em poder computacional, mas o modelo de Turing se destacou por sua simplicidade e intuição, oferecendo uma interpretação "física" do conceito de algoritmo que tornava mais convincente a ideia de representar qualquer processo mecânico.

Adicionalmente, Turing mostrou que seu modelo possuía uma propriedade conhecida como *universalidade*. Essa propriedade está relacionada ao conceito de "computadores de propósito geral", ou seja, dispositivos capazes de receber algoritmos *como entrada* e executá-los passo a passo. Veremos em detalhes o significado deste conceito no Capítulo 6. 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.

7.2 Máquinas de Turing são equivalentes a linguagens de programação

Antes de entrarmos em uma discussão mais profunda para compreender a afirmação de que as "Máquinas de Turing capturam o que entendemos por algoritmos", comecemos com algo mais simples e concreto. Nosso primeiro passo será apresentar um teorema que demonstra que Máquinas de Turing são capazes de representar algoritmos escritos em linguagens de programação.

À primeira vista, é fácil perceber que projetar uma Máquina de Turing para realizar uma tarefa é muito mais trabalhoso do que escrever um programa em uma linguagem de programação de alto nível para executar a mesma tarefa. Entretanto, o fato de que um formalismo exige mais esforço para expressar um algoritmo do que outro não tem relação com o seu *poder computacional*. Basta lembrar que, embora escrever um programa em Assembly seja muito mais trabalhoso do que escrevê-lo em Python, tal programa **pode** ser implementado em ambas as linguagens.

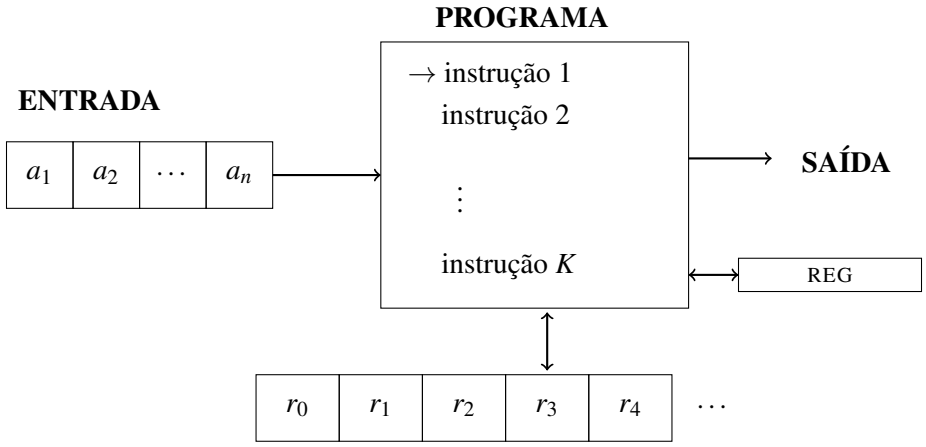
Nesta seção, enunciaremos um teorema que afirma que as Máquinas de Turing são equivalentes aos algoritmos escritos nas linguagens de programação utilizadas atualmente e apresentaremos a ideia principal de sua prova. Como essa formulação é um tanto vaga, adotaremos uma abordagem mais precisa e simples, mostrando que as Máquinas de Turing são equivalentes a programas escritos em linguagem *assembly*. Para isso, apresentaremos uma formulação matemática para programas escritos nessa linguagem, uma vez que qualquer programa em uma linguagem de alto nível pode ser expresso em assembly[4].

[3] A grosso modo, Gödel mostrou é que mesmo para a aritmética, que é uma parte da matemática, um sistema lógico capaz de expressá-la, ou será incompleto (não conseguirá provar todas as verdades) ou será inconsistente (permitirá provar falsidades). Não há como um sistema lógico que expresse a aritmética ter os dois: completude e consistência.

[4] O trabalho de um compilador é converter um programa de alto nível em um programa assembly, que, por sua vez, consiste essencialmente em uma sequência de instruções que o processador é capaz de executar.

7.2.1 Programas Assembly e Máquinas RAM

O modelo matemático que apresentaremos aqui é chamado *Modelo de Máquina RAM*[5], ou simplesmente *Modelo RAM*. O diagrama a seguir ilustra o funcionamento do modelo de Máquina RAM que utilizaremos neste livro:



Um programa para uma Máquina RAM é chamado de *Programa Assembly* e consiste em uma sequência finita de *instruções*. Para sermos precisos, vamos definir exatamente o formato de uma instrução neste modelo.

Considere os seguintes conjuntos A_1 e A_2:

- $A_1 = \{\text{HALT}, \text{READ}, \text{WRITE}\}$
- $A_2 = \{\text{ADD}, \text{MULT}, \text{SUB}, \text{STORE}, \text{LOAD}, \text{JUMP}, \text{JPOS}, \text{JZERO}, \text{JNEG}\}$

Chamaremos de *argumento de instrução* qualquer número inteiro j ou símbolo $\hat{}j$, denominado *endereço de* j.

Definição 7.2.1 — Instrução assembly. Uma *instrução* π pode assumir duas formas:
(1) π é um elemento de A_1;
(2) π é um elemento de A_2 seguido de um argumento.
Além disso, para instruções do tipo (2), vale a seguinte restrição:
Se o argumento é um endereço $\hat{}j$, então a instrução é restrita a uma das seguintes:
LOAD, STORE, ADD, SUB, MULT.

As instruções do tipo (2) são chamadas de *instruções com argumento*. Em uma instrução com argumento, o argumento é escrito após um espaço em branco. Por exemplo, escrevemos STORE 12 separando o nome da instrução e o argumento com um espaço.

Definição 7.2.2 — Programa Assembly (PA). Um *programa assembly* é uma sequência finita $\pi_1, \pi_2, \ldots, \pi_n$ de instruções assembly.

Para maior legibilidade, em vez de escrevermos a sequência de instruções separadas por vírgulas, como na Definição 7.2.2, representamos programas assembly uma instrução por linha, numerando-as conforme sua posição. O exemplo a seguir ilustra essa convenção.

[5] O acrônimo RAM vem do inglês *Random Access Memory*, que reflete a estrutura dos computadores modernos, em que cada posição de memória pode ser acessada diretamente em um único passo, ao contrário das Máquinas de Turing, que podem necessitar de vários passos para deslocar o cabeçote até a posição desejada.

■ **Exemplo 7.1** A seguir, um exemplo de programa assembly:

```
1. READ
2. STORE  ^1
3. READ
4. ADD    ^1
5. STORE  ^1
6. HALT
```

■

Embora um programador consiga ter uma ideia da tarefa realizada pelo programa acima, do ponto de vista matemático isso ainda não está bem definido, pois, a partir da Definição 7.2.2, tudo o que podemos afirmar é que se trata de um programa válido. Ainda não estabelecemos a *semântica* das instruções e, portanto, não definimos formalmente o efeito da execução de cada uma delas em uma máquina RAM.

O funcionamento de uma Máquina RAM é o seguinte: dada uma entrada w, que é uma n-tupla de números inteiros $(a_1, \ldots, a_n)$, a máquina executa o programa Π utilizando registradores $r_0, r_1, r_2, \ldots$ como memória, podendo escrever números inteiros na saída. A máquina possui também um registrador especial, denominado REG.

A ideia é que a instrução READ seja utilizada para ler a entrada. Mais precisamente, na i-ésima execução da instrução READ, o inteiro a_i é armazenado no registrador REG. A execução de um programa consiste em uma sequência de instruções executadas pela máquina. Para isso, a máquina mantém um *contador de programa*, que indica, a cada instante, qual é a próxima instrução a ser executada. O quadro a seguir apresenta a semântica de cada instrução.

O Conjunto de Instruções e a Semântica de cada instrução

A seguir, apresentamos as 16 instruções que compõem o conjunto de instruções da Definição 7.2.1, bem como a semântica de cada uma delas.

READ		A i-ésima chamada desta instrução carrega em REG o valor a_i da entrada.
WRITE		Escreve o conteúdo de REG na saída.
LOAD	i	Carrega o inteiro i em REG.
LOAD	$\hat{}i$	Carrega o conteúdo de r_i em REG.
STORE	$\hat{}i$	Armazena o conteúdo de REG no registrador r_i da memória.
ADD	i	Seja $v(\text{REG})$ o valor contido em REG. Depois da instrução, o registrador REG passa a conter o valor $v(\text{REG}) + i$.
ADD	$\hat{}i$	Sejam $v(\text{REG})$ e $v(r_i)$ os valores em REG e r_i, respectivamente. Depois da instrução, o valor em REG passa a ser $v(\text{REG}) + v(r_i)$.
SUB	i	Seja $v(\text{REG})$ o valor contido em REG. Depois da instrução, o registrador REG passa a conter o valor $v(\text{REG}) - i$.
SUB	$\hat{}i$	Sejam $v(\text{REG})$ e $v(r_i)$ os valores em REG e r_i, respectivamente. Depois da instrução, o valor em REG passa a ser $v(\text{REG}) - v(r_i)$.
MULT	i	Seja $v(\text{REG})$ o valor contido em REG. Depois da instrução, o registrador REG passa a conter o valor $v(\text{REG}) \times i$.
MULT	$\hat{}i$	Sejam $v(\text{REG})$ e $v(r_i)$ os valores em REG e r_i, respectivamente. Depois da instrução, o valor em REG passa a ser $v(\text{REG}) \times v(r_i)$.
JUMP	i	Muda o valor do contador de programa para i.
JZERO	i	Muda o contador de programa para i, caso o conteúdo de REG seja zero.
JPOS	i	Muda o contador de programa para i, caso o conteúdo de REG seja positivo.
JNEG	i	Muda o contador de programa para i, caso o conteúdo de REG seja negativo.
HALT		Interrompe a execução do programa.

Uma vez compreendida a semântica das instruções, podemos entender o que faz o programa do Exemplo 7.1: ele soma dois números e armazena o resultado no registrador r_1. A seguir, outro exemplo de programa assembly:

■ **Exemplo 7.2 — Testando a soma de dois números.** A seguir, um programa que lê três números a, b, c e verifica se $a + b = c$. Se a igualdade for verdadeira, escreve 1 na saída; caso contrário, escreve 0.

```
1.  READ
2.  STORE  ^1
3.  READ
4.  ADD    ^1
5.  STORE  ^1
6.  READ
7.  SUB    ^1
8.  JZERO  11
9.  WRITE  0
10. HALT
11. WRITE  1
12. HALT
```

■

Dizemos que um programa assembly Π *para* se a instrução HALT é executada durante a execução de Π em uma máquina RAM. Note que o programa acima sempre para, pois uma instrução HALT (veja as linhas 10 e 12) será sempre executada, independentemente da entrada do programa.

Como bem sabemos, assim como as máquinas de Turing, programas assembly são capazes de resolver muito mais do que apenas problemas de decisão. Entretanto, como neste capítulo lidaremos especificamente com problemas de decisão, dado um programa assembly Π, será essencial definir formalmente a linguagem do programa Π. Para isso, definimos aceitação e rejeição de uma string de entrada da seguinte forma: uma string é aceita por Π se a máquina para tendo escrito na saída algum valor diferente de zero durante sua execução. Caso contrário, dizemos que o programa não aceita a string de entrada.

Definição 7.2.3 A linguagem de um programa assembly Π, denotada $L(\Pi)$ é o conjunto de strings aceitas pelo programa Π.

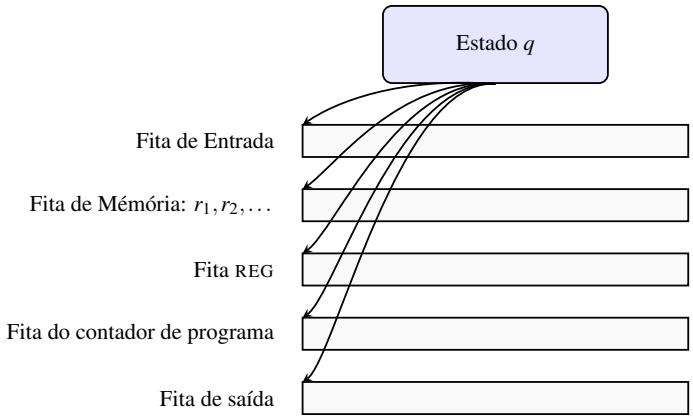
Com isso, vamos enunciar o teorema a seguir.

Teorema 7.2.1 Para todo Programa Assembly Π, existe uma Máquina de Turing M tal que $L(M) = L(\Pi)$.

Ideia da prova: Primeiramente, a ideia é definir 16 máquinas de Turing distintas, uma para cada tipo de instrução (veja, na página anterior, os 16 tipos de instrução apresentados no quadro O CONJUNTO DE INSTRUÇÕES E A SEMÂNTICA DE CADA INSTRUÇÃO). Cada uma dessas 16 máquinas de Turing é definida como uma máquina de cinco fitas. A partir destas máquinas é possível construir uma máquina de Turing maior, que chamaremos de "Máquina de Turing controladora", também tendo cinco fitas. O conteúdo das fitas da máquina controladora representará os seguintes componentes de uma máquina RAM da seguinte forma:

Fita 1: Entrada;
Fita 2: Memória, isto, é conteúdo dos registradores $r_1, r_2, ...$ em uso;
Fita 3: Conteúdo do registrador REG;
Fita 4: Valor do *contador de programa*;
Fita 5: Saída.

Iremos nos referir à fita 1 da máquina controladora como *fita de entrada*, à fita 2 como *fita de memória*, e assim por diante, conforme a ilustração a seguir.



Podemos tornar esta construção mais precisa da seguinte forma: Para qualquer Programa Assembly $\pi_1, \pi_2, \ldots, \pi_k$, é possível especificar uma série de Máquinas de Turing $M_1, M_2, \ldots, M_k$, cada uma correspondendo a uma das k instruções do programa Π de forma que, assim como cada uma das k instruções é de um dos 16 tipos de instrução possível, cada uma destas k máquinas de Turing é de um dos 16 tipos possíveis de máquinas correspondente. Partindo do conjunto destas k máquinas, podemos construir a Máquina de Turing controladora M, também com cinco fitas, que executa em ciclos.

A cada ciclo da máquina controladora, a sub-rotina M_i é executada, sendo i o valor armazenado na fita do contador de programa. Após executar M_i, o valor do contador é atualizado da seguinte forma: se a sub-rotina não for do tipo JUMP, o novo valor é $i+1$; caso contrário, a alteração já terá sido feita pela própria sub-rotina. □

Exercício 7.1 (Opcional) Apresente uma MT de 5 fitas, conforme a ideia apresentada na prova do Teorema 7.2.1 para executar a seguinte tarefa:

(a) Uma máquina que simule a instrução ADD ^ 3: a máquina deve somar o conteúdo da fita REG ao conteúdo de r_3 na fita de memória, e guardar o resultado na fita REG, e terminar sem nenhuma alteração nas demais fitas.

(b) Uma máquina que simula a instrução LOAD 4: a máquina deve escrever ⌞4⌟ na fita REG, e terminar sem nenhuma alteração nas demais fitas.

(c) Uma máquina que simula a instrução READ: a máquina deve ler o conteúdo da fita entrada e escrever na fita REG, e terminar sem nenhuma alteração nas demais fitas.

(d) Uma máquina que simula a instrução JUMP 5: a máquina deve escrever na fita do contador de programa o número 5, e terminar sem nenhuma alteração nas demais fitas. ■

O que o enunciado do Teorema 7.2.1 afirma é que se existe um programa assembly que resolve um problema, então existe uma Máquina de Turing que resolve o mesmo problema. A recíproca também é verdadeira: para toda Máquina de Turing, existe um Programa Assembly equivalente, o que significa que ambos os modelos são computacionalmente equivalentes.

Teorema 7.2.2 Para toda Máquina de Turing M, existe um Programa Assembly Π tal que $L(\Pi) = L(M)$.

Ideia da prova: Dada uma Máquina de Turing M, podemos implementar um programa que contenha matriz que armazena a tabela de transições de M e uma variável que guarda um índice correspondente a cabeça de leitura da máquina. A cada passo, o programa execute sobre a entrada exatamente o que M executaria, atualizando a entrada e a variável que guarda a posição da cabeça de leitura de M. □

7.3 Máquinas de Turing e outros modelos de computação

Além dos programas que executam em máquinas RAM, apresentados na seção anterior, diversos outros modelos matemáticos são equivalentes às Máquinas de Turing. Alguns desses modelos foram propostos ainda na primeira metade do século XX, sendo os mais conhecidos o *cálculo* λ e as *funções* μ*-recursivas* (o apêndice deste livro apresenta brevemente esses formalismos). Esses modelos foram desenvolvidos com o objetivo estritamente matemático de oferecer uma definição formal de algoritmo, sem a intenção, ao menos inicialmente, de estabelecer correspondência com máquinas passíveis de implementação.

No âmbito prático, além da equivalência entre as Máquinas de Turing e os computadores atuais, pesquisas em áreas que buscam construir computadores baseados em substratos físicos "não tradicionais" também têm produzido modelos matemáticos equivalentes às Máquinas de Turing. Ainda é cedo para afirmar quais desses modelos, originados da tentativa de explorar novos substratos físicos para a contrução de computadores, correspondem a tecnologias com potencial real de implementação. De todo modo, podemos ilustrar essa ideia com duas áreas que têm se destacado nos últimos anos. Em menor escala, há a área da computação molecular, mais especificamente a computação baseada em moléculas de DNA[6]. A outra área, atualmente uma das mais ativas e promissoras, é a da computação quântica (veja o quadro COMPUTAÇÃO QUÂNTICA).

COMPUTAÇÃO QUÂNTICA

A ideia que sustenta a pesquisa em computação quântica tem origem em uma observação feita por Richard Feynman [Fey82] e, posteriormente, generalizada por David Deutsch [Deu85]. Feynman notou que, de acordo com as leis da mecânica quântica, a evolução temporal de um sistema físico isolado — composto por átomos, elétrons, fótons ou outros constituintes — pode ser modelada, com precisão arbitrária[a], por uma abstração matemática chamada *circuito quântico*. A proposta é usar o estado físico de certos objetos para representar informação, e realizar o processamento dessa informação por meio da manipulação controlada dos estados possíveis desses objetos, obedecendo às leis quânticas que regem suas transformações.

Deutsch ampliou essa concepção ao propor que qualquer sistema físico, e não apenas sistemas quânticos microscópicos, pode, em princípio, ser simulado por um computador quântico universal. Nessa formulação, não é necessário supor que o sistema opere em um "regime quântico", pois, segundo a própria mecânica quântica, toda a realidade física é quântica em sua base — o comportamento clássico emerge apenas como um limite aproximado. Assim, a tese de Deutsch expressa a noção de que um circuito quântico universal não é apenas um modelo de computação eficiente, mas uma descrição completa de qualquer processo físico possível.

[a] A expressão "precisão arbitrária" deve ser entendida que, em princípio, é possível aproximar a evolução do sistema físico com qualquer grau desejado de exatidão, desde que se disponha de recursos computacionais suficientes.

[6] Há diversos formalismos utilizados em computação molecular e em computação com DNA. No contexto em que o objetivo é realizar *computação de propósito geral*, um dos modelos mais conhecidos é o aTAM (sigla em inglês de *abstract Tile Assembly Model*).

Teorema 7.3.1 — Teorema da Equivalência. Se um algoritmo pode ser escrito em um dos três modelos de computação abaixo, então também pode ser escrito nos outros dois modelos:

(1) Máquinas de Turing com k fitas, para qualquer inteiro $k \geq 1$;
(2) Algoritmos em pseudo-código;
(3) Modelo de Circuitos Quânticos

Esboço da Prova: A equivalência entre (1) e (2) decorre dos Teoremas 7.2.2 e 7.2.1, uma vez que algoritmos em pseudo-código, se escritos com rigor, podem ser convertidos em código assembly. Para provar a equivalência entre (2) e (3), seria necessário definir exatamente o que é um circuito quântico. Embora não apresentemos a definição completa aqui, ressaltamos que a ideia principal é que, no modelo de circuitos quânticos, um algoritmo é representado por uma matriz e a entrada por um vetor. Neste modelo, a execução do algoritmo corresponde à multiplicação do vetor pela matriz. Como é possível escrever um programa para realizar multiplicação de matrizes, a equivalência é válida. □

Mais do que compreender os detalhes da demonstração do Teorema da Equivalência, o essencial neste ponto é destacar que as correspondências apresentadas constituem um **Teorema**. Além das equivalências enunciadas, também é possível demonstrar que as Máquinas de Turing são equivalentes a diversos outros modelos de computação. No quadro MUITO MAIS EQUIVALÊNCIAS, apresentamos alguns desses modelos. Alguns têm interesse essencialmente teórico, como o *cálculo* λ e as *funções* μ*-recursivas*, enquanto outros representam processos físicos, químicos ou biológicos, como o modelo de *circuitos quânticos*, o modelo *ATAM*[7], o modelo *SCRN*[8] e o modelo *P Systems*[9].

MUITO MAIS EQUIVALÊNCIAS

O enunciado do Teorema 7.3.1 poderia ser expandido para incluir muitos outros modelos de computação. A seguir, uma lista de alguns destes modelos:

(1) Máquinas de Turing com fitas n-dimensionais, em vez de fitas unidimensionais;
(2) Qualquer linguagem de programação que rode em computadores conhecidos;
(3) Cálculo λ;
(4) Funções μ-recursivas;
(5) O modelo *P-systems* para computação celular;
(6) O modelo *SCRNs* para computação por reações químicas;
(7) Modelo ATAM para computação com moléculas de DNA;
(8) Modelo de Circuitos Quânticos e Máquinas de Turing Quânticas.

[7] O *Abstract Tile Assembly Model* (ATAM) é um modelo matemático de computação baseado na auto-organização de pequenos blocos segundo regras de ligação entre suas faces. Ele foi proposto para capturar o comportamento informacional de sistemas de automontagem, como moléculas de DNA [Win98].

[8] O modelo de *Stochastic Chemical Reaction Networks* (SCRN) descreve computação como um processo químico no qual espécies interagem por meio de reações elementares. Cada reação transforma um conjunto de moléculas em outro, obedecendo às leis da cinética química [Sol+08].

[9] Os *P systems*, ou *membrane systems*, são um modelo de computação inspirado na estrutura hierárquica das células biológicas. A computação ocorre dentro de membranas aninhadas, onde objetos simbólicos evoluem e são transportados entre regiões conforme regras de reescrita [Pău00].

7.4 O que afirma a Tese de Church-Turing

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 interpretação é que a TCT é uma definição matemática. A segunda intepretação é da TCT como uma afirmação sobre o mundo físico[10].

7.4.1 A Tese de Church-Turing: Uma 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 ε's e δ's, como normalmente vemos em um curso de Cálculo, exprime corretamente o que entendemos *intuitivamente* por continuidade de funções[11]".

Note que a ideia intuitiva que temos de funções contínuas, isto é, funções que podemos "desenhar sem tirar a caneta do papel", é subjetiva. Essa subjetividade impede que possamos fazer descobertas matemáticas sobre elas. Como parece evidente que toda função contínua, segundo essa intuição, pode ser caracterizada pela definição em termos de ε's e δ's que aprendemos em um curso de Cálculo, então **definimos** funções contínuas dessa maneira. Portanto, por definição, uma função é contínua se pode ser caracterizada dessa forma e não é contínua caso contrário.

> A DEFINIÇÃO DE CÍRCULO
>
> A ideia de que uma definição é "obviamente" a definição correta para determinado conceito pode parecer problemática, pois parece haver um salto entre a nossa intuição e a formulação matemática. No entanto, esse salto sempre ocorre quando buscamos apresentar uma definição formal para qualquer conceito. Isso vale para a definição de algoritmos, vale para a definição de funções contínuas e vale até mesmo para definições aparentemente muito simples, como a definição do que entendemos por um círculo.
>
> Para esclarecer essa questão, observe que, embora todos nós tenhamos uma ideia intuitiva do que seja um círculo como um "objeto perfeitamente redondo", para compreendê-lo com precisão e formular afirmações matemáticas rigorosas sobre ele, precisamos de uma definição exata. Uma possível definição seria a seguinte: para todo $r \in \mathbb{R}$ e todo $(c_1, c_2) \in \mathbb{R}^2$, um círculo é o conjunto C de pontos de $\mathbb{R}^2$ que estão à distância r de (c_1, c_2). Alguém poderia questionar se essa é realmente a definição que temos em mente ao pensar em um círculo, mas os matemáticos intuem que qualquer outra definição razoável de círculo acaba sendo, de alguma forma, equivalente a esta. Observe que ao afirmar isso, estamos sustentando a tese de que essa é a definição correta para o conceito intuitivo de círculo.

O ponto-chave é que, como ocorre com outras definições matemáticas, se existe uma máquina de Turing que sempre para para decidir um problema, então, por definição, existe um algoritmo para ele; caso contrário, também por definição, não existe algoritmo para esse problema.

> COMPUTABILIDADE DEFINIDA EM TERMOS MECÂNICOS
>
> *"Computabilidade mecanicamente definida foi estabelecida por Alan Turing sem dúvida alguma."* – Kurt Godel

[10] O termo "afirmação sobre o mundo físico", embora informal, expressa adequadamente a ideia de uma afirmação que pode, em princípio, ser verificada experimentalmente.

[11] Para quem não lembra, segue a definição: Dada $f : \mathbb{R} \to \mathbb{R}$ uma função e A um intervalo de $\mathbb{R}$. Dizemos que f é *contínua em I* se para todo $a \in A$, é verdade que $\forall \varepsilon > 0, \exists \delta > 0$ tal que $|x - a| < \delta \Rightarrow |f(x) - f(a)| < \varepsilon$.

7.4.2 A Tese de Church-Turing: Uma afirmação empiricamente verificável

Uma segunda interpretação é a de que a TCT constitui uma *afirmação empiricamente verificável*. Diferentemente da interpretação anterior, segundo a qual a TCT é entendida como uma definição matemática e, por isso, não pode ser classificada como verdadeira ou falsa, mas apenas como mais ou menos adequada para formalizar a noção intuitiva de algoritmo, essa leitura atribui à tese um conteúdo factual. Assim, ela pode, ao menos em princípio, ser verdadeira ou falsa. Essa leitura tem a vantagem de fornecer um critério empírico para avaliar a tese, que é a possibilidade de confronto com evidências experimentais: a tese deve ser revista caso evidências indiquem que ela não descreve corretamente os limites físicos da computação (veja o quadro A TESE DE CHURCH–TURING É UMA AFIRMAÇÃO CIENTÍFICA? para mais detalhes). O enunciado da tese é o seguinte:

> 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 6, vimos que existe um problema de decisão, conhecido como *problema da parada*, que não pode ser resolvido por máquinas de Turing. Uma vez que uma consequência da TCT é que nenhum objeto fisicamente realizável seria capaz de resolver esse problema, sabemos exatamente que tipo de evidência empírica poderia refutá-la: a construção de um aparato físico capaz de decidir, de forma consistente, o problema da parada. A posição mais comum atualmente, apoiada em múltiplas linhas de argumentação, tanto no que conhecemos sobre as leis da física quanto nas teorias físicas atualmente em desenvolvimento, é a de que a existência de tal aparato é improvável.

No enunciado da TCT, ao nos referirmos a um problema computacional, não estamos limitando o conceito apenas a problemas de decisão. Nesse contexto, tratamos de algo mais geral: uma relação entre certos conjuntos de entradas e saídas possíveis. Por essa razão, a TCT possui uma implicação particularmente interessante. Como o estado de qualquer objeto físico pode ser interpretado como a instanciação de alguma informação, representada por uma string que descreve esse estado, segue-se que a evolução temporal de qualquer objeto físico que realize uma computação bem definida, independentemente de sua complexidade, pode ser simulada por uma Máquina de Turing.

> A TESE DE CHURCH–TURING É UMA AFIRMAÇÃO CIENTÍFICA?
>
> A ideia de que uma afirmação deve ser, em princípio, falsificável para ser considerada científica está associada ao *critério de demarcação* proposto por Karl Popper [Pop59]. Esse critério, contudo, não é consensual na filosofia da ciência. Kuhn enfatiza o papel dos *paradigmas* [Kuh62], Lakatos propõe avaliar teorias no contexto de *programas de pesquisa* [Lak70], e Quine destaca que as hipóteses científicas não são, em geral, testadas isoladamente, mas como parte de uma rede de pressupostos [Qui51]. A Tese de Church–Turing pode ser compreendida à luz dessas diferentes perspectivas: embora sua formulação empiricamente verificável seja falsificável no sentido popperiano, sua aceitação histórica também pode ser compreendida à luz de outras perspectivas da filosofia da ciência, uma vez que diferentes modelos de computação (e.g., probabilística, paralela, quântica) foram incorporados ao quadro teórico sem que surgisse uma evidência de computação fisicamente realizável além da máquina de Turing.

Conexões entre computação e física

Uma importante fonte de evidência em favor da Tese de Church–Turing (TCT) vem de sua compatibilidade com as leis da física. Ao observarmos a natureza em seu nível mais fundamental e considerarmos como os objetos físicos se comportam, quais estados podem assumir, quais graus de liberdade possuem e como evoluem ao longo do tempo, as restrições impostas pelas leis da mecânica quântica parecem sustentar a tese. Um ponto-chave é que a descrição de um sistema físico pode ser aproximada, com precisão arbitrária, pelo modelo matemático conhecido como *circuito quântico*[a]. Sabe-se que tais modelos podem ser simulados por Máquinas de Turing, fato enunciado anteriormente no Teorema da Equivalência. A citação a seguir expressa de forma precisa a vantagem de se adotar essa leitura 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

[a] Apesar do nome *circuito quântico*, esse modelo não corresponde exatamente a um circuito no sentido usual; trata-se, na verdade, de um formalismo matemático para descrever a evolução temporal de sistemas quânticos.

As Leis da computação?

A Tese de Church–Turing, quando vista como uma hipótese empírica acerca dos limites fundamentais da computação, desempenha um papel semelhante ao que as leis da natureza desempenham nas ciências naturais, como a física, a química e a biologia. Por exemplo, enquanto as leis da física procuram caracterizar quais processos físicos podem ocorrer no universo, a Tese de Church–Turing procura caracterizar quais procedimentos efetivamente algorítmicos podem existir, exercendo um papel que pode ser entendido, por analogia, como o de *leis da computação*. Em outras palavras, assim como a posição e o movimento de qualquer partícula no universo devem obedecer às leis da física, a Tese de Church–Turing afirma que toda computação fisicamente realizável pode ser modelada por uma máquina de Turing. Assim, do mesmo modo que as leis da física delimitam o conjunto dos fenômenos fisicamente possíveis, a Tese de Church–Turing delimita o conjunto das computações efetivamente realizáveis.

Observamos, entretanto, que a expressão leis da computação *não é uma terminologia usual na literatura* e não a estamos empregando aqui para introduzir um conceito formal da teoria da computação. Ela é utilizada apenas como uma analogia didática, com o objetivo de destacar o papel desempenhado pela Tese de Church–Turing. A ideia é que, assim como as leis da física estabelecem os limites do que pode ocorrer no mundo físico, a Tese de Church–Turing estabelece os limites do que pode ser realizado por meio de procedimentos efetivamente algorítmicos.

Finalmente, é importante esclarecer que, embora a Tese de Church–Turing não seja uma consequência das leis da física, ela é plenamente *compatível* com elas. Atualmente, a mecânica quântica é considerada a teoria mais fundamental para a descrição dos fenômenos físicos microscópicos e, sob hipóteses bastante gerais, a evolução de um sistema físico descrito por essa teoria pode ser aproximada, com precisão arbitrária, pelo modelo matemático conhecido como *circuito quântico*. Os circuitos quânticos constituem, portanto, um modelo de computação diretamente inspirado na nossa melhor descrição das leis da natureza. Ainda assim, eles não ampliam o conjunto dos problemas computáveis: toda computação quântica pode ser simulada por uma máquina de Turing. Esse fato fornece uma importante evidência empírica em favor da Tese de Church–Turing.

Além de a TCT, enquanto afirmação empiricamente verificável, estar amparada pelo que sabemos de concreto sobre a mecânica quântica, ela também parece tem sustentação em resultados provenientes de áreas na fronteira da física teórica (veja o quadro CONTÍNUO VS DISCRETO).

CONTÍNUO VS DISCRETO

A Tese de Church-Turing, como qualquer tese científica, é passível de debate. Existe uma área da computação conhecida como *hipercomputação*, dedicada ao estudo de modelos que não podem ser simulados por Máquinas de Turing e que, portanto, questionam a TCT. Entretanto, a maioria das propostas que desafiam a TCT são tipicamente variações da antiga ideia de computação analógica[1], uma ideia que parece esbarrar em obstáculos impostos pela física teórica contemporânea. Em particular, o resultado mais importante nessa linha, demonstrado na década de 1970, é conhecido como *limitante de Bekenstein* [Aar13]. Embora alguns parâmetros usados na mecânica quântica sejam contínuos, o limitante de Bekenstein impõe um limite à quantidade de informação — que pode ser entendida em termos do número de estados observáveis em um sistema — que uma região finita do espaço pode conter.

[1] Um dispositivo analógico, no sentido empregado aqui, seria capaz armazenar dados que requerem uma quantidade infinita de informação (por exemplo, armazenar o número π com todos os seus infinitos dígitos) e recuperar essa informação sem erros. No momento, não há comprovação de que seja possível realizar tais tarefas.

Uma consequência bastante discutida da Tese de Church–Turing é a afirmação de que os *cérebros humanos, sendo objetos físicos, podem ser simulados por Máquinas de Turing*. Essa i-deia, como era de se esperar, gera algumas controvérsias por envolver o fator humano.

É importante observar que essa afirmação não significa que já saibamos realizar tal simulação, pois ainda não compreendemos plenamente o funcionamento do cérebro. Entretanto, essa limitação não impede supor que o cérebro obedeça às leis da física e possa, em princípio, ser simulado por um modelo computacional adequado. Essa ideia remonta ao próprio nascimento da computação: em seu célebre artigo sobre o *Teste de Turing* [Tur50], Alan Turing já sugeria que o cérebro humano é um tipo de máquina, no sentido de que segue leis físicas determinísticas ou probabilísticas. Ele, contudo, não reduz explicitamente a mente (no sentido filosófico) a uma Máquina de Turing, mantendo sua discussão no nível comportamental e funcional, e não metafísico.

INTELIGÊNCIA, CONSCIÊNCIA E MENTE

Convém observar que conceitos como *inteligência*, *consciência* e *mente* são fundamentais para a compreensão do ser humano. A Tese de Church–Turing, tal como normalmente é enunciada, não faz nenhuma referência a esses conceitos. Para entender o ponto-chave aqui, vamos usar uma analogia com a física: embora o cérebro humano, enquanto sistema físico, esteja sujeito às leis da física, isso não implica que conceitos como inteligência, consciência e mente se reduzam à descrição dos fenômenos físicos. Da mesma forma, a Tese de Church–Turing caracteriza os limites da computação sem necessariamente implicar uma redução de conceitos como consciência ou pensamento ao modelo matemático da máquina de Turing.

Como já mencionamos, quando o fator humano está envolvido, além do conceito de cérebro, entram também em jogo os conceitos de mente e de consciência. Esses conceitos vão além da descrição puramente física do cérebro e pertencem a debates filosóficos que extrapolam o escopo da Tese de Church–Turing. Esse tema é amplamente estudado na área da *filosofia da mente* [Cha96; Fes19; Sea92]. Para o leitor interessado neste assunto, sugerimos o livro de Edward Feser [Fes19], que oferece uma introdução acessível e amplamente referenciada ao assunto, apresentando as principais correntes da filosofia da mente, como o dualismo, o materialismo, o funcionalismo e o hilomorfismo, entre outras.

7.5 A Tese de Church-Turing Estendida

Nesta seção apresentaremos uma segunda afirmação que, ao contrário da TCT, **não** é a posição normalmente aceita pela comunidade científica. Mas por que discutir uma afirmação que possivelmente não é correta? O ponto é que compreender essa segunda afirmação, conhecida como *Tese de Church-Turing Estendida*, é importante para entendermos desenvolvimentos recentes em teoria da computação, especialmente na área de computação quântica.

O teorema a seguir enuncia um fato importante sobre a equivalência entre Máquinas de Turing e outros modelos de computação.

Teorema 7.5.1 — Teorema da Equivalência Polinomial. No Teorema da Equivalência, além de ser verdade que os modelos (1) e (2) são equivalentes, também é verdade que eles são polinomialmente equivalentes.

Enunciamos o Teorema 7.5.1 de forma um pouco vaga, sem definir exatamente o que queremos dizer por *polinomialmente equivalentes*. No entanto, alunos familiarizados com análise de algoritmos compreendem o sentido do enunciado: não é possível definir um algoritmo em pseudo-código cujo número de passos necessários para a execução seja exponencialmente menor do que o número de transições realizadas por uma Máquina de Turing que o simula.

Note que o item (3) foi deixado de fora do enunciado do Teorema 7.5.1. Isso ocorre porque se conjectura que Máquinas de Turing *não* simulam Circuitos Quânticos com eficiência polinomial. O modelo de Circuitos Quânticos é, até o momento, o único modelo com contrapartida física conhecido para o qual tal conjectura é levantada. Esse modelo constitui a base teórica da pesquisa em computação quântica.

A construção de computadores quânticos é possível em princípio, embora alguns pesquisadores questionem essa possibilidade. Tal questionamento significa afirmar que o modelo de Circuitos Quânticos não seria fisicamente realizável[12]. Esses pesquisadores sustentam que a TCT não apenas é válida, mas que ela é ainda mais sólida do que o consenso atual. Essa afirmação, conhecida como *Tese de Church-Turing Estendida* (TCTE), formulada na década de 1960, diz 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*. Nesse contexto, a palavra "eficiente" significa "polinomial".

Ao contrário da TCT, a TCTE não é consenso científico, pois sua aceitação implicaria considerar o modelo de Circuitos Quânticos como não realista. O consenso atual, entretanto, é que o modelo de Circuitos Quânticos é uma consequência direta das leis da física, conforme são compreendidas atualmente.

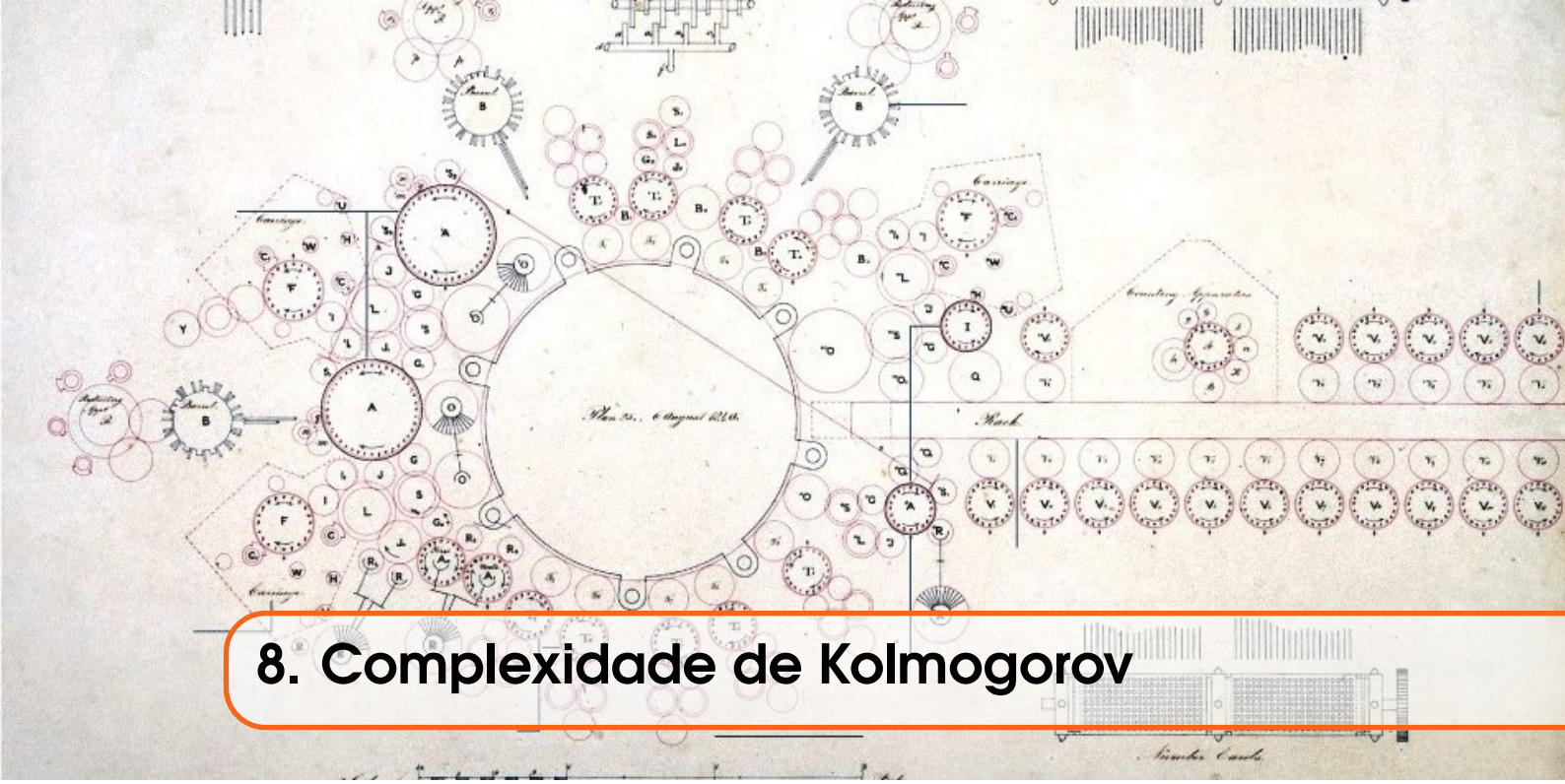
[12] Pequenos computadores quânticos (isto é, contendo poucos *qubits*) e computadores quânticos maiores, mas de propósito específico, já foram construídos. O que se questiona é se o modelo é escalável.

ALGORITMOS QUÂNTICOS

Atualmente, existem problemas para os quais se conhecem algoritmos quânticos (isto é, algoritmos descritos por circuitos quânticos) exponencialmente mais eficientes do que os melhores algoritmos clássicos conhecidos. Quando falamos em algoritmos clássicos, referimo-nos a Máquinas de Turing ou a programas escritos em linguagens de programação convencionais. Contudo, ninguém foi capaz de provar matematicamente que não existam algoritmos clássicos polinomiais para tais problemas. Ainda assim, trabalha-se com a conjectura de que eles não existam e de que o modelo de computação quântica seja exponencialmente mais eficiente do que o modelo clássico para certos problemas específicos. Essa conjectura é expressa pela desigualdade $P \neq BQP$.

Mesmo que essa conjectura seja provada e se conclua que o modelo de computação quântica é inerentemente mais eficiente que o modelo clássico, e ainda que a construção de computadores quânticos viáveis se torne realidade, como se espera, o resultado seria apenas a refutação da TCTE. Por outro lado, **a TCT permaneceria completamente intacta**, pois, em termos do que é possível computar (isto é, desconsiderando a eficiência), computadores quânticos e clássicos são equivalentes. Em outras palavras, o conjunto de problemas que podem ser resolvidos por computadores quânticos é exatamente o conjunto das linguagens recursivas.

Atualmente, embora computadores quânticos sejam capazes de resolver precisamente os mesmos problemas que os computadores clássicos, existe grande interesse em sua construção, já que alguns problemas que poderiam ser resolvidos de forma exponencialmente mais rápida por esses dispositivos são de enorme importância prática e teórica.



8. Complexidade de Kolmogorov

Neste capítulo, faremos uma breve pausa em nosso esforço para compreender o que são algoritmos, computadores e problemas computacionais, e utilizaremos o entendimento que adquirimos até aqui para lançar alguma luz sobre um conceito intimamente relacionado à computação: o conceito de informação.

8.1 Informação, complexidade e aleatoridade

Para introduzir o conceito de informação, vamos recorrer a uma pequena história. Suponha que temos acesso momentâneo a uma string de 480 bits e que nosso objetivo seja descrevê-la para outra pessoa com quem nos encontraremos mais tarde. Para isso, vamos anotá-la em um pedaço de papel. A pergunta que queremos responder neste capítulo é a seguinte: existem strings para as quais essa tarefa é mais difícil do que para outras? Vamos considerar o caso de três strings diferentes, w_1, w_2 e w_3, apresentadas a seguir:

$w_1 =$ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000

$w_2 =$ 000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111

$w_3 =$ 001101001000100111001101001000111100111011111011011110101111110001100111101010111
101011100011100011100101010000010011011010110110100111110011110000001101101111000
000111011010100110100111010000000100110111110101110011101000001011101011011111000
101001111101110011100111111010001001100110011010000100111101010111100110100110010
000100011110000010001110110000101100001011110000101110101110100000101111001000101
110111000001000110010010011110111011001101000001000110100110000000010001110110101

Supondo que queremos descrever **exatamente** a string que vimos, em qual dos três cenários teríamos mais dificuldade para realizar a tarefa? Intuitivamente, w_1 parece ser a string mais fácil de anotar em um pedaço de papel, pois contém pouquíssima informação relevante. Para descrevê-la bastaria anotar a frase *"480 vezes o bit* 0*"* no pedaço de papel.

Um outro extremo é a string w_3, que, por não apresentar nenhum padrão discernível, seria a mais difícil de descrever de maneira compacta, pois não haveria muita alternativa além de anotá-la bit a bit no pedaço de papel. A string w_2 representa um caso intermediário, ainda que razoavelmente próximo ao de w_1, em que há pouca informação útil. As descrições poderiam ser:

Descrição de w_1: "*480 bits* 0"

Descrição de w_2: "*80 bits* 0 *e 80 bits* 1*; repita isso 3 vezes*"

Descrição de w_3: "*a sequência de bits 00110100100010011100110100100011110011101111101101111110001100111101010111010111000111000111001010100000100110110101101101001111100111100000011011011110000011101101010011010011101000000010011011111010111001110100000101110101101111100101001111101110011100111111010001001100110011010000100111101010111100110100110010001000111100000100011101100001011000010111000010111010111010000010111100100010110111000001000110010010011110110110011010000010001101001100000000100011101101012*"

Ao contrário do conceito de computação, que pela Tese de Church-Turing possui uma definição única e universal, o conceito de informação é frequentemente definido de diferentes maneiras, dependendo do contexto ou da área da ciência em que se tenta esclarecer o que significa informação. Independentemente de como queiramos definir o conceito de informação, a questão central aqui, exemplificada pela história apresentada no início do capítulo, é que strings diferentes parecem estar associadas a **quantidades distintas** de informação. Neste capítulo, buscaremos formalizar essas ideias utilizando as ferramentas desenvolvidas ao longo desta Parte 2 do livro.

A forma como, de modo informal, relacionamos a quantidade de informação contida em cada uma das três strings w_1, w_2 e w_3 à sua menor descrição possível está no cerne do conceito de *Complexidade de Kolmogorov*. Veremos que a primeira string possui baixa Complexidade de Kolmogorov (pois tem uma descrição curta), a segunda apresenta uma complexidade um pouco maior, e a terceira possui, em certo sentido, a maior complexidade possível.

O QUE É ALGO COMPLEXO?

O nome **Complexidade** de Kolmogorov sugere relacionar a quantidade de informação contida em uma string com a ideia de complexidade da própria string. Essa escolha é intencional e nos conduz a questões bastante profundas em diversas áreas da ciência.

O problema de definir o que significa algo ser simples ou complexo tem sido amplamente estudado em diferentes campos científicos. As ferramentas que veremos aqui representam apenas a proverbial "ponta do iceberg" do que se sabe sobre esse tema, sob a ótica de uma área da computação conhecida como *teoria algorítmica da informação*.

Na própria teoria algorítmica da informação, entretanto, existem diferentes maneiras de medir a complexidade de uma string. Essas abordagens refinam nossa intuição sobre a relação entre informação e complexidade. Embora esse tema ultrapasse o escopo deste livro, vale mencionar que algumas dessas abordagens aprimoram a maneira como a complexidade é definida aqui. A limitação da abordagem vista aqui fica clara no exemplo da terceira string. Apesar de a string não possuir uma descrição curta, ela não parece realmente complexa, pois se

assemelha a uma string aleatória, algo que poderíamos facilmente produzir se não estivéssemos restritos a replicar exatamente a string em questão.

O próprio Andrey Kolmogorov, na década de 1960, percebeu essa limitação e propôs uma variação de sua teoria para lidar com a ideia de que a verdadeira complexidade não está nem no extremo do caótico (strings aleatórias), nem no extremo da ordem (strings com padrões triviais). Existem várias maneiras de abordar essa questão, sendo que a ideia mais comum é, como já mencionamos, não trabalharmos com descrições exatas das strings, e sim com descrições probabilísticas. Neste caso, uma descrição curta para w_3 seria: "sorteie 400 bits aleatórios".

De qualquer forma, a abordagem que veremos neste capítulo, baseada em descrições exatas das strings, será suficiente para entendermos como os conceitos da teoria da computação podem ser aplicados para compreender melhor noções como informação e complexidade.

8.2 A descrição mínima de uma string

Vamos agora ser mais precisos e formalizar o que queremos dizer com a descrição de uma string x. A ideia central é que, sendo a própria descrição de x também um objeto matemático, ela pode ser representada por uma string binária, que chamaremos de $d(x)$. Por exemplo, a frase que descreve a string w_1, isto é, "*800 bits* 0", pode ser codificada em binário; assim, a string $d(w_1)$ corresponde a essa codificação binária. A pergunta que queremos fazer é a seguinte: qual é a menor descrição possível de uma dada string x?

Primeiramente, observe que a ideia é que seja sempre possível recuperar a string original x a partir de sua descrição $d(x)$. Dessa forma, podemos interpretar $d(x)$ como uma versão "compactada" de x. A maneira de formalizar isso será em termos de computação. Mais precisamente, queremos identificar o menor processo algorítmico capaz de "construir" uma dada string x. Neste ponto do curso já dispomos de um vocabulário preciso para expressar essa ideia: buscamos a menor Máquina de Turing que escreve x em sua fita e, em seguida, para.

8.2.1 Definição de Complexidade de Kolmogorov

A melhor maneira de sermos precisos ao dizer "a menor Máquina de Turing" para realizar alguma tarefa é nos referirmos à menor string $\llcorner M \lrcorner$ tal que a máquina M realize a tarefa designada. Surge, entretanto, uma questão importante: o que exatamente queremos dizer com "a menor Máquina de Turing que escreve x em sua fita"? Queremos nos referir à menor máquina M que inicia a computação com a fita vazia e escreve nessa fita a string x? E se existir uma outra máquina M' muito menor que produza x em sua fita ao receber como entrada uma string w muito curta? Nesse caso, tendo em mãos as strings $\llcorner M' \lrcorner$ e w, seríamos capazes de descrever x.

Com isso em mente, vamos definir a menor descrição de x de forma um pouco mais geral do que simplesmente encontrar uma Máquina de Turing mínima. Definiremos a menor descrição de x como a menor string $\llcorner M \lrcorner w$ tal que $M(w) = x$. Observe que essa definição é mais abrangente, pois permite que, em casos particulares, tenhamos $w = \varepsilon$, de modo que a descrição de x seja simplesmente uma Máquina de Turing mínima.

Definição 8.2.1 — A descrição de uma string. Uma *descrição* de x é uma string $\llcorner M \lrcorner w$ tal que M é uma máquina de turing e $w \in \Sigma^*$ e $M(w) = x$.

Definição 8.2.2 — Descrição mínima. Uma *descrição mínima* de uma string x, denotada $d(x)$, é a menor string $\llcorner M \lrcorner w$ em ordem lexicográfica tal que $\llcorner M \lrcorner w$ é uma descrição de x.

Definição 8.2.3 — Complexidade de Komogorov. A *Complexidade de Komogorov* de uma string x, denotada $K(x)$, é o tamanho da descrição mínima de x, ou seja, $K(x) = |d(x)|$.

Teorema 8.2.1 Existe uma constante c tal que $\forall x \in \Sigma^*,\ K(x) \leq |x| + c$.

Prova: Considere a MT M tal que $\forall x, M(x) = x$, ou seja, a máquina aceita toda string de Σ^* e para com esta string em sua fita. Seja $c = |\llcorner M \lrcorner|$. Com isso $\llcorner M \lrcorner\, x$ é uma descrição de x de tamanho $|x| + c$ e portanto $K(x) \leq |x| + c$.

8.3 Incompressibilidade de informação

Nesta seção vamos mostrar que existem strings que não podem ser comprimidas e entender a relação destas strings com strings aleatórias.

Definição 8.3.1 — Strings compressíveis e incompressíveis. Uma string é x é *c-compressível* se $K(x) \leq |x| - c$. Caso x não seja c-compressível, dizemos que x é *c-incompressível*. Se uma string for 1-incompressível, diremos simplesmente que ela é *incompressível*.

Teorema 8.3.1 Existem strings incompressíveis de todos os tamanhos.

Prova: Vamos contar o número de strings de tamanho n e comparar com o número de descrições menores do que n:

- Número de strings de tamanho n: 2^n.
- Número de descrições menores que n: $\sum_{i=0}^{n-1} 2^i = 2^n - 1$.

Como $2^n > 2^n - 1$, existem mais strings x de tamanho n do que possíveis descrições $d(x)$ destas strings, tal que $|d(x)| < n$, existe pelo menos uma string x de tamanho n que não admite nenhuma descrição menor do que n. Portanto x é incompressível. □

8.3.1 Strings incompressíveis e aleatoriedade

Vamos agora estudar a conexão entre strings que possuem alta Complexidade de Kolmogorov e strings que parecem tipicamente aleatórias. Essa conexão reflete nossa intuição de que é difícil dar uma descrição pequena e exata de uma string que pareça aleatória.

Relembrando as três strings diferentes que vimos anteriormente, tivemos a impressão de que a terceira string era aleatória, enquanto as outras duas não eram. Entretanto, se escolhêssemos uma string de 480 bits ao acaso, todas teriam exatamente a mesma probabilidade de serem selecionadas, mais precisamente, 2^{-480}. Para mostrarmos que strings com alta Complexidade de Kolmogorov se assemelham a strings aleatórias, enfrentamos logo um problema: o que é uma string aleatória? Essa pergunta parece um tanto intangível, já que qualquer string pode ser obtida ao fazermos uma escolha aleatória. O que faremos aqui é contornar essa dificuldade de uma maneira simples.

Definição 8.3.2 — Propriedades. Uma *propriedade* é uma função booleana $P : \Sigma^* \to \{0,1\}$. Dada uma string $x \in \Sigma^*$, se $P(x) = 1$, diremos que x tem a propriedade P ou que a propriedade P é verdadeira para x. Caso contrário, diremos que a string x não tem a propriedade P ou que a propriedade P é falsa para x.

Em outras palavras, uma propriedade é essencialmente a mesma coisa que uma função booleana. Por exemplo, suponha que a propriedade P seja "*ter a mesma quantidade de 0's e 1's*". Essa propriedade pode ser vista como a função que mapeia para 1 as strings que têm a mesma quantidade

de 0's e 1's, e para 0 as demais. Assim, diremos que *a string 001 não tem a propriedade P*, pois $P(001) = 0$. Por outro lado, como $P(00110011) = 1$, diremos que a propriedade de *ter a mesma quantidade de 0's e 1's* é verdadeira para a string 00110011.

Definição 8.3.3 — Propriedade ubíqua. Uma *propriedade ubíqua* é uma propriedade P que satisfaz o seguinte critério: a fração de strings de tamanho n cuja propriedade P é falsa tende a 0 quando n tende ao infinito.

Observe que, quando tomamos aleatoriamente uma string longa o suficiente, a probabilidade de essa string ter uma propriedade ubíqua é alta. De fato, para qualquer propriedade ubíqua, a probabilidade de uma string escolhida ao acaso ter essa propriedade pode ser tão alta quanto quisermos, pois essa probabilidade tende a 1 quando o tamanho da string tende ao infinito. Em outras palavras, uma propriedade ubíqua é precisamente uma propriedade que uma string escolhida de maneira aleatória necessariamente terá, desde que escolhamos uma string suficientemente grande. Assim, faz sentido definir o seguinte:

Definição 8.3.4 — Propriedade de strings aleatórias. Se P é ubíqua, então dizemos que P é uma *propriedade de strings aleatórias*.

Teorema 8.3.2 Seja P uma propriedade computável de strings aleatórias. Seja I o conjunto de strings incompressíveis. Então, a quantidade de strings x em I tal que P é falsa é finita.

Prova: Primeiramente, consideremos o conjunto de strings em que a propriedade P é falsa e dividamos a prova em dois casos, dependendo de esse conjunto ser finito ou infinito.

Caso 1: O conjunto de strings em que a propriedade P é falsa é finito. Esse caso é trivial, pois se o conjunto $\{x \in \Sigma^* \ : \ P(x) = 0\}$ é finito, então o conjunto $\{x \in I \ : \ P(x) = 0\}$ também é finito, já que $I \subseteq \Sigma^*$. Portanto, o teorema está provado.

Caso 2: O conjunto de strings em que a propriedade P é falsa é infinito. A prova deste caso é mais interessante. Considere o algoritmo M que toma como entrada w e itera sobre todas as strings de Σ^* em ordem lexicográfica (ou seja, ε, 0, 1, 00, 01, 10, 11, 000, 001, ...) e retorna a $N(w)$-ésima string x tal que $P(x) = 0$. Note que o algoritmo sempre para, pois existem infinitas strings em que a propriedade P é falsa.

Dada uma string da lista de strings em que P é falsa, em ordem lexicográfica, denote por i_x o número natural correspondente à posição da string x nessa lista. O ponto central é que $\llcorner M \lrcorner \llcorner i_x \lrcorner$ é uma descrição da string x e que essa descrição tem tamanho $c + |i_x|$, onde $c = |\llcorner M \lrcorner|$.

Agora, tome um n suficientemente grande tal que $P(x) = 0$ para uma proporção de, no máximo, $\frac{1}{2^{c+2}}$ das strings de tamanho até n. Note que sempre podemos escolher tal n, pois essa fração tende a zero quando n tende ao infinito e porque existem infinitas strings em que P é falsa.

Entre o total de $2^{n+1} - 1$ strings de tamanho $\leq n$, o conjunto das que não satisfazem P tem tamanho no máximo $\frac{2^{n+1}-1}{2^{c+2}}$. Assim, dada uma string arbitrária x de tamanho n tal que $P(x) = 0$, o índice i_x de x é no máximo $\frac{2^{n+1}-1}{2^{c+2}}$.

Disso, obtemos $i_x \leq \frac{2^{n+1}-1}{2^{c+2}} \leq 2^{n-1-c}$. Logo, $|i_x| \leq n-1-c$ e, portanto, $|\llcorner M \lrcorner \llcorner i_x \lrcorner| \leq c + (n - 1 - c) = n - 1$. Assim, $K(x) \leq n - 1$, ou seja, x é compressível.

Em outras palavras, a partir de um certo n, concluímos que toda string x em que $P(x) = 0$ (note que tomamos x arbitrariamente) é compressível. Portanto, apenas strings de tamanho menor que n podem ter $P(x) = 0$ e pertencer ao conjunto I de strings incompressíveis. Concluímos, então, que I é finito, e a prova está completa. □

Com o devido cuidado, é possível demonstrar que a propriedade "ter uma sequência muito

longa de 0's" não é ubíqua e, portanto, strings escolhidas ao acaso têm baixíssima probabilidade de possuir essa propriedade. Um caso ainda mais extremo seria a propriedade "a string consiste apenas de 0's". Essa propriedade claramente não é ubíqua, pois a fração de strings de tamanho n com essa propriedade tende a 0 quando n tende ao infinito. Não sendo ubíqua, pela nossa definição esta propriedade não é uma **propriedade de strings aleatórias**, o que corresponde à nossa intuição de que a primeira string da primeira página desta seção, que tem baixa Complexidade de Kolmogorov, não "parece" ser aleatória.

Por outro lado, uma propriedade ubíqua que a terceira string considerada no começo desta seção possui é a de ter um número equilibrado de 0's e 1's. Intuitivamente, sabemos que uma string escolhida aleatoriamente tende a ter essa propriedade.

8.4 Incomputabilidade da Complexidade de Kolmogorov

Vamos finalizar este capítulo mostrando que não é possível computar a complexidade de Kolmogorov de uma dada string.

Teorema 8.4.1 Seja w uma string. Não existe MT M_K tal que $M(w) = K(w)$.

Proof. Suponha, por contradição, que existe uma Máquina de Turing M_K que computa a complexidade de Kolmogorov de uma string. Usando M_K podemos construir a seguinte Máquina de Turing D que, dada um número natural n em binário, faz:

(1) Enumera strings $x \in \{0,1\}^*$ em ordem lexicográfica;
(2) Para cada string x enumerada, usa M_K para calcular $K(x)$.
(3) Ao encontrar a primeira string x tal que $K(x) > n$, escreve x na fita e para.

Note que, para toda entrada $\llcorner n \lrcorner$ a máquina D retorna alguma string w_n com $K(w_n) > n$ e para. Ou seja, $D(\llcorner n \lrcorner) = w_n$. Com isso, existe uma uma string w_n com descrição de tamanho $|D| + |\llcorner n \lrcorner|$. Observe que $|\llcorner n \lrcorner|$ tem no máximo $\log_2 n + 1$ bits. Portanto, para alguma constante $c > \llcorner D \lrcorner$, temos $K(w_n) \leq \log_2 n + c$ (a constante c absorve detalhes da codificação e separadores). Por outro lado, pela construção de D (em particular, em *(3)*) temos $K(w_n) > n$. Assim $n < K(w_n) \leq c + \log_2 n$.

Note que para n suficientemente grande esta desigualdade é impossível, já que o lado direito cresce apenas como $\log n$ enquanto o lado esquerdo cresce linearmente em n, uma contradição. Portanto, a hipótese de que M_K computa K é falsa. Concluímos que a função $K(w)$ não é computável. ■

Exercícios

Exercício 8.1 Apresente uma Máquina de Turing M com apenas um estado tal que, para todo $x \in \Sigma^*$, $M(x) = x$. ■

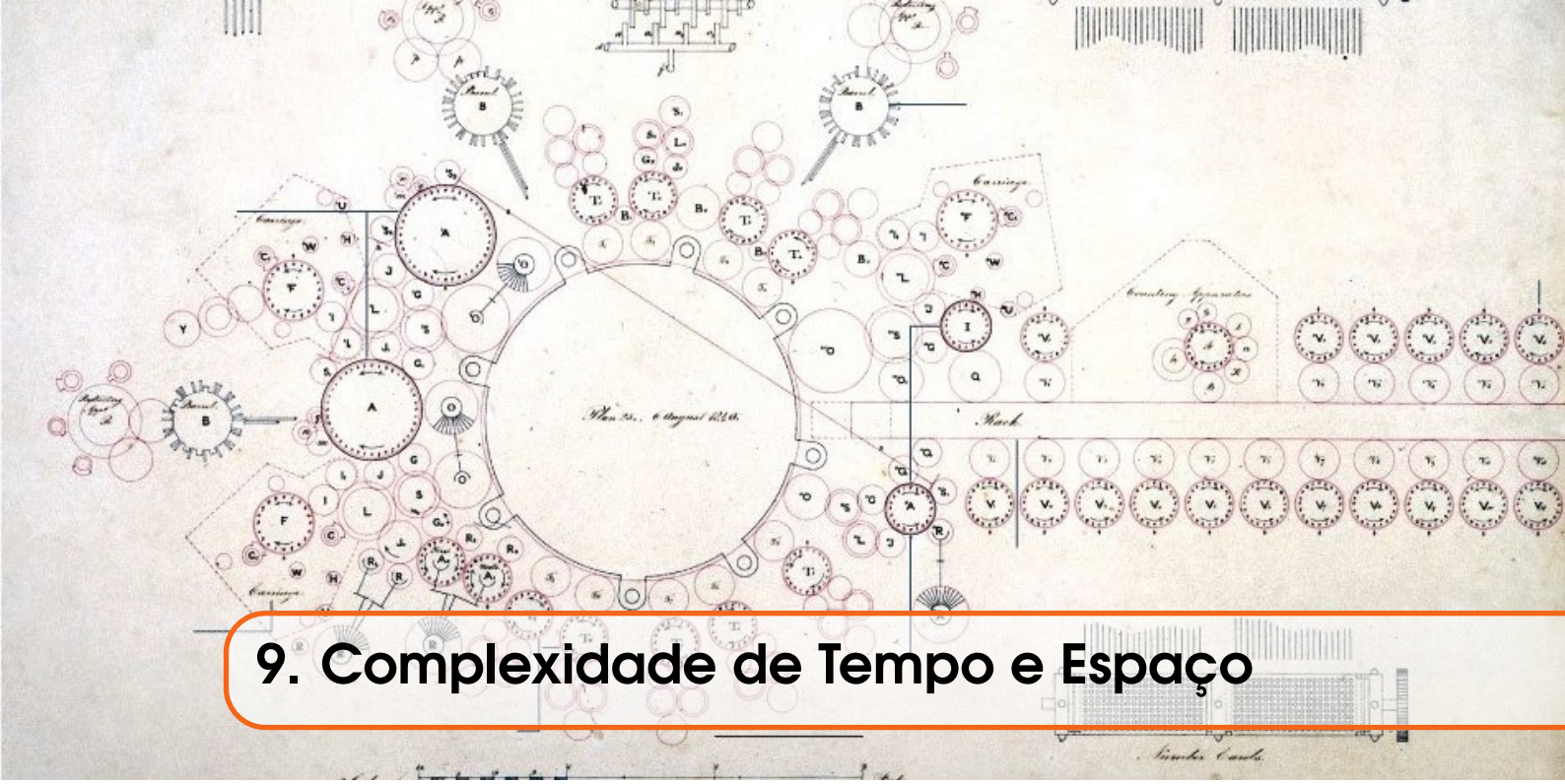
Exercício 8.2 Prove que, para todo $x \in \Sigma^*$, $K(xx) \leq K(x) + c$, em que c é uma constante. ■

Exercício 8.3 Prove que, para todos $x, y \in \Sigma^*$, $K(xy) \leq K(x) + K(y) + c$, em que c é uma constante. ■

Exercício 8.4 Prove a seguinte afirmação: pelo menos $2^n - 2^{n-c+1} + 1$ strings de tamanho n são c-incompressíveis. ■

Exercício 8.5 Prove que existe uma constante c tal que, para todo $x \in \Sigma^*$, a string $d(x)$ é c-incompressível. Em outras palavras, exceto por uma constante aditiva, a descrição mínima $d(x)$ de uma string x já representa a compressão máxima possível de x. Este fato está relacionado a algo que intuitivamente sabemos: a experiência frustrante de tentar comprimir um arquivo que já foi comprimido por um bom compactador. ■

III Parte 3: Complexidade Computacional



9. Complexidade de Tempo e Espaço

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

(1) Dados dois números inteiros de 64 dígitos, calcular a soma dos dois números.
(2) Dado um tabuleiro de xadrez (que também possui 64 posições) 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. Se quisermos ser justos na comparação, podemos imaginar que os números que estamos somando têm 64 dígitos, de modo que estaríamos lidando com instâncias de tamanho aproximadamente comparável à instância representada pelo tabuleiro de xadrez (afinal, um tabuleiro de xadrez possui 64 posições). Ainda assim, com papel e caneta, em poucos minutos conseguimos calcular a soma dos dois números em questão. Por outro lado, para encontrar uma jogada ótima para as peças brancas[1] parece haver uma quantidade astronômica de possibilidades que precisamos considerar. Este segundo problema parece difícil porque, para todas as possíveis ramificações de jogo advindas das possíveis respostas do oponente, tentamos garantir que sempre exista uma próxima jogada ótima, e assim sucessivamente até o final do jogo.

Embora pareça evidente que, dados x e y, o problema de encontrar o número z tal que $z = x + y$ é fácil de resolver, é importante observar que o número procurado tem pelo menos 64 dígitos. Portanto, existem 10^{64} números com o tamanho da resposta desejada. Ou seja, o espaço de busca da solução para o valor que satisfaz $x + y$ é astronomicamente grande, assim como ocorre com o espaço de busca do problema da jogada de xadrez. Ainda assim, em poucos passos, conseguimos sistematicamente encontrar o valor z desejado.

Há algo ainda mais importante do que o fato de que somar dois números de 64 dígitos é mais fácil do que encontrar uma jogada ótima em um tabuleiro de dimensão 8×8. Considere os casos mais gerais dos dois problemas, isto é, o caso em que queremos somar dois números de n dígitos

[1] Neste contexto, dizemos que uma jogada é ótima se existe garantidamente um xeque-mate a partir dela, mesmo que o xeque-mate ocorra, por exemplo, quarenta jogadas adiante. Uma formulação mais precisa do problema seria: encontrar uma jogada ótima ou concluir que as peças pretas garantidamente podem empatar ou derrotar as peças brancas.

e o caso em que queremos encontrar uma jogada ótima em um tabuleiro de xadrez generalizado, jogado em um tabuleiro $n \times n$. O que ocorre é que a dificuldade do segundo problema, que já era astronomicamente maior, cresce exponencialmente com o tamanho do tabuleiro, enquanto a dificuldade do problema da soma cresce de maneira muito mais moderada.

O ponto fundamental é que, embora o espaço de busca nos dois casos cresça exponencialmente com o tamanho n da entrada, no primeiro problema temos um algoritmo muito eficiente para encontrar a solução dentro desse espaço de 10^n possibilidades, ao passo que, no segundo caso, o algoritmo para se encontrar a solução é essencialmente um algoritmo de força bruta, examinando todas as ramificações de jogadas. Nesta parte do curso, nosso objetivo é tornar mais precisa a ideia intuitiva de que alguns problemas são inerentemente mais difíceis de resolver do que outros.

> PROBLEMAS INDECIDÍVEIS *vs* PROBLEMAS INTRATÁVEIS
>
> Nesta parte do curso, os tipos de problemas com os quais estaremos lidando são chamados de *problemas intratáveis*. Tais problemas não admitem algoritmos eficientes, como ocorre no caso do problema do xadrez generalizado, ou, na maior parte dos casos, conjectura-se que não admitem algoritmos eficientes, como no caso dos famosos problemas *NP*-completos. Ainda assim, estaremos lidando com problemas decidíveis.
>
> Podemos pensar que problemas indecidíveis, como o problema da parada, pertencem à categoria dos problemas "impossíveis" e não apenas "difíceis", como o problema do xadrez generalizado ou os problemas *NP*-completos, pois simplesmente não existem algoritmos para resolvê-los. Entretanto, é razoável considerar que instâncias muito grandes de problemas intratáveis, na prática, também podem ser impossíveis de serem resolvidas, devido a limitações físicas de espaço e de tempo que a natureza nos impõe.

9.1 Complexidade de Tempo e de Espaço de Máquinas de Turing

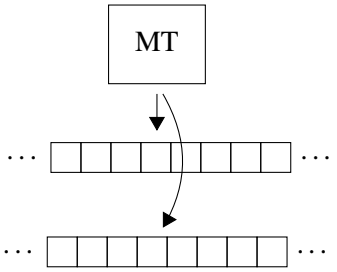
Para tornar precisa a ideia da quantidade de trabalho que um problema exige para que possamos o solucionar, iremos definir o conceito de complexidade de tempo e complexidade de espaço de Máquinas de Turing. A complexidade de tempo de uma máquina, se refere ao número de transições que ela executa em função do tamanho da entrada, enquando e a complexidade de espaço se refere a quantidade de células da fita utilizadas em função do tamanho da entrada.

Nesta parte do curso, será conveniente trabalhar com Máquinas de Turing com duas fitas. O Teorema 7.5.1 nos diz que MTs com uma fita, o modelo que vínhamos utilizando até agora, são equivalentes não apenas em termos de computabilidade, mas também em termos de eficiência[2]. O funcionamento das máquinas com duas fitas que utilizaremos aqui 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, sendo portanto do tipo "read-only".

(2) A segunda fita, chamada de *fita de trabalho*, é uma fita padrão na qual se permite leitura e escrita. A ideia é que nesta fita ocorre a computação propriamente dita.

[2] A 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 duas fitas, isto é, dada uma string de entrada de n bits, o número de transições executadas pela MT de duas fitas 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 é importante para o tipo de questões que estaremos discutindo nesta parte do curso.



Neste modelo em que estamos trabalhando, vamos assumir que, ao final da computação, a segunda fita contém a saída do algoritmo. Portanto, quando escrevemos $M(x) = y$, queremos dizer que, com a entrada x na fita de entrada, o algoritmo termina com y na segunda fita. No caso de problemas de decisão, vamos assumir que a máquina escreve o símbolo 1 ou 0 na segunda fita antes de parar, dependendo de estar aceitando ou rejeitando a string.

Definição 9.1.1 — Complexidade de tempo. A complexidade de tempo de uma Máquina de Turing M é uma função $t_{\mathrm{M}} : \mathbb{N} \to \mathbb{N}$ tal que, para qualquer string de entrada de tamanho n, a máquina para depois de executar no máximo $t_{\mathrm{M}}(n)$ transições.

■ **Exemplo 9.1** MT M_{SOMA} para somar que vimos anteriormente. Para qualquer entrada de tamanho n, M_{SOMA} sempre para depois de fazer, no máximo, $2n+1$ transições. Portanto, a complexidade de tempo de M_{SOMA} é a função $t_{M_{\mathrm{SOMA}}}(n) = 2n+1$. ■

Definição 9.1.2 — Complexidade de espaço. Dada uma Máquina de Turing M, sua complexidade de espaço é uma função $s_{\mathrm{M}} : \mathbb{N} \to \mathbb{N}$ tal que, para qualquer string de entrada de tamanho n, a máquina M para depois de usar no máximo $s_{\mathrm{M}}(n)$ posições da fita de trabalho.

■ **Exemplo 9.2** Suponha que uma MT M, para toda entrada de tamanho n, sempre para usando no máximo $\log n + 7$ posições da fita 2. Neste caso dizemos que a complexidade de espaço de M é $\log n + 7$. ■

Assim como ocorre na análise de algoritmos, em muitos casos poderemos usar notação assintótica. No Exemplo 9.1, dizemos que a complexidade de tempo de M é $O(n^2)$. No Exemplo 9.2, dizemos que a complexidade de espaço de M é $O(\log n)$.

Notação 9.1. *A notação $poli(n)$ 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 $f(n) = poli(n)$.*

Definição 9.1.3 — Complexidade de tempo ou espaço polinomial. Se uma Máquina de Turing M tem complexidade de tempo $poli(n)$, dizemos que *M é polinomial*. Se M tem complexidade de espaço $poli(n)$, dizemos que *M é de espaço polinomial*. Note que a função $poli(n)$ não precisa necessariamente ser um polinômio, basta que a função seja assintoticamente limitada por um polinômio.

Definição 9.1.4 — Complexidade de tempo polinomial 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 têm profundidade $poli(n)$.

Definição 9.1.5 — Complexidade de tempo exponencial. Se uma Máquina de Turing M tem complexidade de tempo $O(2^{poli(n)})$, dizemos que M *é exponencial*.

Observe que, se uma Máquina de Turing não determinística é polinomial, então, independentemente das escolhas não determinísticas feitas ao longo da execução, ela sempre para depois de $poli(n)$ transições. Uma consequência da Definição 9.1.4 é que estamos lidando apenas com MTs não determinísticas que não possuem ramos infinitos em sua árvore de computações possíveis.

ESPAÇO E TEMPO: RECURSOS DISPONÍVEIS PARA SE FAZER COMPUTAÇÃO

Algo interessante de observar é que tempo e espaço são recursos básicos que a natureza nos oferece quando pensamos em realizar computação em um meio físico. Dependendo do substrato físico utilizado para fazer computação, espaço e tempo podem ser entendidos de maneiras distintas.

Por exemplo, em um algoritmo executado em um laptop, espaço corresponde à memória RAM e tempo corresponde ao número de instruções executadas pelo processador. No caso de computação usando moléculas de DNA, espaço corresponde à quantidade de moléculas necessárias para realizar a computação e tempo corresponde ao número de vezes que as moléculas devem interagir (isto é, formar pontes de hidrogênio). Independentemente do modelo de computação adotado, em última análise os recursos específicos utilizados parecem ter correspondência com os conceitos gerais de espaço e tempo conhecidos na física.

9.2 As classes P, NP, PSPACE e EXP

Na seção anterior definimos o que é a complexidade de uma Máquina de Turing. Nesta seção, vamos usar essa definição como base para definir a complexidade inerente de resolver determinados problemas de decisão.

Definição 9.2.1 — Decisão em tempo polinomial. Dizemos que uma linguagem L é *decidível em tempo polinomial* se existe uma MT polinomial que decide L. Dizemos que uma linguagem L é *decidível em tempo polinomial não determinístico* se existe uma MT não determinística polinomial que decide L.

Definição 9.2.2 — Decisão em espaço polinomial. Dizemos que uma linguagem L é *decidível em espaço polinomial* se existe uma MT de espaço polinomial que decide L. Uma linguagem L é *decidível em espaço polinomial não determinístico* se existe uma MT não determinística que decide L em espaço polinomial.

Exercício 9.1 (Decisão em tempo exponencial) Apresente uma definição para o conceito de *decisão em tempo exponencial*. ■

Vamos agora classificar linguagens de acordo com a quantidade de recursos necessários para decidí-las. Tais conjuntos de linguagens são conhecidos como *classes* de linguagens.

Definição 9.2.3 — A classe P. O conjunto de todas as linguagens decidíveis deterministicamente em tempo polinomial é denotado por P.

Definição 9.2.4 — A classe NP. O conjunto de todas as linguagens decidíveis não deterministicamente em tempo polinomial é denotado por NP.

Definição 9.2.5 — A classe PSPACE. O conjunto de todas as linguagens decidíveis deterministicamente em espaço polinomial é denotado por PSPACE.

Definição 9.2.6 — A classe EXP. O conjunto de todas as linguagens decidíveis deterministicamente em tempo exponencial é denotado por EXP.

Teorema 9.2.1 $\mathsf{P} \subseteq \mathsf{NP}$.

Prova: Seja $L \in \mathsf{P}$. Pela Definição 9.2.6, existe uma MT polinomial $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ que decide L. Considere agora 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 $L \in \mathsf{NP}$ e consequentemente $\mathsf{P} \subseteq \mathsf{NP}$. □

Teorema 9.2.2 $\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$.

Exercício 9.2 (Opcional) Prove o Teorema 9.2.2. ∎

9.3 O problema P *vs* NP

O Teorema 9.2.1 nos diz que $\mathsf{P} \subseteq \mathsf{NP}$. Seria possível provar que $\mathsf{P} \neq \mathsf{NP}$? Ou seja, seria possível provar que a classe P está *estritamente* contida na classe NP?

A conjectura normalmente aceita entre os cientistas da computação é que $\mathsf{P} \neq \mathsf{NP}$. Em particular, vários problemas importantes, conhecidos como problemas NP-completos, são candidatos a pertencer à classe $\mathsf{NP} \setminus \mathsf{P}$. O Problema da Satisfatibilidade (simplesmente *SAT*, definido formalmente na próxima seção) é um desses problemas. Mostrar que o problema SAT está na classe NP é uma tarefa simples, como veremos no Capítulo 9. Entretanto, demonstrar que o problema não pertence à classe P é uma tarefa que até o presente momento ninguém conseguiu realizar[3].

Qual é a dificuldade de provar que o problema SAT, ou qualquer outro candidato a estar em $\mathsf{NP} \setminus \mathsf{P}$, não admite um algoritmo polinomial? Como já vimos em capítulos anteriores, em geral não é simples provar que determinado algoritmo não existe. Para mostrar que certo problema de decisão não pertence a P, seria necessário excluir logicamente a possibilidade de que todos os infinitos algoritmos polinomiais falham na tarefa de decidir tal problema.

[3] A demonstração de que o problema não admite algoritmo polinomial ou a apresentação de um algoritmo polinomial, isto é, uma refutação da conjectura $\mathsf{P} \neq \mathsf{NP}$, é um dos maiores problemas em aberto em toda a matemática.

DETERMINISMO *vs* NÃO DETERMINISMO

Algo importante que devemos lembrar é que, muitas vezes, classes de linguagens com definições bastante diferentes podem ser iguais. Por exemplo, no Capítulo 3 vimos que a classe das linguagens aceitas por AFDs é exatamente a mesma classe das linguagens aceitas por AFNs. Por outro lado, em alguns modelos de computação há diferença entre computação determinística e não determinística. Por exemplo, a classe das linguagens aceitas por APs não é a mesma classe das linguagens aceitas por APDs.

Os Teoremas 6.6.2 e 6.6.3, apresentados no Capítulo 5, enunciam que o conjunto de linguagens aceitas por MTs é exatamente o mesmo conjunto de linguagens aceitas por MTNs. Para provar tal resultado, usamos o argumento de que uma MT pode simular uma MTN. O mesmo raciocínio vale para mostrar que MTs decidem as mesmas linguagens que podem ser decididas por MTNs. A pergunta central deste capítulo é a seguinte: uma MT pode sempre simular de maneira eficiente uma dada MTN?

PSPACE *vs* NPSPACE?

O problema P *vs* NP é muito famoso, mas, por outro lado, por que nunca escutamos nada sobre o problema PSPACE *vs* NPSPACE. Qual é o motivo disso? O motivo é que este não é um problema em aberto. Não é tão difícil mostrar que PSPACE = NPSPACE.

Uma maneira alternativa de definir classes de complexidade, útil em algumas circunstâncias, é a seguinte.

Definição 9.3.1 — TIME($f(n)$). Dada uma função $f : \mathbb{N} \to \mathbb{N}$, a classe TIME$(f(n))$ é o conjunto de todas as linguagens que podem ser decididas por uma Máquina de Turing com complexidade de tempo $c \cdot f(n)$, para alguma constante c.

■ **Exemplo 9.3** A classe TIME(n^3) é o conjunto de todas as linguagens que podem ser decididas por uma MT que execute, no máximo, $c \cdot |x|^3$ transições, para alguma constante $c > 0$, para qualquer string x fornecida como entrada. ■

■ **Exemplo 9.4** A classe TIME(2^n) é o conjunto de todas as linguagens que podem ser decididas por uma MT que execute, no máximo, $c \cdot 2^{|x|}$ transições, para alguma constante $c > 0$, para qualquer string x fornecida como entrada. ■

Definição 9.3.2 — SPACE($f(n)$). Dada uma função $f : \mathbb{N} \to \mathbb{N}$, a classe SPACE$(f(n))$ é o conjunto de todas as linguagens que podem ser decididas por uma Máquina de Turing com complexidade de espaço $c \cdot f(n)$, para alguma constante c.

9.3.1 Problemas de Decisão

Nesta seção apresentamos a definição de alguns problemas computacionais envolvendo grafos e lógica booleana. Caso o leitor não tenha familiaridade com conceitos elementares de teoria de grafos, o apêndice do livro contém as definições relevantes utilizadas aqui.

Definição 9.3.3 — Problema da Conectividade de Grafos. O problema de decidir se um dado grafo é conexo é definido pela linguagem

$$L_{\text{CNX}} = \{\llcorner G \lrcorner \in \Sigma^* : G \text{ é um grafo conexo}\}.$$

Definição 9.3.4 — Problema do Grafo Euleriano. O problema de decidir se um grafo é euleriano é definido pela linguagem

$$L_{\text{EUL}} = \{\llcorner G \lrcorner \in \Sigma^* : G \text{ é um grafo euleriano}\}.$$

Definição 9.3.5 — Problema do Grafo Hamiltoniano. O problema de decidir se um dado grafo é hamiltoniano é definido pela linguagem

$$L_{\text{HAM}} = \{\llcorner G \lrcorner \in \Sigma^* : G \text{ é um grafo hamiltoniano}\}.$$

Definição 9.3.6 — Problema do Conjunto Independente. O problema de decisão conhecido como problema do conjunto independente é definido pela linguagem

$$L_{\text{CI}} = \{\llcorner (G,k) \lrcorner \in \Sigma^* : G \text{ contém um conjunto independente de tamanho pelo menos } k\}.$$

Definição 9.3.7 — Problema da Clique. O problema de decisão conhecido como problema da clique é definido pela linguagem

$$L_{\text{CL}} = \{\llcorner (G,k) \lrcorner \in \Sigma^* : G \text{ contém uma clique de tamanho pelo menos } k\}.$$

Para o problema a seguir, lembramos que uma fórmula booleana está em *forma normal conjuntiva* quando é uma conjunção de disjunções. Referimo-nos a tais fórmulas como *fórmulas CNF*[4].

Definição 9.3.8 — Satisfatibilidade de Fórmulas Booleanas (SAT). O problema de decidir se uma dada fórmula booleana em CNF é satisfazível é definido pela linguagem

$$L_{\text{SAT}} = \{\llcorner \phi \lrcorner \in \Sigma^* : \phi \text{ é uma fórmula booleana em CNF satisfazível}\}.$$

■ **Exemplo 9.5** A fórmula $\phi_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3})$ é satisfazível. Uma valoração que satisfaz ϕ_1 é $x_1 = V$, $x_2 = V$ e $x_3 = F$. Assim, $\llcorner \phi_1 \lrcorner$ é uma instância verdadeira de L_{SAT}.

Por outro lado, como $\phi_2 = (\overline{x_1} \vee \overline{x_2}) \wedge (x_1) \wedge (x_2)$ não é satisfazível, concluímos que $\llcorner \phi_2 \lrcorner$ é uma instância falsa de L_{SAT}. ■

Convenção 9.1 (Literais em fórmulas em CNF)**.** *Seja C_i uma cláusula de uma fórmula ϕ em CNF. Assumimos que não ocorrem literais repetidos em C_i. Assumimos também que, se um literal l em C_i corresponde à variável x, então o literal $\overline{x}$ não ocorre em C_i. De maneira semelhante, se $l = \overline{x}$, então o literal x não ocorre em C_i.*

[4] O acrônimo CNF vem do inglês *conjunctive normal form*.

Exercícios

Exercício 9.3 Apresente uma definição para a classe P em termos da definição da classe TIME($f(n)$). ■

Exercício 9.4 Caso estivéssemos utilizando uma Máquina de Turing com apenas uma fita, a classe P seria a mesma ou seria diferente? Justifique sua resposta. Qual seria a desvantagem de definir P-*space* usando MTs com apenas uma fita? ■

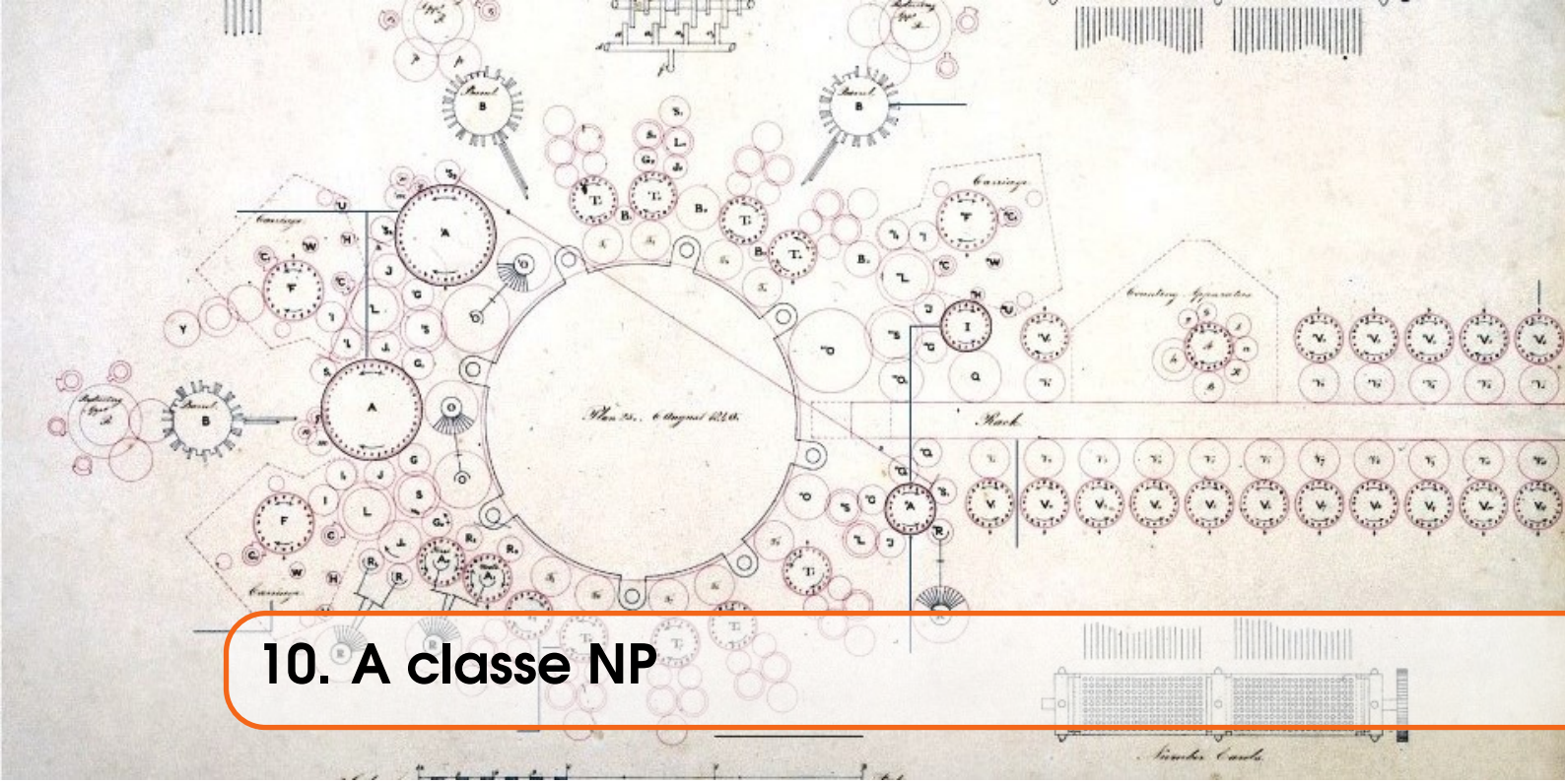
Exercício 9.5 Caso estivéssemos utilizando uma Máquina de Turing com apenas uma fita, seria necessário alterar a definição de complexidade de espaço. Há alguma vantagem em usar MTs com três fitas em vez de MTs com apenas uma fita quando lidamos com complexidade de espaço? ■

Exercício 9.6 Lembrando que $\mathscr{R}$ é o conjunto das linguagens recursivas, prove que $\mathsf{NP} \subseteq \mathscr{R}$. ■

Exercício 9.7 Forneça uma definição para NTIME($f(n)$) de maneira semelhante à Definição 9.3.1, mas considerando Máquinas de Turing não determinísticas. ■

Exercício 9.8 Lembrando que o grafo K_4 é o grafo completo com quatro vértices, responda: $\llcorner K_4 \lrcorner$ é uma instância verdadeira ou falsa do problema L_{EUL}? ■

Exercício 9.9 A string $\llcorner K_4 \lrcorner$ é uma instância verdadeira ou falsa do problema L_{HAM}? ■



10. 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

Figura 10.1: O Problema Sudoku.

Na Figura 10.1 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 admintem solução (instâncias falsas). Na Figura 10.1 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.

10.1 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 $x_1, ..., x_n$ escrita em CNF é uma fórmula booleana que tem a forma $\phi = (l_{11} \vee l_{12} \vee ... \vee l_{1k_1}) \wedge (l_{21} \vee l_{22} \vee ... \vee l_{2k_2}) \wedge ... \wedge (l_{m1} \vee l_{m2} ... \vee l_{mk_m})$, sendo que os *literais* l_{ij} podem ser tanto $x_1, ..., x_n$, quanto $\bar{x}_1, ..., \bar{x}_n$. A rigor, uma instância do problema SAT é uma string (as instâncias verdadeiras são strings $\llcorner\phi\lrcorner \in L_{\text{SAT}}$ e as instâncias falsas são strings $\llcorner\phi\lrcorner \notin L_{\text{SAT}}$), 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 $v = v_1, ..., v_n$ 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 $v_1, ..., v_n$ nos locais da fórmula onde apareçam as variáveis $x_1, ..., x_n$ e verficar 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 10.2 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 encontraríamos 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 $v_1, ..., v_n$ 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 $v_1, ..., v_n$ e verificar se, de fato, $v_1, ..., v_n$ 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 $v_i v_{i+1} \in E(G)$, $i = 1, 2, ..., n-1$ estão presentes no grafo. Para finalizar, verificamos se existe a aresta para "fechar" o ciclo e voltarmos ao vértices 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.

10.2 Certificados e verificação em tempo polinomial

Apresentamos agora a definição formal de certificados e verificação polinomial:

Definição 10.2.1 — Cerficados 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 $poly(n)$ 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, $|c| = poly(|x|)$.

(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).

10.2.1 Verificando o problema SAT em tempo polinomial

Para sermos mais concretos, vamos formalizar as ideias vistas na Seção 10.1 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 $poly(n)$ tal que o tamanho dos certificado das instâncias verdadeiras de tamanho n nunca são maiores $poly(n)$.

Vamos começar com os certificados. Seja uma instância verdadeira $\llcorner \phi \lrcorner$ 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 10.1, um certificado para uma fórmula satisfazível pode ser uma valoração que a satisfaça. Mais precisamente, um certificado para $\llcorner\phi\lrcorner$ é a string de bits $v = v_1v_2...v_n$ 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 $\phi_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3})$ do Exemplo 9.5. Um certificado para $\llcorner\phi\lrcorner$ é a string 110 (pois a valoração $x_1 = V$, $x_2 = V$ e $x_3 = F$ satisfaz ϕ).

Exercício 10.1 Com relação ao problema SAT, mostre strings v, que sugerimos que sejam usadas como certificado, satisfazem a condição $|v| = poly(|\llcorner\phi\lrcorner|)$. ■

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 $|\llcorner\phi\lrcorner|$ 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 $poly(n)$ 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 6, 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 each C_1, C_2, ...C_m do
2:     C_i = FALSE
3:     for each l_ij de C_i do
4:         if v_j satisfaz literal l_ij then
5:             C_i = TRUE
6:             Break
7:     if C_i = FALSE then
8:         Return FALSE
9: Return TRUE
```

No Algoritmo **Verificador_SAT**(ϕ, v), as m cláusulas da fórmula são ϕ de C_1, C_2, ..., C_m, ou seja, ϕ é uma conjunção da forma $C_1 \wedge C_2 \wedge ... \wedge C_m$, sendo que cada cláusula C_i é uma disjunção de literais da forma $C_i = (l_{i1} \vee l_{i2} \vee ... \vee l_{ik_i})$, onde k_i é o número de literais da cláusula C_i. Na Linha 4 do algoritmo é testado se o literal l_{ij} é satisfeito pela valoração v fornecida como entradada. Por exemplo, se $l_{ij} = x_4$, o literal é satisfeito quando $v_4 = V$. Por outro lado, se $l_{ij} = \overline{x}_4$, o literal é satisfeito quando $v_4 = F$.

Exercício 10.2 Mostre que $L_{\text{H}} = \{\llcorner G\lrcorner \; ; \; G \text{ é um grafo hamiltoniano}\}$ pode ser verificado em tempo polinomial. ■

10.2.2 Redefinindo a classe NP

O fato de uma linguagem poder ser verificada em tempo polinomial não nessariamente implica que a linguagem possa ser decidida em tempo polinomial. Veremos a seguir que uma das maneiras de enunciar a conjectura de que $\mathsf{P} \neq \mathsf{NP}$ 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 10.2.1 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 cerficados 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 10.2.2 Seja V a classe de problemas que podem ser verificados em tempo polinomial. Então $\mathsf{V} = \mathsf{NP}$.

Pelo Teorema 10.2.2, podemos definir a classe NP alternativamente da seguinte maneira.

Definição 10.2.2 — Classe NP (definição equivalente). Uma linguagem $L \subseteq \Sigma^*$ está em NP se existe um polinômio $p : \mathbb{N} \to \mathbb{N}$ e uma MT M com complexidade de tempo polinomial (essa MT é chamada de verificador de L) tal que $\forall x \in \Sigma^*$:

$$\boldsymbol{x \in L \Leftrightarrow \exists u \in \Sigma^{p(|x|)}; M(x,u) = 1}$$

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 definião original da classe NP (i.e., Definição 9.2.4), entretanto, ambas se referem exatamente o mesmo conjunto de linguagens.

10.3 Exercícios

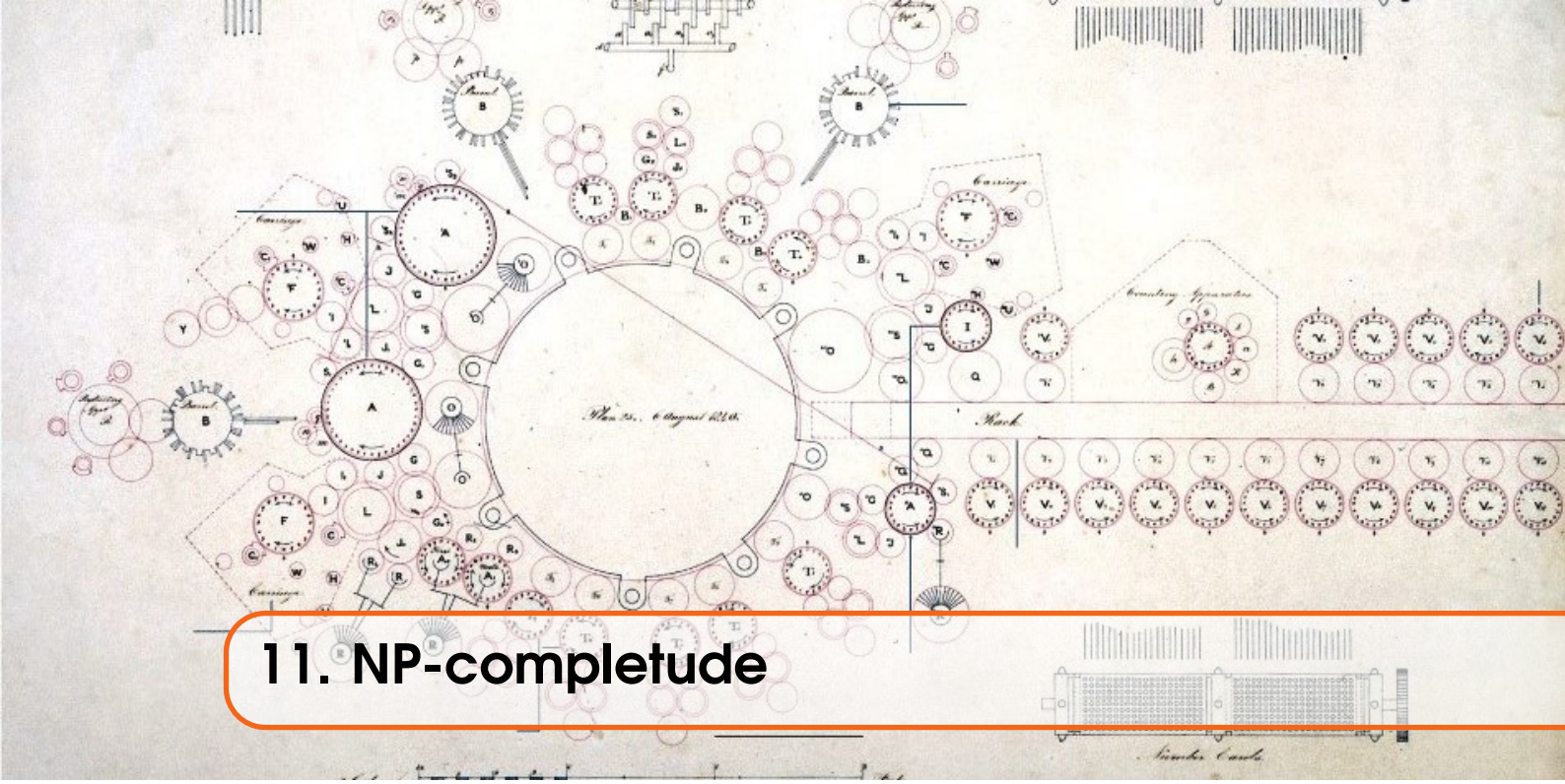
Exercício 10.3 Seja uma linguagem $L \subseteq \Sigma^*$ e seja $\overline{L} = \Sigma^* \setminus L$. Dada uma classe de complexidade $\mathscr{C}$, definimos o complemento de $\mathscr{C}$ da seguinte maneira: $co\text{-}\mathscr{C} = \{\overline{L} : L \in \mathscr{C}\}$. Com relação a esta definição, responda o seguinte:

- Seja L_{EUL} o problema de testar se um grafo não é euleriano. Prove que $L_E \in co\text{-P}$.
- Prove que $\text{P} = co\text{-P}$.

■

Exercício 10.4 Uma conjectura conhecida em complexidade computacional é que $\text{NP} \neq co\text{-NP}$. Por que não poderíamos provar que $\text{NP} = co\text{-NP}$ usando uma estratégia de prova semelhante a estratégia usada no exercício anterior? ■

Exercício 10.5 Na Definição 10.2.1, item (a), um dos requistos é que o certificado tenha tamanho polinomial no tamanho da instância. Por que motivo faz sentido esta restrição? ■



11. 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 certos problemas são os mais difíceis 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.

11.1 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 terema 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. Para que possamos a apresentar o Teorema de Cook-Levin, precisamos primeiro entender o seguinte: Como a existência de um algoritmo polinomial para um dado problema pode implicar a existência de um algoritmo polinomial para um outro dado problema?

Sejam L_1 e L_2 dois problemas de NP. 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? Apresentendo um algoritmo R que, ao tomar uma instância x do problema L_1 como entrada e retorne uma instância y de L_2 como saída. Mas não apenas isto: O ponto chave é que R deve funcionar de forma que se x é uma instância verdadeira de L_1, então 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. Este algoritmo R é chamado de *redução*.

> INSTÂNCIAS DE PROBLEMAS
>
> Lembrando que uma instância de um problema de decisão L é uma string qualquer $x \in \Sigma^*$. Normalmente x é a representação binária de um certo objeto matemático relacionado ao problema em questão. Por exemplo, se o problema em questão é testar se um grafo tem certa propriedade, tipicamente x é a representação em binário 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 11.1.1 — 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 11.1** Vamos mostrar que $L_{\text{CI}} \leq_P L_{\text{CL}}$, ou seja, que o problema da clique é redutível ao problema do conjunto independente. Segue o pseudo-código da redução:

Redução_CI_CL: (G,k)
1: $V(G') = V(G)$
2: $E(G') = \varnothing$
3: $G' = (V(G'), E(G'))$
4: **Para cada** par de vértices distintos u, v de G **do**
5: **if** $uv \notin E(G)$ **then**
6: insira aresta uv em $E(G')$
7: Devolva (G', k)

■

Exercício 11.1 Mostre que a relação $\leq_P$ é transitiva. ■

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 11.1.1 Se $L_1 \leq_P L_2$ e $L_2 \in \mathsf{P}$, então $L_1 \in \mathsf{P}$.

Prova: Como $L_2 \in \mathsf{P}$, 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, $L_1 \in \mathsf{P}$. □

Definição 11.1.2 — Problemas NP-difíceis. A classe NP-difícil é o conjunto de problemas de decisão L tal que $\forall\, L' \in \mathsf{NP}$, $L' \leq_P L$. Se $L \in$ NP-difícil, normalmente dizemos que L *é* NP-*difícil*.

Definição 11.1.3 — 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;
- $L \in \mathsf{NP}$.

Dizemos também que L é NP-*completo* no caso em que $L \in$ NP-completo.

Teorema 11.1.2 Seja $L \in$ NP-*completo*. Se $L \in$ P, então P = NP.

Prova: Suponha que o problema NP-completo L esteja contido na classe P. Seja L' um problema qualquer de NP. Como L é NP-completo, então $L' \leq_P L$, e portanto, pelo Teorema 11.1.1, $L' \in$ P. Consequentemente NP $\subseteq$ P. Como P $\subseteq$ NP (ver Exercício 9.2), então P = NP. □

Teorema 11.1.3 — Teorema de Cook-Levin. L_{SAT} é NP-completo.

Corolário 11.1.4 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.

11.2 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 número inteiro[1], isto é, dado $n \in \mathbb{Z}^+$, queremos retornar a sequência de fatores primos $p_1, ..., p_k$, 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 percurso visitando cada cidade uma única vez. Este tipo de problema é conhecido como problema de otimização. Tais problemas são generalizações 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 encontramos um problema de busca ou otimização, é bastante comum encontrar um problema de decisão que seja semelhante ao problema original estudar a complexidade deste problema de decisão. 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 11.2 — 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. ■

[1] O problema de fatoração é um problema computacional clássico e a dificuldade de resolução do mesmo é a base do protocolo de criptografia RSA.

Exercício 11.2 Mostre que se existe um algoritmo polinomial para o problema MAXCLIQUE, então existe um algoritmo polinomial para o problema da clique. ■

Na seção 11.3, provaremos que o problema da clique é NP-completo. Uma vez que um algoritmo polinomial para o problema de otimização MAXCLIQUE pode facilmente ser adaptado para resolver o problema CLIQUE (Exercício 11.2), conclui-se que se o problema de otimização puder ser resolvido em tempo polinomial, então P = NP. Em outras palavras, para mostrarmos os que o problema de otimização é intratável basta mostrarmos que a versão de decisão de problema é intratatável.

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.

11.3 Provando a *NP*-completude de problemas

Na seção 11.1 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 11.3.1 Suponha que L_1 é um problema NP-completo. Se $L_2 \in \mathsf{NP}$ 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 trasitiva (veja Exercício 11.1), podemos usar o fato que $L \leq_P L_1$ e $L_1 \leq_P L_2$ para concluir que $L \leq L_2$. Como $L_2 \in \mathsf{NP}$, 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 11.3.1, 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 11.3.1.

11.3.1 Provando que o problema do conjunto independente é NP-completo

Nesta seção vamos mostrar que $L_{\text{SAT}} \leq_P L_{\text{CI}}$. 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 11.1.

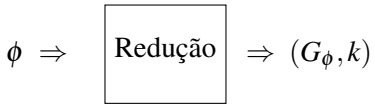


Figura 11.1: A redução toma uma fórmula ϕ e retorna (G_ϕ, k), onde G_ϕ é um grafo e k um número natural tal que ϕ *é satisfazível* $\Leftrightarrow$ G_ϕ *contém um conjunto independente de tamanho* k.

Seja $\phi = C_1 \wedge ... \wedge C_m$ uma instância do problema SAT com n variáveis x_i, $i = 1, ..., n$. Para cada $j = 1, ..., m$, 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)

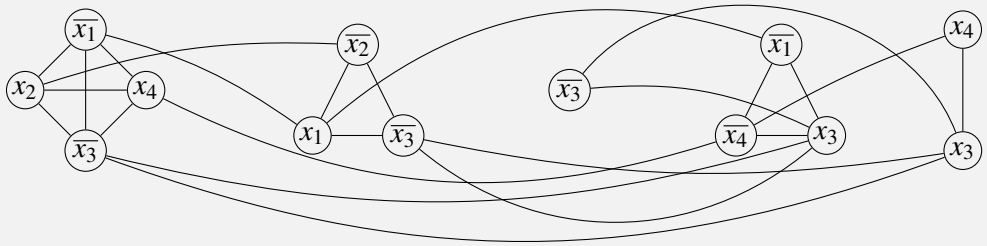
1. Crie um conjunto de vértices V de tamanho $\sum_i^m |A(C_j)|$.
2. Particione o conjunto V em subconjuntos $V_1, ..., V_m$, tal que cada subconjunto V_j tem tamanho $|A(C_j)|$ e os rotule[2] cada vértice de v_j com um elemento diferente do conjunto A_j.
3. Adicione as seguintes arestas: para cada conjunto 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 da 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 $\overline{x_i}$, $i = 1, ..., n$.
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:

$$\phi_1 = (\overline{x_1} \vee x_2 \vee \overline{x_3} \vee x_4) \wedge (x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (\overline{x_3}) \wedge (\overline{x_1} \vee x_3 \vee \overline{x_4}) \wedge (x_3 \vee x_4)$$

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, $(\overline{x_1} \vee x_2 \vee \overline{x_3} \vee x_4)$, corresponde a clique de tamanho 4, mais à esquerda no desenho do grafo. Observe que os literais $\overline{x_1}$, x_2, $\overline{x_3}$ 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 presente nestas cliques, o grafo contém arestas conectando cada literal x_i ao seu complemento $\overline{x_i}$. Por exemplo, o literal $\overline{x_1}$ aparece na primeira cláusula e o literal x_1 na segunda cláusula, portanto há uma aresta ligando os vértices com estes rótulos nas cliques.

[2] Devemos 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.

Importante: Lembramos que a redução precisa garantir que G_ϕ *possui um conjunto independente de tamanho m se e somente se a fórmula* $\phi = C_1 \wedge ... \wedge C_m$ *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 $\overline{x_1}$ 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 $\{v_1, ..., v_m\}$, com um vértice de cada clique diferente. Uma valoração que satisfaz ϕ pode ser obtida a partir dos rótulos dos vértices $\{v_1, ..., v_m\}$.

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 $\overline{x_i}$) e tem tamanho m, o que contradiz o fato que S é um conjunto independente máximo.

Exercício 11.3 Usando o fato que o problema do conjunto independnete é NP-completo, prove que o problema da clique é NP-completo. ■

11.4 Exercícios

Nos exercícios a seguir, as seguintes definições serão necessárias:

Definição 11.4.1 — Isomorfismo de Grafos. Dados dois grafos G_1 e G_2, decidir se o grafo G_1 é isomorfo ao grafo G_2.

Definição 11.4.2 — Problema do caixeiro viajante. Dado um grafo completo G com peso nas arestas, encontrar o caminho hamiltoniano de G com o menor custo.

Definição 11.4.3 — Problema da coloração mínima. Dado um grafo G determinar encontrar uma coloração de G com o menor número possível de cores.

Definição 11.4.4 — Problema da k-coloração. Dado um grafo (G,k) decidir se existe uma coloração de G com k ou menos cores.

Definição 11.4.5 — Problema da 3-Coloração. Dado um grafo G decidir se existe uma coloração de G com 3 ou menos cores.

Definição 11.4.6 — 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 11.4.7 — Problema 3-SAT. Dada uma fórmula ϕ em CNF tal que cada cláusula tenha no máximo 3 literais, decidir se ϕ é satisfazível.

Exercício 11.4 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 ou de busca, apresente uma versão de decisão para o mesmo 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 11.5 Prove que o problema 3-SAT é NP-completo. ■

Exercício 11.6 Prove que o problema da 3-coloração é NP-completo. ■

Exercício 11.7 Prove que o problema da k-coloração é NP-completo. ■

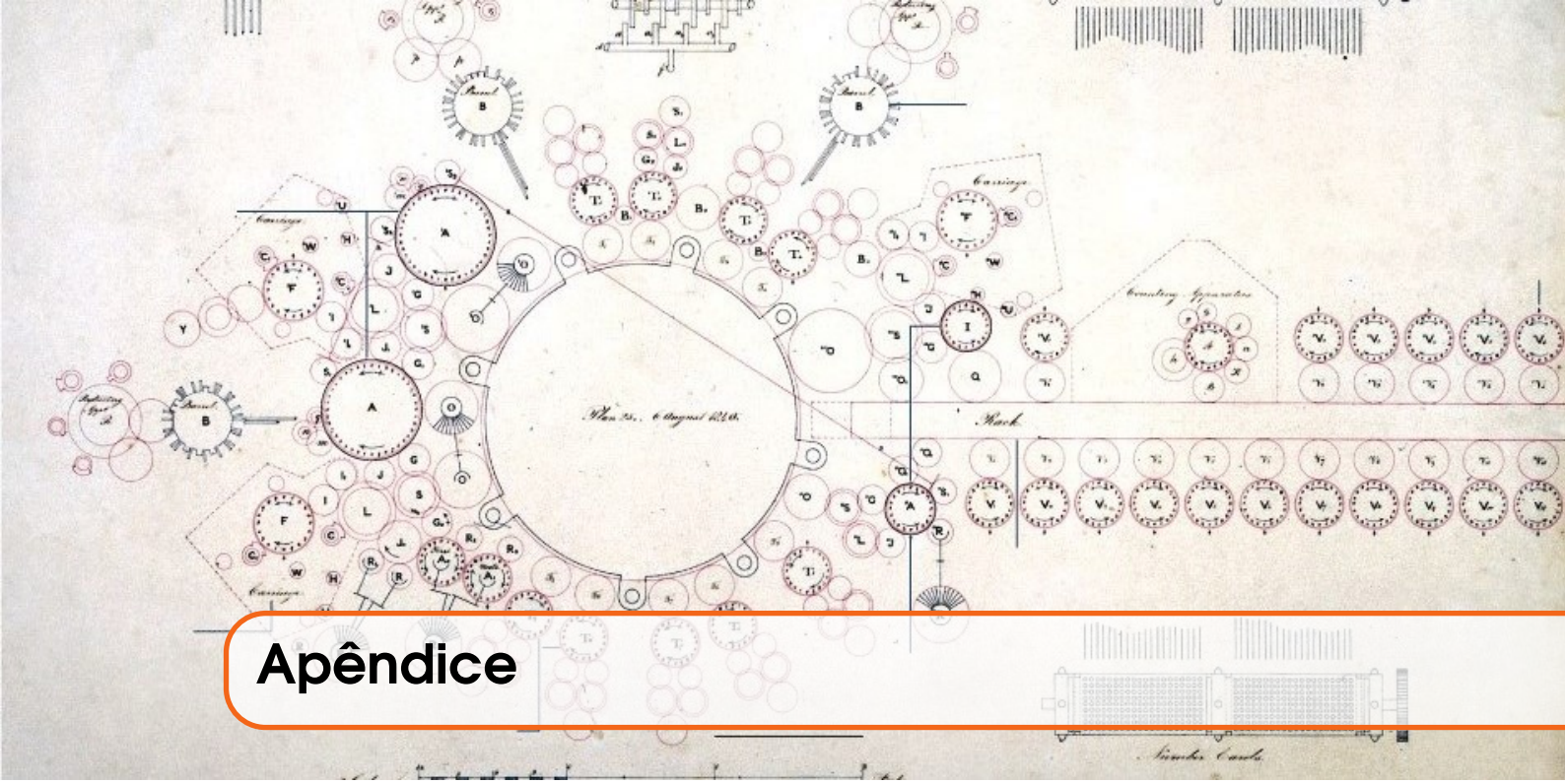
Exercício 11.8 Prove que se existe um algoritmo polinomial para encontrar uma coloração mínima para um grafo, então $\mathsf{P} = \mathsf{NP}$. ■

Exercício 11.9 Seja L_{CV} a versão de decisão do problema de cobertura por vértices que você obteve no Exercício 11.4. Mostre que $L_{\text{CI}} \leq_P L_{\text{CV}}$. ■

Exercício 11.10 Considere o problema NP-completo 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 do problema?
  (b) Apresente um algoritmo polinomial verificador para o problema.

■



Apêndice

Grafos e grafos direcionados

Um *grafo* G é um par $(V(G), E(G))$ onde:

- $V(G)$ é um conjunto finito, chamado de conjunto de vértices.
- $E(G)$ é um conjunto onde cada elemento é um conjunto de dois vértices, chamado de conjunto de arestas

Um *grafo direcionado* G é um par $(V(G), E(G))$ onde

- $V(G)$ é um conjunto finito, chamado de conjunto de vértices
- $E(G)$ é um conjunto de pares de vértices,
  chamado de conjunto de arcos

Quando o $V(G)$ não é definido explicitamente, a convenção é $V(G) = [1..n]$.

Notação simplificada: $G = (V, E)$ (ao invés de $G = (V(G), E(G))$)
Notação simplificada: uv denota a aresta $\{u, v\}$

Grafos ponderados

Um *grafo ponderado* é um par (G, w) onde G é um grafo e w é uma função que associa a cada aresta a de G um *peso* $w(a)$.

- Grafos direcionados ponderados são definidos de maneira análoga.
- Para simplificar, às vezes diremos "grafo ponderado G" ao invés de "grafo ponderado (G, w)".
- Convenção: Se G não é um grafo ponderado, $\forall e \in E(G)$, $w(e) = 1$.
- Notação simplificada: Se $\{u, v\} \in E(G)$, escrevemos $w(u, v)$ para o peso de $\{u, v\}$. (ao invés de escrever $w(\{u, v\})$

Matriz de Adjacência de um grafo (ponderado) G: A *matriz de adjacência* de G é a matriz M_G indexada por $V(G) \times V(G)$ dada por

$$M_G[u, v] = \begin{cases} w(u, v), & \text{se } uv \in E(G), \\ 0, & \text{se } uv \notin E(G). \end{cases}$$

Propriedades de Grafos

Se abaixo as definições de vários conceitos elementares em teoria dos grafos.

- O *complemento* de um grafo G é o grafo $\overline{G}$ que possui conjunto de vértices de $V(\overline{G}) = V(G)$ e tal que $\forall u, v \in V(\overline{G})$, $uv \in E(\overline{G}) \Leftrightarrow uv \notin E(G)$.
- Uma *clique* em um grafo G é um conjunto $S \subseteq V(G)$ tal que $\forall u, v \in S$, $uv \in E(G)$.
- Um *conjunto independente* em um grafo G é um conjunto de vértices $S \subseteq V(G)$ tal que $\forall u, v \in S$, $uv \notin E(G)$.
- Um grafo G é dito *bipartido* se $V(G)$ pode ser particionado em dois conjuntos independentes.
- Um *grafo completo* é um grafo tal que existe uma aresta entre cada par de vértices do grafo. Em outras palavras, $V(G)$ é uma clique.
- Um grafo G é *conexo* se para todos par de vértices u, v do grafo G existe um percurso que saia de u e chegue em v.
- Uma *árvore* é um grafo conexo e acíclico
- Um grafo G é dito *k-colorível* se é possível associar cores aos seus vértices de forma que toda aresta tenha as duas pontas coloridas com cores diferentes.
- Dois grafos G e H são ditos *isomorfos* se existe uma bijeção $f : V(G) \leftrightarrow V(H)$ tal que $uv \in E(G) \Leftrightarrow f(u)f(v) \in E(H)$.

Seja G um grafo com n vértices e seja $\pi = [\pi_1, ..., \pi_n]$ uma permutação de $V(G)$. Observação: neste texto permutações são sempre cíclicas, ou seja, π_{n+1} se refere à π_1. A partir disto, definimos:

- A permutação π é um *circuito hamiltonino* de G se $\{\pi_i, \pi_{i+1}\} \in E(G)$, $i = 1, ..., n$.
- Circuitos hamiltonianos também são chamados de ciclos hamiltonianos.
- Grafos que admitem circuitos hamiltonianos são chamados de *grafos hamiltonianos*.
- A permutação π de G é um *caminho hamiltoniano* se $\{\pi_i, \pi_{i+1}\} \in E(G)$, $i = 1, ..., n-1$.

Seja G um grafo com m arestas. Seja $\pi = [\pi_1, ..., \pi_m]$ uma permutação de $E(G)$. A partir disto definimos:

- A permutação π é um *circuito euleriano* de G se para todo $i \in [1..n]$, $\pi_i = uv$ e $\pi_{i+1} = vw$, para vértices u, v, w de G. Isto é, existe um passeio visitando todas as arestas do grafo.
- Grafos que admitem circuitos eulerianos são chamados de *grafos eulerianos*.

O Problema da Satisfatibilidade

Dadas variáveis booleanas $x_1, ..., x_n$, uma *fórmula booleana na forma normal conjuntiva* é uma conjunção de disjunções de termos que podem ser x_i ou $\overline{x_i}$. Para simplificar, nos referimos a tais fórmulas simplesmente como *fórmulas CNF*.

■ **Exemplo 11.3** A fórmula $\phi_1 = (x_1 \vee x_2) \wedge (x_1 \vee \overline{x_2}) \wedge (\overline{x_3})$ é uma *fórmula CNF*. ■

Terminologia: Em uma fórmula CNF, os termos x_i ou $\overline{x_i}$ são chamados de *literais* da fórmula.

Note que na fórmula ϕ_1 do Exemplo 11.3, os literais são $x_1, x_2, \overline{x_2}$ e $\overline{x_3}$.

Definição 11.4.8 — Problema da Satisfatibilidade (SAT). O problema de decidir se existe uma valoração que satisfaz uma dada fórmula CNF ϕ é chamado de *Problema da Satisfatibilidade*, ou simplesmente Problema *SAT*. Mais precisamente, dada a fórmula ϕ, queremos decidir se existe uma valoração $v_1, v_2, ..., v_n$, sendo $v_i \in \{V, F\}$, de forma que ao substituir o valor v_i no lugar da variável x_i, o valor de verdade da fórmula ϕ é V. Formalmente, o problema de decisão é dado pela linguagem $L_{\text{SAT}} = \{\llcorner\phi\lrcorner \in \Sigma^* : \phi \text{ é uma fórmula CNF satisfazível}\}$.

■ **Exemplo 11.4** A fórmula ϕ_1 do Exemplo 11.3 é satisfazível pois existe uma valoração que a satisfaz: Basta tomar $v_1 = V$, $v_2 = V$, $v_3 = F$

Como a fórmula ϕ_1 é satisfazível, a string $\llcorner\phi_1\lrcorner$ é uma instância verdadeira de L_{SAT}. ■

■ **Exemplo 11.5** A fórmula CNF $\phi_2 = (\overline{x_1} \vee \overline{x_2}) \wedge (x_1) \wedge (x_2)$ não é satisfazível, pois não existe valoração que a satisfaz. Ou seja, $\llcorner\phi_2\lrcorner$ é uma instância falsa de L_{SAT}. ■

Terminologia: Cada disjunção em una fórmula CNF é chamada de *cláusula*.

Note que as cláusulas da fórmula ϕ_2 do Exemplo 11.5 são $(\overline{x_1} \vee \overline{x_2})$, (x_1) e (x_2)

Funções μ-recursivas

As *funções μ-recursivas* são definidas abaixo:

Definição 11.4.9 — Funções μ-recursivas. O conjunto das funções μ-recursivas é o menor conjunto de funções que contém:

Base: As seguintes funções são μ-recursivas:
- Função nula: $Z(x) = 0$;
- Função sucessor: $S(x) = x+1$;
- Funções projeção: $P_i^n(x_1,\ldots,x_n) = x_i$.

Indução:
1. **Operação de Composição:** se as funções $g, h_1, ..., h_k$ são μ-recursivas, $\vec{x}$ é um vetor, então f, definida abaixo, é μ-recursiva:
$$f(\vec{x}) = g(h_1(\vec{x}),\ldots,h_k(\vec{x})).$$
2. **Operação de Recursão primitiva:** se g e h são funções μ-recursivas, $\vec{x}$ é um vetor, então f, definida abaixo, é μ-recursiva:
$$\begin{cases} f(0,\vec{x}) = g(\vec{x}), \\ f(y+1,\vec{x}) = h(y, f(y,\vec{x}),\vec{x}) \end{cases}$$
3. **Operação de Minimização (μ):** se g é uma função μ-recursiva, então
$$f(\vec{x}) = \mu y[g(\vec{x},y) = 0]$$
é μ-recursiva.

Observe que na Operação de Minimização, busca-se o menor y tal que $g(\vec{x},y) = 0$.

■ **Exemplo 11.6** Definição da *função soma* por recursão primitiva:

$$\begin{cases} \text{add}(x,0) = x, \\ \text{add}(x,y+1) = S(\text{add}(x,y)). \end{cases}$$

É uma função *primitiva recursiva* e, portanto, μ-recursiva. ■

■ **Exemplo 11.7** Definição da função que retorna o menor inteiro y tal que $y^2 > x$.

$$f(x) = \mu y[y^2 > x].$$

■

Cálculo λ

O **cálculo** λ é um modelo de computação baseado em expressões matemáticas. Segue a definição:

Definição 11.4.10 — Expressões do cálculo λ. As expressões do cálculo λ (*termos* λ) são definidas recursivamente:

(i) Uma variável x é um termo λ.
(ii) Se M e N são termos λ, então (MN) é um termo λ
(iii) Se x é uma variável e M é um termo λ, então $(\lambda x.M)$ é um termo λ.

A ideia é que uma *abstração* $\lambda x.M$ define uma função com parâmetro x e corpo M e uma *aplicação* MN representa a aplicação da função M ao argumento N. O cálculo λ expressa a ideia de computação como transformação de funções.

■ **Exemplo 11.8 — Função Soma.** Primeiramente, precisamos reprentar números naturais como aplicações de uma função f. Isso é chamado de *codificação de Church*. O número n é representado pelo termo $\overline{n}$ abaixo:

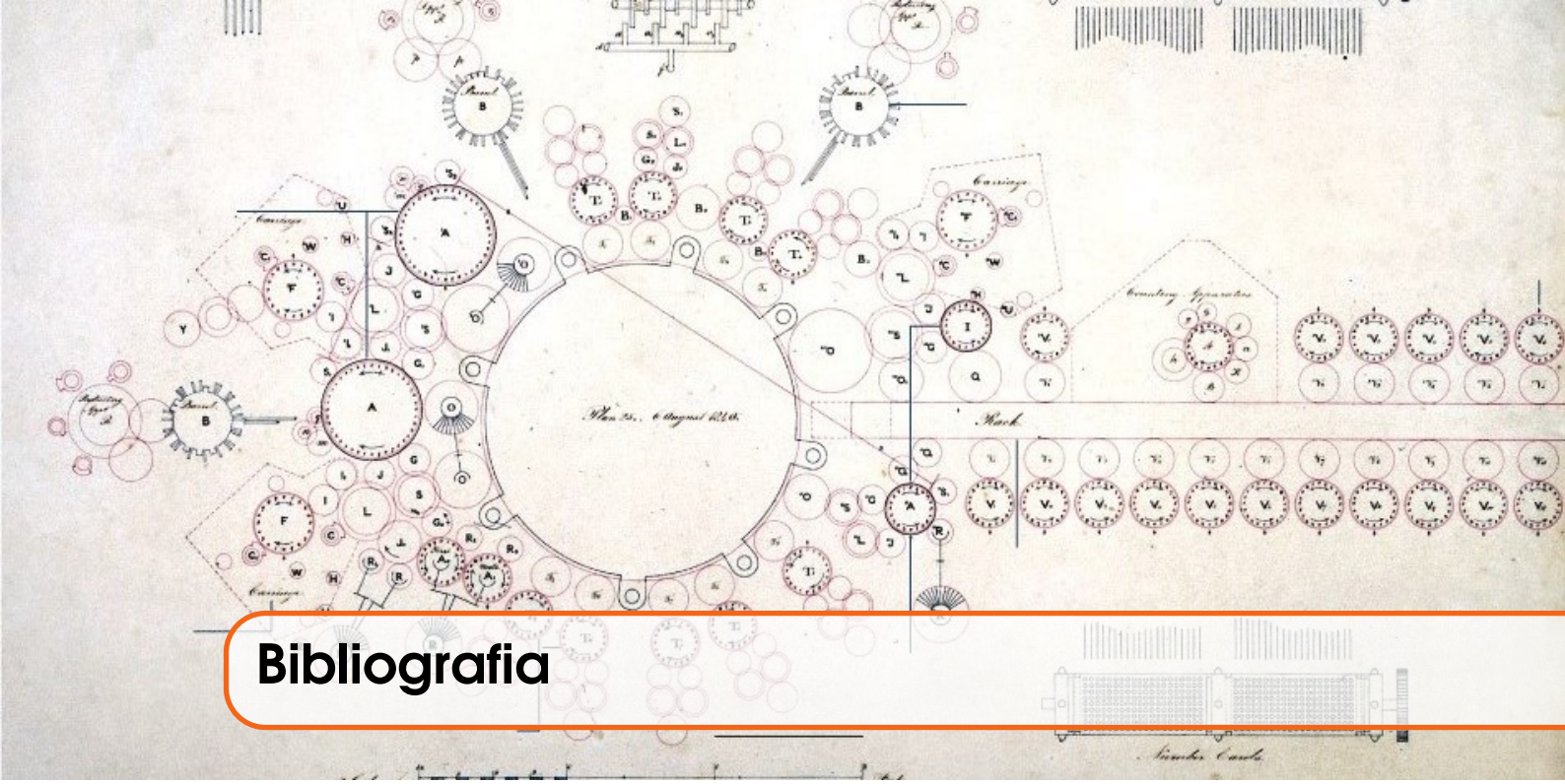
$\overline{n} \equiv \lambda f.\lambda x.\, f^n(x)$, onde f^n significa a aplicação de f a x, n vezes.

A soma de dois números de Church m e n pode ser definida como:

$$\text{SOMA} \equiv \lambda m.\lambda n.\lambda f.\lambda x.\, m\, f\, (n\, f\, x)$$

■

A ideia do exemplo acima é a seguinte: $n\,f\,x$ aplica f n vezes a x, produzindo o número n e $m\,f\,(n\,f\,x)$ aplica f m vezes ao resultado de $n\,f\,x$, ou seja, aplica f $m+n$ vezes a x. Com isso, SOMA $m\,n$ produz o número de Church correspondente a $m+n$.



Bibliografia

Livros

[Aar13] Scott Aaronson. *Quantum Computing Since Democritus*. 1st edition. Cambridge, 2013 (cited on pages 9, 112).

[Cha96] David J. Chalmers. *The Conscious Mind: In Search of a Fundamental Theory*. Obra central sobre o problema da consciência; introduz a distinção entre o "problema fácil" e o "problema difícil". New York: Oxford University Press, 1996 (cited on page 112).

[Fes19] Edward Feser. *Filosofia da Mente - Um guia para iniciantes*. 1st edition. Edições Santo Tomás, Nov. 2019 (cited on page 112).

[HMU06] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to the Theory of Computation*. 2nd edition. Addison Wesley, 2006 (cited on pages 9, 36, 37, 60, 64, 67).

[Kuh62] Thomas S. Kuhn. *The Structure of Scientific Revolutions*. University of Chicago Press, 1962 (cited on page 110).

[Lak70] Imre Lakatos. *Falsification and the Methodology of Scientific Research Programmes*. Edited by Imre Lakatos and Alan Musgrave. Cambridge University Press, 1970, pages 91–196 (cited on page 110).

[LP98] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. 2nd edition. Prentice Hall, 1998 (cited on pages 14, 36).

[Pop59] Karl R. Popper. *The Logic of Scientific Discovery*. Basic Books, 1959 (cited on page 110).

[Sea92] John R. Searle. *The Rediscovery of the Mind*. Crítica ao materialismo reducionista; defesa de uma concepção biológica da consciência. Tradução: *A Redescoberta da Mente*, Martins Fontes. Cambridge, MA: The MIT Press, 1992 (cited on page 112).

[Sip06] Michael Sipser. *Introduction to the Theory of Computation*. 2nd edition. Thomson Course Technology, 2006 (cited on pages 9, 14, 60, 64, 67).

[Sud05] Thomas A Sudkamp. *Languages and Machines: An Introduction to the Theory of Computer Science*. 3rd edition. Addison Wesley, 2005 (cited on page 14).

Articles

[Deu85] David Deutsch. "Quantum theory, the Church–Turing principle and the universal quantum computer". In: *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences* 400.1818 (1985), pages 97–117. DOI: 10.1098/rspa.1985.0070 (cited on page 107).

[Fey82] Richard P. Feynman. "Simulating physics with computers". In: *International Journal of Theoretical Physics* 21.6 (1982), pages 467–488. DOI: 10.1007/BF02650179 (cited on page 107).

[Pău00] Gheorghe Păun. "Computing with membranes". In: *Journal of Computer and System Sciences* 61.1 (2000), pages 108–143. DOI: 10.1006/jcss.1999.1693 (cited on page 108).

[Qui51] Willard Van Orman Quine. "Two Dogmas of Empiricism". In: *The Philosophical Review* 60.1 (1951), pages 20–43 (cited on page 110).

[Sol+08] David Soloveichik et al. "Computation with finite stochastic chemical reaction networks". In: *Natural Computing* 7.4 (2008), pages 615–633. DOI: 10.1007/s11047-008-9067-y (cited on page 108).

[Tur50] Alan M. Turing. "Computing Machinery and Intelligence". In: *Mind* LIX.236 (1950), pages 433–460. DOI: 10.1093/mind/LIX.236.433. URL: https://doi.org/10.1093/mind/LIX.236.433 (cited on page 112).

[Win98] Erik Winfree. "Algorithmic self-assembly of DNA". In: *Ph.D. thesis, California Institute of Technology* (1998). Proposta original do modelo ATAM (cited on page 108).