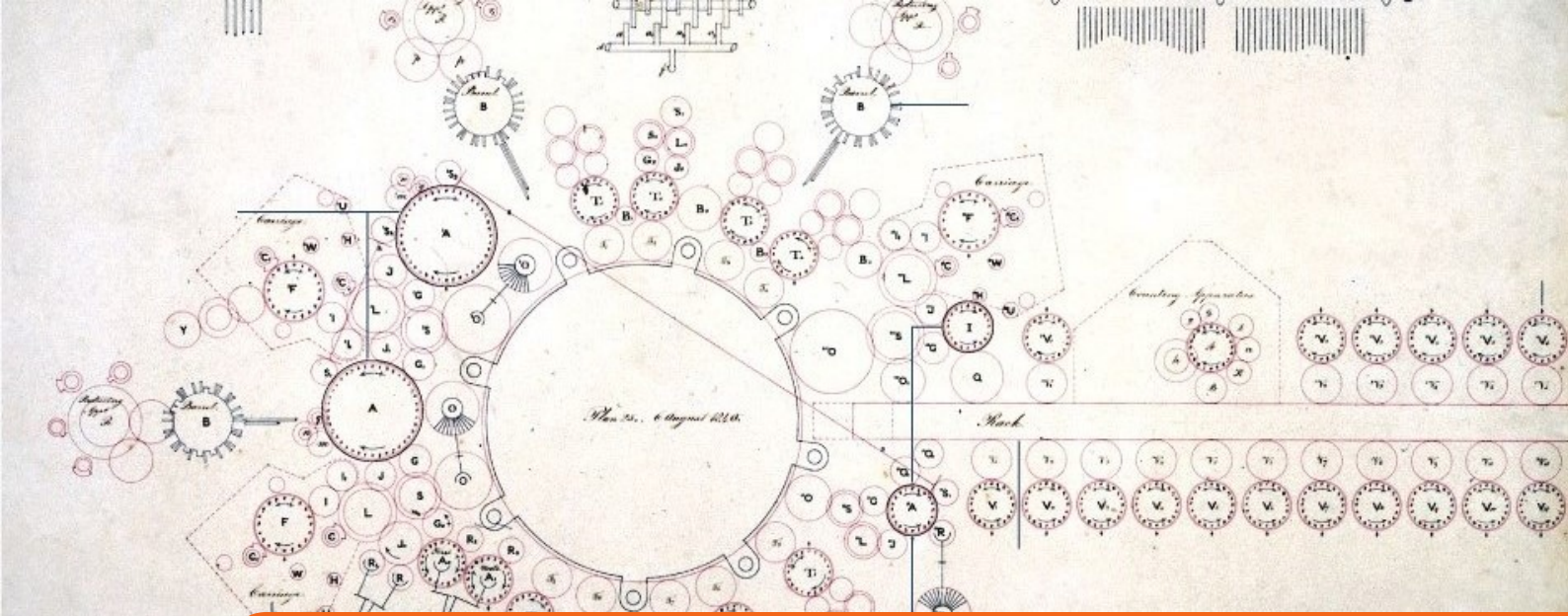
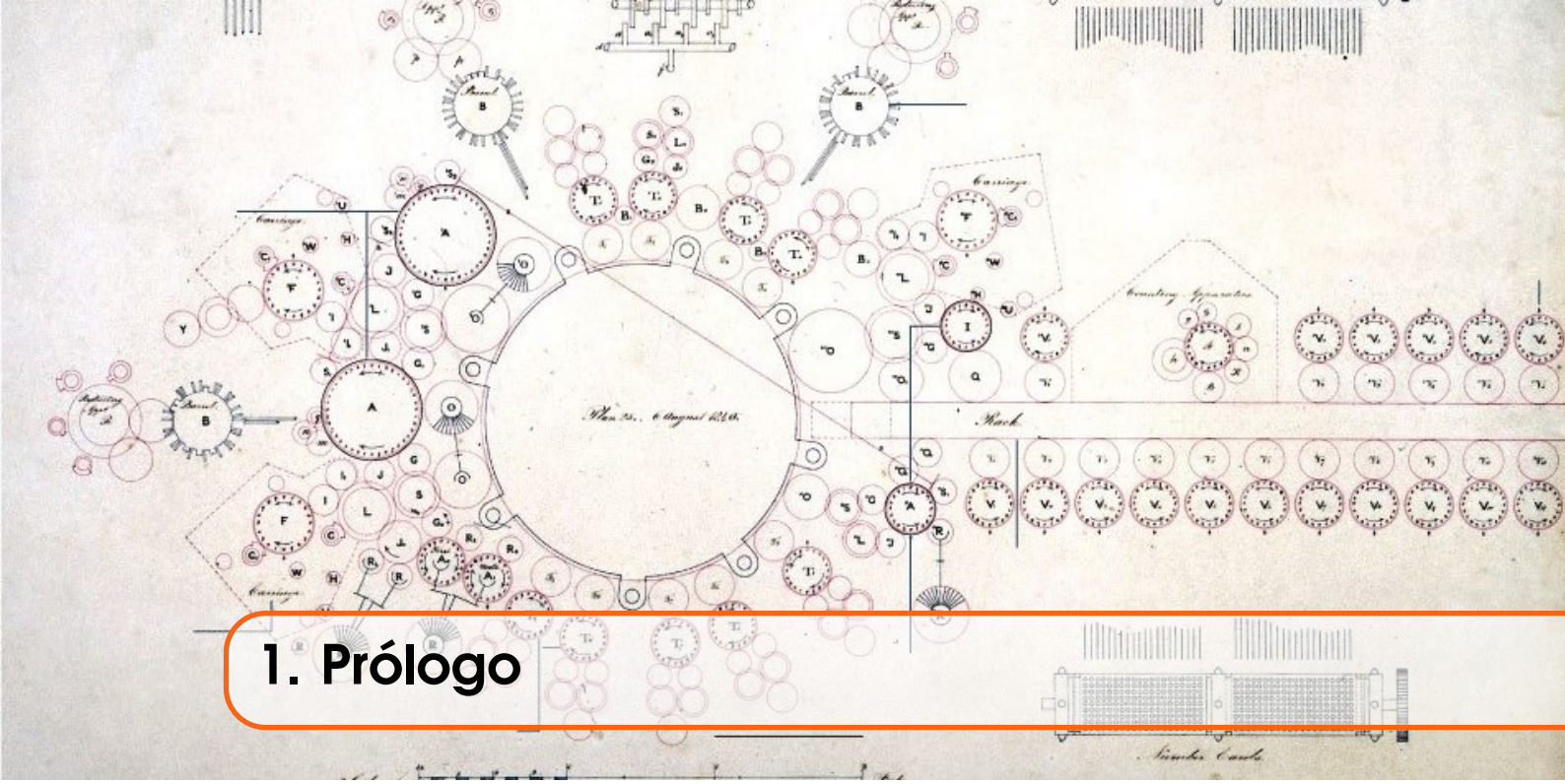
Autômatos, computabilidade e complexidade computacional

Murilo V. G. da Silva



Contents



1. Prólogo

1.1 O que é computação?

A maioria de nós tem uma noção intuitiva do que é um algoritmo ou do que é um computador. Estes dois conceitos são tão corriqueiros que é comum não pararmos para pensar a fundo sobre o que eles realmente significam. Entretanto, quando começamos a fazer perguntas mais profundas sobre computação, as nossas noções intuitivas sobre algoritmos e computadores não são suficientes. Em tais casos sentimos a necessidade sermos precisos. Mas que tipo de perguntas nos exigem sermos tão precisos?

Podemos começar com uma pergunta filosófica antiga: quando pensamos no nascimento da ciência da computação, é comum discutirmos se a computação foi *inventada* ou se as leis da computação foram *descobertas* e, só a partir de tais descobertas, estas leis foram usadas na construção de objetos que chamamos de computadores.

Esta pergunta está claramente no âmbito da filosofia e não é nosso objetivo discutir isso aqui, mas logo de início, este tipo de pergunta acende o seguinte sinal de alerta em pessoas práticas como nós, cientistas da computação, que é o seguinte: alguém está sugerindo que existem supostas "leis da computação" que fazem parte da natureza, foram descobertas e são dignas de serem investigadas. Mas isso não parece estranho? Faz sentido dizer que computação, ou mesmo ideias como algoritmos e computadores são conceitos naturais? Tais conceitos não estariam necessariamente relacionados aos artefatos tecnológicos que temos no nosso dia a dia?

O ponto de partida de qualquer discussão como esta é a ideia de *informação*, uma ideia antiga e intuitiva, mas que começou a entrar de maneira mais regular no vocabulário dos cientistas apenas final do século XIX, especialmente no vocabulário dos físicos, quando começaram a esbarrar no conceito de entropia, enquanto estudavam termodinâmica. Com o avanço da ciência no século XX, uma série de descobertas, em particular a descoberta das leis da mecânica quântica (leis em que a ideia de *informação* quântica é central) e a descoberta da molécula de DNA (molécula que está intrinsicamente ligada a ideia de *informação* genética), começaram a solidificar a intuição de que informação é algo fundamental e que permeia o mundo natural em diferentes níveis de análise.

Com a popularização dos computadores, conceitos como informação e computação (este último podendo ser visto como um processo de transformação de informação) tornaram-se populares. Tais conceitos tornaram-se populares, mas, por outro lado, fortemente associados à artefatos tecnológicos do nosso dia a dia. Com isso, quando descrevemos processos computacionais no mundo natural, como moléculas de DNA armazenando informação, ou cérebros processando informação, é comum acharmos que estamos apenas usando metáforas inspiradas pela tecnologia corrente e esquecendo que o conceito de informação e de computação estão presentes no mundo natural que podem ser estudados como objetos em si, independentes de qualquer contingência tecnológica. Deste modo, quando queremos estudar processos computacionais a fundo, uma das primeiras tarefas que temos pela frente é tentar nos abstrair das tecnologias correntes e sermos extremamente precisos ao falamos de conceitos tão fundamentais quanto algoritmos e computadores.

O fato de que perguntas motivadas por curiosidade intelectual nos forcem a colocar a computação sobre bases sólidas não é supreendente e, embora empolgue o leitor com perfil de cientista, pode alienar o leitor com interesse em problemas práticos mais imediatos. Entretanto, é comum que perguntas bastante práticas também exijam o entendimento aprofundado das ideias que desenvoveremos neste livro. Tais perguntas costumam vir de áreas como, por exemplo, computação quântica ou inteligência artificial. A pergunta recorrente e óbvia que surge em tais contextos é: que tipo de problemas estas novas tecnologias poderão resolver? Ou, ainda, qual é a base científica que faz com que tenhamos razoável confiança de que computadores quânticos ou inteligência artificial são tecnologias viáveis. Não estaríamos tentando construir algo fundamentalmente impossível?

Não precisamos dizer que perguntas como estas são interdisciplinares por natureza e, portanto, para respondê-las devemos não apenas saber sobre computação, mas também sobre outras disciplinas, como física ou biologia. Entretanto, sem sólidas bases científicas em teoria da computação não estaríamos em boa posição para sequer começar a responder perguntas como estas, pois tais perguntas estão intrinsicamente ligadas ao conceito de informação e de transformação de informação. Um dos objetivos centrais deste livro é prover ao estudante os fundamentos da computação para que ele estar em posição para responder perguntas profundas sobre computação de maneira precisa e cientificamente bem informada.

1.2 Algoritmos, problemas e computadores

Além de algoritmos e computadores, um outro conceito que normalmente usamos de maneira intuitiva é o conceito de *problema computacional*. Em nosso dia a dia, nós lidamos com vários problemas computacionais específicos, como o problema de testar se um dado número é primo ou o problema de testar se existe um caminho ligando dois nós em uma rede. Em teoria da computação nós seremos sempre muito precisos e, com isso, uma questão que surge naturalmente é definir exatamente o que *é*, de maneira genérica, um problema computacional.

> DEFINIÇÕES GENÉRICAS E DEFINIÇÕES ESPECÍFICAS
>
> No Capítulo 2 iremos desenvolver algumas ferramentas matemáticas que nos ajudarão a dar definições precisas para o conceito de problema computacional. Mas, mesmo antes de termos estas ferramentas em mãos, vamos refletir brevemente sobre esta ideia de fornecer definições genéricas e, em particular, uma definição genérica para o conceito de problema computacional.
>
> Vamos usar uma analogia para ilustrar o que queremos dizer. Pense na função $f(x) = x^2$ e na função $g(x) = \log x$. Temos aqui dois casos *específicos* e matematicamente bem definidos de funções. Entretanto, sabemos que é perfeitamente possível sermos mais genéricos e fornecermos uma definição para o conceito *genérico* de função[1] tal que as funções $f(x)$ e $g(x)$ anteriores são casos particulares. Ou seja, podemos falar de objetos matemáticos específicos, como $f(x) = x^2$ e $g(x) = \log x$, mas também do objeto matemático genérico que chamamos de *função*. A analogia que queremos fazer aqui é que é possível dar uma definição genérica para problema computacional, tal que os problemas específicos, como testar primalidade de números ou testar a conectividade de redes, são casos particulares.
>
> [1] Para quem está um pouco enferrujado, relembramos que funções podem ser definidas como casos particulares de relações. Para quem também esqueceu o que são relações e não quer ir buscar um livro de matemática discreta, basta lembrar que relações são definidas como o subconjuntos do produtos cartesiano de uma sequência de conjuntos.

Uma vez que começamos a falar de algoritmos e problemas computacionais, uma pergunta que pode surgir é se existe um dado problema para o qual não existe nenhum algoritmo que o resolva. Note que não estamos falando de problemas para os quais nós, hoje, ainda não conhecemos algoritmos que os solucionem, mas que eventualmente venhamos a descobrir tais algoritmos no futuro. O que estamos perguntando aqui é se existem problemas que não podem ser resolvidos por nenhum algoritmo concebível (dentre os infinitos possíveis). Para lidar com esse tipo de questão, além da necessidade de sermos precisos sobre o que é um problema (discutimos brevemente isso no quadro anterior), também temos que ser precisos sobre o que significa um algoritmo.

Neste ponto alguém pode começar a questionar a necessidade de sermos assim tão precisos. A importância de sermos precisos é que somente assim podemos demonstrar matematicamente teoremas do tipo *"existe um problema x tal que para todo y, sendo que y é um elemento do conjunto infinito de todos os possíveis algoritmos, y não é uma solução para o problema x"*.

Aqui reforçamos a breve discussão do quadro anterior a respeito de definições genéricas e definições específicas. No capítulo 5, veremos uma definição matemática genérica para o conceito de algoritmo, de maneira que, todos algoritmos, que também são objetos matemáticos precisos, são instâncias particulares desta definição genérica. Ao contrário do que acontece com a definição matemática do que é um problema computacional, a definição matemática precisa de algoritmo é mais trabalhosa. Um dos objetivos dos Capítulos 3 e 4 é apresentar definições simplificadas de algoritmos (definições que não capturam todos os algoritmos possíveis). Assim ganharemos um pouco de intuição sobre o assunto e nos sentiremos mais confortáveis para definir precisamente o que é um algoritmo no capítulo 5.

REFLETINDO UM POUCO: LIMITE TECNOLÓGICO OU PRINCÍPIO FUNDAMENTAL?

Na discussão anterior mencionamos a possibilidade de existirem problemas insolúveis. Mas a existência destes problemas não reflete apenas uma limitação tecnológica do que atualmente entendemos por computadores e algoritmos? Ou seja, será que tais problemas que não sabemos resolver agora poderiam ser resolvidos por computadores "exóticos" no futuro? Ou a impossibilidade de resolução de alguns problemas seria algo mais fundamental? Até que ponto podemos descartar a possibilidade de que seja possível construir objetos exóticos (condizentes com as leis conhecidas da física) que poderíamos usar como computadores para resolver estes problemas que os computadores atuais não resolvem?

O consenso científico atual, chamado de Tese de Church-Turing, é que qualquer problema solúvel poderia ser resolvido, em princípio, por um computador como concebemos hoje. Este tipo de questão, como qualquer questão científica, é passível de debate. Entretanto, a maioria das propostas que aparecem na literatura tentando atacar esta tese são, em última análise, variações da antiga ideia de computação analógica, uma ideia que parece esbarrar em alguns obstáculos postos pela física teórica contemporânea. Além disso, do ponto de vista experimental, todas estas propostas tem falhado, e, a cada vez que isso ocorre, o consenso em torno da Tese de Church-Turing é fortalecido, como acontece com teses em qualquer outra área de investigação científica. O Capítulo 6 deste livro é dedicado a discussão da Tese de Church-Turing.

Entretanto, observe que não estamos excluindo aqui a possibilidade de que possamos construir computadores muito mais *eficientes* que os atuais. A pesquisa em computação quântica, por exemplo, investiga precisamente a possibilidade de construção de computadores *exponencialmente* mais rápidos do que os atuais na resolução de alguns problemas específicos.

Embora um curso de teoria da computação seja um curso de matemática, e não de física, para que possamos ententer algumas questões fundamentais da área temos que observar que os objetos abstratos estudados em computação (algoritmos, informação) se manifestam de alguma maneira no mundo físico.

O seu laptop rodando Windows é o caso mais óbvio de um processo computacional ocorrendo em um meio físico, mas podemos pensar em uma série de outros exemplos. A computação do logaritmo de um número ocorrendo em uma máquina com alavancas, engrenagens e graxa, como a máquina de Charles Babbage, seria um deles. Um outro exemplo seria uma "gosma cinzenta", como o cérebro de um primata, processando sinais elétricos sensoriais e executando algoritmos de processamento visual. Poderíamos também pensar em moléculas de DNA executando rotinas biológicas ou mesmo em um punhado de átomos, elétrons e fótons interagindo sistematicamente em algum protótipo de computador quântico, de maneira que a manipulação da informação quântica de destas partículas fazem parte do processo de execução de um algoritmo quântico. A Figura 1.1 ilustra tais exemplos.

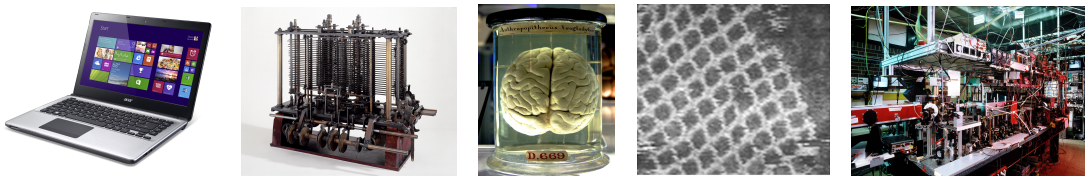


Figure 1.1: Computação ocorrendo em diferentes meios físicos: circuitos eletrônicos, engrenagens mecânicas, estruturas biológicas, moléculas de DNA e partículas subatômicas.

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

Apesar de estarmos repetidamente falando de objetos físicos, já mencionamos que não iremos em lidar com física. O objetivo da teoria da computação é prover a teoria matemática mais geral possível para entendermos processos computacionais. Por processos computacionais, queremos dizer informação e transformações possíveis que esta informação podem sofrer.

O nosso santo graal é um modelo matemático que seja *independente do meio físico*, mas que, ao mesmo tempo, seja abrangente e realista o suficiente tal que se alguma manipulação abstrata de informação seja impossível de se realizar no dado modelo, então a conclusão lógica é que não deve existir nenhum processo físico que realize a tarefa sem que violemos o que sabemos sobre as leis da física. Por quê uma ideia assim seria o nosso santo graal? Por que uma vez que temos um modelo matemático com estas características, basta nos focarmos no modelo para saber o que se pode e o que não se pode ser efetivamente computado.

Claramente este objetivo parece ser bastante ambicioso, pois estamos atrás de um modelo matemático que capture todo e qualquer tipo de manipulação de informação possível no mundo físico[1]. Veremos neste curso que na década de 30 foi proposto um modelo matemático que, apesar de ser razoavelmente simples, parece ter as propriedades que buscamos. Este modelo é conhecido como *Máquina de Turing*. A tese científica que afirma que Máquinas de Turing atingem tal nível de generalidade em termos de poder representar qualquer processo computacional (i.e., transformações possíveis de informação), e que é um dos maiores desenvolvimentos da ciência no século XX, é conhecida hoje em dia como *Tese de Church-Turing*[2].

Os computadores que usamos no dia a dia são objetos precisos e sabemos rigorosamente como eles funcionam (afinal, fomos nós quem os construímos!) e, portanto, temos modelos matemáticos que os descrevem com exatidão. Veremos que os modelos matemáticos que descrevem nossos computadores são equivalentes a certas Máquinas de Turing conhecidas como Máquinas de Turing

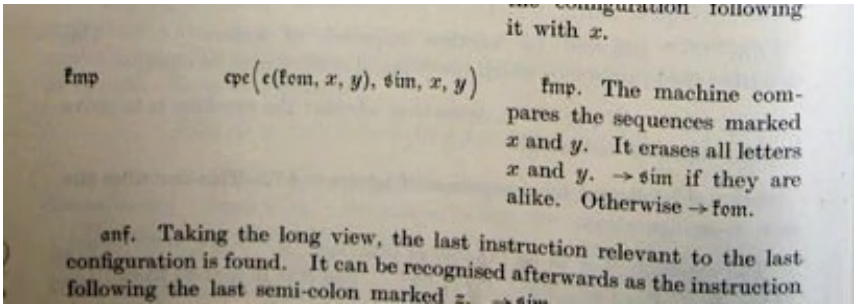
[1] Em particular, a primeira pergunta que pode surgir é: o fato de que nós não conhecemos as leis "finais" da física não seria um impedimento nesta busca? Certamente não, pois levar a sério este tipo de objeção poderia bloquear qualquer tipo de empreendimento científico em qualquer área que seja. O que temos que fazer é o que qualquer área da ciência faz: levar a sério que existe de sólido em física e trabalhar sempre atentando para não contradizer estes fatos sólidos da área.

[2] A Tese de Church-Turing (TCT) tem tipicamente duas interpretações. Uma interpretação é que a TCT é uma definição matemática e a outra, que é a que nos referimos aqui, é que ela é uma afirmação sobre a realidade física. Um ponto fundamental desta segunda versão da TCT é que ela pode, no melhor espírito científico, ser refutada. Isso é importante, pois as únicas hipóteses que importam para a ciência são aquelas que podem ser refutadas por experimentos. Veremos também quais foram as razões que foram tornando esta tese mais e mais sólida com o tempo.

Universais. A propriedade essencial de tais máquinas universais é que elas podem simular[3] qualquer outra Máquina de Turing, e, portanto, pode simular qualquer outro processo computacional com alguma correspondência no mundo físico. Isso significa que, pelo menos em princípio, não existem objetos físicos (o que inclui uma certa espécie de primatas) que possam resolver problemas que não possam ser resolvidos por computadores atuais devidamente programados e com as condições de eficiência e de memória adequadas[4]. A definição matemática de uma Máquinas de Turing Universal é o modelo matemático para o que chamamos de *computador*.

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

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



it with x.

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

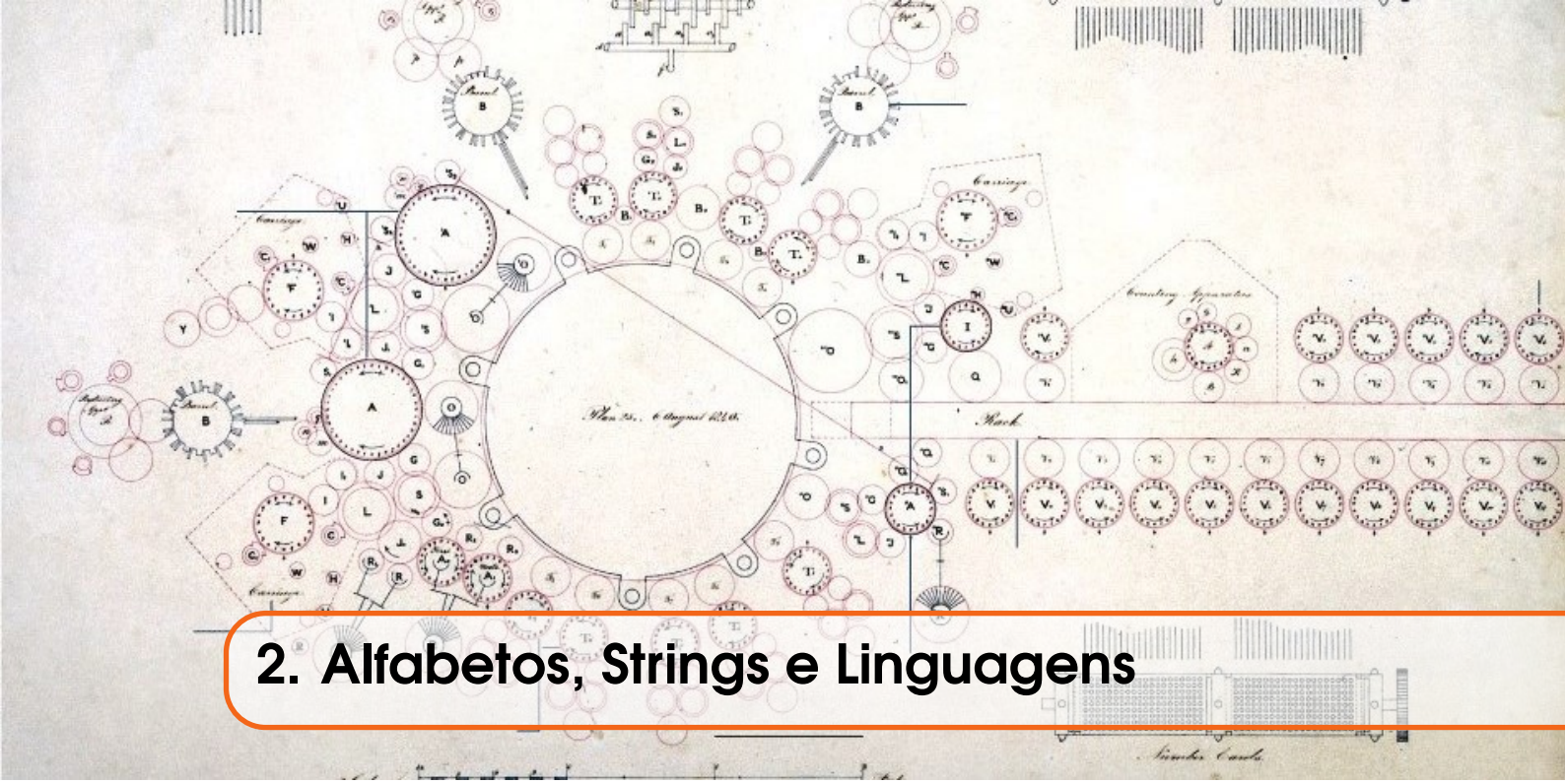
anf. Taking the long view, the last instruction relevant to the last configuration is found. It can be recognised afterwards as the instruction following the last semi-colon marked

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

[3] A palavra "simular" tem um significado matematicamente preciso que, a grosso modo, significa uma certa correspondência de um para um entre dois modelos matemáticos (ou estados e evoluções no tempo dos estados de sistemas físicos, se estivermos falando de objetos no mundo real).

[4] Claramente afirmações assim podem gerar discussões de natureza filosófica, e isso é bastante comum em algumas áreas da ciência quando suas conclusões lógicas são levadas a sério até as últimas consequências e no mundo acadêmico discussões são extremamente saudáveis. Ainda assim, o objetivo deste material é se ater a apresentar o que é consenso científico especificamente em teoria da computação. Ainda assim o leitor interessado é instigado pensar a respeito de algumas de tais consequências.

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 qualquer e *símbolos* são elementos deste conjunto.

■ **Exemplo 2.1** Considere o conjunto $X = \{a, b, c, d\}$. Os elementos a, b, c, d do conjunto X são chamados de *símbolos do alfabeto* X. Note que o conceito de alfabeto nada mais é do que um sinônimo para o conceito matemático de conjunto. ■

Iremos usar vários alfabetos neste livro, sendo que o mais utilizado será o alfabeto $\Sigma = \{0, 1\}$, conhecido como *alfabeto binário*. No mundo das liguagens de programação um alfabeto bastante comum é $\Sigma_{\text{ANSIC}} = \{0, 1, 2, ..., 9, a, b, ..., z, A, B, ..., Z, ..., \&, \#, \%, >, <, ...\}$, que não vamos enumerar símbolo a símbolo aqui, mas que trata-se do 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 deste capítulo se refere exatamente a isso.

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

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

00101 é uma string sobre o alfabeto binário. Um outro exemplo de string é um programa escrito em linguagem C (note que tal programa é uma sequência de símbolos ANSI, incluindo os símbolos especiais usados para quebra de linha, indentação e espaçamento), que pode ser visto matematicamente como uma string sobre o alfabeto Σ_{ANSI}, que mencionamos anteriormente.

Notação 2.1. *Em geral usamos a letra grega* Σ *para denotar alfabetos. No caso em que o alfabeto não estiver explicitamente definido,* Σ *denotará, por convenção, o alfabeto binário* $\{0,1\}$.

Uma *substring* de w é uma subsequência de símbolos de w. Por exemplo, ab, bb e bc são algumas substrings de $abbbbc$. A concatenação de duas strings x e y é a string resultante da justaposição das strings x e y, denotada xy. Por exemplo, a concatenação de $x = 111$ e $y = 000$ é a string $xy = 111000$.

Definição 2.1.3 — Tamanho de uma string. O *tamanho* de uma string w é o número de símbolos de w, e é denotado por $|w|$. Denotamos a *string nula*, ou seja, a string que não contém nenhum símbolo, por ε.

■ **Exemplo 2.2** Dada as strings $aabc$, 001 e ε, usando a nossa notação para tamanho de strings, escrevemos $|aabc| = 4$, $|001| = 3$ e $|\varepsilon| = 0$. ■

Em teoria da computação será bastante comum usar raciocínio indutivo. Este tipo de raciocínio não será útil apenas em demonstrações de teoremas, mas também em várias definições. A seguir, nosso objetivo é definir o que é a reversa de uma string. Intutitivamente, 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 $w = 1100$, então $w^{\text{R}} = 0011$). Vamos formalizar isso usando uma definição indutiva:

Definição 2.1.4 — Reversa de uma string. A reversa de uma string w, denotada w^R, é definida recursivamente da seguinte maneira:

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

Seja Σ um alfabeto e $a \in \Sigma$. Quando escrevemos que uma string sobre este alfabeto é da forma a^n, queremos dizer que a string é formada por n símbolos a consecutivos. Se w é uma string e escrevemos w^n, queremos denotar a concatenação de n strings w. Por exemplo, a string 11111 pode ser escrita como 1^5. Para mais um exemplo, considere a string $w = 10$. Com isso a string 101010101010 pode ser escrita como w^6. Em particular, $x^0 = \varepsilon$ para qualquer string x. No Exercício 2.1, pedimos para o aluno fornecer uma definição formal para o conceito de concatenação de strings.

Definição 2.1.5 — 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: $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$.

Definição 2.1.6 — O conjunto Σ^*. Dado um alfabeto Σ, definimos Σ^* como o conjunto de todas as strings sobre Σ, ou seja,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \ldots$$

2.1.1 Exercícios

Exercício 2.1 Nesta seção, fornecemos uma definição informal para a concatenação de n strings x, denotada x^n. Forneça uma definição formal indutiva para x^n. ■

Exercício 2.2 É verdade que $x^0 = y^0$ para qualquer par de strings x e y? ■

Exercício 2.3 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?
(d) Para qualquer string w, $|w| = |w^R|$?

■

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*.

Vamos a alguns exemplos de linguagens binárias que serão bastante comuns neste curso:

■ **Exemplo 2.3** Dado $\Sigma = \{0,1\}$, são linguagens sobre Σ:

$L_{01} = \{\varepsilon, 01, 0011, 000111, \ldots\}$ = "Linguagem das strings da forma 0^n1^n, para $n \geq 0$".
$L_{\text{PAL}} = \{\varepsilon, 0, 1, 00, 11, 010, 101, 1001, 0110, 0000, \ldots\}$ = "Linguagem dos palíndromos".
$L_{\text{P}} = \{10, 11, 101, 111, 1011, \ldots\}$ = "Linguagem dos números primos (em binário)".
$L_{\text{SQ}} = \{0, 1, 10, 100, 1001, 10000, \ldots\}$ = "Linguagem dos quadrados perfeitos (em binário)".

■

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

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

Observe que strings e linguagens são objetos matemáticos diferentes. Enquanto strings são sequências, linguagens são conjuntos. Enquanto strings são objetos construídos a partir de um número finito de elementos fundamentais (símbolos), linguagens podem conter uma quantidade finita ou infinita de elementos.

LINGUAGENS NATURAIS E 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 no dia a dia para nos comunicarmos. Vamos a alguns exemplos do que queremos dizer:

■ **Exemplo 2.4** Seja $\Sigma_{\text{P}} = \{a, b, c, ..., z\}$ (i.e., o conjunto das letras do alfabeto da lingua portuguesa). Podemos pensar no conjunto L de todas as palavras da língua portuguesa como uma linguagem. Por exemplo, $bola \in L$, $caneta \in L$, $inconstitucional \in L$. ■

■ **Exemplo 2.5** Seja $\Sigma_{\text{P}} = \{a, b, c, ..., z, \sqcup\}$. Se usarmos o símbolo $\sqcup$ como separador de palavras, poderíamos montar um frase como $a \sqcup bola \sqcup verde \sqcup escura$. Se ignorarmos pontuação, poderíamos pensar na língua portuguesa como sendo o conjunto L' de todas as frases válidas da lingua portuguesa sobre Σ_{P}. ■

Neste ponto, algum linguista computacional 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 argumentar que é possível sermos formais definindo L como sendo o conjunto de todas as palavras de algum dicionário específico fixado a priori, mas, para a definição exata de L', admitidamente teríamos uma tarefa bem mais difícil pela frente.

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

L_{C} = "conjunto de todos os programas válidos em linguagem C"

A linguagem L_{C}, definida acima, consiste de todos os (infinitos) possíveis programas válidos escritos em linguagem C. Embora usamos a expressão informal "conjunto de todos os programas válidos em linguagem C" para definir L_{C}, esta linguagem pode ser matematicamente bem definida usando o conceito matemático de *gramática*. Na Seção 4.3 veremos formalmente o que é uma gramática e discutiremos brevemente algumas aplicações deste conceito em construção de compiladores e linguagens de programação.

REFLETINDO UM POUCO: PROBLEMAS COMPUTACIONAIS E LINGUAGENS

O que significa *"o problema de testar se um número n é primo"*? Uma maneira de tentar formalizar isso é pensar que isso é equivalente ao seguinte: 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 2.3.

Para definirmos a linguagem dos números primos, se quiséssemos, também poderíamos fixar outros alfabetos diferentes do binário (e.g., o alfabeto dos dígitos de 0 a 9), mas isso não é tão importante. O ponto central aqui é que uma vez fixado 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 um dado objeto pertence ou não ao conjunto L_{P}.

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

Para quem ainda não está convencido da conexão entre linguagens e problemas, uma maneira intuitiva de pensar a respeito disso é a seguinte. No funcionamento interno de um computador, qualquer objeto matemático é uma sequência de bits. No caso particular do exemplo acima, quando escrevemos um programa para testar se um dado número é primo, em última análise, 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 pertence a linguagem L_P. Não é muito difícil de ver que esse tipo de raciocínio pode ser generalizado para uma série de outros problemas em queremos testar se um dado objeto tem uma certa propriedade.

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

Notação 2.2. *O número natural reprentado 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 $N(w_2) = 15$. Isso torna nossa notação 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 $L_P = \{w \,;\, N(w) \text{ é um número primo}\}$ e $L_3 = \{w \,;\, N(w) \text{ é um múltiplo de } 3\}$.

2.2.1 Exercícios

Exercício 2.4 Seja Σ um alfabeto arbitrário. Com relação a linguagens sobre Σ, responda:

(a) Σ^* é uma linguagem?

(b) $\varnothing$ é uma linguagem?

(c) $\{\varepsilon\}$ é uma linguagem?

(d) O conjunto potência Σ^4 é uma linguagem?

■

Exercício 2.5 Forneça uma definição formal para a linguagem sobre $\Sigma = \{0,1\}$, das strings que representam números múltiplos de 4 escritos em binário. ■

Exercício 2.6 Seja Σ um alfabeto qualquer. É verdade que a linguagem $\{w \,;\, \exists x \in \Sigma, w = xx^R\}$ é a linguagem de todos os palíndromos construídos com símbolos de Σ? Em caso afirmativo, prove. Em caso negativo, forneça um contra-exemplo e, em seguida, forneça uma definição formal adequada para a linguagem dos palíndromos construídos com símbolos de Σ. ■

2.2.2 Operações com linguagens

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

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

■ **Exemplo 2.6** Considere as duas seguintes linguagens: $L = \{0\}$ e $L' = \{1\}$. A concatenação destas duas linguagens é $L \cdot L' = \{01\}$. ■

■ **Exemplo 2.7** Vamos considerar agora um exemplo mais interessante. Se $L = \{aba, bola, lata\}$ e $M = \{x, xx, xxx, ...\}$, a linguagem LM é $\{abax$, $abaxx$, $abaxxx$, ... , $bolax$, $bolaxx$, $bolaxxx$, ... , $latax$, $lataxx$, $lataxxx$, ...$\}$. ■

Exercício resolvido 2.1 Seja Σ o alfabeto binário e L_3 e R as linguagens sobre Σ definidas a seguir: $L_3 = \{w$; $N(w)$ 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: $L_3R = \{w$; $N(w)$ é um múltiplo de 6$\}$. ■

Exercício 2.7 Seja Σ o alfabeto binário e L_3 e R as linguagens sobre Σ definidas a seguir: $L_3 = \{w$; $N(w)$ é um múltiplo de 3$\}$ e $S = \{\varepsilon, 0\}$. Qual é a linguagem L_3S? ■

Exercício resolvido 2.2 Seja Σ o alfabeto binário. Lembrando que o conjunto Σ^* é uma linguagem (este conjunto, que é a união infinita de todas as potências de Σ, contém todas as strings binárias) e sendo $R = \{0\}$. Qual é a linguagem Σ^*R? E qual é a linguagem Σ^*RR?

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

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

■

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

Definição 2.2.3 Fechamento (ou Fecho de Kleene) de L é 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. Formalmente a definição é:

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

■ **Exemplo 2.8** Seja $L = \{000, 111\}$. Então L^* = {ε, 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, em ambos os casos, note que A^* é uma linguagem. Mais precisamente a linguagem de todas as strings construídas usando como "tijolos básicos" elementos do conjunto A. Caso A seja um alfabeto, estes tijolos básicos são símbolos, caso A seja uma linguagem, estes tijolos básicos são strings.

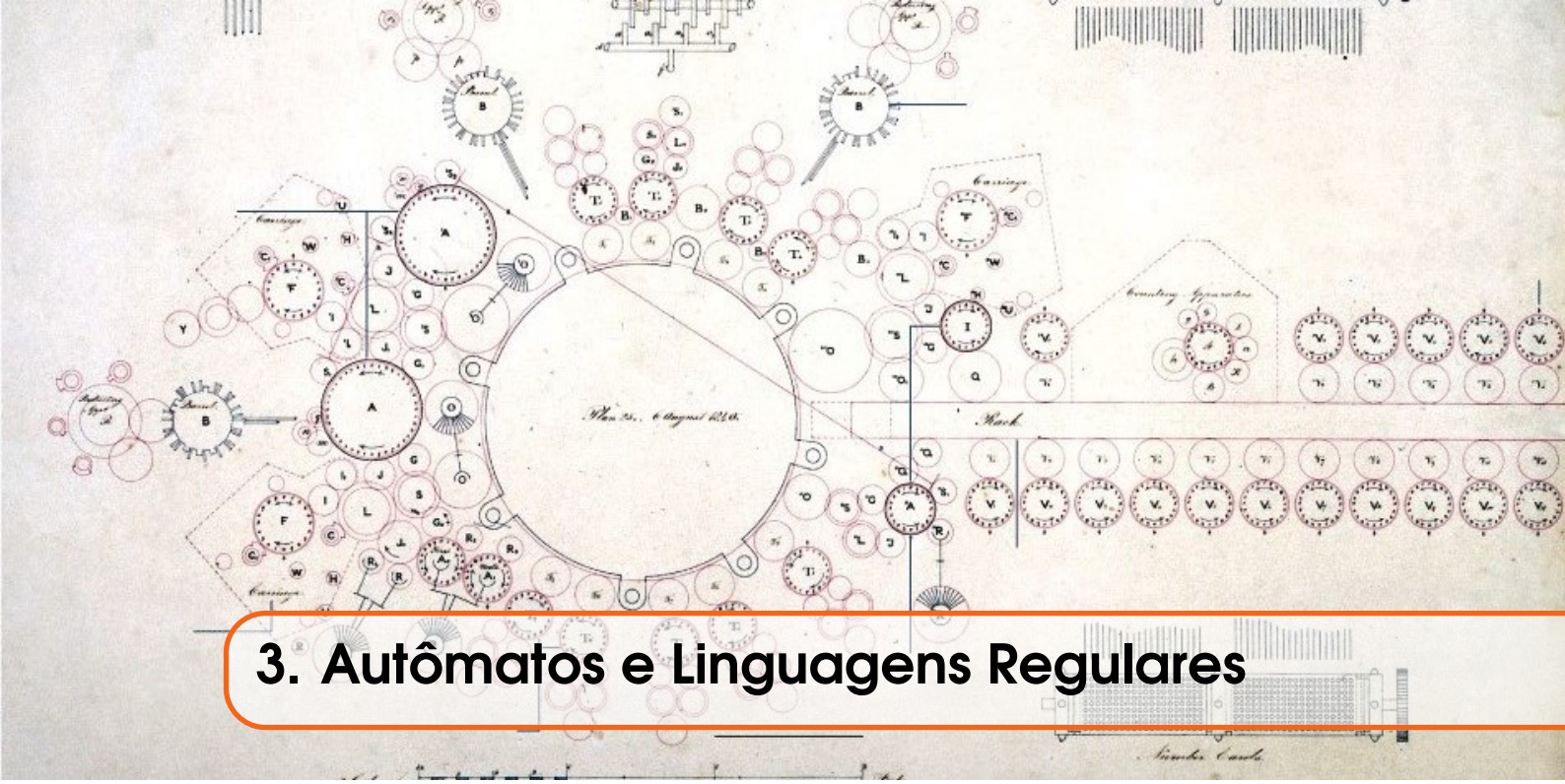
Exercício resolvido 2.3 Considere as linguagens $L = \{1\}$ e $R = \{0\}$. Qual é a linguagem LR^*?

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

Exercício 2.8 Nas questões abaixo, A é sempre um alfabeto e L é sempre uma linguagem. Responda verdadeiro ou falso e justifique sua resposta:

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

■



3. Autômatos e Linguagens Regulares

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

3.1 Autômatos Finitos Determinísticos (DFAs)

Considere a figura abaixo de uma máquina que vende chocolates que custam 6 libras juntamente com um diagrama que descreve o mecanismo de funcionamento interno desta máquina.

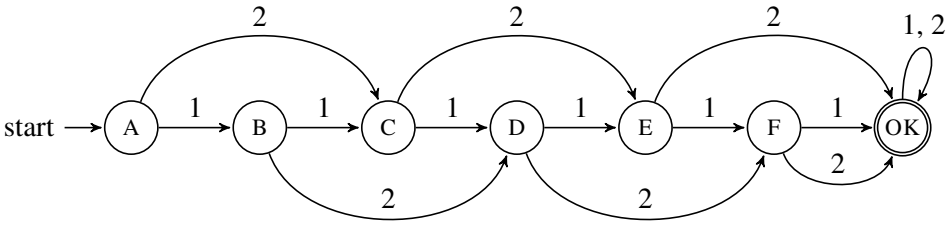


Figure 3.1: Uma máquina que vende chocolates de £6 e aceita moedas de £1 e £2. Ao lado o diagrama simplificado descrevendo o seu mecanismo de funcionamento interno. Por exemplo, se inserirmos uma moeda de £1 e uma moeda de £2, a máquina, atingirá o estado D. A máquina atingirá o estado OK se for inserida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

Diagramas como o da Figura 3.1 são conhecidos como *máquinas de estados* ou *autômatos finitos determinísticos*. A ideia é que a entrada da máquina é uma sequência de moedas (se quisermos, podemos pensar que a entrada é uma "string de moedas"), que podem ser de 1 libra ou 2 libras. A máquina libera o chocolate se a sequência de moedas somar pelo menos 6 libras. O estado A é marcado com a indicação *start* por que este é o estado inicial da máquina. A partir do estado A, a máquina faz uma transição para cada moeda de entrada e libera o chocolate se ao final do processo ela atinge o *estado* OK. Para simplificar supomos que a máquina não fornece troco.

REFLETINDO UM POUCO: MODELOS MATEMÁTICOS E OBJETOS FÍSICOS

Nós sabemos muito bem que desenhos (como o diagrama da Figura 3.1), podem ser ambíguos e geralmente não são considerados definições matemáticas. Nesta seção nós iremos traduzir este diagrama em termos de um conceito matemático preciso. Entretanto, por questões de conveniência, é bastante comum usarmos diagramas no lugar de definições matemáticas.

Um ponto que queremos ressaltar aqui é o seguinte. No Capítulo 1 vimos que podemos pensar em computadores como "instanciações de objetos abstratos e seus possíveis relacionamentos matemáticos em objetos físicos e seu conjunto de possíveis graus de movimento". O que a Figura 3.1 apresenta é exatamente isso (ignorando temporariamente o fato de que estamos usando um diagrama ao invés de uma definição matemática). Para entender isso note o seguinte:

(1) A máquina de chocolates é um objeto físico (embora devemos estar atentos que ela não é um computador de propósito geral, pois ela implementa um único algoritmo, que é o algoritmo que testa se a soma das moedas de entrada é maior ou igual a 6 libras).

(2) O diagrama faz o papel do objeto matemático abstrato, em particular, um objeto matemático que consiste de um conjunto de estados e um conjunto de símbolos (o conjunto $\{1, 2\}$ dos símbolos que aparecem nos rótulos das transições). O diagrama também representa certas relações matemáticas, como os relacionamentos entre estados e símbolos indicados pelas setas saindo de um estado e indo para outro. A definição exata destas relações ficará clara no decorrer desta seção.

O ponto central é que *este objeto matemático e suas relações matemáticas estão "amarradas" em uma correspondência de um para um com o objeto físico e seus graus de movimento* (os graus de movimento são os tipos de movimentações que os mecanismos internos da máquina de vender chocolate podem sofrer).

3.1.1 Modelando matematicamente autômatos

Veremos agora uma definição precisa para diagramas como o que vimos na seção anterior.

Definição 3.1.1 — Autômato Finito Determinístico (DFA). 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 finais*.

Normalmente, diremos simplesmente *autômatos* ou usaremos a sigla *DFA* para se referir aos autômatos finitos determinísticos[1]. Algo importante de se observar é que esta definição é genérica,

[1] A sigla DFA vem da expressão em inglês "deterministic finite automata."

ou seja, se quisermos uma definição matemática particular para o diagrama da nossa máquina de chocolates, teríamos que dar uma *instância específica da Definição 3.1.1*. O exemplo a seguir ilustra o que queremos dizer com isso.

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

$Q = \{\mathrm{A}, \mathrm{B}, \mathrm{C}, \mathrm{D}, \mathrm{E}, \mathrm{F}, \mathrm{OK}\}$;
$\Sigma = \{1, 2\}$;
$\delta_c : Q \times \Sigma \to Q$ é a função definida caso a caso a seguir:
$\delta_c(\mathrm{A}, 1) = \mathrm{B}$, $\delta_c(\mathrm{A}, 2) = \mathrm{C}$, $\delta_c(\mathrm{B}, 1) = \mathrm{C}$, $\delta_c(\mathrm{B}, 2) = \mathrm{D}$, $\delta_c(\mathrm{C}, 1) = \mathrm{D}$, $\delta_c(\mathrm{C}, 2) = \mathrm{E}$,
$\delta_c(\mathrm{D}, 1) = \mathrm{E}$, $\delta_c(\mathrm{D}, 2) = \mathrm{F}$, $\delta_c(\mathrm{E}, 1) = \mathrm{F}$, $\delta_c(\mathrm{E}, 2) = \mathrm{OK}$, $\delta_c(\mathrm{F}, 1) = \mathrm{OK}$, $\delta_c(\mathrm{F}, 2) = \mathrm{OK}$,
$\delta_c(\mathrm{OK}, 1) = \mathrm{OK}$, $\delta_c(\mathrm{OK}, 2) = \mathrm{OK}$.
O estado A é o estado inicial;
O conjunto unitário $\{\mathrm{OK}\}$ é o conjunto de estados finais. ■

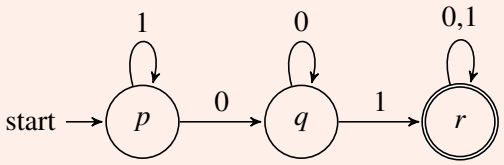
Neste momento pode ser instrutivo lembrar da analogia que fizemos no Capítulo 1, quando discutimos definições genéricas e uma definições específicas. Naquela ocasião mencionamos a definição matemática genérica de "função" e as definições matemáticas de funções particulares, como x^2 ou $\log x$. Na definição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

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

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

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

Solução:



■

A função δ do enunciado do Exercício 3.1 é um tipo de função que é definida ponto a ponto (ou seja, ela não tem uma regra geral como como funções que estamos acostumados a estudar em cálculo, como $f(x) = x^2$, por exemplo). Em teoria da computação, tais tipos de função serão as que vamos lidar com mais frequência. Para facilitar a nossa vida, vamos descrever δ usando uma tabela, como esta apresentada no exemplo a seguir:

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

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

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

No decorrer deste curso, quando formos nos referir a DFAs nós vamos usar tanto 5-tuplas quanto tabelas de transição ou diagramas, dependendo do que for mais conveniente.

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

3.1.2 Aceitação e rejeição de strings

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

A ideia central é ver o DFA como um objeto que receba uma string $w = w_1 w_2 ... w_n$ como entrada, leia esta string símbolo a símbolo, começando com w_1 e terminando com w_n e, no final, nos dê alguma resposta. Durante o processamento desta string, a cada vez que um símbolo w_i é lido, o DFA o descarta w_i e faz uma mudança de estado (uma transição). O DFA irá, passo a passo, descartando símbolos de w e mudando de estados neste processo. O ponto central é o seguinte:

- Em que estado o DFA se encontra após descartar o último símbolo de w?

Digamos que depois da última transição de estados o DFA atinja um dado estado q. Se o estado q for um estado final, nós iremos dizer que o DFA *aceitou* a string w. Se q não for um estado final, nós iremos dizer que o DFA *rejeitou* a string w. Os estados finais também são chamados de estados de aceitação do autômato.

O que precisamos fazer agora é transformar esta ideia intuitiva de que um DFA aceita certas strings e rejeita outras strings em definições matemáticas precisas. Ou seja, queremos saber matematicamente o que significa um certo DFA $(Q, \Sigma, \delta, q_0, F)$ aceitar ou rejeitar uma dada string w. Mas antes disso, vamos a um exercício sobre a ideia intuitiva aceitação e rejeição de strings.

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

Solução: Podemos usar o mesmo DFA usado no Exercício 3.1. ∎

3.1.3 Definição formal para aceitação e rejeição de strings

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

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

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

Indução: $|w| > 0$.
Seja w uma string da forma $w = xa$, sendo que $x \in \Sigma^*$ e $a \in \Sigma$. Então $\hat{\delta}(q,w) = \delta(\hat{\delta}(q,x),a)$.

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

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

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

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

Agora segue a definição mais importante deste capítulo:

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

3.1.4 Exercícios

Exercício 3.2 Seja $\Sigma = \{0,1\}$, construa DFAs que aceitem as linguagens
(a) $L = \{w;\ w \text{ tenha um número ímpar de 1's}\}$.
(b) $L = \{w;\ |w| \leq 3\}$.
(c) $L = \{w\ ;\ w \text{ tem ao mesmo tempo um número par de 0's e um número par de 1's}\}$
(d) $L = \{w;\ \text{se } w' \text{ é uma substring de } w \text{ com } |w'| = 5, \text{ então } w' \text{ tem pelo menos dois 0's}\}$.
(e) $L = \{w;\ w \text{ é terminada em } 00\}$.
(f) $L = \{w;\ \text{o número de 0's em } w \text{ é divisível por 3 e o número de 1's em } w \text{ é divisível por 5}\}$.
(g) $L = \{w;\ w \text{ contém a substring } 011\}$. ∎

Exercício 3.3 Seja um DFA com alfabeto Σ e conjunto de estados Q. Sejam $x, y \in \Sigma^*$, $a \in \Sigma$ e $q \in Q$ quaisquer.

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

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

Exercício 3.4 Considere o seguinte DFA $D = (Q, \Sigma, \delta, q_0, F)$ tal que

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

Qual é a linguagem aceita por D? ■

Exercício 3.5 Forneça um DFA com alfabeto $\Sigma = \{0, 1\}$ que aceite a seguinte linguagem:

$L = \{w \in \Sigma^*;\ w$ seja a representação em binário de um número múltiplo de $3\}$.

Em outras palavras, o DFA deve aceitar strings como 0, 11, 110, 1001, etc. Para simplificar vamos assumir que a string ε representa o número zero (ou seja, o número 0 é representado tanto pela string 0 quanto pela string ε). ■

Exercício 3.6 Prove formalmente que a sua solução para o Exercício 3.5 é correta. ■

Exercício 3.7 Seja uma linguagem L sobre o alfabeto Σ. O complemento da linguagem L, denotado por $\overline{L}$, é definido da seguinte maneira: $\overline{L} = \Sigma^* \setminus L$. Prove que se L é regular, então $\overline{L}$ também é regular. ■

Exercício 3.8 Considere os seguintes DFAs: $D_Q = (Q, \Sigma, \delta_Q, q_0, F_Q)$ e $D_P = (P, \Sigma, \delta_P, p_0, F_P)$ sendo $Q = \{q_0, q_1, ..., q_k\}$ e $P = \{p_0, p_1, ..., p_h\}$.

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

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

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

3.2 Autômatos Finitos não Determinísticos

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

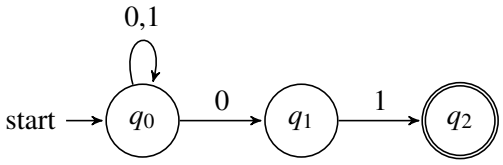


Figure 3.2: Um autômato finito não determinístico.

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

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

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

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

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

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

Em teoria da computação este tipo de situação é razoavelmente comum. O ponto chave é que estes modelos podem ser pensados, em última análise, como *ferramentas matemáticas úteis* no estudo de modelos concretos de computação. No Capítulo 9, veremos que Máquinas de Turing não determinísticas podem ser usadas para definir um conjunto de linguagem conhecido como *NP*, que é parte da famoso (e concreto) problema *P vs NP*. Em um curso mais aprofundado de complexidade computacional a definição de modelos "irreais" de computação, mas que ainda assim sejam matematicamente úteis para lidar com problemas ou modelos concretos de computação, acontece com muita frequência.

3.2.1 Definição formal para autômatos finitos não determiníticos

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

Definição 3.2.1 — Autômato Finito não Determinístico (NFA). Um Autômato Finito não Determinístico, também chamado de NFA, 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 3.2** A definição formal para o diagrama da Figura 3.2 é o NFA $N = (Q,\Sigma,\delta,q_0,F)$, tal que $Q = \{q_0,q_1,q_2\}$, $\Sigma = \{0,1\}$ e a função δ é definida abaixo:

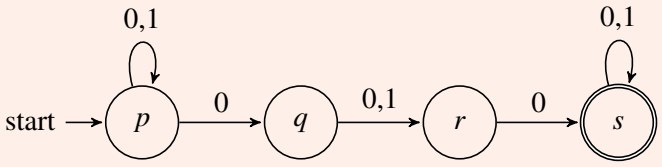
$\delta(q_0,0) = \{q_0,q_1\}$
$\delta(q_0,1) = \{q_1\}$
$\delta(q_1,1) = \{q_2\}$
$\delta(q_1,0) = \delta(q_2,0) = \delta(q_2,1) = \emptyset$ ■

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

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

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

Solução:



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

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

Base: $w = \varepsilon$

A árvore contém apenas o nó raíz q

Indução: $w \neq \varepsilon$

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

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

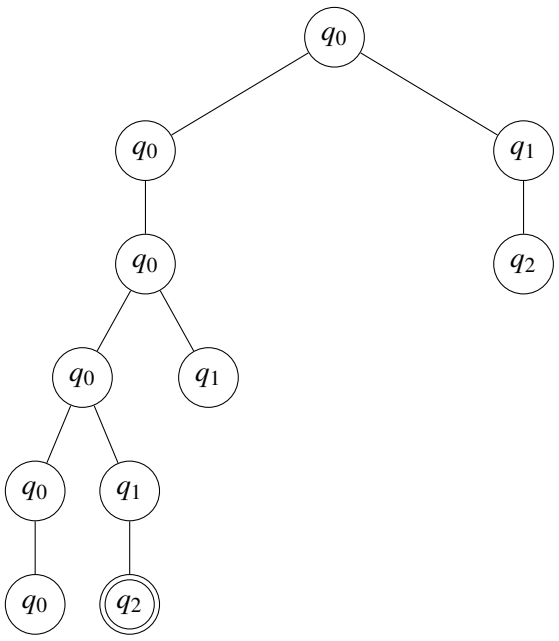


Figure 3.3: A árvore de computações possíveis do NFA N da Figura 3.2 com a string 01001.

Observe que o *nível 0* da árvore (i.e., a raiz) é o estado inicial do NFA e o nível i contém todos os estados que o NFA pode estar depois de i transições do autômato ao processar os i primeiros símbolos da string de entrada. Em particular, dada uma string de tamanho n, o NFA aceita a string se e somente se o n-ésimo nível da árvore contém **pelo menos um estado final**. No exemplo da Figura 3.3 o nível 5 da árvore contém o estado q_2. O fato de que neste nível existe apenas um nó que é um estado final significa que o NFA tem exatamente uma computação possível que aceita a string 01001. A computação é definida pela sequência de estados do caminho ligando a raíz ao estado final em questão.

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

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

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

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

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

■

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

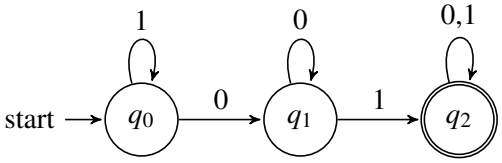


Figure 3.4: Um caso particular em que um NFA que se comporta de maneira determinística.

Exercício 3.9 Desenhe a árvore de computações possíveis para o NFA do Exercício 3.4 com a string de entrada 111010. ■

3.2.2 Aceitação e rejeição de strings por NFAs

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

Definição 3.2.4 Dado um NFA $N = (Q, \Sigma, \delta, q_0, F)$, 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 3.10 Calcule $\hat{\delta}(q_0, 00101)$ do NFA do Exemplo 3.2. ■

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

Exercício resolvido 3.5 Seja $D{=}(Q, \Sigma, \delta, q_0, F)$ um DFA. Apresente um NFA N, tal que $L(D){=}L(N)$.

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

3.2.3 Exercícios

Exercício 3.11 Suponha que queiramos mudar a definição de NFAs para que a *função de transição* tenha a seguinte forma: $\delta : Q \times \Sigma \to (\mathscr{P}(Q) \setminus \varnothing)$. Observe que autômatos segundo esta nova definição nunca morrem. Prove que dado um NFA N, podemos construir um NFA N' segundo nossa nova definição que aceita a mesma linguagem de N. ■

Exercício 3.12 No caso particular de NFAs que são determinísticos, qual é o formato da árvore de computações possíveis? ■

3.3 Equivalência entre DFAs e NFAs

Algo que vamos lidar com frequência neste curso é a diferença de expressividade entre diferentes modelos de computação. Veremos no Capítulo 4 que existem algoritmos que não podem ser expressos na forma de um DFA. Entretanto, tais algoritmos podem ser expressos em outros modelos matemáticos que veremos na sequência deste curso. Neste caso, dizemos que estes outros modelos são mais poderosos (ou mais expressivos) que DFAs.

No *Exercício Resolvido* 3.5 o objetivo foi mostrar que qualquer linguagem que um DFA reconheça, também pode ser reconhecida por um NFA. Isso significa que NFAs são pelo menos tão poderosos como DFAs. Isso é natural, pois NFAs são generalizações de DFAs. A pergunta óbvia que devemos fazer é se NFAs são *estritamente* mais poderosos que DFAs. Veremos nesta seção que a resposta é não. Ou seja, podemos mostrar que se uma linguagem pode ser aceita por um NFA, então existe algum DFA que aceita a mesma linguagem. Isso pode parecer um pouco

surpreendente, pois NFAs, em algumas situações, tem a capacidade de advinhar qual transição deve fazer, uma capacidade que DFAs não tem.

3.3.1 Algoritmo de construção de conjuntos

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

Algorithm 1 Construindo um DFA a partir de um NFA.

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

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

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

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

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

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

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

3.3.2 Algoritmo de construção de conjuntos: versão melhorada

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

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

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

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

Algorithm 2 Construindo um DFA a partir de um NFA: algoritmo melhorado.

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

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

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

Exercício 3.13 Prove o Teorema 3.3.1
(Dica: prove por indução que $\forall w \in \Sigma^*$, $\hat{\delta}_D(q_0, w) \in F_N \Leftrightarrow \hat{\delta}_N(\{q_0\}, w) \cap F_N)$ ■

O próximo teorema enuncia a equivalência entre DFAs e NFAs.

Teorema 3.3.2 Uma linguagem L é aceita por um DFA se e somente se L é aceita por um NFA.

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

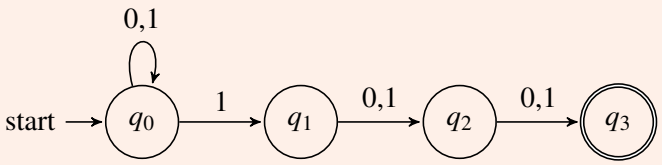
3.3.3 Exercícios

Exercício 3.14 Considere o o NFA dado pela tabela abaixo:

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

Apresente um DFA que aceite a mesma linguagem do NFA acima. ■

Exercício 3.15 Qual a linguagem aceita pelo NFA abaixo?



■

Exercício 3.16 Forneça um DFA equivalente ao NFA do Exercício 3.15. ■

3.4 Autômatos Finitos não Determinísticos com ε-transições

Vamos considerar agora um diagrama de um NFA 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-NFA.

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 clássico livro de Hopcroft, Motwani e Ullman. A ideia é construir um ε-NFA 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);
(3) Possivelmente a descrição do número acaba nos dígitos do item acima, ou, após estes dígitos há um ponto decimal ".";
(4) Opcionalmente, há outra string de dígitos (os dígitos da parte decimal do número);
(5) Faz-se a restrição 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 +–5.5., -8-, 56-, ..56. O autômato é o seguinte:

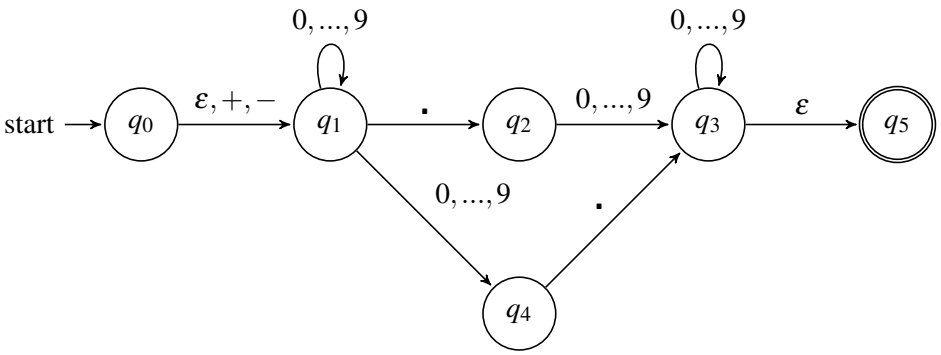


Figure 3.5: Um ε-NFA que aceita números decimais.

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

Definição 3.4.1 — ε-NFA. Um ε-NFA é uma 5-tupla $E = (Q, \Sigma, \delta, q_0, F)$ tal que cada um dos componentes tem a mesma interpretação que um NFA, 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 3.3** O ε-NFA 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	$\varnothing$	$\{q_3\}$	$\varnothing$	$\varnothing$
$^*q_5$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

■

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

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

Definição 3.4.2 — ε-Fecho de estados. Seja um ε-NFA 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).

Agora que temos a Definição 3.4.2 em mãos, podemos definir o conceito de função de transição estendida de um ε-NFA. A ideia é parecida com a definição de função de transição estendida de um NFA, 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-NFAs. Dado um ε-NFA $N = (Q, \Sigma, \delta, q_0, F)$, 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-NFA.

Definição 3.4.4 — Linguagem de ε-NFAs. Seja $E = (Q, \Sigma, \delta, q_0, F)$ um ε-NFA. 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-NFA tal que $L = L(E)$, então dizemos que E aceita L.

3.5 Equivalência entre DFA e ε-NFA

Nesta seção vamos construir um DFA D a partir de um ε-NFA $E = (Q_E, \Sigma, \delta_E, q_0, F_E)$. 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.

Teorema 3.5.1 Se o DFA $D = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ é obtido pelo método visto acima a partir de um ε-NFA $E = (Q_N, \Sigma, \delta_N, q_0, F_N)$, então $L(D) = L(N)$.

Teorema 3.5.2 Uma linguagem é aceita por um DFA se e somente se é aceita por um ε-NFA.

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

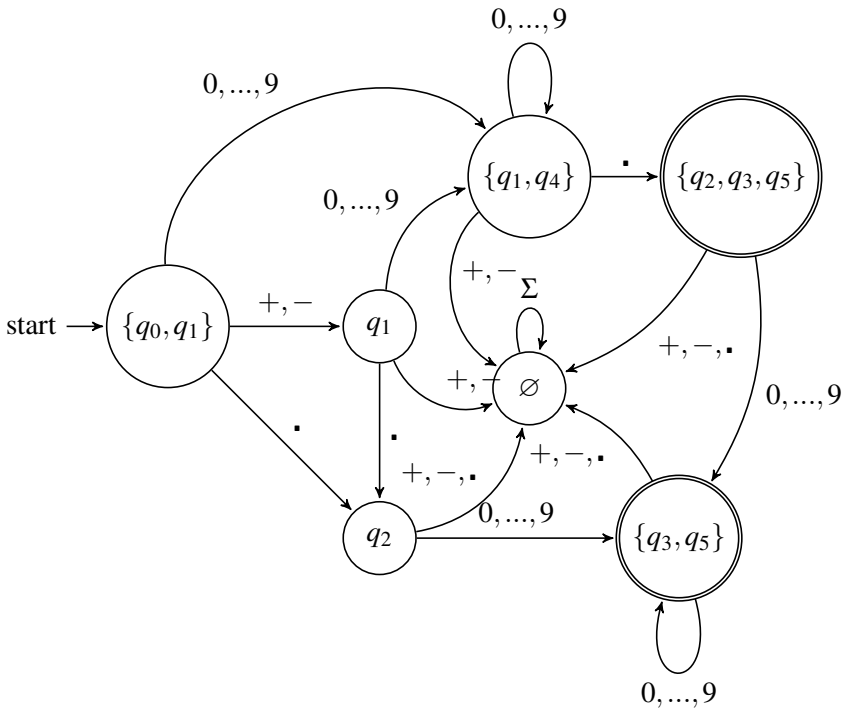
Algorithm 3 Construindo um DFA a partir de um ε-NFA

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

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

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

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



3.5.1 Exercícios

Exercício 3.17 Considere o ε-NFA 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-NFA
(b) Calcule o ε-fecho de cada estado.
(c) Forneça todas strings w tal que $|w| \leq 3$ aceitas pelo autômato.
(d) Converta o ε-NFA em um DFA.

Exercício 3.18 Repita a questão 1 para o ε-NFA dado pela tabela abaixo:

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

3.6 Expressões Regulares

Expressões regulares são expressões que representam linguagens. A ideia é que podemos usar uma expressão como, por exemplo, 01^*0 para denotar uma determinada linguagem. Nesta seção, veremos que esta expressão representa o conjunto de todas as strings que contém "um único 0 seguido por um número arbitrário de 1's seguido de um único 0".

De maneira mais geral, veremos como representar qualquer linguagem regular usando tais expressões. Observe que fizemos uma afirmação forte na frase anterior: "veremos como representar **qualquer linguagem regular** usando tais expressões", ou seja, não existe linguagem regular que não pode ser representada por uma expressão regular da maneira que definiremos abaixo. Mais interessante do que isso, veremos também que **nenhuma expressão regular** representa uma linguagem que não seja regular. Em outras palavras, expressões regulares e autômatos tem precisamente o mesmo poder de expressividade em em termos de linguagens e algoritmos.

3.6.1 Construindo Expressões Regulares

O nosso objetivo agora é definir indutivamente o que tipos de expressões são expressões regulares válidas. Na definição que daremos a seguir, por convenção, expressões regulares são sempre sublinhadas.

Definição 3.6.1 — Expressões Regulares Válidas.

Vamos definir indutivamente quais são as expressões regulares que consideraremos como válidas. O ponto chave é que uma dada expressão regular irá corresponder a uma linguagem. Portanto, ao mesmo tempo que iremos definir quais expressões são válidas, vamos definir exatamente qual linguagem tal expressão estará representando. Para simplificar, a seguinte notação será útil:

Notação: Para uma expressão regular R, iremos denotar $L(R)$ a linguagem expressa por R.

Última observação útil antes de apresentar finalmente a definição: Quando formos nos referir a linguagens, estaremos sendo genéricos e estaremos nos referindo a um alfabeto Σ qualquer. Portanto, quando dizemos $a \in \Sigma$, estamos pensando em a como uma variável que pode assumir qualquer símbolo de Σ. Vamos agora a definição:

Base (expressões regulares elementares):

1. Tanto $\underline{\varepsilon}$ quanto $\underline{\varnothing}$ são expressões regulares e representam, respectivamente, as linguagens $\{\varepsilon\}$ e $\varnothing$. Isto é, $L(\underline{\varepsilon}) = \{\varepsilon\}$ e $L(\underline{\varnothing}) = \varnothing$.
2. Dado $a \in \Sigma$, então $\underline{a}$ é uma expressão regular. Esta expressão representa a linguagem $\{a\}$, ou seja, usando a notação estabelecida, $L(\underline{a}) = \{a\}$.

Indução:

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

■ **Exemplo 3.4** Considere a linguagem $L = \{$"strings contendo 0's e 1's alternados"$\}$. A expressão regular que representa esta linguagem é $\underline{(01)^* + (10)^* + 0(10)^* + 1(01)^*}$. ■

Exercício resolvido 3.6 Justifique passo a passo que a expressão regular do Exemplo 3.4 é uma expressão regular válida e que a linguagem que ela representa é exatamente a linguagem que queríamos representar, i.e., $L = \{\varepsilon, 0, 1, 01, 10, 010, 101, 0101, 1010, 01010, ...\}$.

Solução: O desenvolvimento passo a passo pode ser um pouco tedioso, mas pode ser útil sermos bastante detalhistas pelo menos neste primeiro exemplo.

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

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

Passo 3: Até este ponto já sabemos que $\underline{0}$ e $\underline{1}$ são expressões regulares válidas que representam as linguagens $\{0\}$ e $\{1\}$, respectivamente. Segundo a Definição 3.6.1 (Indução, item 2), concluímos que $\underline{01}$ 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 $(10)^*$ também é uma expressão válida, e $L((10)^*) = \{\varepsilon, 10, 1010, 101010, ...\}$.

Passo 6: Pelos Passos 4 e 5, $(01)^*$ e $(10)^*$ 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), $(01)^* + (10)^*$ é uma expressão válida e $L((01)^* + (10)^*) = \{\varepsilon, 01, 0101, 010101, ...\} \cup \{\varepsilon, 10, 1010, 101010, ...\} = \{\varepsilon, 01, 10, 0101, 1010, 010101, 1010101, ...\}$.

Passo 7: Podemos concluir que $0(10)^*$ é uma expressão regular a partir do fato que 0 e $(10)^*$ 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 $1(01)^*$ é uma expressão regular válida e $L(1(01)^*) = \{1, 101, 10101, 1010101, ...\}$. Com isso, podemos aplicar a Definição 3.6.1 (Indução, item 1) nas expressões $0(10)^*$ e $1(01)^*$ para concluir que $0(10)^* + 1(01)^*$ 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 $E = (01)^* + (10)^* + 0(10)^* + 1(01)^*$ é uma expressão regular válida e $L(E) = \{\varepsilon, 0, 1, 01, 10, 010, 101, 0101, 1010, 010101, ...\}$. ■

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

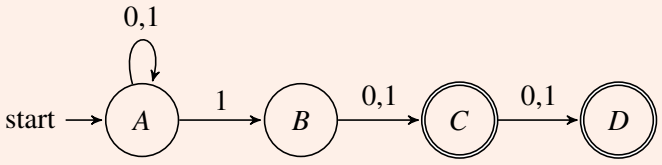
- O operador * tem precedência mais alta e, portanto, deve ser o primeiro a ser aplicado. Com isso, por exemplo, a expressão 01^* é equivalente a $0(1)^*$;
- Na sequência é aplicado o operador de concatenação. Com isso, por exemplo, a expressão $a + bc$ é equivalente a $a + (bc)$;
- Finalmente são aplicadas as uniões, ou seja, o operador $+$;
- Naturalmente usamos parentesis como de costume para alterar a precedência de operadores.

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

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

3.6.2 Exercícios

Exercício 3.19 Obtenha uma expressão regular que represente a linguagem do seguinte ε-NFA:



■

Exercício 3.20 Obtenha um ε-NFA cuja linguagem seja a mesma representada pela expressão regular $(0+1)^*1(0+1)$. ■

Exercício 3.21 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.

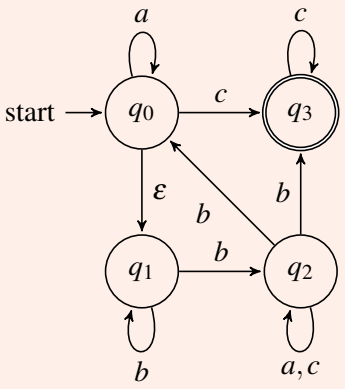
(c) $L \subseteq \Sigma_2^*$ definido por $L = \{w;\ w$ tenha um número par de 0's e um número par de 1's$\}$. ■

Exercício 3.22 Dado o DFA abaixo, encontre uma expressão regular equivalente. Você deve mostrar o desenvolvimento passo a passo de solução.

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

■

Exercício 3.23 Considere o ε-NFA $E = (\{q_0,q_1,q_2,q_3\},\{a,b,c\},\delta,q_0,\{q_3\})$ abaixo:

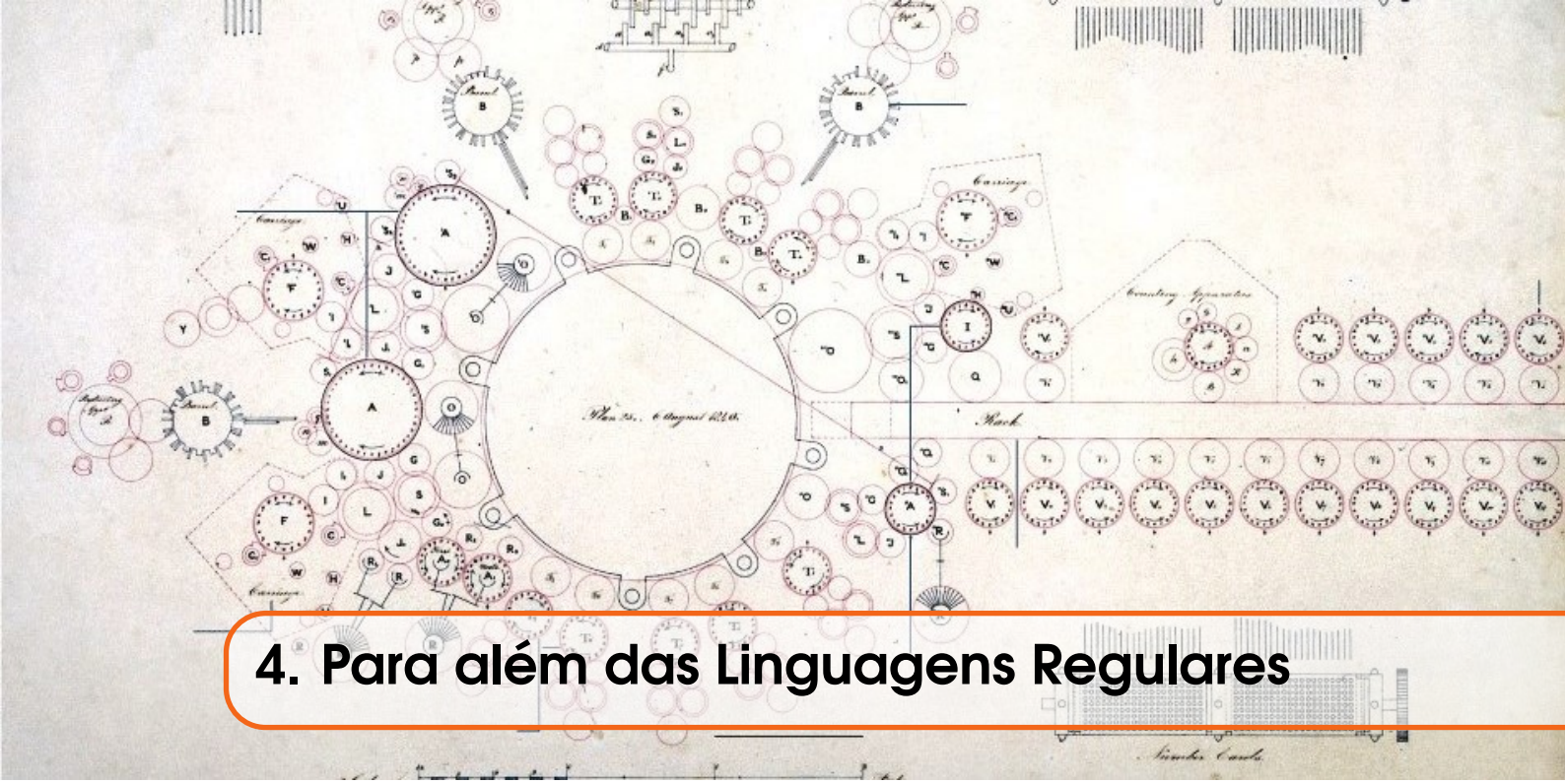


(a) Apresente a tabela de transições da função δ do autômato E.

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

Exercício 3.24 Forneça um ε-NFA que aceite a mesma linguagem da expressão regular $(00+11)^* + (111)^*0$. ■

Exercício 3.25 Construa um ε-NFA que aceite a mesma linguagem da expressão regular $00(0+1)^*$. ■



4. Para além das Linguagens Regulares

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

Se existem linguagens não regulares, como podemos provar tal fato? Primeiramente vamos lembrar que, por definição, uma linguagem é regular se existe um DFA que a aceite. Portanto, dada uma linguagem L, se quisermos mostrar que L é regular, basta que explicitamente apresentemos um DFA a aceite. Entretanto, se quisermos mostrar que L não é regular, temos que provar que não existe nenhum DFA que aceite L. Ou seja, precisamos usar um argumento que exclua logicamente a possibilidade de que cada um dos infinitos possíveis DFAs tenha a propriedade de ser um DFA que aceite L. Este tipo de resultado tende a ser bem mais difícil de se obter. Nosso primeiro objetivo neste capítulo é apresentar uma ferramenta matemática, conhecida como *Lema do Bombeamento*, que será extremamente útil para demonstrarmos que certos DFAs não existem.

4.1 O Lema do Bombeamento para Linguagens Regulares

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

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

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

(1) $y \neq \varepsilon$ (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 ou simplesmente LB. Agora vamos mostrar como podemos fazer uso do LB para mostrar que uma linguagem não é regular. A vantagem de se usar o LB é que não precisamos mostrar explicitamente que um certo DFA não existe. O que acontece aqui é que todo trabalho da prova de inexistência do

DFA fica "encapsulada" dentro da demonstração do LB, que iremos omitir aqui. Segue um exemplo de como fazer uso do Lema do Bombeamento para provar que uma determinada linguagem não é regular.

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

Prova: Suponha que L_{01} é regular. Então o LB nos diz que existe $t \in \mathbb{N}$, tal que se tomaramos uma string w de L grande o suficiente (por grande o suficiente queremos dizer $|w| \geq t$), então $\exists x,y,z \in \Sigma^*$ tal que w pode ser escrita como $w = xyz$ e as seguintes condições são respeitadas: *(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 é uma string para a qual podemos aplicar o LB, pois $w \in L_{01}$ e $|w| \geq t$. Portanto w pode ser escrita na forma $w = xyz$ satisfazendo as três condições apresentadas acima.

Pela condição *(2)*, temos que $|xy| \leq t$ e portanto a string xy contém apenas 0's. Isso tem duas consequências. A primeira é que todos os t símbolos 1 da string w estão contidos em z (note não necessariamente z contém apenas símbolos 1, pois pode acontecer que z contenha alguns símbolos 0 precedendo os t símbolos 1 no caso em que $|xy| < t$, mas isso não é relevante aqui).

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

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:

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

Prova: Suponha que L_p é regular. Então o LB nos diz que existe $t \in \mathbb{N}$, tal que se 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)* $\forall k \geq 0, xy^k z \in L_p$.

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

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

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

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

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

O Teorema 4.1.3 mostra que o problema de reconhecer se um determinado número é primo não pode ser solucionado usando um algoritmo (ou uma máquina) cujo funcionamento possa ser descrito por um autômato finito. Entretanto, observe que assumimos que o alfabeto permitido contém apenas símbolos 1, de maneira que os números primos são representados pelas strings 1^p, onde p é um primo. Podemos nos perguntar se é possível provar um resultado semelhante supondo que o alfabeto de entrada é $\Sigma = \{0,1\}$ e os primos são as strings binárias que representem números primos. A resposta é sim, tal resultado também pode ser demonstrado, mas a demonstração é bem mais trabalhosa. Vamos enunciar tal teorema, mas a prova está fora do escopo deste curso. No enunciado do teorema, relembramos que $N(w)$ é o número natural que a string binária w representa (e.g., se $w = 1001$, então $N(w) = 9$).

Teorema 4.1.4 $L_{\text{P}} = \{w\,;\, N(w) \text{ é um número primo}\}$ não é regular.

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

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

4.2 Autômatos com Pilha de Dados (PDAs)

No capítulo 3 nós apresentamos a definição de DFAs e fomos, aos poucos, incluindo mais habilidades ao modelo. Primeiramente adicionamos a habilidade de “advinhar” qual transição fazer, que chamamos de não determinismo, e, depois, adicionamos as transições ε. Ainda assim, o conjunto de linguagens que podem ser reconhecidas por tais modelos não se alterou. Em outras palavras, o poder de computação dos ε-NFAs é o mesmo dos DFAs. Agora vamos encontrar um cenário diferente. Vamos incrementar o nosso ε-NFA com uma habilidade que, conforme veremos adiante, o tornará mais poderoso. A habilidade é a inclusão de uma pilha de dados ao autômato.

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

4.2.1 O modelo matemático para autômatos com pilha de dados

Definição 4.2.1 — Autômatos com pilha de dados (PDA). Um PDA 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-NFA.
Γ é o alfabeto da pilha.
$Z_0 \notin \Sigma$ é o “símbolo inicial”.
$\delta : Q \times \Sigma \cup \{\varepsilon\} \times \Gamma \to S$, tal que S é um conjunto de pares (q, γ), tal que $q \in Q$ e $\gamma \in \Gamma^*$.

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

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

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

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

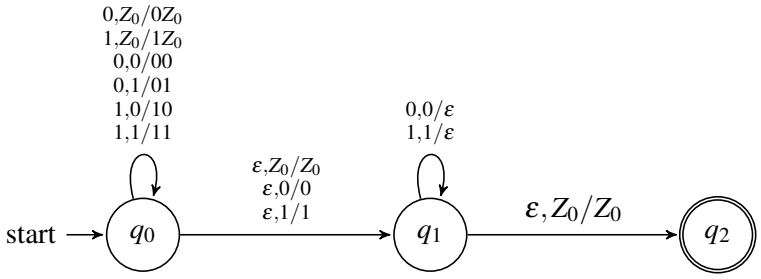


Figure 4.1: Diagrama do PDA P_{RR} que aceita a linguagem L_{RR}.

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

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

4.2.2 Computação com PDAs

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

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

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

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

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

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

Definição 4.2.4 Definimos agora $\vdash_P^*$ indutivamente:

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

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

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

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

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

4.2.3 Aceitação por pilha vazia

Relembramos que quando projetamos um PDA, normalmente tomamos cuidado para que o fundo da pilha sempre contenha o símbolo Z_0, pois o PDA morre quando tenta fazer uma trasição com a pilha vazia. Entretanto, uma vez que temos ao nosso dispor uma pilha de dados, em algumas aplicações é interessante projetar o PDA de maneira que ele morra exatamente quando terminar de ler a string. Alguns de nós com mais experiência em programação de computadores sabe que uma maneira bastante natural de se resolver certos problemas é usar uma pilha de dados e certificar-se que no final ela foi completamente esvaziada.

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

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

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

Observe que as strings de entrada de um dado PDA 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

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

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

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

4.2.4 Autômatos Finitos Determinísticos com Pilha de Dados

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

Definição 4.2.7 — Autômatos Finitos Determinísticos com Pilha de Dados (DPDAs). Um DPDA é um PDA cuja função de transição δ tem as seguintes restrições:

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

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

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

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

4.3 Gramáticas Livre de Contexto

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

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 o símbolo inicial, onde $S \in V$.

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

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

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

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

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

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

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

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

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

4.3.2 Derivações de uma gramática

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

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

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

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

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

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

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

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

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

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

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

4.3.3 Derivação mais a direita e mais a esquerda

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

Exercício 4.3 Verifique que a derivação de G_1 do Exemplo 4.5 é mais a esquerda ■

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

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

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

4.3.4 Árvores de análise sintática

Exercício 4.5 Pesquise a respeito de árvores de análise sintática (também conhecidas como árvores de derivação ou *parse trees*) de gramáticas livres de contexto. ■

4.3.5 Ambiguidade de Gramáticas

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

OBSERVAÇÕES IMPORTANTES:

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

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

4.3.6 Equivalência entre PDAs e gramáticas livre de contexto

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

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

4.3.7 DPDAs e ambiguidade de gramáticas

O Teorema 4.3.1 mostra que o conjunto de linguagens aceitas por PDAs é exatamente o mesmo conjunto das linguagens expressa por gramáticas. Por outro lado, vimos anteriormente que o conjunto das linguagens aceitas por PDAs não é o mesmo que o conjunto de linguagens aceitas por DPDAs. 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 DPDAs.

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

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

4.3.8 Exercícios

Exercício 4.6 Use o Lema do Bombeamento (LB) para provar que a seguinte linguagem sobre $\Sigma = \{0,1\}$ não é regular: $L = \{0^i 1^j ; i > j\}$. ■

Exercício 4.7 Forneça um PDA que aceite a linguagem $\{0^n 1^n ; n \geq 1\}$. ■

Exercício 4.8 Seja $\Sigma = \{(,),x\}$. Forneça um PDA que aceite apenas strings com parenteses balanceados. ■

Exercício 4.9 Forneça uma CFG para a linguagem $\{0^n 1^n ; n \geq 1\}$. ■

Exercício 4.10 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 4.11 Considere a gramática $G = (V,T,P,S)$, onde $V = \{S\}$, $T = \{0,1\}$ e as regras de P são:

$S \to \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$

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

Exercício 4.12 Forneça uma gramática que tem como linguagem expressões bem formadas em lógica proposicional. Lembramos que em uma expressão bem formada em lógica proposicional podemos ter variáveis $x_1, x_2, x_3, ...$, operadores binários $\wedge, \vee, \Rightarrow, \Leftrightarrow$, operador unário $\neg$ e abertura e fechamento de parênteses. ■

Exercício 4.13 Considere a gramática $G = (V, T, P, S)$, onde

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

Onde as regras são:

[STMT] → [IF-THEN] | [IF-THEN-ELSE] | [ASSIGN]

[IF-THEN] → if condition then [STMT]

[IF-THEN-ELSE] → if condition then [STMT] else [STMT]

[ASSIGN] → a=1

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

Exercício 4.14 Para responder as questões abaixo assuma que o alfabeto é $\Sigma = \{a, b, c\}$.

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

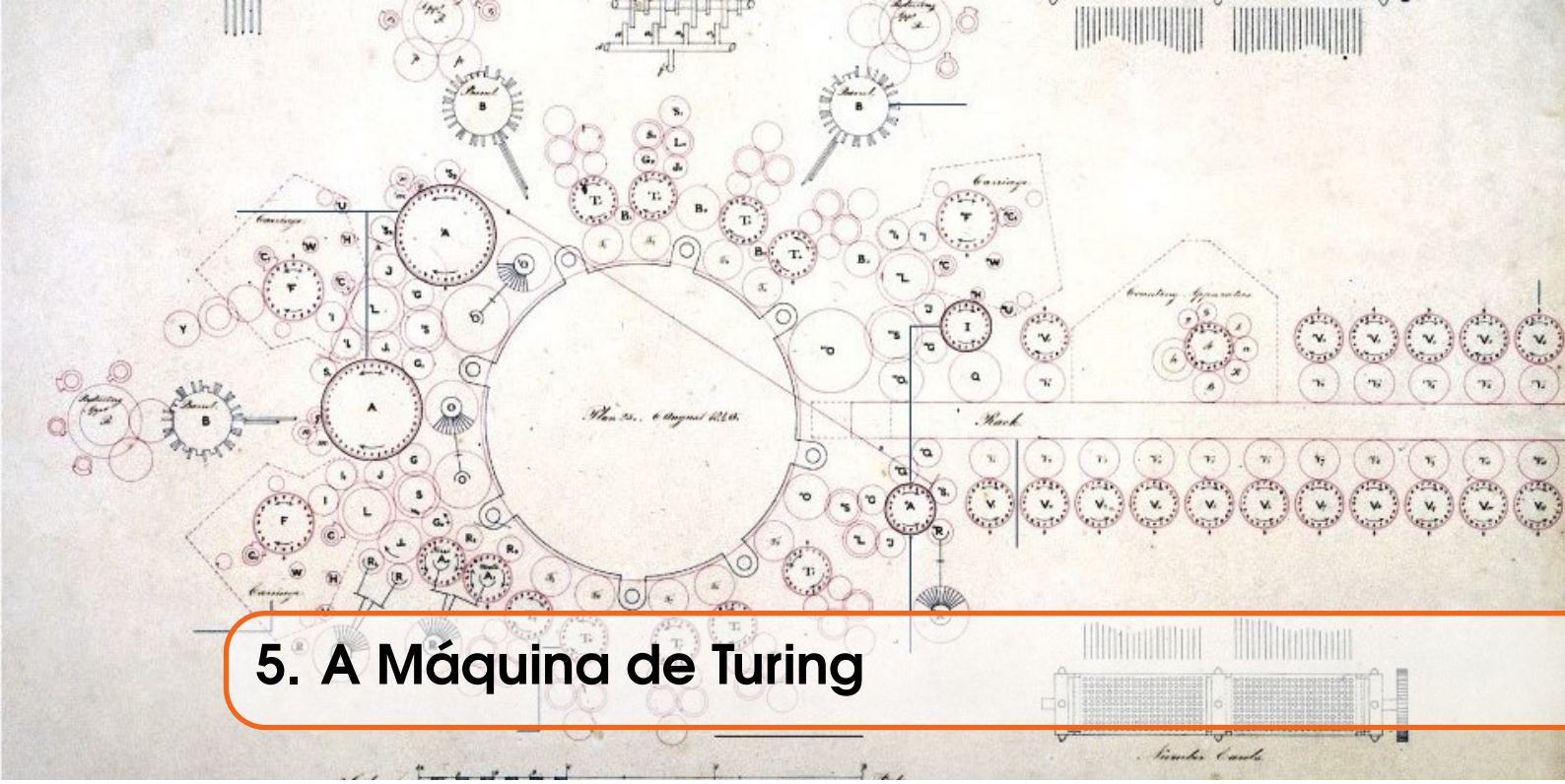
(b) Forneça uma Gramática G tal que $L(G) = \{a^n b^n c^m;\ m, n \geq 0\}$.

■

Exercício 4.15 (OPCIONAL) Prove que $L = \{a^n b^n c^n;\ n \geq 0\}$ não é livre de contexto. ■

Exercício 4.16 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 da questão 4.15 não é livre de contexto. ■

Parte 2: Máquinas de Turing e Computabilidade



5. A Máquina de Turing

Neste capítulo iremos apresentar a definição do modelo matemático proposto por Alan Turing em 1936, conhecido como Máquina de Turing. Usaremos este modelo para fornecer uma definição precisa para o conceito de algoritmo.

5.1 Revisão: problemas computacionais

Antes de falarmos de Máquinas de Turing, vamos brevemente relembrar algo que discutimos no Capítulo 2, que é a ideia de que problemas computacionais podem ser vistos como linguagens. Vimos, por exemplo, que o problema da divisibilidade por 3 e o problema da primalidade podem ser modelados pelas linguagem $L_3 = \{w \in \Sigma^* \; ; N(w) \text{ é um múltiplo de } 3\}$ e $L_P = \{w \in \Sigma^* \; ; N(w) \text{ é um número primo}\}$, respectivamente. A ideia central é que são precisamente as strings de L_3 que incorporam a propriedade "ser divisível por 3" e são precisamente as strings que não estão em L_3 que incorporam a propriedade "não ser divisível por 3", de maneira que a linguagem L_3 é um objeto matemático que captura a essência do problema de divisibilidade por 3. De maneira análoga, a linguagem L_P captura a essência do problema de teste de primalidade.

> **REFLETINDO UM POUCO: PROBLEMAS SÃO SEMPRE LINGUAGENS?**
>
> Uma simplificação que estamos fazendo aqui é que sempre estamos lidando com problemas para os quais a resposta é SIM ou NÃO, ou seja, problemas para os quais as respostas consistem de apenas um bit de informação (afinal, SIM ou NÃO podem ser vistos como 1 ou 0). Problemas com estas características são conhecidos como *problemas de decisão*.
>
> Nós sabemos perfeitamente que nem todo problema computacional é um problema de decisão. Por exemplo, considere o *problema do caminho mínimo em grafos*. Dada uma tripla (G,u,v), sendo que G é um grafo e u,v são vértices de G, a resposta que queremos produzir é o menor caminho conectando os vértices u e v no grafo G. Neste caso, a resposta do problema é um objeto matemático que não pode ser representado por apenas um bit. Mais precisamente, a resposta para este problema é uma sequência de vértices $v_1, v_2, ..., v_k$ tal que $v_i v_{i+1}$ são arestas

presentes em G. Observe que podemos representar uma sequência de vértices usando algum esquema de codificação em binário, ou seja, podemos pensar que a resposta é uma string binária respeitando determinadas propriedades.

Um outro exemplo de problema que não é de decisão é o problema de multiplicar dois números inteiros a e b. A resposta para este problema é o número inteiro $a \cdot b$. Note que neste caso, novamente, podemos pensar que a resposta é uma string, mais precisamente a string w tal que $N(w) = a \cdot b$.

Em outras palavras, enquanto problemas de decisão são tipos de problemas cuja resposta é composta por apenas um bit de informação, problemas como os discutidos acima, são mais gerais e a resposta é um string contendo vários bits de informação. Nos capítulos subsequentes lidaremos com tais tipos de problemas mais gerais, entretanto, é importante salientar que mesmo no cenário restrito a problemas de decisão já seremos capazes de explorar fundamentos e limites da computação.

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

Do ponto de vista formal, dizemos que encontramos uma solução para um problema de decisão L quando apresentamos um "objeto matemático" que possa ser usado para determinar sistematicamente se uma dada string pertence ou não pertence à linguagem L. Em outras palavras, dizemos que solucionamos o problema quando apresentamos uma definição matemática de algoritmo para resolver o problema. Por exemplo, sabemos que para o problema L_3 existe um DFA que é capaz de distinguir strings divisíveis por 3 de strings que não são divisíveis por 3. Por outro lado, vimos que DFAs não são capazes de resolver o problema L_P, ou seja, não são capazes de fazer teste de primalidade.

No Capítulo 4 nós estudamos PDAs, que são modelos mais poderosos do que DFAs. Nós não exploramos PDAs a fundo, mas é possível demonstrar existem vários problemas que também não podem ser resolvidos por PDAS, incluindo a tarefa de testar a primalidade de um número. Entretanto sabemos que é fácil escrever um algoritmo em nossa linguagem de programação favorita que teste se um dado número é primo. Em outras palavras, existem algoritmos que não podem ser escritos na forma de DFAs ou PDAs, e, portanto, DFAs e PDAs sãos modelos matemáticos que não são capazes de capturar todos os algoritmos possíveis e imagináveis.

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

CONTEXTO HISTÓRICO: DFAS, PDAS E MÁQUINAS DE TURING

Neste texto nós fomos introduzindo modelos de computação que são incrementalmente mais poderosos. Começamos com DFAs (que são equivalentes a NFAs e ε-NFAs), seguidos por PDAs e, neste capítulo, vamos introduzir Máquinas de Turing. Apresentar estes modelos nesta sequência é interessante do ponto de vista pedagógico, mas não reflete a sequência histórica em que estes modelos foram aparecendo na literatura científica. Curiosamente, Alan Turing propôs seu modelo na década de 30, enquanto que os modelos de computação vistos nos Capítulos anteriores apareceram na literatura por volta da década de 50 e 60.

5.2 Definição da Máquina de Turing

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

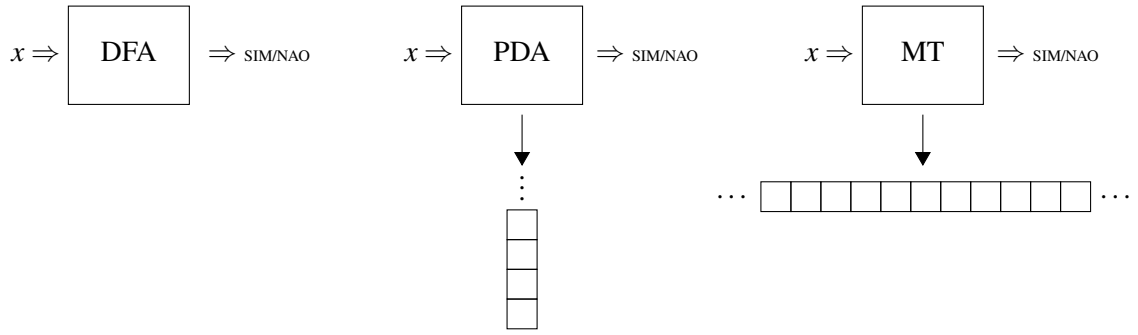


Figure 5.1: Comparação entre um DFA, um PDA e uma Máquina de Turing. Os DFAs são máqinas de estados e PDAs são máquinas de estados com acesso a uma memória em forma de pilha de dados. A Máquina de Turing é essencialmente uma máquina de estados com uma memória em forma de *fita* de dados. A máquina pode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

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

- Q é o conjunto finito de estados
- Σ é o alfabeto de entrada.
- Γ é o alfabeto da fita, tal que $\Sigma \subseteq \Gamma$.
- $\delta : (Q \setminus F) \times \Gamma \to Q \times \Gamma \times D$ 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.

Figure 5.2: Alan Turing

Algo que mencionamos no início deste curso, mas que gostaríamos de reforçar, é que de agora em diante, exceto quando explicitamente dito o contrário, estaremos assumindo que Σ é sempre o alfabeto binário. Por questões de simplicidade, vamos assumir que a string x de entrada no início da computação encontra-se localizada na fita de dados[1]. Isso simplifica um pouco a nossa definição da função δ (lembre que o domínio da função δ de PDAs era uma tripla (estado, símbolo, símbolo) e agora o domínio da função δ de MTs é apenas um par (estado, símbolo). A cada momento a máquina estará em um determinado estado e terá acesso a uma posição específica da fita e, em tal situação, o vocabulário que usaremos é o seguinte: diremos que a *cabeça de leitura* da máquina está *escaneando* uma determinada *célula* da fita de dados.

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

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 $x = x_1x_2...x_n$, 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 5.3 exemplifica isso.

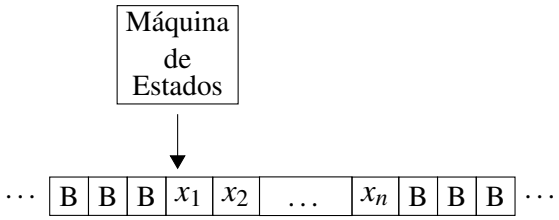


Figure 5.3: Máquina de Turing escaneando o primeiro símbolo da string $x = x_1x_2...x_n$ armazenada na fita.

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 argumentar 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 e, de fato, um dos atrativos de um objeto idealizado assim é a sua simplicidade.
>
> Entretanto, o que torna este objeto idealizado realmente útil não é apenas a sua simplicidade (afinal, DFAs são ainda mais simples que Máquinas de Turing), mas o fato de que ele, apesar da simplicidade, é extremamente poderoso.

Dissemos que MTs são "semelhantes a DFAs" (ao invés de dizer que são "exatamente DFAs") 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 DFAs. Em Máquinas de Turing, a função δ não precisa estar definida em todos os pares (q,X). Isso também ocorreu na definição de NFAs, mas nós iremos tratar este caso de maneira diferente aqui, pois a idéia é que MTs, ao contrário de NFAs, sejam modelos determinísticos de computação. Em 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*[2].

[2]Observe que uma máquina parar é um passo puramente determinístico, ao invés de ser um ramo que "morreu",

A causa do comportamento não determinístico de NFAs 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 3.11). No caso de MTs não há mais de uma transição definida em um certo par (q,X), o que torna o comportamento da máquina completamente determinístico. Dada uma MT $M = (Q,\Sigma,\Gamma,\delta,q_0,B,F)$, a cada aplicação da função $\delta(q,X)$ ocorre exatamente um dos dois casos abaixo:

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

5.2.2 Diagrama de estados de uma Máquina de Turing

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

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

Ainda não definimos o exatamente o que significa a linguagem de uma dada MT, mas observe que as únicas strings que fazem a MT da Figura 5.4 atingir seu estado final são as strings que pertencem a linguagem $L = \{0^n1^n; n \geq 1\}$. Além disso, note que uma vez que a máquina atinge o estado final, ela obrigatoriamente para sua execução (afinal, não há transições definidas no estado final). Algo que requer uma análise um pouco mais criteriosa, mas que também é verdade é que para toda string de $\Sigma^* \setminus L$, esta máquina irá em determinado momento atingir um estado $q \notin F$ enquanto escaneira o símbolo $A \in \Gamma$ tal que não há transição definida para o par (q,A). Em outras palavras, a máquina sempre para a computação em um estado que não é final se a string não pertence a L.

5.2.3 Linguagem de uma Máquina de Turing

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

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

Seja $M = (Q,\Sigma,\Gamma,\delta, q_0,B,F)$ uma Máquina de Turing, $X_1...X_n \in \Gamma^*$ e $q \in Q$. A descrição instantânea $X_1X_2...X_{i-1}qX_iX_{i+1}...X_n$ indica que M encontram-se no estado q, com a string $X_1...X_n$ em sua fita de dados e com a cabeça de leitura posicionada sobre o símbolo X_i. Além disso, a fita contém uma sequência infinita de símbolos B tanto à esquerda de X_1 e quanto à direita de X_n. Note que alguns símbolos X_i podem ser eventualmente iguais a B.

dentre muitas computações possíveis. O termo *morre* será usado mais adiante, quando formos definr Máquinas de Turing *não* determinísticas (ocasião em que o conceito de morrer terá uma interpretação semelhante ao conceito visto em NFAs).

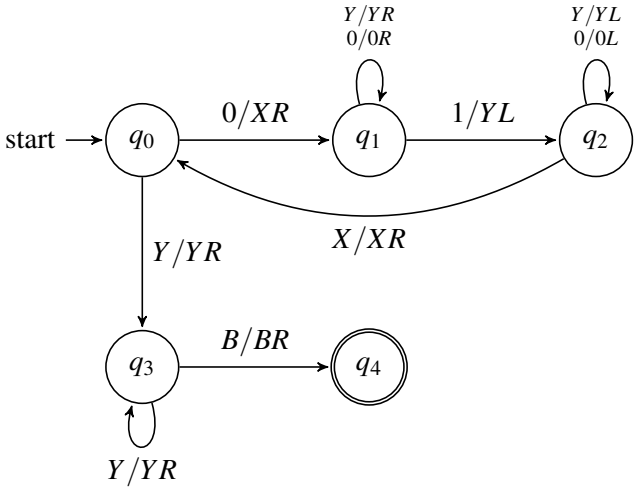


Figure 5.4: Diagrama de estados de uma Máquina de Turing.

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

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

$$X_1X_2...X_{i-1}qX_iX_{i+1}...X_n \vdash_M X_1X_2...X_{i-2}pX_{i-1}YX_{i+1}...X_n.$$

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

(1) Se $i = 1$, então escreveremos $qX_1...X_n \vdash_M pBYX_2...X_n$
(2) Se $i = n$ e $Y = B$, então escreveremos $X_1X_2...X_{n-1}qX_n \vdash_M X_1X_2...X_{n-2}pX_{n-1}$

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

Exercício 5.1 Apresente definições de quando é possível escrever $I_1 \vdash_M I_2$, no caso em que $\delta(q, X_i) = (p, Y, R)$ e no caso em que $\delta(q, X_i) = (p, Y, S)$. ■

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

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

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

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

A seguinte definição será útil várias situações.

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

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

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

Definição 5.2.7 — Linguagens aceitas por MTs. Dada uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, a linguagem $L(M) = \{w \in \Sigma^*; q_0 w \vdash_M^* I_F$, tal que I_F é um ID final de $M\}$ é chamada de *linguagem de M* ou linguagem *aceita* por M.

Definição 5.2.8 — Linguagens Recursivamente Enumeráveis. Se L é uma linguagem aceita por alguma Máquina de Turing, então a linguagem L é dita *Recursivamente Enumerável*. O conjunto de todas as linguagens recursivamente enumeráveis é denotado por $\mathscr{RE}$.

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

- Como na computação com Máqinas de Turing não existe o conceito de terminar de ler a string de entrada (como acontecia com DFAs e PDAs), existe a possibilidade de que a máquina continue computando indefinidamente em algumas circunstâncias, ou seja, a máquina pode ficar em *loop infinito*.

- Como definimos a função δ como nunca sendo definida em um estado final (note que F é excluído do domínio de δ na Definição 5.2.1), a máquina sempre para quando a string é aceita. Uma consequência disso é que se $L \in \mathscr{RE}$, então existe uma MT M tal que $\forall w \in L$, a máquina M *aceita w e em seguida finaliza sua execução*. Por outro lado, observe que se uma dada string $w \notin L$ que não é aceita por M, então isso pode significar duas coisas: M pode ter parado sua execução em um estado que não é final ou M pode ter ficado em loop infinito.

- A nossa busca por uma definição matemática formal para um algoritmo é essencialmente uma busca por uma definição genérica do 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. A possibilidade deste modelo matemático definir procedimentos que possam rodar indefinidamente pode ser útil em algumas circutnstâncias e, de fato, isso será útil adiante. Entretanto, dentre o conjunto de todas MTs, caso queiramos nos restringir a procedimentos que semprem fornecem alguma resposta depois de uma quantidade finita de passos (que essencialmente é um "algoritmo" para se resolver algum problema), teremos que nos restringir a MTs que semprem parem depois de um número finito de passos.

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, é importante observar que a definição de MTs permite a existência de máquinas que fiquem em loop infinito. Portanto, quando usarmos o apenas o termo "Máquina de Turing", sem especificar que ela sempre para, não *necessariamente* estaremos nos referindo a um algoritmo.

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

Definição 5.3.2 — Decidindo Problemas. Seja L um problema de decisão. Se existe um Algoritmo M tal que $L(M) = L$, dizemos que M *decide* L. Usando um vocabulário diferente, mas que significa a mesma coisa, podemos dizer: 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 5.3.3 Se existe MT que decide uma dada linguagem L, dizemos que L é uma *Linguagem Recursiva*. O conjunto de todas as linguagens recursivas é denotado por $\mathscr{R}$.

Note que quando L é **decidida**, a máquina M tal que $L = L(M)$ por definição tem a propriedade de sempre parar, o que não era uma restrição para linguagens **aceitas** por alguma MT.

Definição 5.3.4 Seja M uma MT e $x \notin L(M)$. Se a execução de M para quando a string x é fornecida como entrada, diremos que a M *rejeita* a string de entrada.

Observe que para qualquer string w de entrada, um algoritmo vai sempre aceitar ou rejeitar w, ou seja, por definição um algoritmo nunca fica em loop infinito. Por outro lado, quando dizemos que uma MT M aceita uma linguagem L, não podemos afirmar isso, pois podem existir strings $w \notin L$ tal que M fica em loop infinito com a entrada w. Neste caso não dizemos que M rejeita w, mas simplesmente que M não aceita w.

Exercício 5.2 Apresente uma MT M_3 que aceite, mas que não decida a seguinte linguagem:
$L_3 = \{w \in \Sigma^* \, ; N(w) \text{ é um múltiplo de } 3\}$. ■

Exercício 5.3 Com relação a MT M_3 e a linguagem L_3 do Exercício 5.2 responda:

(a) A máquina M_3 implica que $L_3 \in \mathscr{RE}$?

(b) Como M_3 não decide L_3 (apesar de aceitar esta linguagem), é possível concluir que $L_3 \notin \mathscr{R}$? Justifique sua resposta

(c) Caso a afirmação (b) esteja incorreta, ou seja, a existência de M_3 não necessariamente diz algo sobre L_3 estar ou não estar em $\mathscr{R}$, diga qual a relação de L_3 com $\mathscr{R}$ e justifique sua resposta (ou seja, prove que $L_3 \in \mathscr{R}$ ou prove que $L_3 \notin \mathscr{R}$, dependendo de qual caso for verdade).

■

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

Máquinas de Turing e Algoritmos

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

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

Por outro lado, a descrição exata de quais são todos os programas que podem ser escritos em Python ou C é bastante complicada (embora possível de ser dada). A vantagem de trabalharmos com Máquinas de Turings é precisamente o fato de que podemos provar teoremas genéricos sobre algoritmos (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 dependendo da linguagem usada em questão) usando uma simples 7-tupla. Todos os programa concebíveis são instâncias particulares de uma 7-tupla da Definição 5.2.1.

Uma pergunta que responderemos no Capítulo 7 é a seguinte. Será que existe uma linguagem que esteja em $\mathscr{RE}$, mas que não esteja em $\mathscr{R}$? Um outra pergunta próxima a esta é a seguinte: existe alguma linguagem qualquer (independente de estar ou não em $\mathscr{RE}$, pois ainda não sabemos se existem linguagens fora do conjunto $\mathscr{RE}$) que não esteja contida em $\mathscr{R}$?

5.3.1 Exercícios

Exercício 5.5 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 5.6 Responda se existe uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ tal que:

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

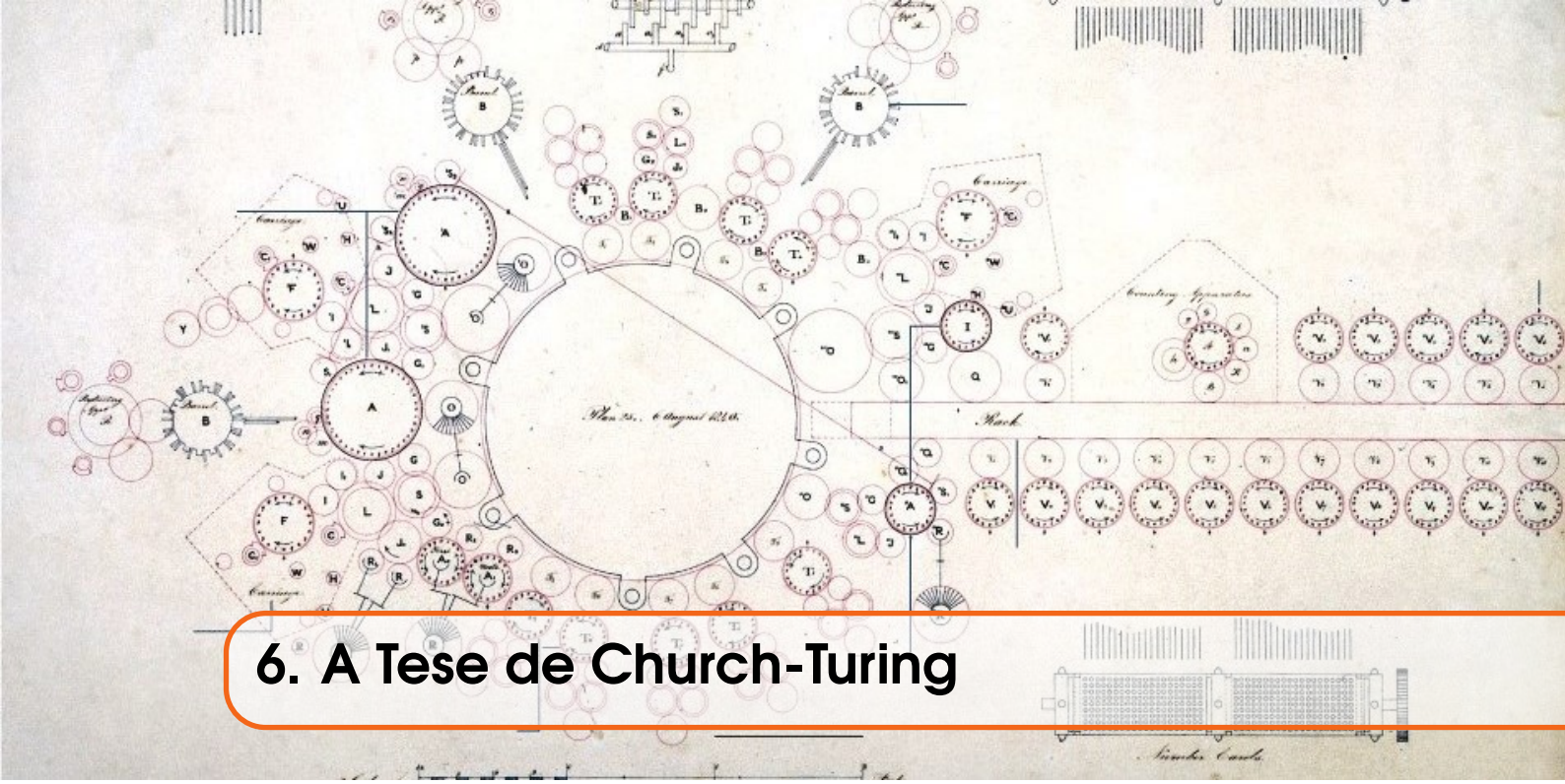
■

Exercício 5.7 Observe que a M_{COPY} a MT da Questão 1 não é o tipo máquina que normalmente temos projetado durante o curso, ou seja, não é uma máquina que testa se uma dada string de entrada tem ou não certa propriedade e no fim aceita ou rejeita a string de acordo com essa propriedade. Em outras palavras, o conceito de qual é a linguagem da máquina M_{COPY} não é algo central para o tipo de computação que ela realiza. Mas ainda assim a definição matemática $L(M_{\text{COPY}})$ é bem precisa. Pergunta: qual é linguagem $L(M_{\text{COPY}})$? ■

Exercício 5.8 Forneça uma definição formal para uma MT com 3 fitas. ■

Exercício 5.9 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 5.10 Seja M uma MT que sempre para com as características da Questão 5.9. Mostre que podemos projetar uma MT M' que sempre para com $L(M) = L(M')$ com as seguintes características: a MT tem 1 fita "read only" de entrada e 1 fita "read/write" de memória e uma terceira fita "read/write" de saída que a máquina escreve um bit de informação antes da máquina parar. O bit de informação é 0, se a string é rejeitada e 1 se a string é aceita. ■



6. A Tese de Church-Turing

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

6.1 Perspectiva histórica

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

Isso motivou o matemático David Hilbert a desafiar a comunidade matemática em 1928 a encontrar um algoritmo que tome como entrada uma afirmação matemática e que responda SIM, se a afirmação é verdadeira (ou seja, a afirmação é um teorema) ou NÃO se a afirmação é falsa[1]. O observe que, dada a maneira como Hilbert enunciou o problema, ele não considerava a possibilidade de que tal algoritmo não viesse a existir.

Resolver o problema proposto por Hilbert, conhecido como *Entscheidungsproblem* (problema da decisão, em alemão), era uma tarefa nada modesta, pois o objetivo era encontrar um algoritmo extremamente poderoso que automatizaria todo o processo de se "fazer matemática". Qualquer pessoa seguindo este algoritmo seria capaz de provar qualquer teorema[2]. Em 1936, primeiramente Alonzo Church, e, logo em seguida, Alan Turing provaram que tal algoritmo não existe.

[1] Neste caso estamos assumindo que os axiomas e as regras de inferência são algo fixo, conhecido a priori. Mas, poderíamos também pensar que o algoritmo, juntamente com a afirmação matemática, toma como entrada o conjunto finito de axiomas e regras de inferência que ele pode usar para provar ou refutar a afirmação matemática em questão.

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

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

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

6.2 Máquinas de Turing são equivalentes a linguagens de programação

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

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

Nesta seção nós vamos enunciar um teorema que afirma que Máquinas de Turing são equivalentes aos algoritmos escritos em linguagens de programação usadas hoje em dia. A demonstração do teorema não envolve nenhum conceito abstrato complicado, é muito longa e técnica e não é o nosso objetivo aqui. O nosso objetivo é obsevar que isso é um **Teorema Matemático**! A demonstração, caso algum estudante curioso tenha interesse em verificar, pode ser vista na seção 2.6 do livro [PAP94]. Para que possamos enunciar o teorema, vamos definir um modelo matemático equivalente a programas escritos em assembly, uma vez que um programa escrito em uma linguagem de programação de alto nível pode sempre ser expresso por um programa em assembly[3].

6.2.1 Programas Assembly

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

- $A_1 = \{\text{HALF}, \text{HALT}\}$
- $A_2 = \{\text{ADD}, \text{SUB}, \text{READ}, \text{STORE}, \text{LOAD}, \text{JUMP}, \text{JPOS}, \text{JZERO}, \text{JNEG}, \text{ZERO}\}$
- $B = \{j, \hat{}j\}$, onde j é uma string de $\{0,...,9\}$.

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

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

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

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

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

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

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

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

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

```
ADD     33
READ    10
READ    ^10
HALF
STORE   12
HALT
```

■

SEMÂNTICA DE INSTRUÇÕES E EXECUÇÃO DE PROGRAMAS ASSEMBLY

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

Não iremos apresentar a semântica a fundo de cada uma das instruções da nossa definição e o que significa *executar um programa assembly*, uma vez que estes conceitos são intuitivos para quem tem experiência com programação. O aluno curioso é motivado a ver este conceitos em detalhe na seção 2.6 do livro [PAP94].

Nós sabemos que Programas assembly são capazes de resolver bem mais do que apenas problemas de decisão (Máquinas de Turing também tem esta propriedade). Entretanto, uma vez que estamos lidando por hora com problemas de decisão, será essencial definir o conceito de aceitação de linguagens por programas assembly.

Apesar de não estarmos nos preocupando com a semântica exata de PAs, não teremos como

escapar de alguns conceitos elementares. Primeiramente, precisamos pensar como um PA faz a leitura da string de entrada. Dado um PA, vamos assumir que o i-ésimo bit da string de entrada $x = x_1x_2...x_n$ que este programa toma é acessível usando-se a instrução READ i. Depois disso, a ideia geral é que a execução do programa consiste de uma série de instruções sendo *chamadas* (não necessariamente em ordem, uma vez instruções como JUMP servem para mudar o fluxo de execução do programa).

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

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

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

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

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

6.3 Máquinas de Turing e outros modelos de computação

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

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

Ainda é um pouco cedo para afirmar quais destes modelos advindos da tentativa de se usar substratos físicos não tradicionais refletem tecnologias que podem sair do papel. De maneira geral, duas áreas tem sido mais ou menos proeminentes nos dias de hoje. Em menor escala, uma destas áreas é de computação molecular, mais precisamente, computação usando moléculas de DNA[4].

A outra área, que certamente é umas das mais ativas atualmente, é a área de computação quântica. A ideia que sustenta a pesquisa em computação quântica é a seguinte: de acordo com

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

as leis da mecânica quântica, a evolução no tempo de um conjunto de objetos de um sitema físico (estes objetos podem ser átomos, elétrons, fótons, etc) podem ser modelados por uma abstração chamada de *circuito quântico*. A ideia básica é a mesma que estamos discutindo desde o início do curso: usar o estado de um objeto (ou conjunto de objetos) para registrar informação e fazer o processamento desta informação por meio da manipulação dos estados em que estes objetos possam se encontrar. Entratanto, como aqui os objetos usados são suficientemente pequenos, os tipos de manipulações possíveis (i.e., os tipos de de transformações permitidas que levam um estado a outro) são regidas pelas leis da mecânica quântica, que são as leis que ditam quais transformações são possíveis de ocorrer em última análise em qualquer sistema imaginável.

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

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

O enunciado do Teorema 6.3.1 está um pouco vago, pois não definimos com precisão vários destes modelos matemáticos (e.g., Cálculo λ, funções μ-recursivas, Linguagem C, Linguagem Java, etc) e usamos vocabulário impreciso, como "outros modelos matemáticos" e "MT com outros alfabetos"[5]. O nosso objetivo central neste ponto não é apresentar os detalhes destas equivalências e sim reforçar que estas equivalências são **Teoremas Matemáticos**. Na próxima seção vamos discutir brevemente algo diferente, que é a tese de que MTs não apenas são equivalentes a outros modelos matemáticos, mas modelam efetivamente qualquer tipo possível de transformação de informação, ou seja, computação, que possa ocorrer no mundo físico.

6.4 A Tese de Church-Turing e suas interpretações

A *Tese de Church-Turing (TCT)* é afirmação de que Máquinas de Turing "capturam o conceito de computação efetiva". Há duas interpretações que normalmente são feitas desta tese. A primeira interpretação é que a TCT é uma definição matemática. A segunda intepretação é da TCT como uma afirmação sobre o mundo físico.

6.4.1 A TCT como definição matemática

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

Uma afirmação como esta acima pode ser debatida e algumas pessoas podem até discordar a respeito da afirmação, mas no fim ela é aceita por que ela é útil tem funcionado muito bem na prática

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

desde a criação do Cálculo. O que acontece é que a ideia original de função contínua (funções que podemos "desenhar sem tirar a caneta do papel") era subjetiva, e essa subjetividade impede que possamos fazer avanços matemáticos a respeito de funções contínuas. Uma vez que não parece existir alguma função contínua (como intuitivamente concebemos) que não possa ser expressa em termos desta definição usando epsilons e deltas, então os matemáticos **definiram** funções contínuas desta maneira. Portanto, funções que não podem ser expressas desta maneira **por definição não são contínuas** e funções uque podem ser expressas desta maneira **por definição são contínuas** .

A afirmação de que uma Máquina de Turing que sempre para é exatamente a definição de um algoritmo pode ser vista como algo útil e faz sentido na prática. Algumas pessoas poderiam debater se existe ou não existe um formalismo matemático capaz de representar algo que reconheçamos subjetivamente como um algoritmo, mas que não pode ser expresso na forma de Máquinas de Turing. Embora discussões como esta pareçam um pouco "anos 30", ainda assim faz sentido para alguns debater esta questão pois trata-se de uma questão de definição matemática. Ainda assim, de maneira semelhante a analogia com as funções contínuas, é consenso matemático que quando dizemos que se existe uma MT que sempre para para decidir um dado problema, então **por definição existe um algoritmo para o problema** e se não existe uma MT que sempre para para decidir um dado problema, então **por definição não existe um algoritmo para o problema**.

6.4.2 A TCT como afirmação sobre o mundo físico

Uma outra interpretação, razoavelmente mais comum, que se faz da TCT é que ela é uma *afirmação sobre o mundo físico*. Esta interpretação é interessante por que ela tende a eliminar debates infrutíferos, uma vez que há um critério objetivo para se refutar tal afirmação, caso ela venha a ser falsa. O critério que nos referimos é o mesmo usado para qualquer afirmação sobre o mundo físico: a afirmação deve ser descartada caso seja refutada experimentalmente. O que a Tese de Church-Turing afirma 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 7, veremos que existe um problema de decisão, chamado *problema da parada*, que não pode ser resolvido por Máquinas de Turing. Uma vez que uma consequência da TCT é que nenhum objeto no mundo físico seja capaz de resolver este problema, saberíamos precisamente o que tipo de evidência empírica precisaríamos para refutar a TCT: um aparato que resolva consistentemente o problema da parada. O consenso atual, dado o que sabemos sobre as leis da física (e mesmo sobre os fragmentos do que sabemos a respeito da direção que a física parece estar tomando) é que a existência de tal objeto parece ser bastante improvável.

No enunciado da TCT, quando nos referimos a um problema computacional, não nos referimos apenas a problemas de decisão. Neste contexto estamos nos referimos a algo extremamente amplo (essencialmente qualquer processo sistemático de tranformação informação[6]). Por conta disto, a TCT tem uma implicação bastante significativa. Uma vez que podemos ver o estado de um objeto físico qualquer como a instanciação de alguma informação (i.e., a descrição do estado que o objeto se encontra é a informação em si), sabemos que a evolução no tempo de tal objeto, não importando quão complicado seja este objeto, pode ser simulado por uma Máquina de Turing.

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

Conexões entre computação e física

A Tese de Church-Turing é normalmente aceita por que, ao observarmos a natureza em seu nível mais fundamental e levarmos em consideração como objetos se comportam, quais estados que estes objetos podem estar, quais são seus graus de movimentos possíveis, e quais são os tipos de evolução que estes objetos podem sofrer no tempo, as restrições impostas pelas leis da mecânica quântica parecem sustentar a tese. Um ponto chave é que a descrição de um objeto pode ser aproximado com precisão arbitrária pelo modelo matemático conhecido com circuito quântico[1], e sabemos que estes modelos podem ser simulados por Máquinas de Turing (enunciamos este fato no Teorema 6.3.1). A citação abaixo expressa bem a vantagem que alguns cientistas vêem ao intepretar TCT como afirmação sobre a realidade física e por que a comunidade científica tende sustentar esta versão da tese.

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

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

Contínuo vs Discreto?

A Tese de Church-Turing, como qualquer questão científica, é passível de debate. Existe uma área da computação, conhecida como *hiperpcomputação*, que é dedicada a estudar modelos que desafiem a TCT. Entretanto, a maioria das propostas que questionam a TCT são tipicamente variações da antiga ideia de computação analógica[1], uma ideia que parece esbarrar em alguns obstáculos postos pela física teórica contemporânea. Em particular, o resultado mais importante nesta linha, demonstrado na década de 70, é chamado de *Limitante de Bekenstein*. Embora alguns parâmetros usados na mecânica quântica sejam contínuos, o Limitante de Bekenstein impõe um limite a quantidade de informação (que pode ser pensada em termos da quantidade de estados que podem ser observados em um sistema) que uma região finita do espaço pode conter.

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

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

Turing e Inteligência artificial

Umas das consequências mais discutidas da Tese de Church-Turing é a afirmação de que

cérebros humanos, sendo estes objetos físicos, podem ser simulados por Máquinas de Turing.

Observe que a afirmação não é que já sabemos como fazer tal simulação, pois não sabemos exatamente como cérebros funcionam. A afirmação também não é a de que a simulação de um cérebro por uma Máquina de Turing é a melhor estratégia para se implementar inteligência artificial. Outro argumento normalmente confundido com esta afirmação é a de que a arquitetura específica de um cérebro é semelhante a arquitetura de computadores atuais (embora cérebros tenham evoluído para processar informação, processamento de informação pode ser feito usando-se muitas arquiteturas diferentes). Estas questões são interessantes e dignas de pesquisa, mas não são relevantes aqui, pois a afirmação em questão é bem simples e fundamental: um cérebro, como qualquer objeto físico, pode ser simulado em princípio por uma Máquina de Turing.

Algo importante de se ressaltar é que a discussão da possibilidade da inteligência humana ser simulada por máquinas não é algo recente. Não é o caso de que esta ideia foi ganhando força lentamente e somente está em voga agora por que inteligência artificial tornou-se uma das mais ativas áreas da ciência atualmente. Esta observação apareceu juntamente com nascimento computação e o próprio Alan Turing trabalhou nesta questão em seu famoso artigo em que o *Teste de Turing* é proposto.

6.5 A Tese de Church-Turing estendida

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

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

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

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

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

A construção de computadores quânticos é possível em princípio, mas alguns pesquisadores questionam esta possibilidade. Este questionamento significa dizer que o modelo de circuitos quânticos não são modelos fisicamente realizáveis[7]. O que estes pesquisadores fazem é afirmar que não somente a TCT é sólida, mas que ela é mais sólida do que o consenso atual. Esta afirmação, conhecida como Tese de Church-Turing Estendida (TCTE), e que data da década de 60, afirma o seguinte: *todo problema computacional que possa ser resolvido de maneira eficiente no mundo*

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

físico, pode ser resolvido de maneira eficiente por uma Máquina de Turing. Neste enunciado, a palavra eficiente quer dizer polinomial. Ao contrário do que acontece com a TCT, a TCTE não é consenso científico, pois tal afirmação sugeriria que o modelo de circuitos quânticos não é realista. Entrentanto, o consenso científico é que o modelo de circuitos quânticos é simplesmente consequência das leis da mecânica quântica.

ALGORITMOS QUÂNTICOS

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

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

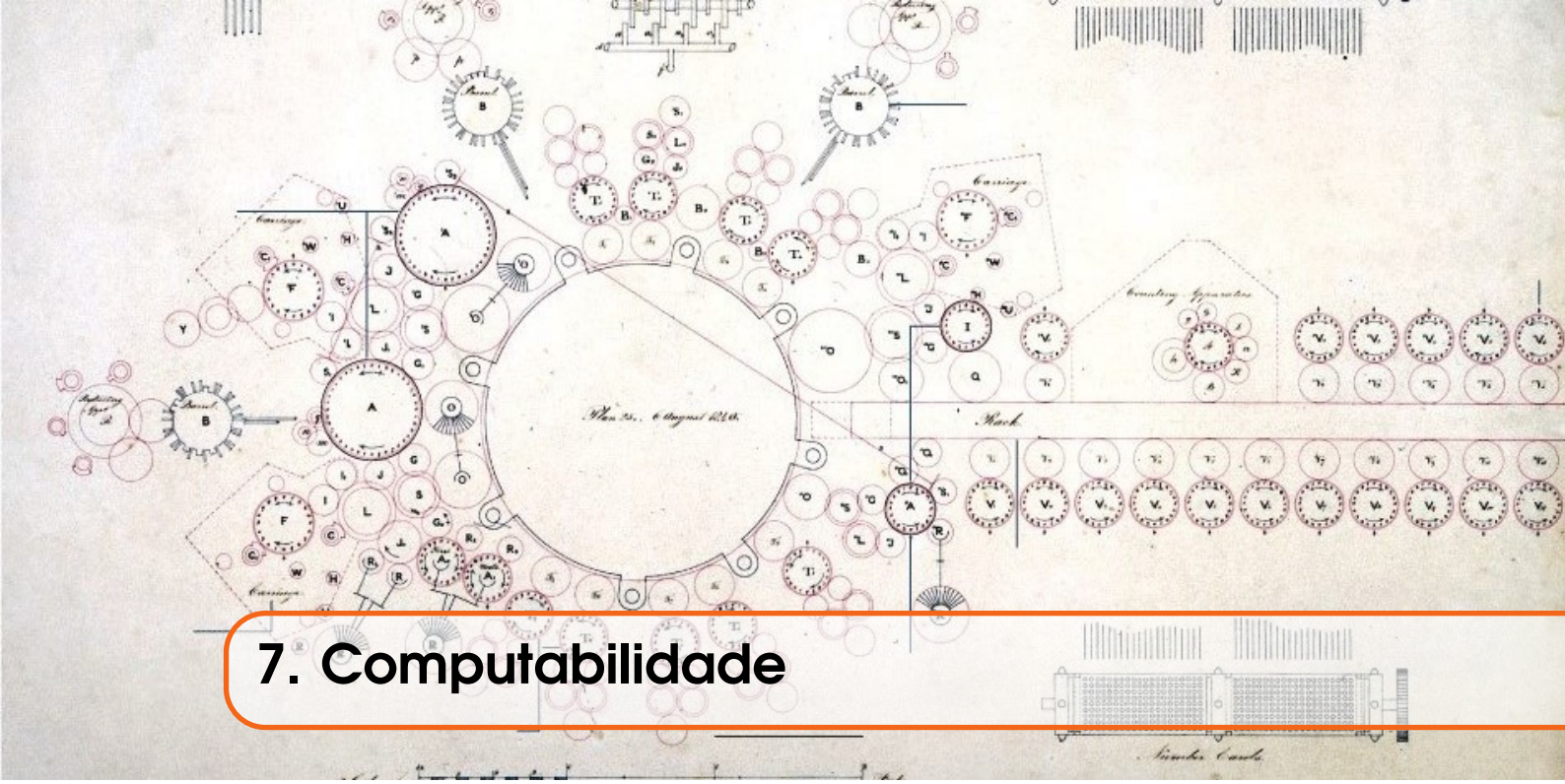
O que ocorre atualmente é que embora computadores quânticos sejam capazes de resolver precisamente os mesmos problemas que computadores clássicos, tem-se bastante interesse na construção de computadores quânticos, pois alguns problemas computacionais que eles poderiam resolver de maneira exponencialmente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

6.5.1 Exercícios

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

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

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



7. Computabilidade

Neste capítulo vamos estudar dois resultados que Alan Turing apresentou, juntamente com a definição de sua máquina, em seu famoso artigo de 1936. O primeiro é a prova de que existem problemas para os quais não existem algoritmos que os resolvam. O segundo é a existência da Máquina de Turing Universal, uma instância de Máquina de Turing capaz de simular todas as possíveis Máquinas de Turing. A Máquina de Turing Universal pode ser vista como um modelo matemático que descreve o que entendemos por um computador.

7.1 Funções computáveis

Uma outra maneira de ver a ideia de Máquinas de Turing resolvendo problemas de decisão, é vê-las calculando funções booleanas do tipo $f : \{0,1\}^* \to \{0,1\}$. Entretanto, temos que ter um pouco de cuidado diferenciando MTs que sempre param de MTs que possam eventualmente entrar em loop infinito.

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

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

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

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

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

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

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

Observe que a expressão $M(x) = y$ não faz sentido se M rejeita y ou se M não para. Também devemos tomar cuidado não confundir a notação acima com a Notação 7.1 no caso da string y conter apenas um bit. O contexto deve estar sempre claro no uso desta notação.

Definição 7.1.2 — 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.

7.2 Codificando objetos matemáticos em binário

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

7.2.1 Notação para Máquinas de Turing tomando vários argumentos de entrada

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

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

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

Na realidade esta é uma dificuldade que é encontrada na prática em computadores de hoje em dia e que é superada de diversas maneiras, como, por exemplo, fazendo o uso de sistemas de codificação (e.g., tabela ASCII, que nos permite definir símbolos de espaçamento, quebra de linha, etc). No caso de MTs, nós também podemos usar estratégias parecidas.

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

MÚLTIPLAS ENTRADAS: SOLUÇÃO SIMPLES

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

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

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

7.2.2 Representando objetos matemáticos

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

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

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

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

Podemos pensar em várias maneiras de se representar um grafo usando uma string e, em geral, nós não vamos nos preocupar com a maneira exata de se fazer isso. Entretanto, quando estivermos usando uma representação binária para algum objeto matemático, precisamos ter certeza que de que é possível realizar tal tarefa (por exemplo, se r é um número real, o objeto $\llcorner r \lrcorner$ não faz sentido algum). No caso de um dado grafo G, a maneira mais comum de representá-lo é concatenar os bits da matriz de adjacência de G linha por linha[1], como no exemplo a seguir.

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

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).

O ponto mais importante que queremos levantar nesta seção é que Máquinas de Turing também poder ser representadas por strings binárias, afinal, MTs também são objetos matemáticos bem definidos e finitos. Usando novamente nossa analogia com computadores reais, um algoritmo

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

implementado em alguma linguagem de programação (que sabemos que são equivalentes a MTs) armazenado na memória de um computador nada mais é do que uma sequência de bits na memória deste computador. Ou seja, um algoritmos pode ser vistos como strings. No caso de Máquinas de Turing, se uma MT M toma como entrada uma outra MT M', a ideia é que a MT M' nada mais é do que uma sequência de 0's e 1's na fita da MT M.

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

Exercício 7.2 Seja M a Máquina de Turing da Figura 5.2.2. Apresente a string $\llcorner M \lrcorner$. ∎

7.2.3 Problemas de Decisão

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

Definição 7.2.1 — Primalidade. O problema de decidir se um número natural é primo pensado formalmente como a seguinte linguagem: $L_{\text{P}} = \{\llcorner n \lrcorner \in \Sigma^* \; ; \; n \text{ é um número primo } \}$.

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

Definição 7.2.3 — Conectividade de Grafos. O problema de decidir se um dado grafo é conexo é definido pela linguagem $L_{\text{C}} = \{\llcorner G \lrcorner \in \Sigma^* \; ; \; G \text{ é um grafo conexo}\}$.

Definição 7.2.4 — Grafos Eulerinos. O problema de decidir se um grafo é euleriano é definido pela linguagem $L_{\text{E}} = \{\llcorner G \lrcorner \in \Sigma^* \; ; \; G \text{ é um grafo euleriano}\}$.

Definição 7.2.5 — Grafos Hamiltonianos. O problema de decidir se um dado grafo é hamiltoniano é definido pela linguagem $L_{\text{H}} = \{\llcorner G \lrcorner \in \Sigma^* \; ; \; G \text{ é um grafo hamiltoniano}\}$.

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

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

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

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

Exercício 7.4 A string $\llcorner K_4 \lrcorner$ é uma instância verdadeira ou falsa do problema L_{H}? ■

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

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

Primo: (N)
1: **if** $N = 1$ **then**
2: Return False
3: **for** $i = 2; i \leq \sqrt{N}; i$++ **do**
4: **if** N mod $i = 0$ **then**
5: Return False
6: Return True

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

PSEUDO-CÓDIGOS OU MÁQUINAS DE TURING?

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

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

7.4 O problema da Parada

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

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

O objetivo desta 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 7.4.1 — O problema da parada. O *problema da parada* é definido pela linguagem $L_{\text{H}} = \{\llcorner M \lrcorner\, x \;;\; \text{tal que } M \text{ é uma MT}, x \in \Sigma^* \text{ e } M(x) \neq \nearrow\}$.

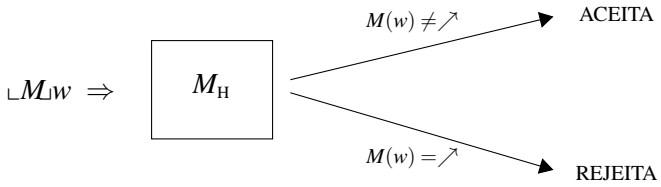
Resolver o problema da parada significaria fornecer uma MT M_{H} que decida L_{H}. Ou seja, uma MT M_{H} que tome $(\llcorner M \lrcorner, x)$ como entrada e que tenha o seguinte comportamento:

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

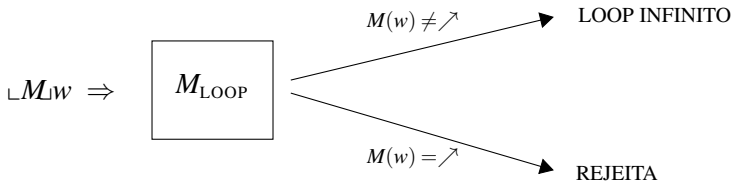
Teorema 7.4.1 — Teorema da Parada. Não existe algoritmo que decida a linguagem L_{H}.

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 $M(w) = \nearrow$, então a máquina M_{H} deve rejeitar a string de entrada. Por outro lado, se $M(w) \neq \nearrow$ (note que não importa se $M(w) = 0$ ou se $M(w) = 1$), então M_{H} deve aceitar a string de entrada. O diagrama abaixo ilustra o funcionamento de M_{H}:

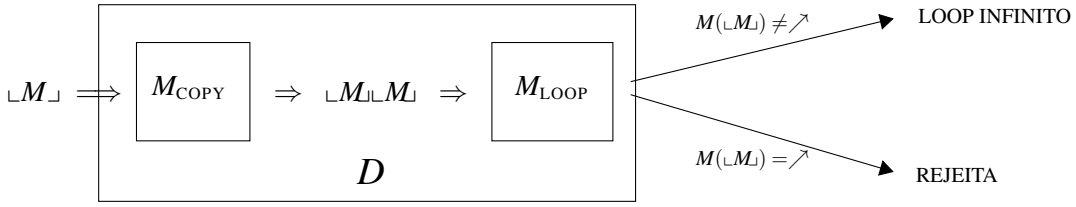


Dado que a máqina M_{H} existe, vamos agora concluir que a máquina M_{LOOP}, que vamos descrever a seguir, também existe. A máquina M_{LOOP} é essencialmente M_{H} com uma pequena modificação. Dada uma string que M_{H} aceite, ao invés da máquina aceitar e parar, a máquina deve entrar em loop infinito.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



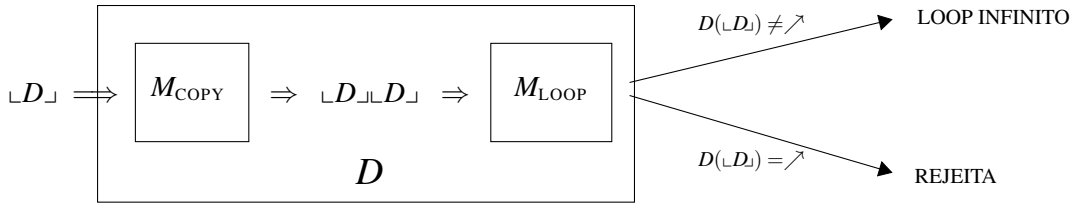
Antes de seguir em frente, devemos antentar ao tipo de argumentação que estamos usando. Estamos trabalhando com a existência de M_{H}, pois esta foi a nossa suposição inicial, mas será que, de fato, M_{LOOP} existe? Nosso argumento foi que podemos obtê-la fazendo uma modificação em M_{H}. Para termos 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 7.8 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 5.5), 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 descrito pela figura a seguir).



A construção formal de D é intuitiva, mas os estudantes que gostam de demonstrar teoremas de maneira extrememete rigorosa são encorajados a resolver o Exercício 7.9, 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: $D(\llcorner M \lrcorner) = \nearrow \Leftrightarrow M(\llcorner M \lrcorner) \neq \nearrow$. 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 $D(\llcorner D \lrcorner) \neq \nearrow \Leftrightarrow D(\llcorner D \lrcorner) = \nearrow$, o que é uma contradição lógica. Com isso concluímos que M_H não existe. □

7.5 A Máquina de Turing Universal

Considere uma MT $\mathscr{U}$ que toma como entrada uma outra máquina M e uma string x e simule o comportamento de $M(x)$, ou seja, $\mathscr{U}$ "executa" a MT M quando esta tem como argumento de 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 $M(x)$ (caso $M(x) = \nearrow$, a máquina $\mathscr{U}$ deve ficar em "loop infinito"). Adicionalmente, quando vemos nossas máquinas de Turing computando funções não booleanas, se $M(x) = y$, então $\mathscr{U}(\llcorner M \lrcorner, x) = y$

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

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

- Até agora estávamos vendo Máquinas de Turing como "software" e definimos algoritmos como sendo Máquinas de Turing que sempre param (MTs que ficam em loop infinito, por definição, não são algoritmos, mas elas podem ser pensadas como sendo programas de computador que ficam em loop infinito).

- No caso da Máquina de Turing Universal é bastante razoável pensarmos nela como um modelo matemático para um computador. Uma MT universal $\mathscr{U}$ tem a capacidade de rodar qualquer outra MT M com qualquer possível entrada de dados x, e, ao final, retornar a saída de $M(x)$. A máquina $\mathscr{U}$ pode ficar executando M indefinidamente se $M(x) = \nearrow$. Isso é essencialmente o que um computador faz. Claramente, essa visão de "software" e "computador" pode ser maleável, pois muitas vezes temos implementações de algoritmos feitas em hardware de propósito específico e, por outro lado, também temos softwares que funcionam como uma Máquina de Turing Universal. Alguns exemplos de softwares que podem ser vistos com MTs universais são, por exemplo, interpretadores de linguagens de programação ou softwares emuladores[2].

- Finalmente, note que existência de uma MT universal do ponto vista físico (i.e., a possibilidade de se implementar fisicamente uma MT Universal) é algo bastante poderoso. Quando o conceito foi concebido por Alan Turing em 1936 não existiam computadores e muito menos software. Entretanto, a indústria de software de hoje seria uma impossibilidade matemática se o objeto matemático $\mathscr{U}$ não existisse sob a luz da Tese de Church-Turing, pois em tal situação não seria possível construir computadores capazes de rodar cada algoritmo possível e imaginável. Em tal cenário, para cada problema específico precisaríamos implementar o respectivo algoritmo que o resolve diretamente em hardware.

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

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

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 conteúdo estaria presente na fita de M durante a computação de $M(x)$. A terceira fita de $\mathscr{U}_3$ será uma fita de memória auxiliar. Durante o processo de simulação da máquina M, vamos armazenar nesta terceira fita dois números em binário. O primeiro número corresponde ao estado que a máquina M sendo simulada se encontra. O segundo número é um índice que corresponde a posição da fita que a cabeça de leitura de M está posicionada.

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 parar. No caso em que M não tenha uma transição definida e esteja em um estado que não seja final (ou seja, M irá rejeitar a entrada), a MT $\mathscr{U}_3$ irá para um estado especial que é um estado que não tem nenhuma transição definda e

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

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 7.5.2 L_{H} é recursivamente enumerável.

Exercício 7.6 Foneça uma prova para o Teorema 7.5.2. ■

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

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

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 7.6, $L_{\text{H}} \in \mathscr{RE}$. Pelo Teorema 7.4.1, $L_{\text{H}} \notin \mathscr{R}$. Portanto $\mathscr{R} \subsetneq \mathscr{RE}$.

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

Exercício 7.7 Foneça uma prova para o Teorema 7.5.4. ■

7.6 Máquinas de Turing não determinísticas (MTN)

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

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

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

Do ponto de vista de definição matemática, a única diferença que MTNs tem em relação a MTs é que, ao invés de uma função de transição δ, MTNs tem um par de funções (δ_0,δ_1). A questão agora é *interpretar* a definição de MTNs para que possamos definir como é o processo de computação não determinística neste caso. A ideia é que, a cada passo, a máquina 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 veremos, o processo de computação de uma Máquina de Turing como uma sequência de descrições instantâneas.

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

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

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

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

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

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

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

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

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

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

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

Uma observação importante é que uma árvore de computações possíveis de MTNs pode ter alguns ramos infinitamente longos[3] nos casos em que ramos da computação fiquem em loop infinito. Entretanto, se uma MTN N aceita uma string x, existe pelo menos um ramo finito nesta árvore, e a folha no final deste ramo é um ID final.

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

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

Idéia da prova: Uma MT pode simular uma NTM.

[3] 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 7.6.2 Se M é uma MT, então existe uma MTN N tal que $L(M) = L(N)$.

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

7.7 Exercícios

Exercício 7.8 Seja uma MT $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ que decide a linguagem L. Mostre como obter uma MT M' com o seguinte comportamento:
(1) Se $M(x) = 1$, então $M'(x) = \nearrow$
(2) Se $M(x) = 0$, então $M'(x) = 0$. ■

Exercício 7.9 Seja M_{COPY} a máquina fornecida no Exercício 5.5. Seja M_{LOOP} a máquina cujo funcionamento é explicado na prova do Teorema 7.4.1. Prove que se M_{COPY} M_{LOOP}, então existe uma máquina D tal que $D(\llcorner D \lrcorner) \neq \nearrow \Leftrightarrow D(\llcorner D \lrcorner) = \nearrow$, o que é uma contradição lógica. ■

Exercício 7.10 Prove de maneira formal que L_{H} (a linguagem da parada) é recursivamente enumerável. ■

Exercício 7.11 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. ■

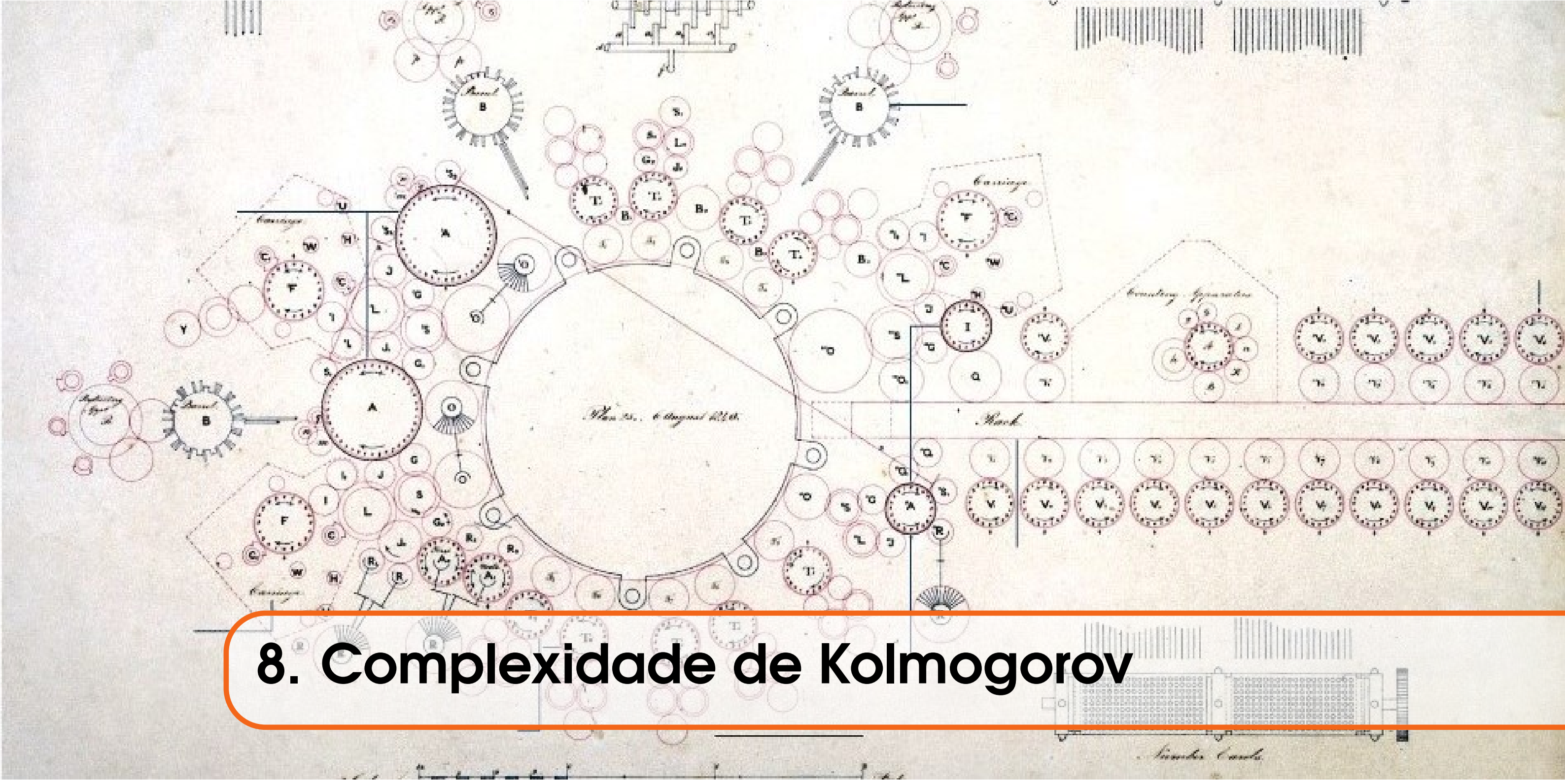
Exercício 7.12 Agora vamos mostrar explicitamente que uma linguagem específica não é recursivamente enumerável. Seja L_H a linguagem da parada. Prove que $\overline{L_{\text{H}}} \notin \mathscr{RE}$. ■

Exercício 7.13 Prove o seguinte problema é indecidível. Dada uma Máquina de Turing M, queremos decidir se $M(\varepsilon) \neq \nearrow$. ■

Exercício 7.14 Prove que $L = \{\llcorner M \lrcorner \, ; \exists x \in \Sigma^* \text{ tal que } M(x) \neq \nearrow\}$ é indecidível. ■

Exercício 7.15 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 $M(\varepsilon) = \nearrow$, então $M'(\llcorner M \lrcorner) \neq \nearrow$ (ou seja, M' para, independente de aceitar ou rejeitar).

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



8. Complexidade de Kolmogorov

Neste capítulo vamos fazer uma pequena pausa em nosso esforço para entender o que são algoritmos, computadores e problemas e vamos usar o entendimento que ganhamos até aqui para tentar jogar luz sobre um conceito intimamente relacionado a computação, que é o conceito de informação. Vamos ilustrar isso com uma pequena história.

Suponha que temos acesso momentâneo a uma string de 800 bits e que nosso objetivo é descrever essa string para alguma outra pessoa mais tarde. Para tal, nós podemos memorizar a string ou anotá-la em um pedaço de papel, dependendo de como anda a nossa memória. A pergunta que queremos responder é a seguinte. Existem strings para as quais esta tarefa é mais fácil ou mais difícil do que outras?

8.1 Informação, complexidade e aleatoridade

Seguindo a nossa discussão acima, considere os seguintes três cenários, sendo cada um deles temos acesso a uma string diferente, digamos w_1, w_2 e w_3, apresentadas a seguir.

$w_1 =$ 00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000000000000000000

$w_2 =$ 00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111
00000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111

$w_3 =$ 01101000010100110101000001010101010011101110100001010000110111011001101101 1
11010010010011010110100001110101010100111010010110101000111010101011010111111011
00111010101011110101110011101110111010010010011000100011010111111011010010111 0
10001010100110101010010101010110100010110001001110111111011010100111101011110011
11001110100011101010011010001111010101011110101010110100111010010110101000111010
10101110101111010010111010010101001010101001101111010111011010100100111010100101
10001110101001010100010101101101010011101010101000001101010101110101010010101111
01000011110101000110101011111101010101010001011101010101001010110001110101010101
00011101010101001101000100101010110100010111010101010111101010101001011010111010 1
00111010101000001101010001110101010100111010010111100101011010101011010001101001

Supondo que queremos descrever **exatamente** a string que vimos, em qual dos três cenários teríamos mais dificuldade para realizar esta tarefa? Intuitivamente, w_1 parece ser a que temos mais facilidade de memorizar (note que, de maneira semelhante, ela parece ser mais fácil de anotar em um pedaço de papel), pois ela parece conter pouquíssima informação relevante. Para descrevê-la bastaríamos dizer que ela consiste de 800 ocorrências do bit 0 (seja para memorizá-la, quanto para anotar no papel a frase "800 vezes o bit 0"). Por outro lado, a string w_3, por não conter nenhum padrão discernível, seria a string que nos daria mais trabalho para memorizar, uma vez que não teríamos muita opção senão memorizá-la bit a bit (o que não é uma tarefa simples!) ou anotar a própria string inteira, bit a bit, em um pedaço de papel. A string w_2 é um caso intermediário, mas, ainda assim, razoavelmente próximo do caso da string w_1, caso em que não há muita informação útil. Mais concretamente, temos:

Descrição de w_1: "*800 bits* 0"

Descrição de w_2: "*80 bits* 0 *e 80 bits* 1*; repita isso 4 vezes*"

Descrição de w_3: "*a sequência de bits 01101000010100110101000001010101010011101110100001010000110111011001101101 1 11010010010011010110100001110101010100111010010110101000111010101011010111111011 00111010101011110101110011101110111010010010011000100011010111111011010010111 0 10001010100110101010010101010110100010110001001110111111011010100111101011110011 11001110100011101010011010001111010101011110101010110100111010010110101000111010 10101110101111010010111010010101001010101001101111010111011010100100111010100101 10001110101001010100010101101101010011101010101000001101010101110101010010101111 01000011110101000110101011111101010101010001011101010101001010110001110101010101 00011101010101001101000100101010110100010111010101010111101010101001011010111010 1 00111010101000001101010001110101010100111010010111100101011010101011010001101001*

Ao contrário do conceito do computação, que pela Tese de Church-Turing tem uma definição única e universal, o conceito de informação muitas vezes é definido de diferentes maneiras, dependendo do contexto ou da área da ciência em que tenta-se definir o que significa informação.

Em Ciência da Computação, a nossa primeira reação é associar informação à strings de bits (quem tem inclinação a pensar em termos de física, poderia fazer a correspondência de strings com estados ou alguns "padrões" ocorrendo no meio físico usado para registrar a informação). A ideia está na direção correta, entretanto, a questão central aqui, exemplificada pele história que contamos no início do capítulo, é que strings diferentes parecem estar associadas a **quantidades diferentes** de informação. Ou seja, dizer simplesmente que **a string é a informação si** não parece muito adequado. Neste capítulo vamos tentar fomalizar estas ideias usando as ferramentas que aprendemos nesta Parte 2 do livro.

A maneira como informalmente relacionamos a quantidade de informação contida em cada uma das três strings w_1, w_2 e w_3 como sendo a menor descrição da string correspondente é a ideia central por trás do conceito de *Complexidade de Kolmogorov*. Neste capítulo veremos que a primeira string tem baixa Complexidade de Kolmogorov (ela tem uma descrição pequena), a segunda tem uma complexidade um pouco maior e a terceira tem, de certa maneira, maior complexidade possível.

> O QUE É ALGO COMPLEXO?
>
> O nome "Complexidade de Kolmogorov" sugere relacionar a quantidade de informação de uma string com a ideia de complexidade da string. Isso é intencional e nos coloca em contato com algumas questões bastante profundas em ciência.
>
> O problema de se definir o que significa algo simples ou complexo é bastante estudado em diferentes áreas de investigação científica. As ferramentas que veremos aqui apenas são apenas a proverbial "ponta do iceberg" do que sabemos sobre este assunto sob a ótica de uma área da computação conhecida como teoria algorítmica da informação.
>
> Relembrando que strings em geral podem ser usadas para descrever objetos (matemáticos) quaisquer, faz sentido pensar que a primeira string possa representar um objeto extremamente simples, a segunda string possa representar um objeto um pouco mais complexo que o primeiro, enquanto a terceira strings representa um objeto bastante complexo. Esta essencialmente é a relação que pode ser feita entre complexidade e quantidade de informação "intrínseca" ao objeto em questão.
>
> Na realidade, mesmo na própria área de teoria algorítmica da informação, existem maneiras diferentes de se medir a complexidade de uma string que podem afiar um pouco mais nossa intuição sobre informação e complexidade. Embora isto esteja fora do escopo deste livro, umas das ideia é a seguinte. Um dos primeiros argumentos contrários a teoria que veremos aqui é que terceira string não é de fato complexa, pois não passa de "ruído aleatório". O próprio Andrey Kolmogorov, na década de 60, observou isso e propôs uma variação de sua teoria para lidar com a ideia de que algo complexo não parece estar nem no extremo do "caótico" (strings aleatórias), nem no extremo da ordem (strings com padrões simples). Existe uma série de maneiras para se lidar com essa ideia, mas a intuição é que se ao invés de trabalharmos com descrições exatas da string, trabalhássemos com descrições probabilísticas, poderíamos pensar que a terceira string teria, de fato, uma descrição curta: a grosso modo algo como "w_3 é uma string aleatória".

8.2 A descrição mínima de uma string

Vamos agora ser precisos e formalizar o que queremos dizer com a descrição de uma string x. A ideia chave é que sendo a própria descrição de x também um objeto matemático, ela também uma string binária, que chamaremos de $d(x)$. Por exemplo, a frase que descreve a string w_1, i.e., "*800 bits* 0", pode ser codificada em binário, ou seja, a string $d(w_1)$ seria essa codificação em binário. A pergunta que queremos fazer é a seguinte. Qual 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 da string $d(x)$ e, sendo assim, podemos ver $d(x)$ como uma versão "compactada" de x. Vamos

definir a quantidade de informação de uma string como a menor descrição possível desta string. A maneira que vamos fazer isso é em termos de computação. Mais precisamente, o que queremos é o menor processo algorítmico que "constrói" uma dada string x. Nesta altura do curso já temos um vocabulário preciso para definir isso: queremos saber qual é a menor Máquina de Turing que escreve x em sua fita e depois para.

8.2.1 Definição de Complexidade de Kolmogorov

A melhor maneira de sermos precisos quando dizemos "a menor Máquina de Turing" para realizar alguma tarefa é nos referirmos a menor string $\llcorner M \lrcorner$, tal que a MT M realize a designada tarefa. Mas existe outra questão: quando dizemos a menor MT que escreve x em sua fita, queremos dizer exatamente o quê? Queremos dizer a menor MT M que começa a computação com a fita vazia e escreve nesta fita a string x? Mas e se existir uma outra MT M' muito menor e que escreve x em sua fita se começarmos a computação fornecendo alguma string w muito pequena de entrada? Isso significa que tendo as strings $\llcorner M' \lrcorner$ e w em mãos, nós podemos descrever x. Com isso em mente, vamos definir a menor descrição de x de uma maneira um pouco mais geral do que a ideia de encontrar uma determinada Máquina de Turing mínima. Vamos definir a menor descrição de x como a menor string $\llcorner M \lrcorner w$ tal que $M(w) = x$. Note que esta definição é mais genérica, pois ela permite que em casos particulares possamos ter $w = \varepsilon$, e portanto a descrição de x é simplesmente uma MT mínima.

Definição 8.2.1 — A descrição e descrição mínima de uma string. Uma *descrição* de x é uma string $\llcorner M \lrcorner w$ tal que $M(w) = x$. Uma *descrição mínima* de x, denotada $d(x)$, é a menor string $\llcorner M \lrcorner w$ em ordem lexicográfica tal que $M(w) = x$. Por menor string em ordem lexicográfica queremos dizer que caso haja empate entre várias descrições de tamanho mínimo, o desempate é feito pela ordem lexicográfica das strings.

Definição 8.2.2 — Complexidade de Komogorov de uma string. 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 (a ideia é simples e é o objetivo do Exercício 8.1). 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

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 estudar agora a conexão entre strings que possuem alta Complexidade de Kolmogorov e strings que parecem tipicamente aleatórias. Esta conexão reflete a nossa intuição de que é difícil dar uma descrição pequena e exata de uma string que pareça aleatória. Neste capítulo vamos apresentar apenas os conceitos elemetares, mas isso já será suficiente para provar alugns teoremas relevantes da área.

Relembrando as três strings diferentes que vimos na página anterior, nós tivemos a impressão de que a terceira string era aleatória, enquanto outras duas strings não eram. Entretanto, se fôssemos escolher uma string de 400 bits ao acaso, as três strings teriam exatamente a mesma probabilidade de serem escolhidas, mais precisamente, 2^{-400}. Para mostrarmos que strings com alta Complexidade de Kolmogorov são semelhantes a strings aleatórias, logo de cara temos um problema: o que é uma string aleatória? Esta pergunta parece um tanto intangível, uma vez que qualquer string pode ser obtida quando fazemos uma escolha aleatória. O que vamos fazer aqui é contornar isso de uma maneira simples, mas engenhosa.

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, digamos que a propriedade P é "*ter a mesma quantidade de 0's e 1's*". Esta propriedade pode ser vista como a função que mapeia strings que tem a mesma quantidade de 0's e 1's para 1 e as demais strings para 0. Por exemplo, diremos que *a string 001 não tem a propriedade* P, pois $P(001) = 0$. Por outo lado, como $P(00110011) = 1$, diremos que a propriedade de "*ter a mesma quantidade de 0's e 1's* é verdadeira para a strings 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 desta string ter uma propriedade ubíqua é alta. De fato, para qualquer propriedade ubíqua, a probabilidade de uma string escolhida ao acaso ter esta propriedade pode ser tão alta quanto queiramos, pois esta 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 obrigatóriamente terá, desde que ecolhamos uma string grande o suficiente. Portanto, faz sentido definir o seguinte:

Definição 8.3.4 — Propriedade de strings aleatórias. Se P é uma 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 vamos olhar o conjunto de strings em que a propriedade P é falsa e dividir a

prova em dois casos, dependendo se este conjunto é finito ou infinito.

Caso 1 – O conjunto de strings em que a propriedade P é falsa é finito : Este caso é trivial de provar, pois se o conjunto de strings de $\{x \in \Sigma^* \; ; P(x) = 0\}$ é finito, então o conjunto $\{x \in I \; ; P(x) = 0\}$ também é finito, pois $I \subseteq \Sigma^*$ e 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 toda string de Σ^* em ordem lexicográfica (ou seja, ε, 0, 1, 00, 01, 10, 11, 000, 001,...) e retorna a $N(w)$-ésima string x que satisfaça $P(x) = 0$. 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 que é posição da string x nesta lista. O ponto chave é que $\llcorner M \lrcorner \llcorner i_x \lrcorner$ é uma descrição da string x e que esta descrição tem tamanho $c + |i_x|$, onde $c = \llcorner M \lrcorner$.

Agora tome um n grande o suficiente tal que tenhamos $P(x) = 0$ para uma proporção de no máximo $\frac{1}{2^{c+2}}$ no conjunto de strings de tamanho até n. Note que sempre podemos tomar tal n, pois esta fração tende a zero quando n tende ao infinito e por que existem infinitas strings em que P é falsa.

Vamos ver agora, dentre o total de de $2^{n+1} - 1$ strings de tamanho $\leq n$, qual o tamanho do conjunto que tem a propriedade falsa. Sabemos que a fração do total é no máximo $\frac{1}{2^{c+2}}$ e portanto o tamanho deste conjunto de strings com a propriedade falsa é no máximo $\frac{2^{n+1}-1}{2^{c+2}}$. Com isso, dada uma string arbitrária x de tamanho n tal que $P(x) = 0$, o índice i_x de x é no máximo $\frac{2^{n+1}-1}{2^{c+2}}$.

A partir daí temos $i_x \leq \frac{2^{n+1}-1}{2^{c+2}} \leq 2^{n-1-c}$. Logo $|i_x| \leq n-1-c$ e portanto $\llcorner M \lrcorner \llcorner i_x \lrcorner \leq c + (n - 1 - c) = n - 1$. Com isso $K(x) \leq n - 1$, 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, e portanto apenas strings menores do que n podem ter $P(x) = 0$ e estarem no conjunto I de strings incompressíveis. Logo I é finito e a prova está concluída. □

Como mencionamos anteriormente, não temos as ferramentas matemáticas aqui para ir a fundo no assunto, mas a moral da história é a seguinte: a propriedade "ter uma sequência muito longa de 0's" não é ubíqua e portanto strings escolhidas ao acaso tem baixíssima probabilidade de ter esta propriedade. Um caso 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**, e isso corresponde a 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 (dentre várias outras) que a terceira string considerada no começo desta seção tem é ter um número equilibrado de 0's e 1's e, intuitivamente, sabemos que uma string escolhida aleatoriamente tem esta propriedade.

8.4 Incomputabilidade e Complexidade de Kolmogorov

Vamos fechar a aula de hoje mostrando um resultando negativo: Computar a Complexidade de Kolmogorov de uma dada string é um problema incomputável:

Teorema 8.4.1 Dada uma string x, não existe algoritmo que computa $K(x)$.

8.5 Exercícios

Exercício 8.1 Apresente uma MT M com apenas um estado tal que $\forall x \in \Sigma^*, M(x) = x$. ■

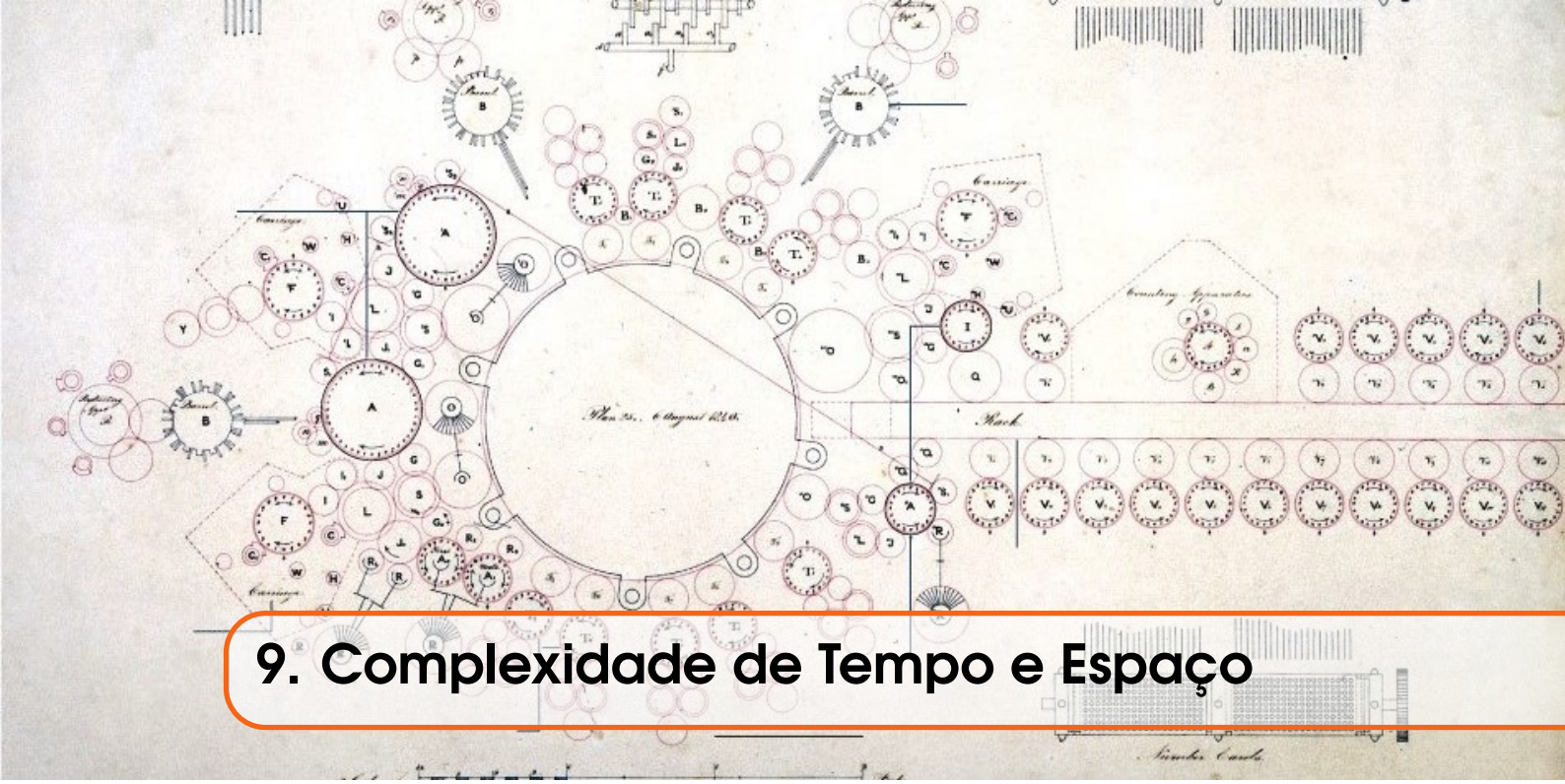
Exercício 8.2 Prove que $\forall x \in \Sigma^*, K(xx) = K(x) + c$, tal que c é uma constante. ■

Exercício 8.3 Prove que $\forall x, y \in \Sigma^*, K(xy) = K(x) + K(y) + c$, tal 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 $\forall x \in \Sigma^*$, $d(x)$ é c-incompressível. Em outras palavras, para toda string x, exceto por uma contante, a sua descrição mínima $d(x)$ já é a compressão máxima de x. Este fato está ligado a algo que intuitivamente sabemos, que é a nossa experiência frustrante de tentar comprimir um arquivo que já foi comprimido usando um bom compactador de arquivos. ■

III Parte 3: Complexidade Computacional



9. Complexidade de Tempo e Espaço

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

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

Resolver o primeiro problema parece ser bem mais fácil do que resolver o segundo problema. Se quisermos ser justos na comparação, podemos imaginar que os números que estamos somando têm 64 dígitos, assim estaríamos lidando com instâncias do problema de tamanho mais ou menos parecidas com a instância do problema no tabuleiro de xadrez (afinal, um tabuleiro de xadrez têm 64 posições). Ainda assim, com papel e caneta em dois minutos calculamos a soma dos dois números em questão. Por outro lado, para encontrar uma jogada ótima para as peças brancas[1] parece haver uma quantidade astronômica de possiblidades que teremos que levar em consideração. Este segundo problema parece ser difícil, pois para todas as possíveis ramificações de jogo advindas de todas as possíveis respostas do oponente, estamos tentando garantir que sempre deva existir uma próxima jogada ótima, e assim sussessivamente até o final do jogo.

Se por um lado realizar uma jogada ótima no xadrez parece ser difícil, por outro, pode parecer "óbvio" que dados x e y, o problema de encontrar o número z, tal que $z = x + y$ é um problema simples. Entretanto, é importante observar o seguinte: existem 10^{64} números de 64 dígitos. Se supormos que o número z que estamos procurando tenha 64 dígitos (potencialmente z pode ser ainda maior, mas vamos ignorar isso) o espaço de possíveis valores que a soma $x+y$ pode assumir também é astronomicamente grande. Ainda assim, em poucos passos, sistematicamente nós chegamos ao valor z que estamos procurando.

Há algo ainda mais importante do que o fato de que somar números de 64 dígitos é mais fácil

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

encontrar uma jogada ótima para as peças brancas em um tabuleiro 8×8. No caso mais geral, ou seja, o caso em que queremos somar dois números de n dígitos e o caso em que queremos encontrar uma jogada ótima em um tabuleiro de "xadrez generalizado" (uma versão do xadrez jogado em um tabuleiro $n \times n$), a dificuldade de se resolver o segundo problema, que já era astronomicamente maior, cresce vertiginosamente com o crescimento do valor de n.

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

> PROBLEMAS INDECIDÍVEIS *vs* PROBLEMAS INTRATÁVEIS
>
> Nesta parte do curso, os tipos de problemas que estaremos lidando são chamados de *problemas intratáveis*. Tais problemas não admitem algoritmos eficientes (como o caso do problema do xadrez generalizado) ou, na maior parte dos casos, conjectura-se não admitir algoritmos eficientes (este é o caso dos famosos problemas *NP*-completos). Ainda assim estaremos lidando com problemas decidiveis. Podemos pensar que problemas indecidíveis, como o problema da parada, estão na categoria dos problemas "impossíveis" (e não meramente "difíceis"). Entretanto, é razoável pensar que instâncais grandes de problemas intratáveis, na prática, também podem ser impossíveis de serem resolvidas.

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 leva para resolver um dado problema e a complexidade de espaço se refere a quantidade de células da fita que ela usa na resolução deste problema.

Nesta parte do curso, será conveniente trabalhar com Máquinas de Turing com 3 fitas. O Teorema 6.5.1 nos diz que MTs tradicionais são equivalentes não apenas em termos de computabilidade (ou seja, problemas que estes dois modelos podem resolver são os mesmos), mas também em termos de eficiência[2] a MTs com 3 fitas em termos de computabilidade, são equivalentes a MTs tradicionais, e portanto podemos trabalhar com O funcionamento destas máquinas é descrito a seguir.

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

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

(3) A terceira fita, chamada de *fita de saída* (também chamada de fita de resposta). Nesta fita permite-se apenas escrita (a fita é do tipo "write-only");

A principal vantagem de termos uma fita de saída é facilitar a nossa vida quando estivermos resolvendo problemas de não sejam de decisão (i.e., queremos obter uma string de Σ^* como saída, ao invés de apenas uma resposta SIM/NÃO). Neste caso vamos assumir que no final da computação

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

a terceira fita contém a saída do algoritmo. Portanto, neste caso, quando formos escrever $f(x) = y$, queremos dizer que com a entrada x na fita de entrada, o algoritmo termina com y na fita de saída.

No caso de problemas de decisão, vamos assumir que a máquina escreve o símbolo 1 ou 0 na fita de saída antes de parar (dependendo do caso em que a máquina vai aceitar ou rejeitar a string). No Exercício 5.10 mostramos que, se uma Máquina de Turing sempre para, podemos assumir que a máquina escreve o bit de resultado 1 ou 0 na terceira fita antes de efetivamente parar a sua execução.

Definição 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 w de entrada de tamanho n, a máquina para depois de executar no máximo $t_{\mathrm{M}}(n)$ transições.

■ **Exemplo 9.1** Seja M uma Máquina de Turing. Se para qualquer string de entrada w de tamanho n, M sempre para depois de fazer, no máximo, $n^2 + 3n$ transições, dizemos que a complexidade de tempo de M é $n^2 + 3n$. ■

Definição 9.1.2 — Complexidade de espaço. Dada uma MT M, a sua complexidade de espaço é uma função $s_{\mathrm{M}} : \mathbb{N} \to \mathbb{N}$ tal que, para qualquer string de entrada w de tamanho n, a máquina M para depois de usar no máximo $s_{\mathrm{M}}(n)$ posições da fita de trabalho.

■ **Exemplo 9.2** Seja M uma Máquina de Turing. Se, para qualquer string de entrada w de tamanho n, M sempre para depois de fazer, no máximo, $\log n + 7$ transições. Neste caso a complexidade de tempo de M é $\log n + 7$. ■

Assim como estamos acostumado a fazer em análise de algoritmos, na maior parte dos casos nós estaremos usando 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 $O(\log n)$.

vspace0.2cm

Notação 9.1. *Em muitas situações escreveremos $poly(n)$ para nos referirmos 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 dizemos que $f(n) = poly(n)$.*

vspace0.3cm

Definição 9.1.3 — Complexidade de tempo/espaço polinomial. Se uma MT M tem complexidade de tempo $poly(n)$, dizemos que *M é polinomial*. Se M tem complexidade de espaço $poly(n)$, dizemos que *M é de espaço polinomial*. Note que a função $poly(n)$ não necessariamente precisa ser um polinômio, pois basta que ela seja assitoticamente limitada por um polinômio.

Definição 9.1.4 — Complexidade de tempo/espaço em MTs não determinísticas. Uma Máquina de Turing não determinística é polinomial se dada uma entrada de tamanho n, *todos os ramos da árvore de computações possíveis tem profundidade $poly(n)$*. Com isso, queremos dizer que para todas as possíveis escolhas não determinísticas da MTN, ela sempre para depois de $poly(n)$ transições.

Observe que uma consequência da Definição 9.1.4 é que, em particular, estaremos lidando apenas com MTNs que não possuem ramos infinitos em sua árvore de computações possíveis.

ESPAÇO E TEMPO: RECURSOS DISPONÍVEIS PARA SE FAZER COMPUTAÇÃO

Algo interessante de se observar é que tempo e espaço são recursos básicos que a natureza nos oferece quando pensamos em efetivamente realizar computação em um meio físico. Dependendo do substrato físico que estivermos utilizando para fazer computação, espaço e tempo podem ser traduzidos de diferentes maneiras.

Por exemplo, em um algoritmo rodando em um laptop, espaço significa memória RAM e tempo significa número de intruções executadas pelo processador. No caso de computação usando moléculas de DNA, espaço significa a quantidade de moléculas necessárias para se realizar a computação e tempo significa o número de vezes que as moléculas devem "interagir" (i.e., fazer pontes de hidrogênio). Independente do modelo de computação que estamos usando, em última análise, os recursos específicos utilizados parecem ter correspondência com os conceitos gerais de espaço e tempo que conhecemos em física.

9.2 As classes P, NP e P-space

Na seção anterior nós definimos o que é a complexidade de uma Máquina de Turing. Nesta seção, nós vamos usar esta definição como base para definir a complexidade inerente de se resolver determinados problemas de decisão.

Definição 9.2.1 — Decisão em tempo polinomial. Uma linguagem L pode ser *decidida deterministicamente em tempo polinomial* se existe uma MT polinomial M que decide L. Uma linguagem L é *decidida não deterministicamente em tempo polinomial* se existe um MT não determinística que decide L em tempo polinomial.

Definição 9.2.2 — Decisão em espaço polinomial. Uma linguagem L pode ser *decidida deterministicamente em espaço polinomial* se existe uma MT de espaço polinomial M que decide L. Uma linguagem L é *decidida não deterministicamente em espaço polinomial* se existe um MT não determinística que decide L em espaço polinomial.

Vamos agora classificar linguagens de acordo com a quantidade de recursos necessários para que possamos decidí-las. Tais conjuntos de linguagens são conhecidos como *classes* de linguagens (ou classes de problemas). De agora em diante, sempre que dissermos *classes de linguagens*, estamos nos referindo a conjuntos de linguagens.

Definição 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 P-space. O conjunto de todas as linguagens decidíveis deterministicamente em espaço polinomial é denotado por P-space.

Definição 9.2.6 — A classe NP-space. O conjunto de todas as linguagens decidíveis não deterministicamente em espaço polinomial é denotado por NP-space.

O PROBLEMA P VS NP

Uma vez definidas as quatro classes acima, podemos começar a fazer o tipo de pergunta que temos feito desde o começo do curso. Estas classes são todas distintas?

Algo importante que devemos lembrar é que, muitas vezes, classes de linguagens definidas de maneiras diferentes podem eventualmente ser iguais. Por exemplo, no Capítulo 3 vimos que a classe das linguagens aceitas por DFAs era precisamente a mesma classe das linguagens aceitas por NFAs. Mas, por outro lado, em alguns modelos de computação, há diferença entre computação determinística e não determinística. Por exemplo, a classe das linguagens aceitas por PDAs não é a mesma classe das linguagens aceitas por DPDAs.

Os Teoremas 7.6.1 e 7.6.2, vistos no Capítulo 5, enuciam que o conjunto de linguagens aceitas por MTs é exatamente o mesmo conjunto das linguagens aceitas por MTNs. Para provar que *MTs* aceitam as mesmas linguagens aceitas por MTNs, o argumento que usamos é que uma MT pode simular uma MTN (o mesmo argumento vale para provar que MTs *decidem* as mesmas linguagens que podem ser decididas por MTNs). A pergunta chave neste capítulo é a seguinte: uma MT pode sempre simular de maneira *eficiente* uma dada MTN?

No capítulo seguinte veremos o problema SAT pode ser resolvido por Máquinas de Turing não determinísticas de tempo polinomial, algo que conjectura-se ser impossível de ser realizado por Máquinas de Turing determinísticas. O mesmo ocorre com vários outros problemas conhecidos como problemas NP-completos.

Exercício 9.1 Mostre que $\mathsf{P} \subseteq \mathsf{NP}$. ∎

Solução: Seja $L \in \mathsf{P}$. Pela Definição 9.2.3, existe uma MT polinomial $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ que decide L. Agora considere a Máquina de Turing não determinística $N = (Q, \Sigma, \Gamma, (\delta, \delta), q_0, B, F)$. A máquina N comporta-se exatamente da mesma maneira que M, portanto N também decide L em tempo polinomial. Logo $L \in \mathsf{NP}$ e consquentemente $\mathsf{P} \subseteq \mathsf{NP}$. □

Se quisermos provar que $\mathsf{P} \neq \mathsf{NP}$, teríamos que ser capazes de mostrar que a classe P está *estritamente* contida na classe NP, ou seja, devemos provar que existe pelo menos um problema que esteja em NP, mas que não esteja em P.

A conjectura normalmente aceita em ciência da computação é que $\mathsf{P} \neq \mathsf{NP}$. Em particular, vários problemas importantes, conhecidos como problemas NP-completos, são os problemas candidatos a pertencerem à classe $\mathsf{NP} \setminus \mathsf{P}$. O Problema da Satifatibilidade, conhecido simplemente como problema SAT (veja a Definição 7.2.6, no Capítulo 7), é um destes problemas. Mostrar que o problema SAT está na classe NP é uma tarefa simples (veremos isso no próximo capítulo). A parte difícil, e que é um dos maiores problemas matemáticos em aberto atualmente, é mostrar que o problema não pertence a classe P.

Qual é a dificuldade de se provar que o problema SAT (ou qualquer outro problema candidato a estar em $\mathsf{NP} \setminus \mathsf{P}$) não admite um algoritmo polinomial? Como já vimos em capítulos anteriores, em geral, não é uma tarefa fácil provar que determinado algoritmo não existe. Para demonstrarmos que $L_{\text{SAT}} \notin P$ teríamos que excluir logicamente a possibilidade de que todos os infinitos algoritmos polinomiais falham na tarefa de decidir L_{SAT}.

P-SPACE *vs* NP-SPACE?

O problema P *vs* NP é muito famoso, mas, por outro lado, por que nunca escutamos nada sobre o problema P-space *vs* NP-space. Qual é o motivo disso? O motivo é que este não é um problema em aberto. Não é tão difícil mostrar que P-space = NP-space. A prova é simples, mas está fora do escopo deste curso.

Uma maneira diferente de se definir classes de complexidade, e que pode ser útil em algumas circunstâncias, é a seguinte.

Definição 9.2.7 — TIME($f(n)$). Dada uma função $f : \mathbb{N} \to \mathbb{N}$, o conjunto de toda linguagem que pode ser decidida por uma Máquina de Turing com complexidade de tempo $O(f(n))$ é denotado por TIME($f(n)$).

■ **Exemplo 9.3** A classe TIME(n^3) é a classe de toda linguagem que pode ser decidida por uma MT que execute $\mathcal{O}(n^3)$ transições, tal que n é o tamanho da string de entrada. ■

■ **Exemplo 9.4** A classe TIME(2^n), que é conjunto de todas as linguagem que podem ser decidida por uma MT que execute $O(2^n)$ transições, tal que n é o tamanho da string de entrada. ■

Definição 9.2.8 — SPACE($f(n)$). Dada uma função $f : \mathbb{N} \to \mathbb{N}$, o conjunto de toda linguagem que pode ser decidida por uma Máquina de Turing com complexidade de espaço $O(f(n))$ é denotado por SPACE($f(n)$).

9.3 Exercícios

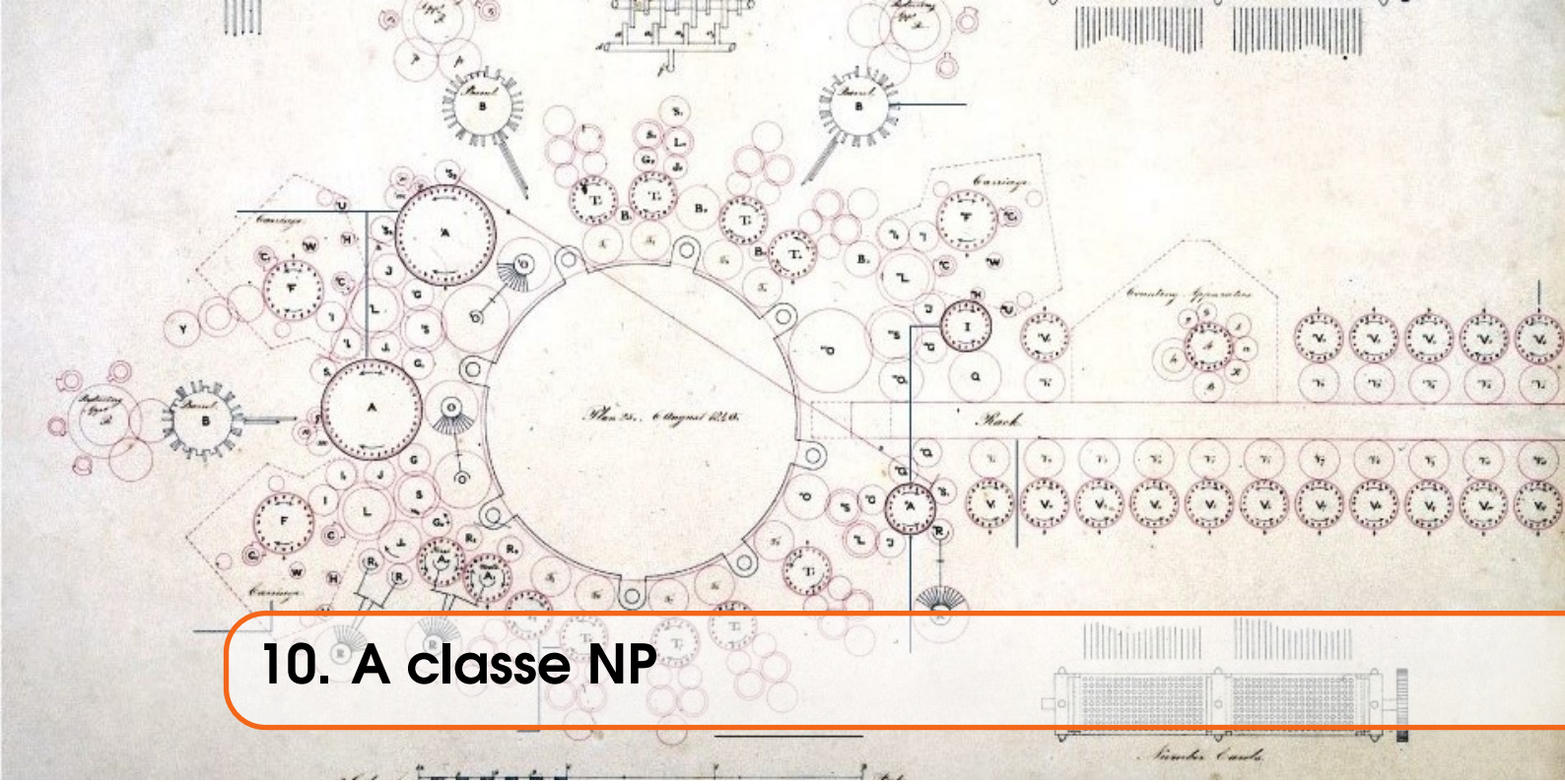
Exercício 9.2 Defina a classe P usando a definição de TIME($f(n)$). ■

Exercício 9.3 Caso estivéssemos utilizando uma Máquina de Turing usando apenas 1 fita, a classe P seria a mesma ou seria diferente? Justifique a sua resposta. Qual seria a desvantagem de se definir P-*space* usando MTs que possuam apenas 1 fita? ■

Exercício 9.4 Caso estivéssemos utilizando uma Máquina de Turing usando apenas 1 fita, teríamos que alterar a nossa definição de complexidade de espaço. Há alguma vantagem em se usar MTs com 3 fitas ao invés de MTs com apenas 1 fita quando estamos lidando com complexidade de espaço? ■

Exercício 9.5 Lembrando que $\mathscr{R}$ é o conjunto das linguagens recursivas, prove que NP $\subseteq \mathscr{R}$. ■

Exercício 9.6 Forneça uma definição para NTIME($f(n)$) de maneira semelhante a Definição 9.2.7, mas agora considerando Máquinas de Turing não Determinísticas. ■



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

Figure 10.1: O Problema Sudoku.

Na Figura 10.1 à esquerda é 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, provando que tínhamos uma instância verdadeira em mãos. Agora considere um problema de decisão diferente: dado um grid 9×9 preenchido (como o da figura a direita), queremos decidir se 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 (em breve 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, no momento, 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 de instâncias falsas, concluir que a fórmula não é satisfazível.

Algo que pode ter passado despercebido nesta discussão é que instâncias verdadeiras do problema SAT têm um *certificado* que atesta elas são realmente verdadeiras. 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

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) Se $x \in L$, existe pelo menos uma string $c \in \Sigma^*$, chamada de *certificado de* x tal que $V(x,c) = 1$. Além disso, temos requisito de que o certificado c não pode ser muito grande em relação a w. De maneira precisa, requeremos que $|c| = poly(|w|)$.

(b) Se $x \notin L$, então $\forall c \in \Sigma^*$, $V(x,c) = 0$ (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 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 a variável x_i pode aparecer em ϕ na forma original ou na forma negada). Como já discutimos, 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 $v = v_1 v_2 ... v_n$ 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.

Exercício 10.1 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 e o certificado v tem tamanho exatamente n (um bit para cada variável que aparece na fórmula) e 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 7, quando estamos lidando com problemas concretos, é muito mais conveniente apresentar o pseudo-código de um algoritmo ao invés de usar Máquinas de Turing. O verificador, apresentado no Algoritmo 10.2.1, 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.

Algorithm 4 Um algoritmo Verificador para o problema SAT.

Verificador_SAT: (ϕ, v)

```
1:  for i = 1; i ≤ m; i++ do        /* Para cada uma das m cláusulas */
2:      C_i = False                 /* Assume inicialmente que C_i não vai ser satisfeita */
3:      for j = 1; j ≤ k_i; j++ do  /* Para cada literal da cláusula */
4:          IND = GetVar(l_ij)      /* Obtém índice da variável que aparece no literal */
5:          if LitSat(l_ij, v_IND) then  /* Verifica se o bit v_IND da valoração v faz l_ij = V */
6:              C_i = True          /* Em caso afirmativo a cláusula foi satisfeita */
7:              Break               /* Sai do loop interno para testar próxima cláusula */
8:      if C_i = False then         /* Se todos literais falharam, a cláusula fica falsa */
9:          Return False            /* Portanto a v não satisfaz φ */
10: Return True                     /* Passou no teste da linha 10 para toda cláusula */
```

No Algoritmo 10.2.1, 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. A função GetVar(l_{ij}) é uma função que retorna o índice da váriável que aparece no literal l_{ij}, por exemplo, tanto no caso em que $l_{ij} = x_4$, quanto no caso em que $l_{ij} = \bar{x}_4$, a função retorna 4. A função LitSat(l_{ij}, v_{IND}) testa se o literal l_{ij} é satisfeito pela valoração em questão. Por exemplo, se $l_{ij} = x_4$, o literal é satisfeito quando $v_4 = 1$. Por outro lado, se $l_{ij} = \bar{x}_4$, o literal é satisfeito quando $v_4 = 0$.

Exercício 10.2 Mostre de maneira formal 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 P $\neq$ NP é conjecturar que existam problemas que possam ser verificados em tempo polinomial, mas que não possam 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 uma instância do problema Sudoku (o problema que mencionamos no início deste capítulo) admite solução.

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 $\mathscr{C}$ a classe de problemas que podem ser verificados em tempo polinomial. Então $\mathscr{C} = \mathsf{NP}$.

Dado o Teorema 10.2.2, podemos definir a classe NP da maneira abaixo e ainda assim estaremos falando da mesma classe de complexidade.

Definição 10.2.1 — 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 $\forall x \in \Sigma^*$:

$$x \in L \Leftrightarrow \exists u \in \Sigma^{p(|x|)}; M(x,u) = 1$$

Observe que, 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).

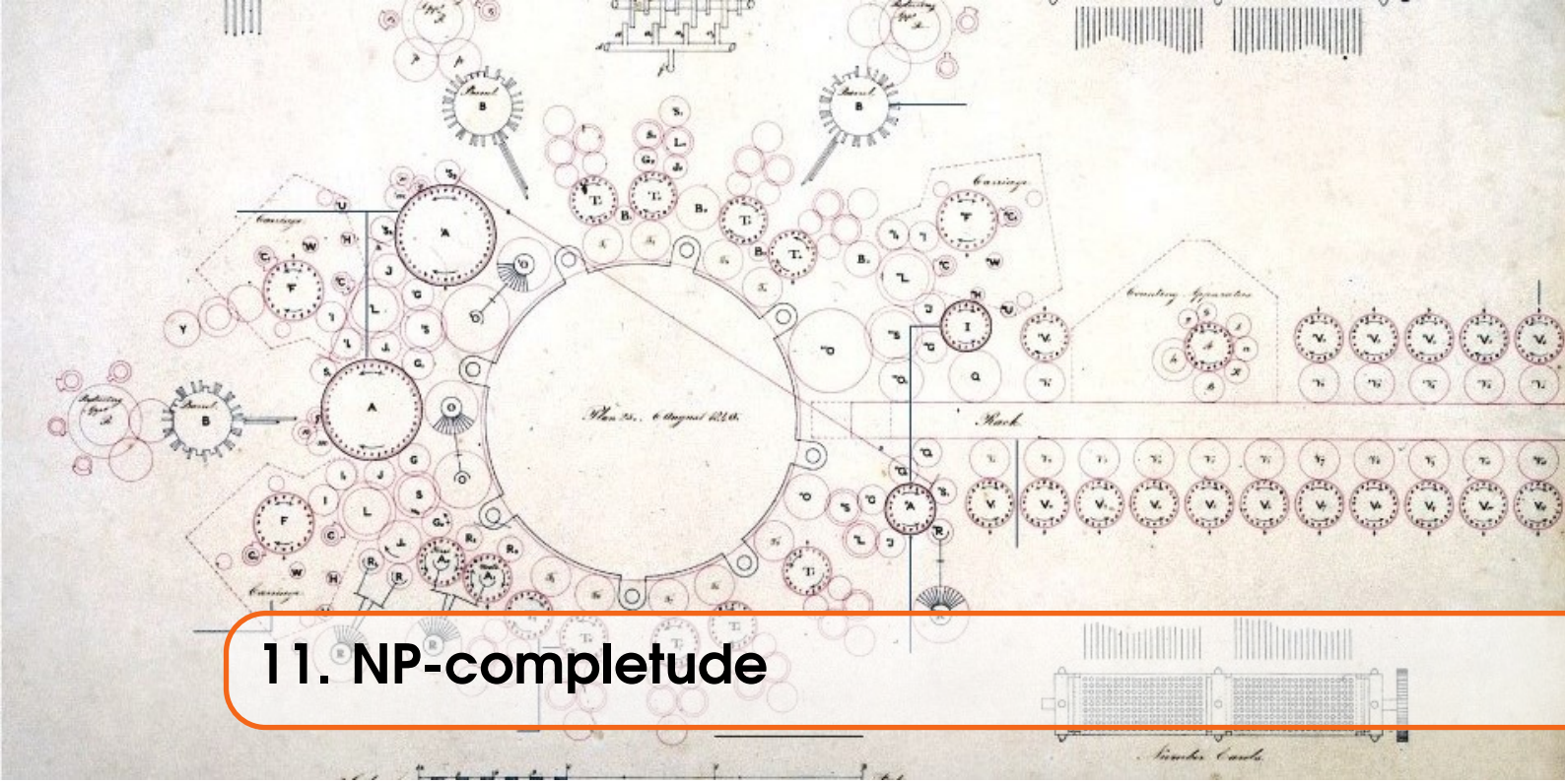
10.3 Exercícios

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_E o problema de testar se um grafo não é euleriano. Prove que $L_E \in co\text{-P}$.
- Prove que $\mathsf{P} = co\text{-P}$.

■

Exercício 10.4 Uma conjectura conhecida em complexidade computacional é que $\mathsf{NP} \neq co\text{-NP}$. Por que não poderíamos provar que $\mathsf{NP} = co\text{-NP}$ usando uma estratégia de prova semelhante a estratégia usada no exercício anterior? ■



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. Como podemos tornar precisa a ideia de que um certo problema é o mais difícil da classe NP? O que vamos fazer é provar que se existe um algoritmo polinomial que resolve 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.

Antes de apresentar o Teorema de Cook-Levin, vamos atacar a seguinte questão. Como pode a existência de um algoritmo polinomial para um problema L_1 implicar a existência de um algoritmo polinomial para outro problema L_2, sendo que os dois problemas podem ser potencialmente completamente diferentes? O primeiro passo para entender isso é entender a ideia de *redução polinomial*.

Suponha que existe um algoritmo polinomial M_1 para L_1. A ideia central é transformar o problema L_2 em L_1. Como isso é feito? Apresentendo um (outro) algoritmo R que, dada uma instância y do problema L_2, retorne uma instância x de L_1. Este algoritmo R é chamado de redução. O ponto chave é que R deve ter a seguinte propriedade: se y é uma instância verdadeira de L_2, x também deve ser uma instância verdadeira de L_1, e, por outro lado, se y é uma instância falsa de L_2, x também deve ser uma instância falsa de L_1.

> O QUE ERA UMA INSTÂNCIA MESMO?
>
> Lembrando que uma instância de um problema L é uma string qualquer $x \in \Sigma^*$ que codifica um objeto (por exemplo, se o problema em questão for um problema sobre grafos, tipicamente x é a codificação de um grafo). Se $x \in L$, dizemos que x é uma instância verdadeira de L e se $x \notin L$, dizemos que x é uma instância falsa de L.

Mas qualquer redução R basta? Não, o algoritmo R, que transforma instâncias de L_2 para L_1, deve ser polinomial. Desta maneira podemos obter uma algoritmo polinomial M_2 para resolver L_2 usando uma combinação de R com M_1. Para resolver L_2, primeiramente transformamos o problema L_2 em L_1 usando R e, depois, usamos M_1 para resolver L_1.

Definição 11.1.1 — Redução de tempo polinomial. Sejam L_1 e L_2 linguagens sobre Σ. Dizemos que L_2 é *polinomialmente redutível* à L_1 se existe uma MT polinomial R tal que $y \in L_2 \Leftrightarrow R(y) \in L_1$. Neste caso escrevemos $L_2 \leq_P L_1$. Para simplificar, muitas vezes diremos simplesmente que L_2 é *redutível* a L_1 (ao invés de dizer "polinomialmente" redutível).

■ **Exemplo 11.1** Nas seção 11.3 veremos que L_{SAT} é redutível ao problema L_{H}, portanto, a maneira formal de escrever isso é $L_{\text{SAT}} \leq_P L_{\text{H}}$. Concretamente, o que iremos fazer para provar que existe esta redução é apresentar um *algoritmo polinomial que trasforma fórmulas satisfazíveis em grafos hamiltonianos em fórmulas não satisfazíveis em grafos não hamiltonianos*. ■

Teorema 11.1.1 Se $L_2 \leq_P L_1$ e $L_1 \in \mathsf{P}$, então $L_2 \in \mathsf{P}$.

Prova: Como $L_1 \in \mathsf{P}$, então existe uma MT polinomial M_1 que decide L_1. Como $L_2 \leq_P L_1$, existe uma MT polinomial R tal que $y \in L_2 \Leftrightarrow R(y) \in L_1$. Consire o algoritmo polinomial M_2 que, dado $x \in \Sigma^*$, $M_2(x) = M_1(R_1(x))$. Portanto, $M_2(x) = 1$, se $x \in L_2$, e $M_2(x) = 0$, se $x \notin L_2$. Consequentemente M_2 decide L e, assim, $L_2 \in \mathsf{P}$. □

Definição 11.1.2 — Problemas NP-difíceis. $L \in \mathsf{NP}$-*difícil* se $\forall\, L' \in \mathsf{NP}$, $L' \leq_P L$;

Definição 11.1.3 — Problemas NP-completos. $L \in \mathsf{NP}$-*completo* se as duas condições abaixo são satisfeitas:

- L é NP-difícil;
- $L \in \mathsf{NP}$.

Alternativamente também é comum dizer que "L é NP-completo" ao invés de $L \in$ NP-completo. O mesmo vale para problemas NP-difíceis.

Teorema 11.1.2 Seja $L \in \mathsf{NP}$-*completo*. Se $L \in \mathsf{P}$, então $\mathsf{P} = \mathsf{NP}$.

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, $L' \in \mathsf{P}$. Consequentemente $\mathsf{NP} \subseteq \mathsf{P}$. Como $\mathsf{P} \subseteq \mathsf{NP}$ (ver Exercício 9.1), então $\mathsf{P} = \mathsf{NP}$. □

Teorema 11.1.3 — Teorema de Cook-Levin. L_{SAT} é NP-completo.

Prova: A prova deste Teorema está fora do escopo deste curso. □

Corolário 11.1.4 Se L_{SAT} admite um algoritmo polinomial, então é $\mathsf{P} = \mathsf{NP}$.

Uma vez que a maior parte dos pesquisadores da área de complexidade computacional conjectura que $\mathsf{P} \neq \mathsf{NP}$, uma consequência é que é bastante improvável que o problema SAT (e também qualquer problema NP-completo ou NP-difícil) admita um algoritmo polinomial.

Exercício 11.1 Mostre que a relação $\leq_P$ é transitiva. ■

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 (um problema ligado a protocolos de criptografia), isto é, dado n, 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 é o seguinte: dado um grafo com peso nas arestas representando as distâncias entre pares de cidades, encontrar o menor *tour* percorrendo cada cidade uma única vez (problema do caixeiro viajante). Este tipo de problema é conhecido como problema de otimização. Tais problemas são variações de problemas de busca em que a solução que estamos buscando satisfaz algum critério de maximização ou minimização.

> A VERSÃO DE DECISÃO DE UM PROBLEMA
>
> Na área de complexidade computacional, quando temos um problema de busca ou otimização, é bastante comum definirmos um problema de decisão semelhante ao problema original e trabalharmos com esta versão de decisão do problema. Qual é vantagem disso? Problemas de decisão, embora mais simples, comumente ainda 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 (o que não ocorre com problemas de busca e otimização, que não podem ser definidos 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. ▪

O problema acima não é de decisão, mas poderíamos pensar em uma versão de decisão deste mesmo problema da seguinte maneira: dado um par (G,k), determinar se G tem uma clique de tamanho pelo menos k. Formalmente, a versão de decisão do problema é seguinte:

Definição 11.2.1 — O problema da clique (CLIQUE). O problema de decisão, conhecido como problema CLIQUE, é definido pela linguagem $L_{\text{CL}} = \{\llcorner (G,k) \lrcorner \in \Sigma^* \; ; \; G \text{ é um grafo que contém uma clique de tamanho pelo menos } k\}$.

Exercício 11.2 Mostre que se existe um algoritmo polinomial para o problema MAXCLIQUE, então existe um algoritmo polinomial para o problema de decisão CLIQUE. ▪

Na seção 11.3, provaremos que o problema CLIQUE é NP-completo. Uma vez que um algoritmo polinomial para o problema de otimização MAXCLIQUE pode facilmente ser adaptado para resolver o problema de decisão CLIQUE (Exercício 11.2), conclui-se que se o problema de otimização puder ser resolvido em tempo polinomial, então P = NP. Em outras palavras, para mostrarmos que ambos os problemas são intratáveis, nos basta mostrar que 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. A versão de decisão deste problema é a seguinte:

Definição 11.2.2 — Problema do conjunto Independente (CI). O problema de decisão, conhecido como problema do conjunto independente, é definido pela linguagem $L_{\text{CI}} = \{\llcorner(G,k)\lrcorner \in \Sigma^* \text{ ; } G \text{ é um grafo que contém um conjunto independente de tamanho pelo menos } k\}$.

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. O teorema abaixo será essencial para que seguir em frente.

Teorema 11.3.1 Suponha que L_1 é um problema NP-completo. Se $L_2 \in \text{NP}$ 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 $L_2 \in \text{NP}$, concluímos que L_2 é NP-completo. □

A demonstração de que o problema L_{SAT} é NP-completo, resultado do Teorema de Cook-Levin, não é simples e está fora do escopo deste curso. Entretanto, para que possamos mostrar a NP-completude de outros problemas nós não teremos tanto trabalho. 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 seguinte.

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.

$$\phi \Rightarrow \boxed{\text{Redução}} \Rightarrow (G_\phi,k)$$

Figure 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)

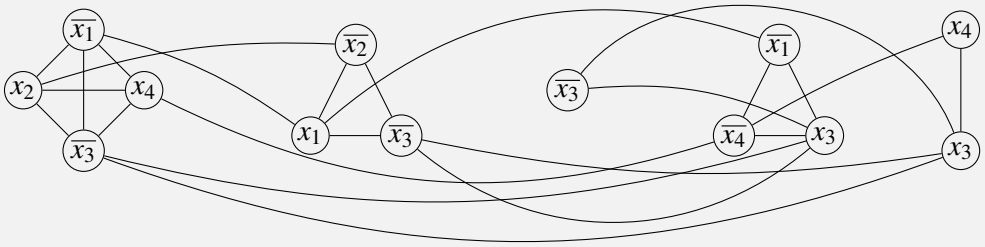
1. Crie um conjunto de vértices V de tamanho $\sum_{i}^{m} |A(C_j)|$.
2. Particione o conjunto V em subconjuntos $V_1, ..., V_m$, tal que cada subconjunto V_j tem tamanho $|A(C_j)|$ e os rotule[1] cada vértice de v_j com um elemento diferente do conjunto A_j.
3. Adicione as seguintes arestas: para cada 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 presente 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.

Pegunta: *Por 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?*

O motivo 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

[1] 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 ter rótulos iguais.

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.

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 do fluxo máximo. Dado (G, s, t), tal que G é um grafo capacitado e $s, t \in v(G)$, determinar o fluxo máximo de G partindo de s e chegando em t.

Definição 11.4.8 — Problema do emparelhamento bipartite. Dado um grafo bipartite G decidir se existe um emparelhamento perfeito de G.

Definição 11.4.9 — Problema 3-SAT. Dada uma fórmula ϕ em CNF tal que cada cláusula tenha no máximo 3 literais, decidir se ϕ é satisfazível.

Exercício 11.3 Para cada um dos problemas definidos anteriormente, faça o seguinte:

- Se o problema for de decisão, apresente uma definição formal para o problema usando uma linguagem.
- Se o problema for de otimização, apresente uma versão de decisão para o problema e em seguida apresente uma definição formal usando uma linguagem.
- Prove que a versão de decisão de cada um destes problemas está em **NP**.

■

Exercício 11.4 Seja L_{EB} o problema do Emparelhamento Bipartite e L_{FLUXO} a versão de decisão do problema de fluxo em grafos. Mostre que $L_{\text{EB}} \leq_P L_{\text{FLUXO}}$ e conclua que $L_{\text{EB}} \in \mathsf{P}$ usando a versão de decisão do algoritmo de Ford-Fulkerson. ■

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 Prove que o problema da clique (Definição 11.2.1) é NP-completo. ■

Exercício 11.10 Seja L_{CV} a versão de decisão do problema de cobertura por vértices que você obteve no Exercício 11.3. Mostre que $L_{\text{CI}} \leq_P L_{\text{CV}}$. ■