



UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA

**Apontamentos
para
Linguagens Formais e Autómatos**

Curso de Informática (2ºano)
Daniel Graça

Ano letivo 2011/12

Conteúdo

1	Introdução	1
1.1	Motivação	1
1.2	Alfabetos e linguagens	2
2	Autómatos finitos	5
2.1	Introdução	5
2.2	O modelo	5
2.3	Linguagens regulares	12
2.4	Expressões regulares	15
2.5	Linguagens não-regulares	22
3	Linguagens livres de contexto	25
3.1	Autómatos de pilha	25
3.2	Gramáticas livres de contexto	30
3.3	Linguagens que não são livres de contexto	34
4	Teoria da Computabilidade	39
4.1	Máquinas de Turing	39
4.2	Tese de Church-Turing	47
4.3	O problema da paragem	55
5	Complexidade computacional	59
5.1	Introdução	59
5.2	As classes P e NP	62
5.3	Problemas NP-completos	66

Capítulo 1

Introdução

1.1 Motivação

O hardware informático tem evoluído de forma espantosa, tendo seguido nas últimas 4 décadas a Lei de Moore:

O poder de computação de um chip duplica a cada 18 meses

Por outras palavras, o poder de computação tem crescido de forma exponencial. Na realidade esta lei não tem sido só válida para a velocidade de processamento de um CPU, mas também para outros dispositivos digitais: a memória, o número de píxeis das CCD's das câmaras digitais, etc.

Este desenvolvimento também se refletiu no software, com novas metodologias e linguagens de programação a progredir de forma muito acentuada, não sendo difícil que um engenheiro informático fique rapidamente desatualizado.

Torna-se portanto importante desenvolver um conjunto de conhecimentos básicos sobre o que é computação, mas que sejam independentes do tipo de computador utilizado, ou da linguagem de programação (se é C ou Prolog, etc.). É sobre estas questões que a teoria da computação se debruça.

A teoria da computação é importante para um informático para lhe dar mais maturidade intelectual. Por exemplo, permite compreender se novas soluções tecnológicas constituem pequenos melhoramentos (como o aumento do número de transístores no CPU), ou se pelo contrário estas constituem autênticas revoluções (como promete ser o possível desenvolvimento de computadores quânticos). Por exemplo, o primeiro vírus informático surgiu na década de 1970. No entanto a sua existência (e a forma de os conceber) é conhecida desde a década de 1930, 40 anos antes desse evento, e mesmo antes da utilização generalizada de computadores. Portanto a aparição de vírus não foi surpresa para aquelas pessoas que tinham conhecimentos de teoria da computação (é bem possível que tenha sido uma dessas pessoas a programar os primeiros vírus).

Outra característica importante da teoria da computação é que nos permite compreender melhor os limites da Informática. Assim como um cirurgião não é bom cirurgião

se não conhece os limites das técnicas cirúrgicas que aplica, o mesmo se poderá dizer de um informático que não conhece os limites da programação.

Há problemas que se sabem serem impossíveis de resolver por meio de um computador, mesmo que se pudesse esperar triliões de anos, e há outros que embora sejam resolúveis teoricamente, não o são na prática porque exigem demasiado tempo ou memória. A teoria da computação investiga e fornece resultados sobre estes problemas.

Do que é que isso lhe serve? Imagine, por exemplo, que o seu patrão lhe pede um programa para realizar uma determinada tarefa. Você tenta, mas não consegue. Você pode sempre dizer ao seu patrão que só conseguiu um algoritmo que leva dias ou meses a resolver o problema dele, e mais nada, deixando-o a pensar que você é um informático de segunda e que fez mal em tê-lo contratado. Ou então pode explicar que não é possível desenvolver algoritmos eficazes para resolver este tipo de problemas e convencê-lo que, em vez de despedi-lo, o melhor será tentar apostar numa heurística (um programa que dá uma resposta aproximada, em vez de exata), ou então outra abordagem.

1.2 Alfabetos e linguagens

Definição 1.2.1. Um *alfabeto* é um conjunto finito e não-vazio de símbolos.

Em geral iremos utilizar letras gregas maiúsculas para designar alfabetos. Por exemplo, podemos considerar $\Sigma = \{0, 1\}$ como sendo o alfabeto binário, ou ainda o alfabeto $\Pi = \{\$, \ddagger, a\}$. No, entanto, e seguindo a notação usual na literatura, quando se considerar um só alfabeto, ele será representado pela letra “sigma” maiúsculo: Σ .

Definição 1.2.2. Uma *palavra* sobre um alfabeto Σ é uma sequência finita de símbolos de Σ . O *comprimento* de uma palavra w sobre Σ é o número de símbolos presentes em w , denotado por $|w|$. Em particular a *palavra vazia*, denotada de ε , é a palavra com zero ocorrências de símbolos.

Por exemplo, 010010 é uma palavra sobre o alfabeto binário $\Sigma_1 = \{0, 1\}$ e da mesma forma *abracadabra* é uma palavra sobre o alfabeto $\Sigma_2 = \{a, b, \dots, z\}$, com comprimentos 6 e 11, respetivamente. Estritamente falando, a definição de comprimento da palavra w é incorreta, pois estamos a falar no número de *ocorrências* de símbolos de Σ , e não simplesmente no número de símbolos de w (por exemplo, *abracadabra* só tem 5 símbolos: a, b, c, d, r , mas há 11 ocorrências de símbolos de Σ_2 nesta palavra). Mas esta é uma notação normalmente utilizada que iremos manter. O conjunto das palavras de comprimento k sobre Σ é dado por

$$\Sigma^k = \{a_1 \dots a_k \mid a_i \in \Sigma, \text{ para } i = 1, \dots, k\}.$$

Em particular, $\Sigma^0 = \{\varepsilon\}$. Por exemplo, se tomarmos $\Sigma = \{0, 1\}$, tem-se $\Sigma^0 = \{\varepsilon\}$, $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$, $\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$, ...

Como uma palavra não tem comprimento fixo, o conjunto das palavras sobre Σ , designado por Σ^* , é o conjunto definido por:

$$\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \dots$$

Como iremos utilizar frequentemente o conjunto dos números inteiros não-negativos, vamos assumir nesta cadeira, como usual na teoria da computação, que $\mathbb{N} = \{0, 1, 2, \dots\}$. Note-se que esta notação difere da tradicionalmente utilizada em Portugal, em que $\mathbb{N} = \{1, 2, \dots\}$ e $\mathbb{N}_0 = \{0, 1, 2, \dots\}$.

Às vezes também vamos utilizar um expoente num símbolo ou palavra. Em geral a notação w^k , para $k \in \mathbb{N}$ e $w \in \Sigma^*$, designa a palavra

$$w^k = \underbrace{ww \dots w}_{k \text{ vezes}}$$

Por exemplo, $0^3 = 000$, $(10)^2 = 1010$, $(101)^0 = \varepsilon$.

Definição 1.2.3. Dadas duas palavras $x = x_1 \dots x_k$, $y = y_1 \dots y_m$ sobre Σ , definimos a sua *concatenação* $x \circ y$ (também representada simplesmente como xy) como sendo a palavra

$$x_1 \dots x_k y_1 \dots y_m.$$

Nos exemplos que se seguem $\Sigma = \{0, 1\}$. Por exemplo, dadas palavras $u = 1011$ e $v = 11$, definidas sobre $\Sigma = \{0, 1\}$, tem-se $uv = 101111$ e $vu = 111011$ (note-se que a ordem pela qual se faz a concatenação importa!).

Definição 1.2.4. Dada uma palavra $x = x_1 \dots x_k$, a sua *palavra reversa* é x escrita ao contrário, i.e. é $x^R = x_k \dots x_1$. Se $x = x^R$, x diz-se um *palíndromo*.

Por exemplo, $u^R = 1101$ e $v^R = 11$. Além do mais, como $v = 11 = v^R$, concluímos que v é um palíndromo.

Definição 1.2.5. Uma linguagem sobre um alfabeto Σ é um subconjunto L de Σ^* .

Por exemplo, $L = \{1, 11, 111\}$ é uma linguagem sobre Σ , assim como o conjunto vazio \emptyset , ou o conjunto $\{w \in \Sigma^* \mid w \text{ tem o mesmo número de ocorrências de } 0\text{'s do que } 1\text{'s}\}$. No entanto, $L = \{a, b, c\}$ já não é uma linguagem sobre Σ .

Definição 1.2.6. Sejam A e B linguagens sobre o alfabeto Σ . Definimos as seguintes operações cujo resultado são também linguagens sobre Σ .

- **União:** $A \cup B = \{x \in \Sigma^* \mid x \in A \text{ ou } x \in B\}$.
- **Concatenação:** $A \circ B = \{xy \in \Sigma^* \mid x \in A \text{ e } y \in B\}$.
- **Operador de fecho:** $A^* = \{x_1 x_2 \dots x_k \in \Sigma^* \mid k \geq 0 \text{ e } x_i \in A\}$.

Dito de outro modo, A^* é o conjunto de todas as palavras que podem ser obtidas concatenando palavras de A tantas vezes quantas quisermos. Definimos também

$$A^k = \underbrace{A \circ \dots \circ A}_{k \text{ vezes}} = \{x_1 x_2 \dots x_k \in \Sigma^* \mid x_i \in A\}$$

para $k \in \mathbb{N}$.

Por exemplo, tomando $A = \{10, 11\}$, $B = \{0, 111\}$, tem-se

$$A \cup B = \{0, 10, 11, 111\}$$

$$A \circ B = \{100, 10111, 110, 11111\}$$

$$A^* = \{\varepsilon, 10, 11, 1010, 1011, 1110, 1111, 101010, 101011, \dots\}$$

$$A^0 = \{\varepsilon\}$$

$$A^1 = \{10, 11\}$$

$$A^2 = \{1010, 1011, 1110, 1111\}$$

(concatenando zero palavras de A , obtém-se só a palavra vazia. Concatenando uma palavra de A , obtém-se as palavras 10 e 11. Concatenando duas palavras de A , obtém-se 1010, 1011, 1110, 1111. Concatenando três palavras de A , obtém-se ...).

Se estiver definida uma ordem sobre Σ , podemos definir a *ordem lexicográfica* em Σ^* como sendo a mesma ordem que é utilizada num dicionário, exceto que palavras mais curtas precedem palavras mais longas. Por exemplo, a ordem lexicográfica sobre $\{0, 1\}$ é dada por (assumindo que já temos a ordem definida por $0 < 1$ sobre este alfabeto)

$$\varepsilon <_{\Sigma} 0 <_{\Sigma} 1 <_{\Sigma} 00 <_{\Sigma} 01 <_{\Sigma} 10 <_{\Sigma} 11 <_{\Sigma} 000 <_{\Sigma} \dots$$

Formalmente diz-se que $u <_{\Sigma} v$, para $u, v \in \Sigma^*$ se $|u| < |v|$, ou se $|u| = |v|$ então existe k tal que $u_i = v_i$ para $i = 1, \dots, k-1$ e $u_k < v_k$.

Capítulo 2

Autómatos finitos

2.1 Introdução

Uma questão essencial da Teoria da Computação é: o que é um computador? Parece uma pergunta trivial, mas um computador é uma máquina muito complexa, e não é fácil inferir sobre as suas capacidades e limitações. Por essa razão utilizamos modelos idealizados destas máquinas, retendo apenas as características essenciais do seu funcionamento, obtendo assim resultados que são válidos para qualquer tipo de computador.

Nesta cadeira vamos estudar 3 modelos de computação, cada um com mais poder computacional do que o anterior. Grosso modo, os 3 modelos que vamos estudar são constituídos por dois elementos: uma memória; uma estrutura de controlo que pode aceder ao input e à memória, e que executa instruções pré-determinadas (o programa). Na prática, esta estrutura de controlo corresponde ao microprocessador num computador. Os três modelos utilizam o mesmo tipo de estrutura de controlo, e a diferença fundamental (mesmo que o não pareça à primeira vista) é no tipo de memória utilizado.

Os três modelos são, por ordem crescente de capacidades computacionais: autómatos finitos (memória limitada), autómatos de pilha (memória ilimitada, mas do tipo LIFO: last in, first out), e máquinas de Turing (memória ilimitada de acesso aleatório).

2.2 O modelo

Os autómatos finitos consistem na idealização de um computador que tem acesso apenas a uma quantidade limitada de memória. Na prática, computadores desse género podem ser utilizados para controlar um elevador, as portas automáticas de um supermercado, o dispositivo de injeção de um motor de combustão interna, etc. (a quantidade de memória é fixa e não é possível em geral aumentá-la). Normalmente iremos representar autómatos finitos por meio de diagramas, por ser mais fácil compreendê-los desta forma. Um exemplo de autómato finito *determinístico* (AFD) pode ser encontrado na Fig. 2.1.

Um AFD é constituído por vários estados, representados pelas bolas amarelas. Tem

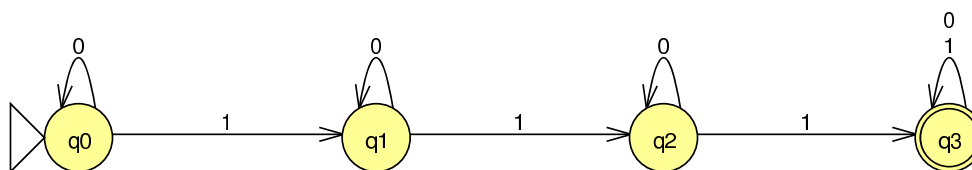


Figura 2.1: Um exemplo de autômato finito determinístico.

sempre um estado inicial, identificado pelo triângulo à sua esquerda, que no caso deste autômato é o estado q_0 . Poderá ter 0, 1, ou mais estados finais, que são identificados através de um círculo dentro do estado. No caso do AFD da Fig. 2.1 existe um só estado final, o estado q_3 . Associado a um AFD está sempre um alfabeto Σ . No caso do exemplo, o alfabeto é $\Sigma = \{0, 1\}$. Dado um estado, terá de haver sempre uma transição para outro estado (possivelmente ele próprio) associada a cada elemento de Σ . No nosso exemplo, dado um estado, terá de existir sempre uma transição (representada por uma seta) associada ao elemento 0 e outra associada ao elemento 1. **Todas** as possíveis transições têm de estar definidas num AFD.

Um input para um AFD é uma palavra em Σ^* que será aceite ou rejeitada pelo AFD. Vamos ver com um exemplo, como é que um AFD trabalha. Consideremos o AFD da Fig. 2.1 com o input $w = 0110$. Inicialmente a computação começa no estado inicial, q_0 . O AFD lê o primeiro símbolo de w , que é um 0. As regras de transição determinam que do estado q_0 com um 0, nos mantemos no estado q_0 . Depois lemos o símbolo seguinte do input, que é um 1. Então passamos para o estado q_1 . Lendo o símbolo seguinte de w , um 1, transitamos para o estado q_2 . Finalmente lemos o último símbolo de w , e a respetiva regra de transição diz para nos mantermos no estado q_2 . Neste momento a computação acabou. A palavra w será aceite se a computação acabou num estado final, e será rejeitada caso contrário. Como q_2 não é estado final, a palavra w é rejeitada. Se tomássemos como input $v = 1011$, esta palavra já será aceite. Não é difícil verificar que este AFD aceita todas as palavras binárias que contenham pelo menos três 1's, e rejeita todas as restantes.

Esta descrição por meio de diagramas é a que será normalmente utilizada nos exercícios. A “memória” de que falamos na Secção 2.1 está implicitamente codificada nos estados. Por exemplo, no AFD da Fig. 2.1, o estado q_0 memoriza o facto de que não apareceu nenhum 1 no input até ao presente momento, o estado q_1 memoriza o facto que já apareceu exatamente um 1, e o estado q_3 que já apareceram três ou mais 1's no input.

Formalmente, um AFD pode ser descrito da seguinte forma.

Definição 2.2.1. Um *autômato finito determinístico* é um 5-tuplo $(Q, \Sigma, \delta, q_0, F)$, onde

1. Q é um conjunto finito, o conjunto dos *estados*;
2. Σ é um alfabeto;

3. $\delta: Q \times \Sigma \rightarrow Q$ é a função de transição;
4. $q_0 \in Q$ é o estado inicial;
5. $F \subseteq Q$ é o conjunto dos estados finais.

Utilizando esta notação, um AFD funciona da seguinte forma. Dada uma palavra $w = w_1 \dots w_k$ sobre Σ , começamos no estado inicial q_0 e vamos ver que novo estado dá a função de transição para este estado e para o símbolo w_1 . Se $\delta(q_0, w_1) = r_1$, deste novo estado r_1 , vamos ler o símbolo seguinte w_2 da palavra w , obtendo o novo estado $r_2 = \delta(r_1, w_2)$. Repetimos este processo até chegar ao último símbolo de w . Aí o novo estado será $r_k = \delta(r_{k-1}, w_k)$ e o AFD para. Se r_k é um estado final, então o AFD aceitou w , caso contrário rejeitou a palavra. Formalmente:

Definição 2.2.2. Sejam $M = (Q, \Sigma, \delta, q_0, F)$ um autômato finito determinístico e $w = w_1 \dots w_k$ uma palavra sobre Σ . Então M aceita w se existe uma sequência de estados r_0, r_1, \dots, r_k tal que:

1. $r_0 = q_0$;
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ para $i = 0, \dots, k-1$;
3. $r_k \in F$.

Definição 2.2.3. Dizemos que o autômato finito M reconhece a linguagem L se $L = \{w \mid M \text{ aceita } w\}$.

Por exemplo, a linguagem associada ao AFD da Fig. 2.1 é

$$L = \{w \in \{0, 1\}^* \mid w \text{ contém pelos menos três ocorrências de } 1\text{'s}\}.$$

Note-se que, em AFD's, cada passo da computação é determinístico: dado o estado atual e o próximo símbolo a ser lido, sabemos exatamente qual será o próximo estado. No entanto, muitas vezes será útil permitir *não-determinismo*. Numa máquina não-determinística, dado um estado atual e um símbolo, poderão existir *várias* escolhas para o próximo estado (a palavra *várias* inclui o caso em que só há uma ou nenhuma escolha). Um exemplo, de autômato finito não-determinístico (AFND) é dado na Fig. 2.2. Como se pode ver, há estados que admitem mais do que uma transição para um dado símbolo (por exemplo, o estado q_1 admite transições para os estados q_1 e q_2 com o símbolo 1), e há estados que não admitem transições para certos símbolos (por exemplo, o estado q_0 não admite transições para o símbolo 1). Esta é a característica fundamental do não-determinismo: a possibilidade de várias transições para um dado estado e símbolo. Como o não-determinismo é uma generalização do comportamento determinístico, concluímos que todo o autômato finito determinístico é também não-determinístico.

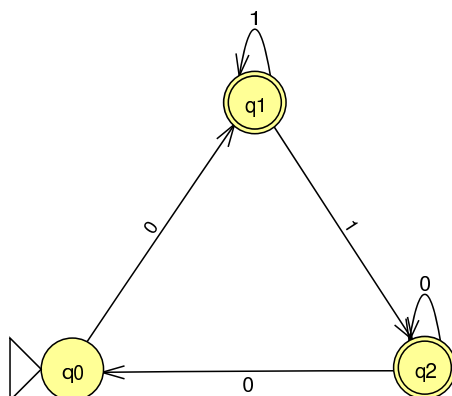


Figura 2.2: Um exemplo de autômato finito não-determinístico.

Quando há mais do que uma possibilidade de transição para um dado estado e símbolo, há também mais do que uma possibilidade de computação. Por exemplo, no autômato da Fig. 2.2, se tomarmos como input a palavra $v = 01$, há duas computações possíveis: $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_1$ e $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$. Para uma palavra ser aceite por um AFND, é preciso que exista *pelo menos uma* computação que acabe num estado final. A computação não pode acabar a meio do input (por exemplo, no autômato da Fig. 2.2, a palavra 00 é rejeitada, já que temos $q_0 \xrightarrow{0} q_1$, mas de q_1 já não podemos fazer nada com o 0 seguinte do input – a computação “morre” a meio do input, pelo que não é considerada). No exemplo anterior, a palavra v é aceite pelo AFND porque as duas computações acabam ambas num estado final (mas bastava uma das computações acabar num estado final para a palavra ser aceite). No entanto, a palavra 1 já não é aceite (não existe nenhuma computação associada a esta palavra e, em particular, não existirá nenhuma computação que acabe num estado final), mas a palavra 010 já é aceite (há duas computações associadas a esta palavra: $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_0$ e $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_2$. Como a última computação acaba num estado final, a palavra é aceite).

Na definição de um autômato finito não-determinístico, iremos também introduzir mais uma característica que é comum na literatura, e que se pode mostrar não alterar a capacidade computacional deste modelo: a possibilidade de ter transições com a palavra vazia ε . Um exemplo, é dado na Fig. 2.3, que é basicamente o AFND da Fig. 2.2, com mais uma transição que utiliza a palavra vazia.

Normalmente, dada uma palavra em Σ^* , a palavra vazia não aparece lá diretamente. Por exemplo, se $\Sigma = \{0, 1\}$, qualquer palavra em Σ^* só terá 0's e 1's. No entanto, a palavra vazia ε pode ser sempre acrescentada “à vontade”, porque não altera a palavra. Por exemplo, $010 = \varepsilon 010 = \varepsilon \varepsilon 010 = \varepsilon 01\varepsilon 0 = \varepsilon 0\varepsilon 1\varepsilon 0\varepsilon \varepsilon \varepsilon = \dots$. Portanto, se quisermos ver se 010 é aceite por um AFND que tem uma transição com a palavra vazia, temos de considerar, para além da palavra 010, todos os seus “desdobramentos” que contenham a palavra vazia, e depois computamos como se ε fosse um símbolo igual aos outros.

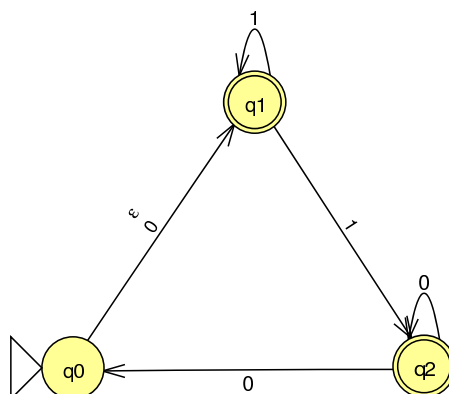


Figura 2.3: Um exemplo de autômato finito não-determinístico com transição utilizando a palavra vazia ε .

Por exemplo, para o desdobramento 010ε e para o AFND da Fig. 2.3 só temos uma computação: $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_0 \xrightarrow{\varepsilon} q_1$. Como o último estado é final, a palavra 010ε é aceite. A palavra 010 é aceite se pelo menos um dos seus “desdobramentos” é aceite. Como neste caso 010 admite um desdobramento que é aceite (010ε . Outro desdobramento seria a própria palavra 010), a palavra 010 será aceite pelo AFND da Fig. 2.3.

Formalmente:

Definição 2.2.4. Um *autômato finito não-determinístico* é um 5-tuplo $(Q, \Sigma, \delta, q_0, F)$ onde

1. Q é um conjunto finito de estados;
2. Σ é um alfabeto;
3. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q) = \{A \mid A \subseteq Q\}$ é a *função de transição*;
4. q_0 é o estado inicial;
5. $F \subseteq Q$ é o conjunto dos *estados finais*.

Como já mencionamos, num AFND podem existir vários caminhos de computação para uma palavra $w \in \Sigma^*$. Essa palavra é reconhecida se existe pelo menos um caminho de computação que acaba num estado final.

Definição 2.2.5. Seja $M = (Q, \Sigma, \delta, q_0, F)$ um autômato finito não-determinístico e w uma palavra sobre Σ . Então M *aceita* w se w pode ser escrita como $w = w_1 \dots w_k \in \Sigma \cup \{\varepsilon\}$ e se existe uma sequência de estados r_0, r_1, \dots, r_k tal que:

1. $r_0 = q_0$;

2. $r_{i+1} \in \delta(r_i, w_{i+1})$ para $i = 0, \dots, k-1$;
3. $r_k \in F$.

Da mesma forma, podemos dizer que M reconhece a linguagem L se $L = \{w \mid M \text{ aceita } w\}$. Obviamente que todo o AFD é também um AFND. Apesar de poder parecer à primeira vista que AFND's têm mais poder computacional do que os AFD's, tal não é verdade, como mostra o seguinte teorema.

Teorema 2.2.6. *Seja L a linguagem reconhecida por um autômato finito não-determinístico. Então existe um autômato finito determinístico que reconhece a mesma linguagem L .*

Demonstração. Seja $N = (Q, \Sigma, \delta, q_0, F)$ um AFND que reconhece L . Queremos construir um AFD M que reconhece L . Primeiro assumimos que não existem transições utilizando a palavra vazia ε . Vamos construir $M = (Q', \Sigma, \delta', q'_0, F')$ da seguinte forma:

1. $Q' = \mathcal{P}(Q)$.
2. Para cada $R \in Q'$ e $a \in \Sigma$, toma-se

$$\delta'(R, a) = \bigcup_{r \in R} \{\delta(r, a)\}.$$

3. $q'_0 = \{q_0\}$.
4. $F' = \{R \in Q' \mid R \text{ contém um estado final}\}$.

Não é difícil ver que M reconhece L . Falta-nos agora ver o caso em que as transições podem utilizar a palavra vazia ε . Para isso vamos definir para cada estado R de M um conjunto $E(R)$ de estados que podem ser alcançados a partir de transições com ε , tendo o cuidado de incluir os próprios membros de R . Formalmente, se $R \subseteq Q$, então tomamos

$$E(R) = \{q \in Q \mid q \text{ pode ser alcançado com } 0 \text{ ou mais transições utilizando a palavra } \varepsilon\}.$$

Posto isto, basta alterar δ' e q'_0 na definição de M para o seguinte: $q'_0 = E(\{q_0\})$ e

$$\delta'(R, a) = \bigcup_{r \in R} E(\delta(r, a)).$$

□

Note-se que a demonstração do Teorema 2.2.6 é construtiva: dado um AFND, essa demonstração indica-nos um processo que nos permite obter um AFD que reconhece a mesma linguagem. Para dar um exemplo, consideremos o AFND da Fig. 2.3. Vamos utilizar o processo indicado na demonstração do Teorema 2.2.6 para obter um AFD que reconhece a mesma linguagem.

O conjunto dos estados do AFND é $Q = \{q_0, q_1, q_2\}$. Então o conjunto dos estados do AFD que lhe está associado pela construção anterior é

$$\mathcal{P}(Q) = \{\emptyset, \{q_0\}, \{q_1\}, \{q_2\}, \{q_0, q_1\}, \{q_0, q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}.$$

Para simplificar a notação, e porque alterar o nome dos estados não muda a linguagem reconhecida por um AFD, vamos antes designar os estados do AFD por

$$Q' = \{q'_\emptyset, q'_0, q'_1, q'_2, q'_{0,1}, q'_{0,2}, q'_{1,2}, q'_{0,1,2}\}. \quad (2.1)$$

Como temos transições com a palavra vazia, o estado inicial do AFD é

$$\bar{q}_0 = E(\{q_0\}) = \{q_0, q_1\} = q'_{0,1}$$

(não se utilizou a notação q'_0 da demonstração para o estado inicial, para não se confundir com o estado q'_0 de (2.1)). Os estados finais de Q' são aqueles que estão associados a um conjunto que contém um estado final. Por exemplo, q'_\emptyset não está associado a nenhum estado e portanto não está associado a nenhum estado final. Logo q'_\emptyset não é final no AFD. Da mesma forma, q'_0 está associado ao estado q_0 do AFND, que não é final, logo q'_0 também não é final. Por outro lado, $q'_{0,1}$ está associado aos estados q_0, q_1 do AFND. Como pelo menos um deles é final (q_1), então $q'_{0,1}$ será um estado final no AFD. Procedendo desta forma concluí-se que o conjunto dos estados finais do AFD é

$$F' = \{q'_1, q'_2, q'_{0,1}, q'_{0,2}, q'_{1,2}, q'_{0,1,2}\}.$$

Falta só definir as transições no AFD. Note-se que, para cada estado de Q' , deverá estar definida **uma e uma só** transição para cada símbolo de $\{0, 1\}$, e não haverá transições para a palavra vazia. Se falhar alguma destas condições, o autómato obtido será não-determinístico, o que não é o nosso objetivo. Por exemplo, tomemos o estado $q'_{1,2}$, que está associado aos estados q_1 e q_2 do AFND. Quando tomamos o símbolo 0, o que acontece para estes dois estados? q_1 não vai para lado nenhum (o conjunto formado pelo resultado da transição é \emptyset) e q_2 vai para ele próprio e q_0 (o conjunto formado pelo resultado da transição é $\{q_0, q_2\}$). Agora unimos o resultado das duas transições para o símbolo 0 ($\emptyset \cup \{q_0, q_2\} = \{q_0, q_2\}$) e aplicamos o operador E a este conjunto (o operador retorna o conjunto, mais os estados que podem ser alcançados utilizando unicamente transições com a palavra vazia), sendo o resultado $E(\{q_0, q_2\}) = \{q_0, q_1, q_2\}$. Portanto, quando consideramos o estado $q'_{1,2}$ e o símbolo 0, o resultado da transição será o estado $q'_{0,1,2}$. Da mesma forma, se consideramos o estado $q'_{1,2}$ e o símbolo 1, o resultado da transição será o estado $q'_{1,2}$. Procedendo desta forma para todos os restantes estados de Q' , obtemos as regras de transição. Portanto um AFD que reconhece a

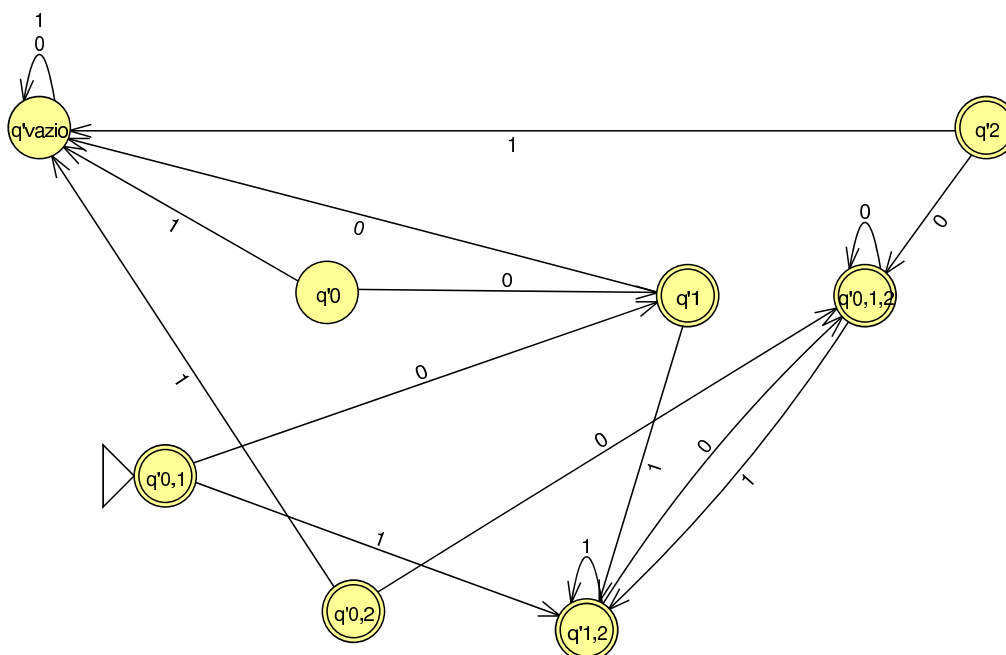


Figura 2.4: Autômato finito determinístico que reconhece a mesma linguagem que o autômato finito não-determinístico da Fig. 2.2.

mesma linguagem que o AFND da Fig. 2.3 será o AFD representado na Fig. 2.4. É possível eliminar estados redundantes, que nunca podem ser alcançados a partir do estado inicial e que portanto nunca aparecem numa computação (por exemplo $q'_{0,2}$), e isso não altera o resultado do Teorema 2.2.6.

2.3 Linguagens regulares

Vimos na secção anterior que a classe de linguagens reconhecidas por AFD's e AFND's coincide. Por isso podemos apresentar a seguinte definição.

Definição 2.3.1. Uma linguagem L diz-se *regular* se é reconhecida por algum autômato finito.

Vamos agora mostrar que a classe das linguagens regulares é fechada para as operações de união, concatenação e operador de fecho.

Teorema 2.3.2. A classe de linguagens regulares é fechada para o operador de união. Por outras palavras, se A e B são linguagens regulares, também o será $A \cup B$.

Demonstração. Sejam $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ e $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ AFND's que reconhecem A e B , respetivamente. Então construímos um AFND $M = (Q, \Sigma, \delta, q_0, F)$ que reconhece $A \cup B$ da seguinte forma.

1. $Q = Q_1 \cup Q_2 \cup \{q_0\}$, onde q_0 é um estado que não pertence a Q_1 nem a Q_2 (acrescenta-se um novo estado inicial).
2. $F = F_1 \cup F_2$ (os estados finais são aqueles que já o eram antes).
3. δ é definido da seguinte forma: para todo o $q \in Q$ e todo o $a \in \Sigma \cup \{\varepsilon\}$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{se } q \in Q_1 \\ \delta_2(q, a) & \text{se } q \in Q_2 \\ \emptyset & \text{se } q = q_0 \text{ e } a \neq \varepsilon \\ \{q_1, q_2\} & \text{se } q = q_0 \text{ e } a = \varepsilon \end{cases}$$

(do novo estado inicial fazem-se transições com a palavra ε para os antigos estados iniciais de M_1 e M_2 , mantendo todas as outras transições de M_1 e M_2).

□

Por exemplo, um AFND que reconhece a união das linguagens reconhecidas pelos autômatos finitos das Fig. 2.2 e 2.3 é o dado na Fig. 2.5.

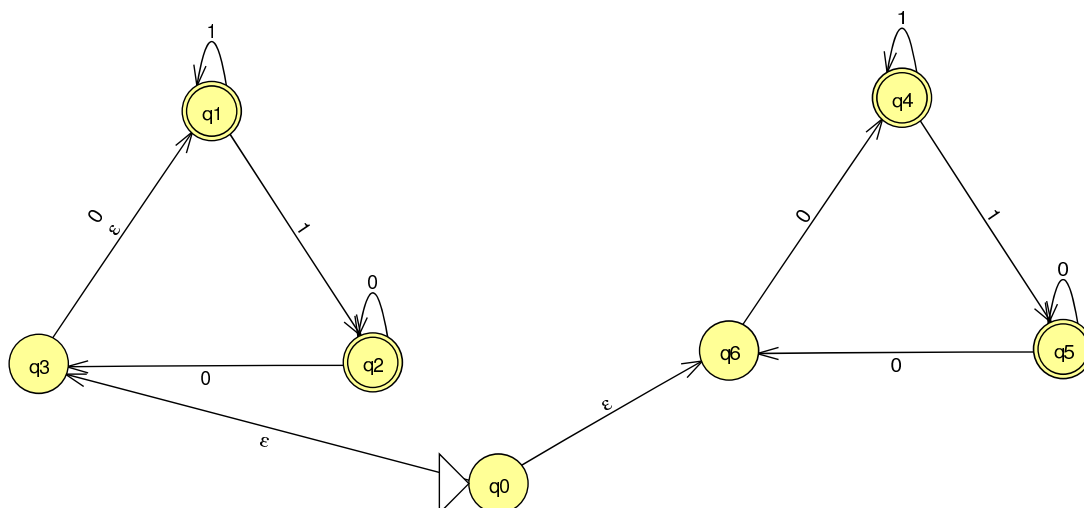


Figura 2.5: Autômato finito não-determinístico que reconhece a união das linguagens reconhecidas pelos autômatos das Fig. 2.2 e 2.3.

Teorema 2.3.3. *A classe de linguagens regulares é fechada para o operador de concatenação. Por outras palavras, se A e B são linguagens regulares, também o será $A \circ B$.*

Demonstração. Sejam $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ e $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ AFND's que reconhecem A e B , respetivamente. Então construímos um AFND $M = (Q, \Sigma, \delta, q_0, F)$ que reconhece $A \circ B$ da seguinte forma:

1. $Q = Q_1 \cup Q_2$.
2. $q_0 = q_1$ (o estado inicial do novo autômato é o estado inicial de M_1).
3. $F = F_2$ (os estados finais do novo autômato são os estados finais de M_2).
4. δ é definido da seguinte forma: para todo o $q \in Q$ e todo o $a \in \Sigma \cup \{\varepsilon\}$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{se } q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q, a) & \text{se } q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_2\} & \text{se } q \in F_1 \text{ e } a = \varepsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

(mantêm-se as transições, acrescentando transições com a palavra ε dos antigos estados finais de M_1 para o antigo estado inicial de M_2).

□

Por exemplo, se L_1 e L_2 forem as linguagens reconhecidas pelos AFND's das Fig. 2.2 e 2.3, respetivamente, então o AFND da Fig. 2.6 reconhece $L_1 \circ L_2$.

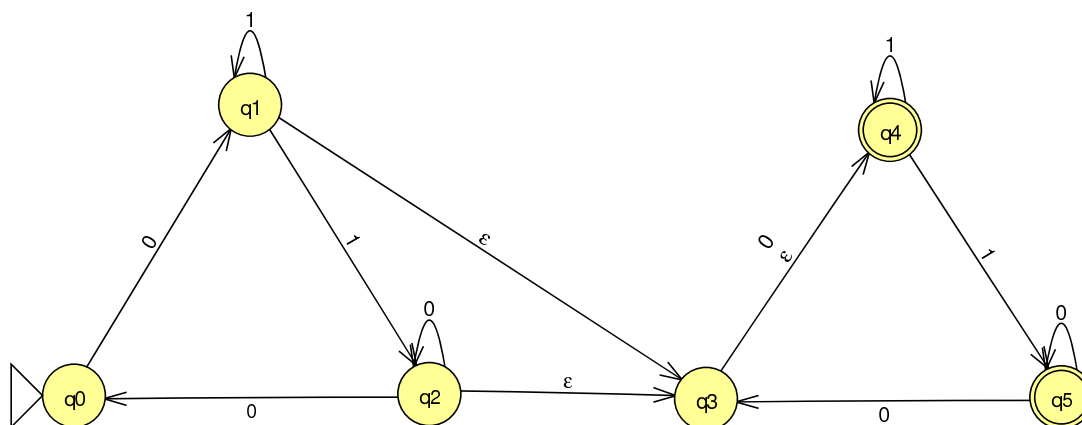


Figura 2.6: Este autômato finito reconhece $L_1 \circ L_2$, onde L_1 e L_2 são as linguagens reconhecidas pelos autômatos finitos das Fig. 2.2 e 2.3, respetivamente.

Teorema 2.3.4. *A classe de linguagens regulares é fechada para o operador de fecho. Por outras palavras, se A é uma linguagem regular, também o será A^* .*

Demonstração. Seja $N = (Q_1, \Sigma, \delta_1, q_1, F_1)$ um AFND que reconhece A . Vamos construir um AFND $M = (Q, \Sigma, \delta, q_0, F)$ que reconhece A^* da seguinte forma:

1. $Q = \{q_0\} \cup Q_1$. O estado inicial q_0 é novo e deve satisfazer $q_0 \notin Q_1$ (acrescenta-se um novo estado inicial).

2. $F = \{q_0\} \cup F_1$ (os antigos estados finais mantêm-se finais, e o novo estado inicial é também final).
3. δ é definido da seguinte forma: para todo o $q \in Q$ e todo o $a \in \Sigma \cup \{\varepsilon\}$,

$$\delta(q, a) = \begin{cases} \delta_1(q, a) & \text{se } q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q, a) & \text{se } q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q, a) \cup \{q_0\} & \text{se } q \in F_1 \text{ e } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon \end{cases}$$

(fazem-se transições com a palavra ε dos estados finais de N para o novo estado inicial, e faz-se uma transição com a palavra ε deste estado para o antigo estado inicial).

□

Por exemplo, se L_1 for a linguagem reconhecida pelo AFND da Fig. 2.2, então o AFND da Fig. 2.7 reconhece L_1^* (NOTA: É importante adicionar um novo estado inicial. Não basta tornar o estado inicial anterior num estado final, porque há casos em que este procedimento falha).

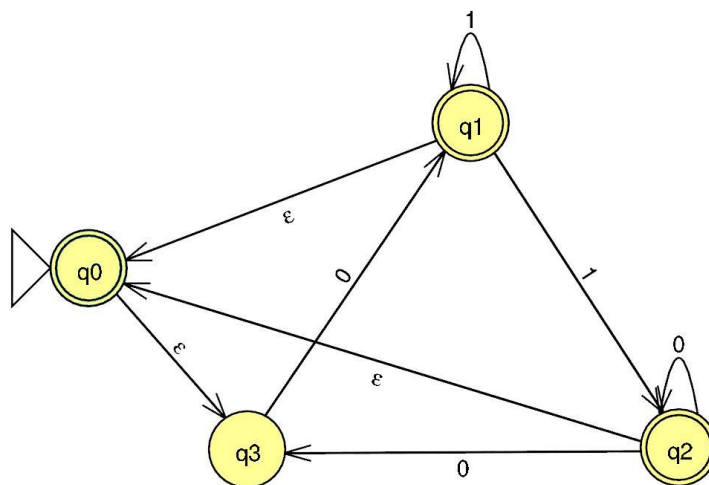


Figura 2.7: Autômato finito que reconhece L_1^* , onde L_1 é a linguagem reconhecida pelo autômato finito da Fig. 2.2.

2.4 Expressões regulares

Em certas aplicações informáticas, pretende-se procurar expressões que satisfaçam determinado formato. Por exemplo, no ambiente MS-DOS, poderíamos invocar o comando

`dir f*.*`

para procurar todos os ficheiros no diretório corrente que comecem pela letra “f” (um comando similar para sistemas Unix seria o `ls`). Outra aplicação será procurar determinada palavra num ficheiro de texto, etc. Isso pode ser modelado pela noção de expressão regular que passamos a introduzir.

Definição 2.4.1. Uma *expressão regular* sobre um alfabeto Σ é uma expressão à qual associamos uma linguagem sobre Σ . Uma expressão regular pode somente ser obtida pelas seguintes regras:

1. Se $a \in \Sigma$, então \mathbf{a} é uma expressão regular associada à linguagem $\{a\}$,
2. ε e \emptyset são expressões regulares associadas às linguagens $\{\varepsilon\}$ e \emptyset , respetivamente,
3. Se R_1 e R_2 são expressões regulares associadas às linguagens L_1 e L_2 , respetivamente, então também são expressões regulares $(R_1 \cup R_2)$, $(R_1 \circ R_2)$ e (R_1^*) , estando associadas às linguagens $L_1 \cup L_2$, $L_1 \circ L_2$ e L_1^* , respetivamente.

Por exemplo, o Unix (Linux) tem incluídas funcionalidades que lhe permitem trabalhar diretamente com expressões regulares. Mais detalhes podem ser encontrados em [HMU06]. Na prática, e para simplificar a notação, vamos utilizar as seguintes convenções quando escrevermos uma expressão regular: (i) sempre que estamos a concatenar duas expressões regulares, omitimos o símbolo \circ ; (ii) o operador de fecho tem precedência sobre os restantes operadores; (iii) o operador de concatenação tem precedência sobre o operador de união. Utilizando estas precedências, podemos eliminar muitos dos parêntesis. Por exemplo, tomando $\Sigma = \{0, 1\}$, a expressão $001^*0 \cup 01$ corresponde à expressão regular $(0 \circ 0 \circ (1^*) \circ 0) \cup (0 \circ 1)$ (a ordem pela qual a concatenação é feita não importa). Palavras que estão associadas a esta expressão regular são, por exemplo, 01, 000, 0010, 00110. A linguagem associada a $001^*0 \cup 01$ é dada por $\{w \in \{0, 1\}^* \mid w = 01 \text{ ou } w = 001^k0 \text{ para algum } k \in \mathbb{N}\}$.

Outra convenção que será usada frequentemente nos exercícios práticos é utilizar a expressão $\{\mathbf{a}, \mathbf{b}\}$ para designar $\mathbf{a} \cup \mathbf{b}$. Por exemplo a expressão $0\{0, 1\}^*$ corresponde à expressão regular $0 \circ (0 \cup 1)^*$. Se o alfabeto utilizado for $\Sigma = \{0, 1\}$, também escreveremos $0\Sigma^*$.

Vamos agora mostrar que o poder descritivo das expressões regulares é equivalente ao dos autómatos finitos.

Teorema 2.4.2. *Uma linguagem é regular se e só se está associada a alguma expressão regular.*

Repare-se que o teorema tem um “se e só se” pelo que estamos a lidar com uma equivalência. Vamos mostrar as duas implicações em dois lemas distintos.

Lema 2.4.3. *Se uma linguagem está associada a uma expressão regular, então ela é regular.*

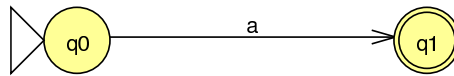


Figura 2.8: Um autómato finito que reconhece a linguagem $\{a\}$.

Demonstração. Se a expressão regular for **a** para certo $a \in \Sigma$, então o AFND representado na Fig. 2.8 reconhece $\{a\}$.

Se a expressão regular for ε , então o AFND representado na Fig. 2.9 reconhece $\{\varepsilon\}$. Para o caso de \emptyset , qualquer AFD sem estados finais reconhece \emptyset . Finalmente, o caso da união, concatenação e operador de fecho já foram todos tratados na secção anterior.

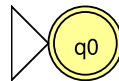


Figura 2.9: Um autómato finito que reconhece a linguagem $\{\varepsilon\}$.

□

Por exemplo, a linguagem associada à expressão regular $(00)^* \cup 10$ é reconhecida pelo AFND da Fig. 2.10 (podiam-se ter utilizado as construções anteriores para a concatenação e fecho, mas aqui procedeu-se a algumas simplificações para o AFND não ficar muito complexo).

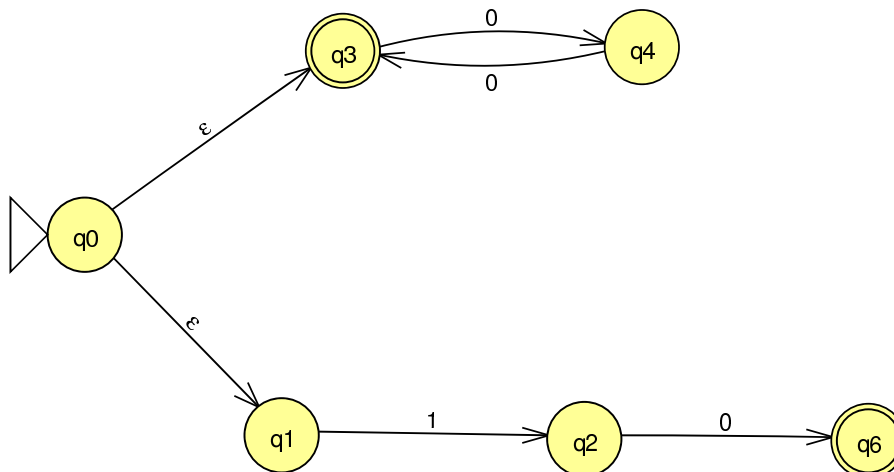


Figura 2.10: Autómato finito que reconhece a linguagem associada à expressão regular $(00)^* \cup 10$.

Lema 2.4.4. *Se uma linguagem é regular, então ela está associada a alguma expressão regular.*

Demonstração. A demonstração vai ser feita com recurso a autômato finitos generalizados (AFG). Basicamente um AFG é um AFND em que cada transição pode ter não só um símbolo de Σ ou ε , mas sim qualquer expressão regular sobre Σ . De resto, o autômato comporta-se como usualmente, exceto que cada transição pode ler logo um grupo de símbolos do input. Uma descrição formal pode ser encontrada em [Sip05]. Para além da característica essencial de cada transição poder admitir expressões regulares, vamos assumir, sem perda de generalidade, as seguintes condições:

- Um AFG tem só um estado final, do qual não partem transições para outros estados (se o autômato original não satisfizer esta condição, pode-se convertê-lo noutro equivalente adicionando um novo estado q_f , que será o único estado final, e transições com a palavra vazia que vão dos “antigos” estados finais para q_f).
- O AFG tem um estado inicial que não recebe transições de nenhum outro estado (se o autômato original não satisfizer esta condição, pode-se convertê-lo noutro equivalente adicionando um novo estado inicial q_i , e uma transição com a palavra vazia que vai de q_i para o estado inicial “antigo”).
- Entre dois estados, excetuando os casos mencionados acima, existe sempre uma e uma só transição (supõe-se que a transição de um estado para ele próprio com a palavra ε existe sempre, embora não seja representada nos diagramas. Se não existir nenhuma transição entre dois estados distintos, cria-se uma nova transição associada à expressão regular \emptyset , que não vamos representar mais à frente para simplificar os diagramas. Se houver mais do que uma transição, por exemplo, se existirem duas transições do estado q_i para o estado q_j , uma associada ao símbolo 0 e outra associada a ε , substituir essas transições por uma só que tem a união das expressões regulares associadas às transições anteriores. No exemplo anterior, passaria a existir uma só transição do estado q_i para o estado q_j , que estaria associada à expressão regular $\varepsilon \cup 0$).

O processo para obter uma expressão regular associada à linguagem reconhecida por um AFND será o seguinte: (i) converter o AFND para um AFG com k estados \mathcal{U}_k ($k \geq 2$, pois terá de ter um estado inicial e um estado final) utilizando os passos anteriores (ii) utilizar um processo (que será introduzido de seguida) que, para $k > 2$, permite determinar um AFG \mathcal{U}_{k-1} , equivalente a \mathcal{U}_k , mas que tem só $k - 1$ estados. Repetindo recursivamente o passo (ii), obtemos um sequência de AFG's equivalentes $\mathcal{U}_k, \mathcal{U}_{k-1}, \dots, \mathcal{U}_2$. Quando obtivermos o AFG \mathcal{U}_2 , que só terá o estado inicial e o estado final, e uma única transição entre estes estados, a expressão regular associada ao AFND original será a expressão regular associada a esta transição de \mathcal{U}_2 .

Falta só determinar o processo que permite converter \mathcal{U}_k em \mathcal{U}_{k-1} para $k > 2$, e mostrar que tem as propriedades desejadas.

Como \mathcal{U}_k tem $k > 2$ estados, há pelo menos um estado que não é inicial nem final. Seja q_r um desses estados, que vamos remover para obter \mathcal{U}_{k-1} . Mas para fazer isso vamos ter de “reparar” as transições para que os AFG se mantenham equivalentes. Suponhamos que em \mathcal{U}_k , o estado q_i vai para o estado q_r através da expressão regular R_1 , q_r vai para ele próprio através da expressão R_2 , q_r vai para q_j através da expressão R_3 e q_i vai para q_j através da expressão R_4 (ver Fig. 2.11). Então a transição do estado q_i para o estado q_j no AFG \mathcal{U}_{k-1} estará associada à expressão $(R_1)(R_2)^*(R_3) \cup R_4$.

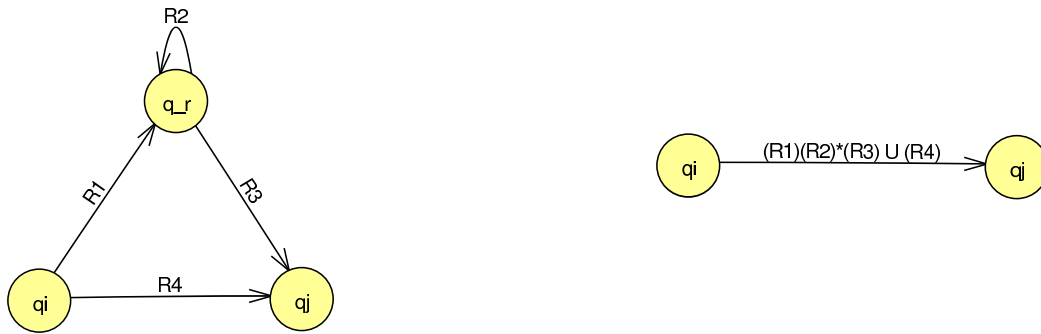


Figura 2.11: Suprimindo um estado na demonstração do Lema 2.4.4.

É fácil ver que se uma palavra w é aceite por \mathcal{U}_{k-1} , ela também será aceite por \mathcal{U}_k . Por outro lado, se w é aceite por \mathcal{U}_k , terá de existir uma sequência de estados

$$q_{inicial} \rightarrow q_{i_1} \rightarrow q_{i_2} \dots \rightarrow q_{final}$$

associada a este input. Se esta sequência de estados não inclui q_r , então é óbvio que w é também aceite por \mathcal{U}_{k-1} . Se inclui o estado q_r , existe uma primeira ocorrência deste estado na sequência, sendo o estado imediatamente anterior q_i , e uma última ocorrência, sendo o estado imediatamente seguinte q_j . A expressão regular na transição de q_i para q_j em \mathcal{U}_{k-1} descreve todas as palavras que levam q_i para q_j em \mathcal{U}_k . Portanto w será também aceite por \mathcal{U}_{k-1} . \square

A demonstração anterior é construtiva: dado um AFND, permite-nos obter uma expressão regular associada à linguagem reconhecida por este AFND. Por exemplo, consideremos o AFND da Fig. 2.12. Utilizando a construção do lema anterior, concluímos que uma expressão regular associada à linguagem reconhecida por este AFND é

$$(ab \cup aaa^*b)^* \circ (\varepsilon \cup aaa^*).$$

O processo que permite mostrar isso encontra-se esquematizado nas Fig. 2.13, 2.14, 2.15, e 2.16.

Resumindo, as linguagens regulares admitem duas formas distintas, mas equivalentes, de serem caracterizadas: uma através de máquinas (autômatos finitos), e outra mais descritiva (expressões regulares).

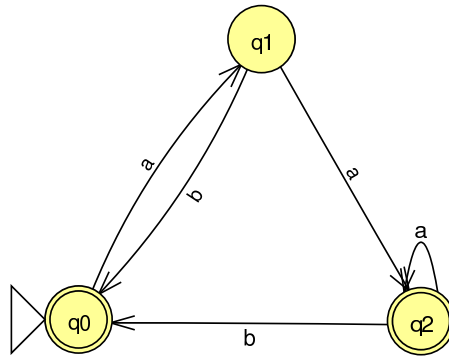


Figura 2.12: Exemplo de um AFND.

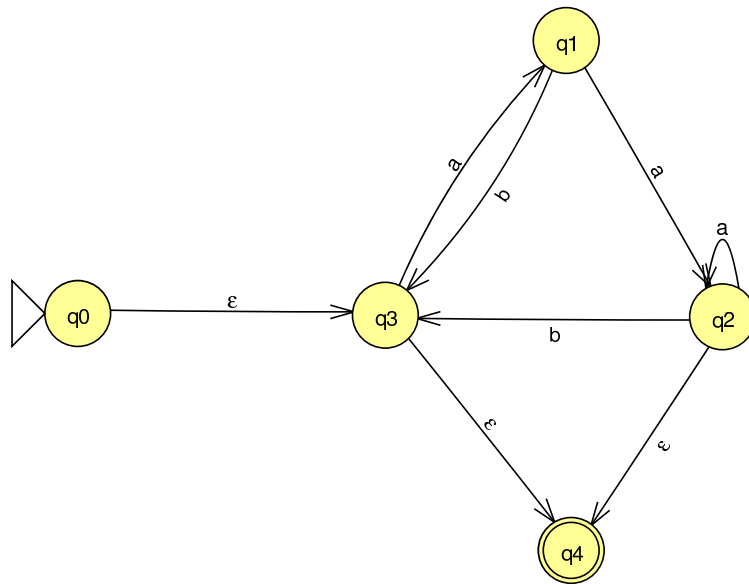


Figura 2.13: Obtenção da expressão regular associada ao AFND da Fig. 2.12: passo 1.

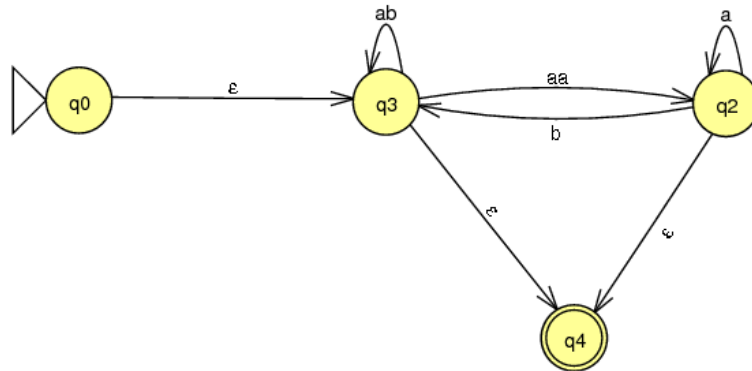


Figura 2.14: Obtenção da expressão regular associada ao AFND da Fig. 2.12: passo 2.

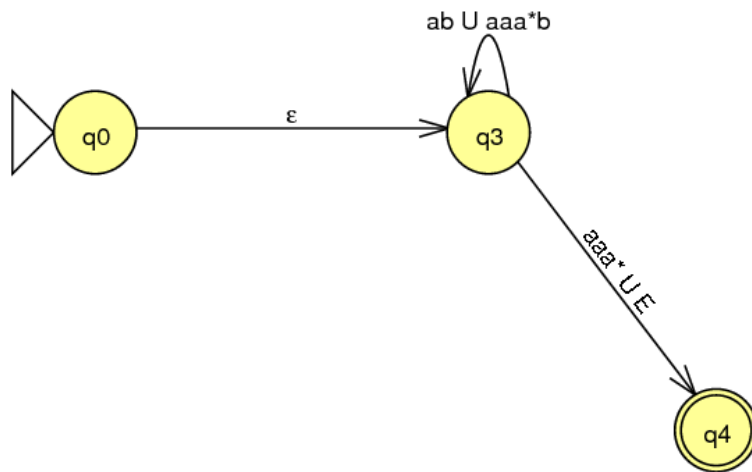


Figura 2.15: Obtenção da expressão regular associada ao AFND da Fig. 2.12: passo 3. A expressão “E” corresponde à palavra vazia ϵ .

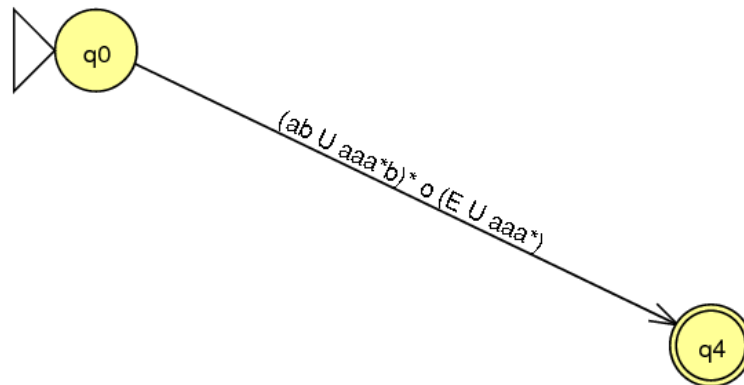


Figura 2.16: Obtenção da expressão regular associada ao AFND da Fig. 2.12: passo 4. A expressão “E” corresponde à palavra vazia ε .

2.5 Linguagens não-regulares

Em geral, para compreender quais as capacidades de um sistema, temos de ser capazes de identificar as suas limitações. Nesta secção iremos apresentar um método que, em certos casos, nos permite dizer que uma dada linguagem não é regular. Por exemplo, a linguagem

$$L = \{0^k 1^k \in \{0, 1\}^* \mid k \in \mathbb{N}\} \quad (2.2)$$

parece ser não-regular, porque parece que necessitamos de ter acesso a uma quantidade ilimitada de memória para registar o número de 0's que já apareceram durante a leitura do input, para depois compará-los com o número de 1's que hão de aparecer. No entanto, este argumento não é suficiente para mostrar que L não é regular, pois existe sempre a possibilidade de alguém inventar um método engenhoso que permita reconhecer L .

Podemos, no entanto, mostrar que L não é regular através do seguinte resultado.

Teorema 2.5.1 (Lema da bombagem). *Se A é uma linguagem regular, então existe um número p (o comprimento da bombagem) com a propriedade que se w é uma palavra em A com comprimento maior ou igual a p , então w pode ser escrita como $w = xyz$, onde x, y, z são palavras e:*

1. Para cada $i \in \mathbb{N}$, $xy^i z \in A$,
2. $|y| \geq 1$,
3. $|xy| \leq p$.

Demonstração. Sejam $M = (Q, \Sigma, \delta, q_0, F)$ um AFD que reconhece a linguagem A e p o número de estados de M . Seja ainda $w = w_1 \dots w_n$ uma palavra de A com comprimento

$n \geq p$. Seja r_1, \dots, r_{n+1} a sequência de estados que M segue para aceitar w . Esta sequência tem $n + 1 \geq p + 1$ estados. Como só existem p estados, terá de haver dois estados repetidos na sequência r_1, \dots, r_{n+1} . Suponhamos então que $r_j = r_k$, para $j < k \leq p + 1$. Se tomarmos $x = w_1 \dots w_{j-1}$, $y = w_j \dots w_{k-1}$, e $z = w_k \dots w_n$, não é difícil ver que as 3 condições do teorema são satisfeitas. \square

Em geral, para mostrar que uma dada linguagem L não é regular, utilizamos uma demonstração por absurdo: supomos que L é regular, pelo que o Lema da bombagem deveria ser válido. Depois mostramos que uma das condições do Lema da bombagem não se verifica, pelo que temos um absurdo, o que implica que L não seja regular.

Por exemplo, vamos mostrar que a linguagem definida pela equação (2.2) não é regular. Suponhamos, por absurdo, que a linguagem L é regular. Então existe um comprimento da bombagem p que satisfaz as condições do Lema da bombagem (não sabemos o valor desse p : pode ser 2, 7, ou mesmo 1000103. Por isso designamo-lo simplesmente por ' p '). Em particular, a palavra

$$w = 0^p 1^p = \underbrace{0 \dots 0}_{p \text{ vezes}} \underbrace{1 \dots 1}_{p \text{ vezes}}$$

pertence a L e tem comprimento $|w| = 2p \geq p$ (é importante fazer aparecer o p na expressão de w para garantir que tenha comprimento $\geq p$), pelo que se lhe aplica o Lema da bombagem. Então podemos decompor w em $w = xyz$, onde $x, y, z \in \{0, 1\}^*$ são palavras que satisfazem as condições 1, 2 e 3 do Lema da bombagem. Pela condição 3, $|xy| \leq p$. Mas como xy constitui o início da palavra w , e como os primeiros p símbolos desta palavra são 0's, só pode ser

$$xy = 0^k = \underbrace{0 \dots 0}_{k \text{ vezes}}$$

com $k \leq p$. Em particular, isso implica que y só pode ser constituído por 0's, isto é $y = 0^j = \underbrace{0 \dots 0}_{j \text{ vezes}}$ com $j \leq k$. Por outro lado, pela condição 2 do Lema da bombagem,

$|y| \geq 1$, pelo que $j \geq 1$. Isto implica que

$$xy^2z = \underbrace{0^{k-j}}_x \underbrace{0^j}_y \underbrace{0^j}_y \underbrace{0^{p-k} 1^p}_z = 0^{p+j} 1^p.$$

Como $p + j > p$, concluímos que $xy^2z \notin L$. Mas isto é absurdo, pois se L é regular, terá de satisfazer o Lema da bombagem e, em particular, a condição 1 que implica que $xy^2z \in L$. Então a nossa hipótese (L é regular) terá de ser falsa, isto é, L não é uma linguagem regular.

Apesar de nas aulas utilizarmos o Lema da bombagem para mostrar que uma linguagem não é regular, chama-se a atenção que há linguagens não-regulares (ver Exercício 33 das folhas práticas) que satisfazem o Lema da bombagem. Por outras palavras, se uma linguagem não satisfazer as condições do Lema da bombagem, de certeza que não é regular, mas se as satisfazer, pode ou não ser uma linguagem regular (nada se pode concluir neste caso).

Capítulo 3

Linguagens livres de contexto

Vimos no capítulo anterior que nem todas as linguagens são reconhecidas por autómatos finitos. Um exemplo é a linguagem introduzida em (2.2). Por este motivo precisamos de modelos computacionais mais poderosos que permitam, ao contrário dos autómatos finitos, utilizar uma quantidade ilimitada de memória. Uma aplicação deste tipo de linguagem é, por exemplo, na compilação de linguagens de programação.

3.1 Autómatos de pilha

No capítulo anterior, introduzimos um tipo de máquina, com memória limitada, que reconhece as linguagens regulares: o autômato finito. Em geral, um autômato finito funciona segundo a descrição esquemática apresentada na Fig. 3.1. Existe uma estrutura de controlo, que representa os estados e as funções de transição. Existe um input, e o autômato vai lendo o input da esquerda para a direita, uma casa de cada vez, atualizando a estrutura de controlo.

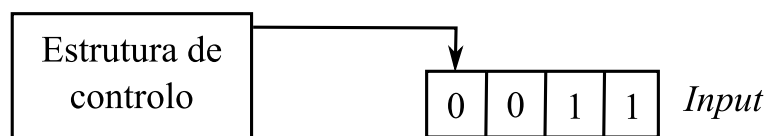


Figura 3.1: Esquema de um autômato finito.

Agora vamos estender os autómatos finitos, permitindo-lhe acesso a uma quantidade ilimitada de memória através de uma pilha (i.e. através de uma memória de tipo LIFO: last in, first out), originando os chamados autómatos de pilha (AP). Este modelo funciona como um AFND, com a diferença de incorporar uma pilha, tal como esquematizado na Fig 3.2.

Agora a regra de transição não depende apenas do símbolo atualmente lido no input e do estado atual, mas também do símbolo lido no topo da pilha. Esta regra vai alterar

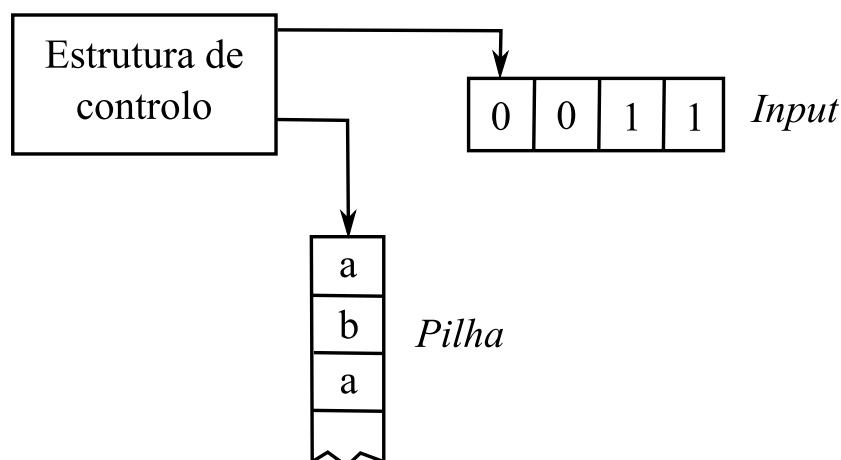


Figura 3.2: Esquema de um autômato de pilha.

não só o estado, como já acontecia nos autômatos finitos, mas também o símbolo que está no topo da pilha. Os AP também podem ser descritos através de diagramas. Um exemplo é dado na Fig. 3.3.

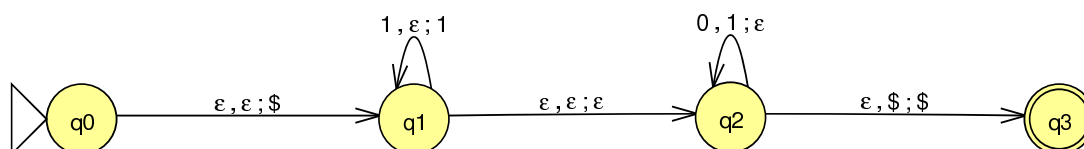


Figura 3.3: Exemplo de um autômato de pilha.

As transições são representadas por expressões do tipo $a, b; c$ ou, equivalentemente, do tipo $a, b \rightarrow c$ (que é, na minha opinião, mais clara, embora a notação com o ponto e vírgula seja a utilizada pelo software JFLAP). O símbolo a corresponde ao símbolo atualmente lido no input, o símbolo b corresponde ao símbolo atualmente lido na pilha (está no topo da pilha). O símbolo c corresponde ao novo símbolo a escrever na pilha e que substitui b . Por exemplo, a transição mais à esquerda do AP da Fig. 3.3 diz que quando se está a ler ϵ no input e ϵ no topo da pilha, então o ϵ do topo da pilha deve ser mudado para o símbolo $\$$. Repare-se que estamos a permitir não-determinismo, assim como a utilização da palavra vazia ϵ , tal como se fez no caso dos AFND. Podemos ter várias computações, dependendo das várias possibilidades que as transições oferecem e das várias possibilidades de incorporar ϵ no input e na pilha.

Na realidade, o conteúdo da pilha pode ser considerado como uma palavra s . Inicialmente é a palavra vazia (a pilha está vazia), mas cada vez que acrescentamos um novo símbolo na pilha, este símbolo é concatenado à esquerda da palavra s . Logo o símbolo do topo da pilha (que está atualmente a ser lido) é o símbolo mais à esquerda de s . Por

exemplo, o conteúdo da pilha do AP da Fig 3.2 é $s = aba\dots$. Como podemos sempre acrescentar palavras vazias à vontade e, no fundo, como o que conta para as computações é o topo da pilha, vamos ter de considerar as computações referentes à palavra $aba\dots$ (ou seja, o topo da pilha é a) e as computações referentes à palavra $\varepsilon aba\dots$ (ou seja, o topo da pilha é ε). A computação funciona como nos AFND.

Vamos tomar como exemplo o AP da Fig. 3.3 e o input $v = 1100$. Inicialmente a pilha está vazia pelo que $s = \varepsilon$. Para começarmos uma computação do estado inicial, é preciso que o símbolo lido no input seja ε . Então reescrevemos o input de forma equivalente como $v = \varepsilon 1100$. Lemos o ε do input, o ε do topo da pilha, e a regra de transição diz-nos que devemos transitar para o estado q_1 , substituindo o ε do topo da pilha por $\$$ (esta transição é tipicamente efetuada para marcar o fim da pilha com um símbolo especial, que tomamos como sendo $\$$), obtendo $s = \$$. Depois lemos o primeiro 1 do input (poderíamos também ler um ε para transitar para o estado q_2 . Mas quando fôssemos ler o 1 seguinte do input – e temos de ler todo o input – a computação “morria” porque não existe transição definida de q_2 para o símbolo 1 pelo que este ramo da computação não interessa). Só temos transições a partir do estado q_1 se o topo da pilha tiver um ε . Mas $s = \$ = \varepsilon \$$, pelo que podemos tomar a pilha como sendo $\varepsilon \$$. Assim transitamos novamente para o estado q_1 e o símbolo ε do topo da pilha foi substituído por um 1, i.e. temos agora $s = 1 \$$. Por outras palavras, acrescentamos um 1 à pilha (fizemos uma operação *push*). Lendo o 1 seguinte do input, de forma semelhante ao caso anterior, vamos continuar no estado q_1 e a pilha terá o conteúdo $s = 11 \$$.

Agora já não temos nenhum 1 no input. A única forma de continuar a computação é utilizar as palavras vazias no input e na pilha, ou seja tomar $v = \varepsilon 11\varepsilon 00$ e $s = \varepsilon 11 \$$. Com isto transitamos para o estado q_2 , e a pilha continua a ter o conteúdo $s = \varepsilon 11 \$ = 11 \$$. Agora temos duas possibilidades de transição. Como o símbolo no topo da pilha não é $\$$, a única possibilidade de transição é ler um 0 do input e um 1 da pilha. Este 1 da pilha é transformado na palavra vazia ε , pelo que $s = \varepsilon 1 \$ = 1 \$$, ou seja extraiu-se o símbolo 1 do topo da pilha (fez-se uma operação de tipo *pop*). Em geral, uma transição do tipo $a, \varepsilon \rightarrow b$ faz uma operação de tipo *push* na pilha e uma transição do tipo $a, b \rightarrow \varepsilon$ faz uma operação de tipo *pop* na pilha – ver Fig. 3.4 e 3.5.

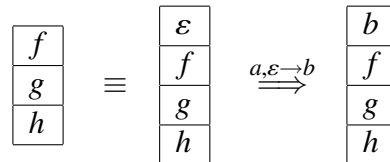


Figura 3.4: Efetuando uma operação *push* numa pilha através de uma transição do tipo $a, \varepsilon \rightarrow b$.

Em seguida lê-se o último 1 do input, obtendo-se $s = \varepsilon \$ = \$$. A computação para a palavra $v = \varepsilon 11\varepsilon 00$ acaba aqui, no estado q_2 . Como não acaba num estado final, esta computação não aceita v . Mas ainda podemos continuar a computação, para a seguinte

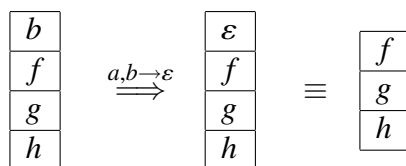


Figura 3.5: Efetuando uma operação *pop* numa pilha através de uma transição do tipo $a, b \rightarrow \varepsilon$.

variante de $v = \varepsilon 11\varepsilon 00\varepsilon$. Neste caso lê-se o último ε do input e o $\$$ da pilha (garante que a pilha está vazia) e transita-se para o estado q_3 . Como acabaram todos os símbolos para esta variante do input, e o estado atingido é final, conclui-se que v é aceite, isto é, aceitamos a palavra 0011.

Da mesma forma se conclui que as palavras $\varepsilon, 10, 111000, 1^k 0^k$ são aceites e, por exemplo, as palavras $0, 1, 01, 0011, 11000, 110$ são rejeitadas pelo AP da Fig. 3.3. Assim podemos associar a este AP M uma linguagem, a constituída pelas palavras aceites por M :

$$L(M) = \{0^k 1^k \in \{0, 1\}^* | k \in \mathbb{N}\}. \quad (3.1)$$

Note-se que esta linguagem não é regular (já foi visto na Secção 2.5). Formalmente um AP pode ser definido da seguinte forma.

Definição 3.1.1. Um autómato de pilha é um 6-tuplo $(Q, \Sigma, \Gamma, \delta, q_0, F)$ onde:

1. Q é um conjunto finito (de estados),
2. Σ é um alfabeto (de entrada),
3. Γ é um alfabeto (da pilha),
4. $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q \times (\Gamma \cup \{\varepsilon\}))$ é a função de transição,
5. $q_0 \in Q$ é o estado inicial,
6. $F \subseteq Q$ é o conjunto dos estados finais.

Considere-se o autómato de pilha $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. Uma palavra $w \in \Sigma^*$ é *aceite* pelo autómato de pilha M se w pode ser escrita como $w = w_1 \dots w_n$, onde cada $w_i \in \Sigma \cup \{\varepsilon\}$, e se existem sequências de estados $r_0, r_1, \dots, r_n \in Q$ e de palavras $s_0, s_1, \dots, s_n \in \Gamma^*$ tais que as seguintes três condições são satisfeitas:

1. $r_0 = q_0$ e $s_0 = \varepsilon$ (a computação começa com o estado inicial e a pilha vazia);
2. Para $i = 1, \dots, n$, tem-se $(r_i, b) \in \delta(r_{i-1}, w_i, a)$, onde $s_{i-1} = at$ e $s_i = bt$ para certos $a, b \in \Gamma \cup \{\varepsilon\}$ e $t \in \Gamma^*$ (a sequência de estados e de conteúdos da pilha é uma possível computação de M);

3. $r_n \in F$ (o último estado é final).

Os dois primeiros passos correspondem a uma *computação* do AP M (esta computação pode ou não acabar num estado final). A *linguagem reconhecida* por um autômato de pilha M será obviamente a linguagem

$$L(M) = \{w \in \Sigma^* \mid M \text{ aceita } w\}.$$

Definição 3.1.2. Uma linguagem é *livre de contexto* se é a linguagem reconhecida por algum autômato de pilha.

Teorema 3.1.3. *Toda a linguagem regular é livre de contexto.*

Demonstração. Se a linguagem L é regular, ela é reconhecida por um AFD M . Pegando em cada regra de transição de M , do tipo $\delta(q, a)$, basta altera-la para $\delta'(q, a, \varepsilon) = \{(\delta(q, a), \varepsilon)\}$ para obter um AP M' que reconhece a mesma linguagem L . Na prática isso equivalente a mudar o símbolo a em cima de uma seta de transição num AFD (ou AFND) para $a, \varepsilon \rightarrow \varepsilon$ para obter o AP correspondente. No fundo, este AP não usa a pilha, pelo que se comporta como um AFND, reconhecendo a mesma linguagem que M . \square

Por exemplo, utilizando a construção da demonstração do teorema anterior, concluímos que a linguagem reconhecida pelo AFND da Fig. 2.2 é reconhecida pelo AP da Fig. 3.6.

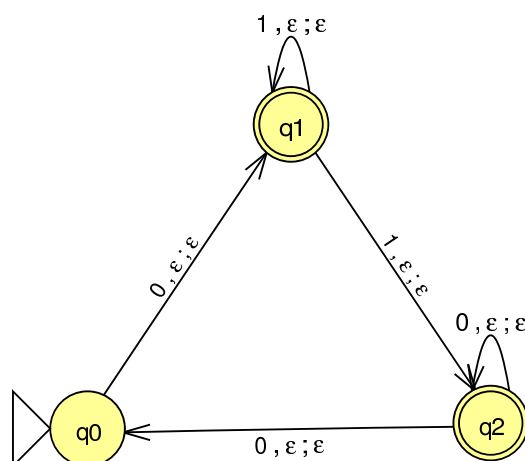


Figura 3.6: Um exemplo de autômato de pilha que reconhece a mesma linguagem que o autômato finito da Fig 2.2.

concluímos também que a classe das linguagens livres de contexto inclui estritamente a classe das linguagens regulares, pois a linguagem dada por (3.1) é livre de contexto, mas não é regular.

3.2 Gramáticas livres de contexto

No caso das linguagens regulares, demos duas caracterizações equivalentes mas distintas para esta classe, uma através de máquinas e outra através de expressões. Vamos fazer a mesma coisa para as linguagens livres de contexto. Já vimos como podem ser caracterizadas através de máquinas (utilizando os autómatos de pilha), e agora vamos ver como podem ser descritas de forma mais algébrica, através das gramáticas livres de contexto.

Antes de darmos a definição formal de gramática livre de contexto, vamos ver um exemplo de gramática livre de contexto, que designaremos de G_1 :

$$\begin{aligned} A &\rightarrow 0A1 \\ A &\rightarrow B \\ B &\rightarrow \# \end{aligned}$$

G_1 consiste num conjunto de regras de substituição (neste caso, 3 regras). Cada regra tem um símbolo que aparece à esquerda da seta, designado de variável (G_1 tem por variáveis A e B), e à direita da seta existe uma palavra que é constituída por variáveis, e por outro tipo de símbolos, os terminais (em G_1 os terminais são os símbolos $0, 1, \#$). Uma das variáveis é designada como sendo a variável inicial (nesta cadeia tomaremos a convenção de ser a variável que aparece à esquerda da seta da primeira regra. No caso da gramática G_1 será a variável A). Depois vamos gerando palavras começando com a variável inicial, e utilizando regras de G_1 para ir expandindo, passo a passo, cada variável, até que o resultado só contenha terminais. Assim é possível gerar palavras sobre o alfabeto dos terminais. Por exemplo, a palavra $0\#1$ pode ser gerada pela gramática G_1 , já que

$$A \Rightarrow 0A1 \Rightarrow 0B1 \Rightarrow 0\#1.$$

Da mesma forma, a palavra $000\#111$ também pode ser gerada pela gramática G_1 , pois

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111.$$

Em geral, já que só nos interessa o resultado final, e para não descrever todos os passos intermédios, iremos escrever simplesmente $A \xRightarrow{*} 000\#111$ (i.e. $000\#111$ pode ser obtido de A aplicando regras de G_1). É fácil ver que a gramática G_1 gera exatamente todas as palavras com o formato $0^k\#1^k$, onde $k \in \mathbb{N}$. Portanto, podemos definir a linguagem da gramática livre de contexto G_1 como sendo

$$L(G_1) = \{0^k\#1^k \in \{0, 1, \#\}^* \mid k \in \mathbb{N}\}.$$

Note-se que, na definição de G_1 , há duas regras que têm a variável A . Para simplificar a notação, iremos fundir as duas regras numa só, separando as palavras à direita das setas pelo símbolo '|'. Por outras palavras, iremos utilizar a seguinte descrição equivalente

para G_1 :

$$\begin{aligned} A &\rightarrow 0A1 \mid B \\ B &\rightarrow \#. \end{aligned}$$

Vamos agora definir formalmente os conceitos que acabamos de introduzir.

Definição 3.2.1. Uma *gramática livre de contexto* é um 4-tuplo (V, Σ, R, S) , onde:

1. V é um conjunto finito (de *variáveis*),
2. Σ é um conjunto finito, que não contém elementos de V (conjunto dos *terminais*),
3. $R \subseteq V \times (V \cup \Sigma)^*$ é o conjunto das *regras*,
4. S (a *variável inicial*) é um elemento de V .

Para simplificar a notação e aproximá-la da que já introduzimos informalmente, se $A \in V$, $u \in (V \cup \Sigma)^*$, e $(A, u) \in R$, então escrevemos esta regra como $A \rightarrow u$. Novamente, se houver duas regras $A \rightarrow u$ e $A \rightarrow v$ que comecem com A , então escrevemos

$$A \rightarrow u \mid v.$$

Se $x, y, u \in (V \cup \Sigma)^*$, $A \in V$, e $A \rightarrow u$ é uma regra da gramática, dizemos que xAy origina xuy , e escrevemos $xAy \Rightarrow xuy$. Escrevemos $u \xRightarrow{*} v$ se $u = v$ ou se existe uma sequência u_1, u_2, \dots, u_k tal que

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \dots \Rightarrow u_k \Rightarrow v.$$

A esta sequência chamamos *derivação* de v a partir de u . A linguagem gerada por uma gramática livre de contexto é o conjunto de todas as palavras que podem ser derivadas a partir do símbolo inicial.

Definição 3.2.2. A *linguagem de uma gramática livre de contexto* $G = (V, \Sigma, R, S)$ é o conjunto

$$L(G) = \{w \in \Sigma^* \mid S \xRightarrow{*} w\}.$$

Por exemplo, a gramática $G = (\{S\}, \{0, 1\}, \{S \rightarrow 0S1, S \rightarrow \varepsilon\}, \{S\})$, onde as regras podem ser escritas como

$$S \rightarrow 0S1 \mid \varepsilon,$$

origina a linguagem $L(G) = \{0^k 1^k \mid k \in \mathbb{N}\}$. A gramática dada pela regra

$$S \rightarrow S$$

não origina nenhuma palavra, pelo que a linguagem desta gramática livre de contexto é \emptyset .

Às vezes, dada uma gramática G , é possível ter duas derivações distintas de uma palavra a partir do símbolo inicial. Por exemplo, a gramática definida com as seguintes regras:

$$\langle \text{EXPR} \rangle \rightarrow \langle \text{EXPR} \rangle + \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle \times \langle \text{EXPR} \rangle \mid \langle \text{EXPR} \rangle | a$$

permite gerar a palavra $a + a \times a$ através de duas derivações. Neste caso a gramática diz-se ambígua.

Definição 3.2.3. Uma palavra w diz-se *ambígua* na gramática livre de contexto G se existem duas derivações distintas da palavra a partir do símbolo inicial. Neste caso diz-se também que a gramática G é *ambígua*.

Muitas vezes é possível transformar uma gramática ambígua noutra não-ambígua e que gera a mesma linguagem. No entanto, este procedimento nem sempre é possível.

Tal como no capítulo anterior, temos um resultado que relaciona linguagens de gramáticas livres de contexto com linguagens livres de contexto.

Teorema 3.2.4. *Uma linguagem é livre de contexto se e só se é a linguagem de alguma gramática livre de contexto.*

A demonstração deste resultado vai ser dividida em duas partes.

Lema 3.2.5. *Se uma linguagem é a linguagem de uma gramática livre de contexto, então ela é reconhecida por um autómato de pilha.*

Demonstração. Aqui apenas iremos apresentar um esboço da demonstração. Os detalhes formais podem ser encontrados em [Sip05], por exemplo.

Seja $L(G)$ uma linguagem livre de contexto gerada pela gramática G . Vamos construir um autómato de pilha que aceita uma palavra w se e só se $w \in L(G)$. A descrição informal desse autómato é dada pelos seguintes passos:

1. Meter o símbolo \$ na pilha e depois a variável inicial de G .
2. Repetir os seguintes passos:
 - (a) Se o topo da pilha é um símbolo de variável A , de forma não-determinística seleccionar umas das regras de G de expansão para A e substituir A na pilha pelos correspondentes símbolos da regra.
 - (b) Se o topo da pilha é um terminal a , ler o símbolo seguinte do input e compará-lo com a . Se são iguais, repetir a regra 2, se são diferentes, rejeitar (neste ramo de computação não-determinística).
 - (c) Se o topo da pilha é o símbolo \$, ir para um estado final [Oinput será aceite apenas se foi completamente lido].

□

Lema 3.2.6. *Se uma linguagem é reconhecida por um autômato de pilha, então ela é livre de contexto.*

Demonstração. Seja $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ um autômato de pilha. Queremos construir uma gramática livre de contexto G tal que w é aceite por P se e só se $w \in L(G)$. Antes de construirmos G vamos assumir, sem perda de generalidade, que (i) P tem um único estado final q_{final} (ver o exercício 50), (ii) P esvazia a pilha antes de aceitar uma palavra e (iii) cada transição adiciona ou retira um símbolo não-nulo da pilha (i.e. efetua uma operação *push* ou *pop*. Uma transição que substitua um símbolo por outro na pilha pode sempre ser decomposta em duas transições deste tipo efetuadas sequencialmente). A nossa gramática G terá a seguinte propriedade: para quaisquer estados p, q de P , existe uma variável A_{pq} que gera todas as palavras que levam o estado p com a pilha vazia para o estado q com a pilha vazia. Neste caso, $A_{q_0q_{final}}$ vai gerar todas as palavras aceites por P .

A gramática G é constituída pelas variáveis $\{A_{pq} | p, q \in Q\}$ e a variável inicial é obviamente $A_{q_0q_{final}}$. As regras de transição são:

- Para todos os $p, q, r, s \in Q, t \in \Gamma$ e $a, b \in \Sigma$, se $(r, t) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, t)$, então introduzir a regra $A_{pq} \rightarrow aA_{rs}b$ em G .
- Para todos os $p, q, r \in Q$, introduzir a regra $A_{pq} \rightarrow A_{pr}A_{rq}$ em G .
- Para todo o p , introduzir a regra $A_{pp} \rightarrow \varepsilon$ em G .

Esta gramática tem as propriedades desejadas, como se pode mostrar por indução (a estrutura onde se aplica a indução depende do caso que estamos a considerar):

(1) (Demonstração por indução no número de passos da derivação de w a partir de A_{pq}) Queremos mostrar que se A_{pq} gera a palavra w , então w leva o estado p com a pilha vazia para o estado q com a pilha vazia. O caso base é trivial, já que consiste numa única aplicação de uma regra de G para obter w . Logo a regra em causa só pode ser do tipo $A_{pp} \rightarrow \varepsilon$, donde $p = q$, e é óbvio que a palavra ε leva o estado p com a pilha vazia para o estado p com a pilha vazia. Suponhamos agora que o resultado é verdadeiro para palavras geradas por $\leq k$ aplicações de regras de G . Suponhamos que $A_{pq} \xrightarrow{*} w$ com $k + 1$ aplicações de regras de G . Então a primeira regra utilizada nessa derivação será $A_{pq} \rightarrow aA_{rs}b$ ou $A_{pq} \rightarrow A_{pr}A_{rq}$.

No primeiro caso teremos $w = ayb$, $A_{rs} \xrightarrow{*} y$ e y pode ser derivado de A_{rs} com k aplicações de regras de G . Aplica-se portanto a hipótese de indução a y , pelo que P pode ir com a pilha vazia de r para s (partindo do estado p , com o símbolo a vamos para o estado r e metemos t na pilha. Fazemos a computação de r a s utilizando y e, depois de acabar essa computação, só temos t na pilha. Lendo um b no input e o t da pilha, podemos apagar o t da pilha e ir para o estado q). Além do mais, como $A_{pq} \rightarrow aA_{rs}b$, tem-se que $(r, t) \in \delta(p, a, \varepsilon)$ e $(q, \varepsilon) \in \delta(s, b, t)$, pelo que P pode ir com a pilha vazia de

p para q .

No segundo caso teremos $w = yz$ e $A_{pr} \xRightarrow{*} y$, $A_{rq} \xRightarrow{*} z$ com $\leq k$ aplicações de regras de G . Pela hipótese de indução, P pode ir de p para r com a pilha vazia (lendo a porção y do input), e daí para q com a pilha vazia (lendo a porção z do input), o que mostra o resultado.

(2) Queremos mostrar que se w leva o estado p com a pilha vazia para o estado q com a pilha vazia, então w pode ser gerado a partir de A_{pq} . A indução é feita no número de passos da computação de P . O caso base ($n = 0$) é trivial, já que se começamos no estado p , ficamos no estado p . A única palavra que consegue fazer isso é ε , que pode ser gerada pela regra $A_{pp} \rightarrow \varepsilon$. Suponhamos agora que o resultado é válido para computações de comprimento $\leq k$ passos. Seja w uma palavra que leva o estado p com a pilha vazia para o estado q com a pilha vazia em $k + 1$ passos. Há dois casos a ter em conta na computação de p para q com a pilha vazia: (a) No estado p é inserido um símbolo t na pilha que fica lá até ser removido na transição final que leva ao estado q ; (b) a propriedade anterior não se verifica, i.e. a pilha fica vazia numa posição intermédia da computação.

No caso (a) sejam a e b os símbolos do input lidos no primeiro passo e último passos, respetivamente, r o estado depois do primeiro passo, e s o estado antes do último passo. Então $(r, t) \in \delta(p, a, \varepsilon)$, $(q, \varepsilon) \in \delta(s, b, t)$ e, pela definição de G , $A_{pq} \rightarrow aA_{rs}b$. Se $w = ayb$, então y leva o estado r com a pilha vazia (ou com t) para o estado s com a pilha vazia (ou com t , respetivamente) em $k - 1$ passos. Logo aplica-se a hipótese de indução e $A_{rs} \xRightarrow{*} y$, donde $A_{pq} \xRightarrow{*} ayb = w$.

No caso (b), seja r um estado intermédio em que a pilha está vazia. Então é possível ir do estado p com a pilha vazia para o estado r com a pilha vazia, e daí para o estado q com a pilha vazia, sendo estas duas computações feitas em $\leq k$ passos. Por hipótese de indução, $A_{pr} \xRightarrow{*} x$, $A_{rq} \xRightarrow{*} y$, com $w = xy$. Mas então a regra $A_{pq} \rightarrow A_{pr}A_{rq}$, válida para este caso, dá-nos $A_{pq} \xRightarrow{*} w$. □

3.3 Linguagens que não são livres de contexto

Embora a classe das linguagens livres de contexto seja mais extensa do que a classe das linguagens regulares, existem linguagens não livres de contexto. Podemos mostrar isso, por exemplo, através do seguinte resultado.

Teorema 3.3.1 (Lema da bombagem para linguagens livres de contexto). *Se A é uma linguagem livre de contexto, então existe um número p (o comprimento da bombagem) com a propriedade que se w é uma palavra em A com comprimento maior ou igual a p , então w pode ser escrita como $w = uvxyz$, onde u, v, x, y, z são palavras e:*

1. Para cada $i \in \mathbb{N}$, $uv^i xy^i z \in A$,
2. $|vy| \geq 1$,

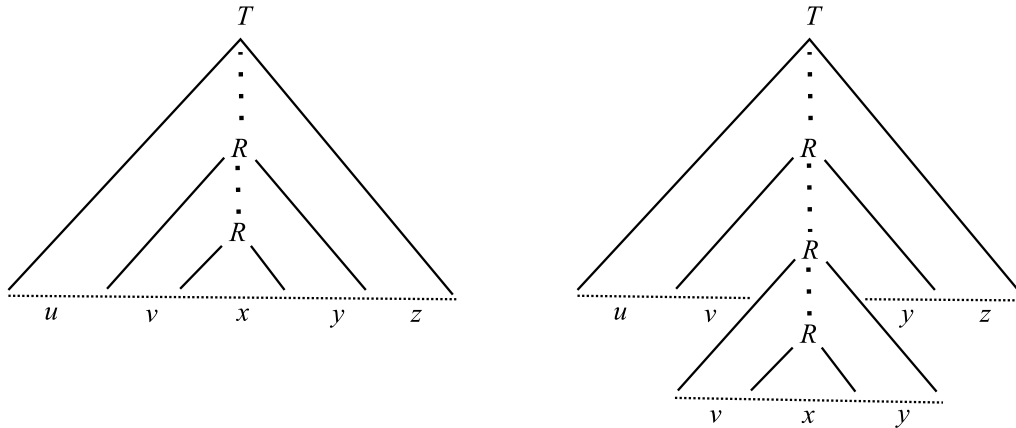


Figura 3.7: Árvores de derivação.

3. $|vxy| \leq p$.

Demonstração. Seja G uma gramática livre de contexto que gera a linguagem A e seja b o número máximo de símbolos no lado direito de uma regra. Podemos supor, sem perda de generalidade, que $b \geq 2$ (se houver uma regra $A \rightarrow B$, ela pode ser substituída por $A \rightarrow B|C$, $C \rightarrow C|B$).

Dada essa gramática, podemos fazer uma árvore de derivações a partir do símbolo inicial S , e é fácil verificar que cada vértice não pode ter mais de b filhos. Portanto, no nível 1 da árvore temos, no máximo, b folhas, no nível 2, b^2 folhas, e no nível k , b^k folhas. Logo, se a árvore que gera uma palavra tem profundidade $\leq k$, então a palavra não pode ter comprimento superior a b^k . Seja $|V|$ o número de variáveis de G , e tome-se $p = b^{|V|+2}$. Portanto qualquer árvore que gera uma palavra de comprimento $\geq p$ terá profundidade $\geq |V| + 2$.

Seja w uma palavra de comprimento $\geq p$. Vamos utilizar o *princípio do pombo*¹ para mostrar que w satisfaz as propriedades indicadas no teorema. Seja \mathcal{A} uma árvore de derivação da palavra w , tal que nenhuma outra árvore de derivação de w tenha menos vértices do que \mathcal{A} . Esta árvore terá profundidade $\geq |V| + 2$. Então haverá um caminho descendente de comprimento $\geq |V| + 2$ em que aparecem pelo menos $|V| + 1$ variáveis (a última folha é um símbolo e não uma variável). Pelo princípio do pombo, nessa sequência de símbolos haverá um símbolo R repetido, com duas ocorrências nos últimos $|V| + 1$ vértices que aparecem no caminho mencionado acima.

Dividimos a palavra w de acordo com a Fig. 3.7. Cada ocorrência de R tem uma subárvore debaixo dela. Na ocorrência de cima de R , a subárvore gera a palavra vxy , enquanto que a subárvore da ocorrência em baixo de R gera x . Como ambas as árvores são geradas pela mesma variável, podemos substituir uma subárvore pela outra: substituindo

¹Este princípio afirma que se n pombos devem ser metidos em m pombais, com $n > m$, então há de certeza um pombo com mais de um pombo.

a árvore maior pela menor, obtemos uxz . Substituindo a árvore menor pela maior, de forma iterada, podemos gerar a palavra $uv^i xy^i z$ para $i \geq 2$. Isto mostra o ponto 1 do teorema.

Para obter a condição 2, temos de garantir que v e y não são ambas ε . Se, por absurdo, $v = y = \varepsilon$, então $w = uxz$ e podemos substituir a subárvore da ocorrência de cima de R pela subárvore da ocorrência de baixo de R . Esta árvore continua a gerar w , mas tem menos vértices do que \mathcal{A} , o que é absurdo (\mathcal{A} é uma árvore mínima de derivação de w).

Para a condição 3, como as ocorrências que consideramos de R se situam nos últimos $|V| + 1$ vértices, então a subárvore de R que gera vxy tem profundidade $\leq |V| + 2$, e só pode gerar uma palavra de comprimento $\leq b^{|V|+2} = p$. \square

Vamos agora dar um exemplo de como aplicar o Lema da bombagem (para linguagens livres de contexto), para mostrar que uma linguagem não é livre de contexto. O procedimento é semelhante ao das linguagens regulares, mas agora há mais casos a tratar porque, para as linguagens livres de contexto, as condições do Lema da bombagem são um pouco mais complexas.

Seja

$$L = \{0^k 1^k 0^k \in \{0, 1\}^* \mid k \in \mathbb{N}\}. \quad (3.2)$$

Vamos mostrar que esta linguagem não é livre de contexto. Suponhamos, por absurdo, que a linguagem L é livre de contexto. Então existe um comprimento da bombagem p que satisfaz as condições do Lema da bombagem. Em particular, a palavra

$$w = 0^p 1^p 0^p = \underbrace{0 \dots 0}_{p \text{ vezes}} \underbrace{1 \dots 1}_{p \text{ vezes}} \underbrace{0 \dots 0}_{p \text{ vezes}}$$

pertence a L e tem comprimento $|w| = 3p \geq p$ (é importante fazer aparecer o p na expressão de w para garantir que w tenha pelo menos p símbolos), pelo que se lhe aplica o Lema da bombagem. Então podemos decompor w em $w = uvxyz$, onde $u, v, x, y, z \in \{0, 1\}^*$ são palavras que satisfazem as condições 1, 2 e 3 do Lema da bombagem. Pela condição 3, $|vxy| \leq p$. Vamos considerar dois casos:

1. A palavra vxy começa no primeiro bloco de 0's da palavra w . Como vxy tem, no máximo, p símbolos, só pode ser $vxy = 0 \dots 0 = 0^k$ ou $vxy = 0 \dots 0 1 \dots 1 = 0^k 1^j$ com $j, k \geq 1$. Pela condição 2, $|vy| \geq 1$. Logo v e y não podem ser simultaneamente a palavra vazia ε . Suponhamos, sem perda de generalidade, que $v \neq \varepsilon$ (se for $v = \varepsilon$, então terá de ser $y \neq \varepsilon$, e um raciocínio semelhante aplica-se a y). Como v é uma subpalavra (prefixo) de vxy , terá de ser $v = 0^l$ ou $v = 0^l 1^i$, com $l, i \geq 1$. Mas então

$$uv^2 xy^2 z = uvvxyyz$$

terá uma palavra com pelo menos $p + l$ zeros, seguido de, possivelmente um bloco com 1's e 0's, seguindo-lhe um bloco de p 1's, seguido de um bloco de p 0's.

Logo $uv^2xy^2z \notin L$. Mas isto entra em contradição com a condição 1 do Lema da bombagem.

2. A palavra vxy não começa no primeiro bloco de 0's da palavra w . Então vxy começa no bloco de 1's, ou no segundo bloco de 0's. Logo só pode ser $vxy = 1\dots 1 = 1^k$, $vxy = 1\dots 10\dots 0 = 1^k0^j$ ou $vxy = 0\dots 0 = 0^j$ com $j+k \geq 1$. Pela condição 2, $|vy| \geq 1$. Logo v e y não podem ser simultaneamente a palavra vazia ε . Suponhamos, sem perda de generalidade, que $v \neq \varepsilon$ (se for $v = \varepsilon$, então terá de ser $y \neq \varepsilon$, e um raciocínio semelhante aplica-se a y). Como v é uma subpalavra (prefixo) de vxy , terá de ser $v = 1^l$, $v = 1^l0^i$, ou $v = 0^i$, com $l+i \geq 1$. Mas então

$$uv^2xy^2z = uvvxxyyz$$

começará com um bloco de p 0's, seguido de um bloco de p 1's, seguido de uma palavra com pelo menos $p+l$ 1's e $p+i$ 0's, onde $i+l \geq 1$. Logo $uv^2xy^2z \notin L$, mas isto entra em contradição com a condição 1 do Lema da bombagem.

Como em ambos os casos chegamos a um absurdo, concluímos que a linguagem L não é livre de contexto.

Apesar de nas aulas utilizarmos o Lema da bombagem para mostrar que uma linguagem não é livre de contexto, novamente, há linguagens não-livres de contexto (ver Exercício 58 das folhas práticas) que satisfazem o Lema da bombagem. Por outras palavras, se uma linguagem não satisfizer as condições do Lema da bombagem (para linguagens livres de contexto), de certeza que não é livre de contexto, mas se as satisfizer, nada se pode concluir.

Capítulo 4

Teoria da Computabilidade

Até agora abordamos dois modelos de computação, autômatos finitos e autômatos de pilha, que apesar de úteis, não são suficientemente poderosos para capturar o poder computacional de um computador. Isso será feito através de um modelo computacional, a máquina de Turing, que iremos estudar neste capítulo.

Anteriormente, como vimos, os autômatos finitos são modelos com uma quantidade de memória finita, enquanto que um autômato de pilha consegue aceder a uma quantidade infinita de memória, através de uma pilha. O problema é que se queremos aceder a um símbolo que está no fundo da pilha, vamos ter de fazer várias operações *pop*, perdendo toda a informação que está por cima desse símbolo. Isso já não acontece se utilizarmos uma memória de acesso aleatório, e é por esta razão que máquinas de Turing são um modelo mais poderoso do que autômatos de pilha. No entanto, quando definirmos máquinas de Turing, não vamos utilizar uma memória de acesso aleatório, mas sim uma fita onde podem ser lidos e escritos símbolos, sendo possível mostrar que este modelo tem o mesmo poder computacional que outros modelos que utilizam memórias de acesso aleatório (por exemplo, é equivalente às *Random Access Machines*, etc. – ver Secção 4.2).

As linguagens aceites por máquinas de Turing são designadas de linguagens recursivamente enumeráveis. Adicionando mais um requerimento, obtemos a classe das linguagens recursivas. Tal como no caso das linguagens regulares e das linguagens livres de contexto, as linguagens recursivamente enumeráveis e as recursivas admitem caracterizações de carácter mais algébrico (através de funções recursivas, do cálculo lambda, etc.), mas que não serão abordadas nesta cadeira.

4.1 Máquinas de Turing

As máquinas de Turing são um modelo computacional introduzido em 1936 por Alan Turing. É um modelo teórico que nos permite concluir resultados realistas sobre computadores, especialmente acerca das suas limitações.

Uma máquina de Turing é constituída por vários elementos. Tem uma fita infinita,

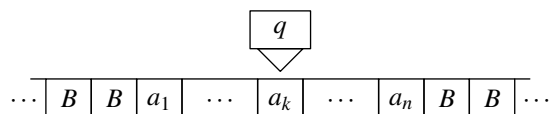


Figura 4.1: Uma máquina de Turing.

dividida em células que contêm símbolos de um alfabeto, e que é utilizada como memória (ver Fig. 4.1). Existe também uma cabeça de leitura que só pode ler uma célula de cada vez e que está ligada a uma estrutura de controlo que se encontra num determinado estado, de entre n possíveis (na figura esse estado é q). Em cada momento a cabeça de leitura lê o símbolo da célula onde está posicionada. Dependendo *apenas desse símbolo e do estado atual*, a máquina faz os seguintes três passos simultaneamente: (i) atualiza o estado atual; (ii) altera (ou não) o conteúdo da célula que está a ler; (iii) mantém a cabeça de leitura no mesmo sítio ou, em alternativa, move-a uma célula para a direita ou uma célula para a esquerda.

Como é que uma máquina de Turing (MT) aceita uma palavra w ? Se $w = a_1 \dots a_n$, então inicialmente os símbolos a_1, \dots, a_n estão escritos de forma consecutiva em células adjacentes, e supomos que as restantes células contêm um símbolo especial, o *branco* (B na figura). A cabeça de leitura é colocada sobre a célula que contém a_1 (i.e. sobre o símbolo mais à esquerda do input) e o estado é o chamado *estado inicial*. A partir deste momento podemos começar a computação, que vai sendo executada passo-a-passo até atingir um estado *final* (caso em que o input w é aceite), ou que não haja nenhuma transição definida (caso em que o input w é rejeitado). Pode acontecer que um input não seja aceite nem rejeitado, i.e. a computação pode continuar indefinidamente.

As MT podem ser definidas graficamente de modo semelhante aos autómatos finitos e aos autómatos de pilha. Um exemplo é dado na Fig. 4.2.

Nesta figura (obtida com recurso ao software JFLAP), a seguinte notação é utilizada: (i) o símbolo \square é o símbolo branco (ou seja $\square = B$); (ii) uma transição tem uma etiqueta do tipo $a; b, M$, onde a é o símbolo atualmente lido pela cabeça de leitura, b é o símbolo a escrever na célula que contém o a , e M é o movimento que a cabeça de leitura vai efetuar (na notação do JFLAP: $L \rightarrow$ left, $R \rightarrow$ right, $S \rightarrow$ stay). Nos diagramas das aulas também vamos utilizar etiquetas do tipo $a \rightarrow b, M$. Por exemplo, na transição que vai de q_0 a q_1 , deve-se ler como: “se no estado q_0 e com a cabeça de leitura a ler o símbolo 0, então substituir esse 0 por um X , mover a cabeça de leitura uma célula para a direita, e mudar o estado para q_1 ”.

O diagrama da Fig. 4.3 (obtida também com recurso ao software JFLAP) mostra os primeiros 5 passos de computação da MT da Fig. 4.2 com input 001100. O que faz esta MT? Ela aceita exatamente a linguagem

$$L = \{0^k 1^k 0^k \in \{0, 1\}^* | k \in \mathbb{N}\},$$

que vimos não ser livre de contexto na Secção 3.3. Portanto as máquinas de Turing

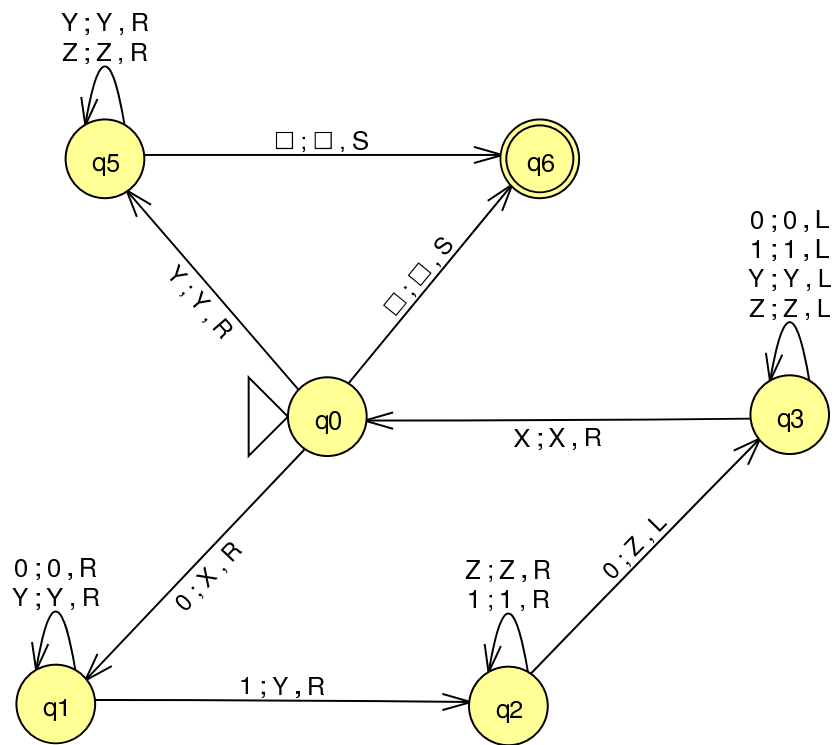


Figura 4.2: Um exemplo de máquina de Turing.

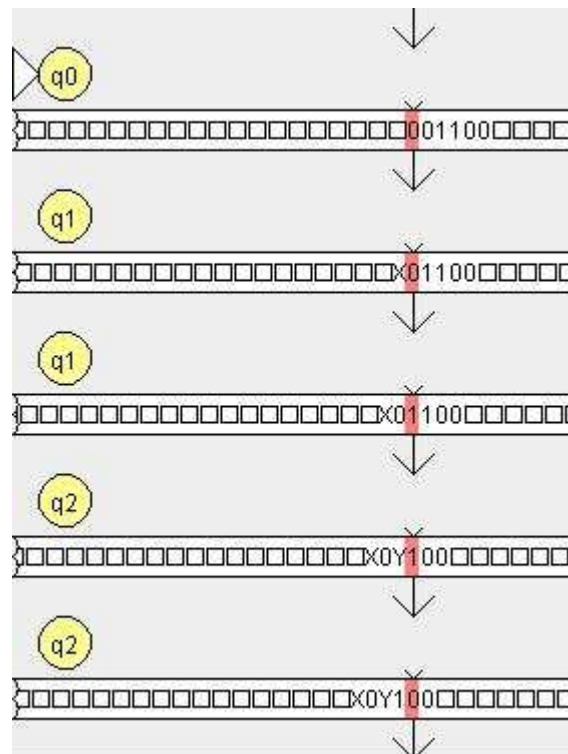


Figura 4.3: Os primeiros 5 passos da computação da máquina de Turing da Fig. 4.2 com input 001100.

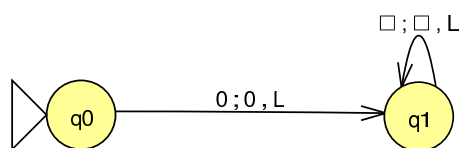


Figura 4.4: Uma máquina de Turing cuja computação não para para o input $w = 00$.

conseguem reconhecer linguagens que são não regulares, nem livres de contexto. Como funciona a MT? Ela aceita imediatamente a palavra vazia ε . Se o input começa por um 1, ele é imediatamente rejeitado. Se começa com um 0, marcamos-lo com um X e procuramos o 1 seguinte à direita. Se não existir, a palavra é rejeitada. Se houver um 1, marcamos-lo com um Y. Agora procuramos um zero à direita desse 1. Se não existir, o input é rejeitado. Se existir, marcamos o 0 com um Z. Agora voltamos para o início da palavra (mais corretamente até ao primeiro X já marcado – não interessa voltar mais para trás do que isto) – e repetimos este processo até esgotar os zeros do primeiro bloco de 0's (é quando transitamos de q_0 para q_5). Aí vamos percorrendo a palavra, garantindo que já não há 0's e 1's, isto é que só há Y's e Z's, até chegar ao símbolo branco. Neste momento sabemos que podemos aceitar o input. Se um input não deve ser aceite, então há sempre um ponto onde já não há é possível efetuar mais transições com este input, pelo que ele é rejeitado.

Note-se que esta MT para sempre, aceitando ou rejeitando o input (diz-se que reconhece a linguagem L), mas pode acontecer que para certos inputs uma MT não pare. Por exemplo, a MT da Fig. 4.4 não para para o input $w = 00$ (a cabeça de leitura desloca-se indefinidamente para a esquerda).

Vamos agora definir formalmente estes conceitos.

Definição 4.1.1. Uma *máquina de Turing* é um 7-tuplo $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ onde:

1. Q é um conjunto finito (de *estados*),
2. Σ é um alfabeto (dos inputs),
3. Γ é um alfabeto (da fita), contendo o símbolo especial B (o símbolo *branco*), e satisfazendo $\Sigma \subseteq \Gamma$,
4. $\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{D, E, P\}$ é a *função de transição*,
5. $q_0 \in Q$ é o *estado inicial*,
6. F é o conjunto dos *estados finais*.

No ponto 4 da definição anterior D, E, P designam movimentos da cabeça de leitura: uma célula para a direita (D), uma célula para a esquerda (E), ou então fica parada (P).

Em cada momento da computação apenas um número finito de células não terá o símbolo B . Portanto, assumindo que a cabeça de leitura está a ler uma certa célula, o conteúdo da fita a partir da primeira célula mais à esquerda que não contém um B até à célula que a cabeça de leitura está a ler, inclusive, pode ser escrito como uma palavra $v \in \Sigma^*$. Da mesma forma, o conteúdo da fita começando a partir da célula lida pela cabeça de leitura, inclusive, até à célula mais à direita que não contém o símbolo B , pode ser codificado noutra palavra $w \in \Sigma^*$. Então se q_i for o estado atual da MT, o triplo (v, q_i, w) determina completamente o estado atual da computação: conhecendo apenas esta informação podemos continuar a computação sem quaisquer problemas. A estes triplos dá-se o nome de *configurações*. Por exemplo, no 3^o passo da computação indicada na Fig. 4.3, a configuração correspondente é dada por $(X0, q_1, 1100)$.

Repare-se que a uma dada configuração segue-se outra configuração e por aí fora, bastando para isso ir seguindo as regras de transição da máquina de Turing, até chegar a uma configuração (u, q, w) em que q é um estado final, a que se chama uma *configuração aceitadora*, ou até chegar a uma configuração a que não se lhe segue nenhuma outra configuração (a computação “morre” neste momento), chamada *configuração rejeitadora*. Uma *configuração de paragem* é uma configuração que é ou aceitadora, ou rejeitadora.

Em particular, no início da computação, a configuração será do tipo (ε, q_0, w) , onde w é o input da MT. Este tipo de configuração é chamado de *configuração inicial*.

Definição 4.1.2. Uma máquina de Turing *aceita* a palavra $w \in \Sigma^*$ se existe uma sequência de configurações C_1, \dots, C_k tal que:

1. C_1 é a configuração inicial de M para a palavra w ,
2. A regra de transição de M determina que a cada configuração C_i se siga a configuração C_{i+1} ,
3. C_k é uma configuração aceitadora.

A *linguagem de M* (ou *linguagem aceite por M*) é a classe

$$L(M) = \{w \in \Sigma^* \mid M \text{ aceita } w\}.$$

Definição 4.1.3. Uma linguagem diz-se *recursivamente enumerável* (r.e.) se é a linguagem de alguma máquina de Turing.

Por exemplo, as linguagens das MT das Fig. 4.2 e 4.4 são, respetivamente, $\{0^k 1^k 0^k \in \{0, 1\}^* \mid k \in \mathbb{N}\}$ e \emptyset .

Repare-se que, no caso anterior, quando um input não é aceite podem acontecer duas coisas: (i) o input é rejeitado; (ii) a MT nunca chega a parar. Como normalmente nos dá jeito que a MT pare ao fim de algum tempo, introduzimos a seguinte definição.

Definição 4.1.4. Uma linguagem L diz-se *recursiva* se é a linguagem de alguma máquina de Turing M , com a propriedade adicional que para qualquer $w \in \Sigma^*$, a computação de M com input w tem sempre de acabar numa configuração de paragem. Neste caso dizemos também que M *decide* L , ou ainda que M *reconhece* L .

Por exemplo, a MT da Fig. 4.2 decide a linguagem $\{0^k 1^k 0^k \in \{0, 1\}^* | k \in \mathbb{N}\}$ da Equação (3.2) mas, apesar de \emptyset ser a linguagem da MT da Fig. 4.4, essa MT não decide \emptyset , porque há inputs (por exemplo, o input 0) que nunca chegam a uma configuração de paragem (por outras palavras, utilizando tempo finito e apenas a computação da MT, não conseguimos dizer se o input pertence ou não à linguagem).

Note-se que toda a linguagem recursiva é obviamente r.e., mas como teremos ocasião de ver mais à frente, o resultado recíproco não é verdadeiro. O seguinte teorema permite relacionar linguagens recursivas e recursivamente enumeráveis com as classes de linguagens introduzidas nos capítulos anteriores.

Teorema 4.1.5. *Toda a linguagem livre de contexto é também recursiva.*

Demonstração. Se L é uma linguagem livre de contexto, então está associada a uma gramática livre de contexto G , com variável inicial S . A ideia é obter todas as possíveis derivações a partir de S . No entanto, a demonstração formal não será dada nas aulas (há o problema de alguns ramos de computação poderem levar tempo infinito, e isto implica que a demonstração seja um pouco mais complicada do que parece à primeira vista), mas pode ser encontrada em [Sip05]. \square

Note-se que a classe das linguagens livres de contexto está estritamente incluída na classe das linguagens recursivas, já que a linguagem da Equação (3.2) é recursiva, mas não é livre de contexto.

Às vezes teremos necessidade de computar uma função $f : \Sigma^* \rightarrow \Sigma^*$ com uma MT. Por outras palavras, dada uma palavra inicial $w \in \Sigma^*$, queremos calcular a partir desse input uma nova palavra $f(w) \in \Sigma^*$. Por exemplo, tomemos $f : \{0\}^* \rightarrow \{0\}^*$ como sendo uma função que dada uma palavra $0^k \in \{0\}^*$ retorna $f(0^k) = 0^{2k}$, i.e. f duplica o tamanho do input. Esta função é computada pela MT da Fig. 4.5, se utilizarmos a noção de computação indicada abaixo.

Definição 4.1.6. Uma função $f : \Sigma^* \rightarrow \Sigma^*$ é computada por uma máquina de Turing M se, para cada input $w \in \Sigma^*$, o seguinte acontece:

1. Se $f(w)$ está definido, então inicialmente a cabeça de leitura está a ler o símbolo mais à esquerda do input. Para além do input, a fita só tem brancos. Depois a computação é iniciada, e continua até chegar a uma configuração aceitadora. Neste instante a fita conterá somente o resultado $f(w)$ (o resto da fita só terá símbolos brancos), estando a cabeça de leitura a ler o símbolo mais à esquerda de $f(w)$.

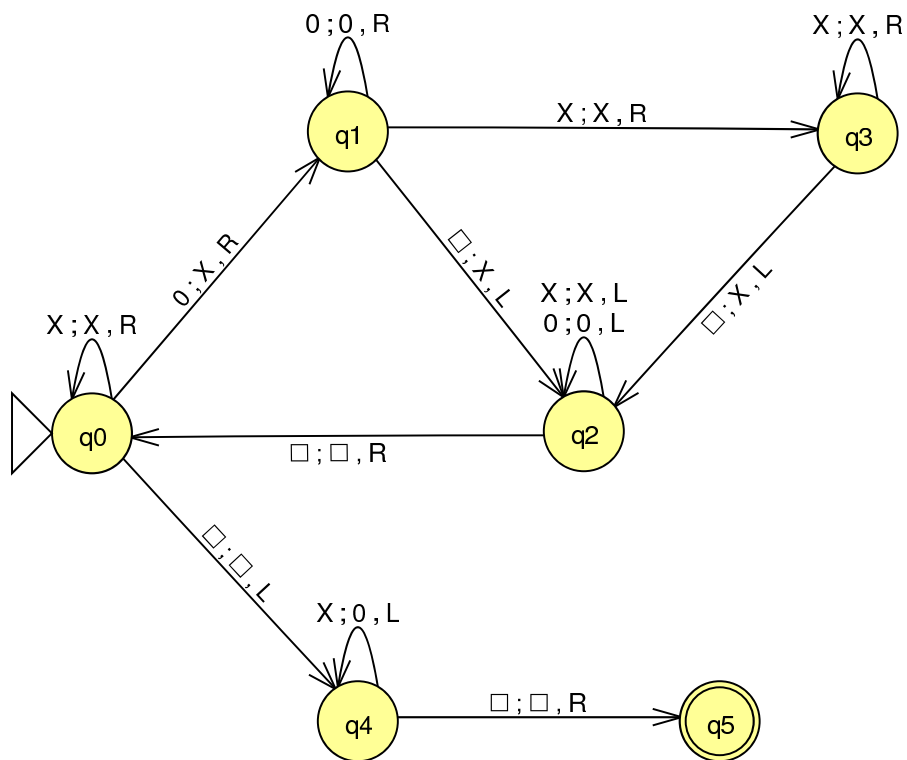


Figura 4.5: Máquina de Turing que duplica o tamanho do input, quando este é constituído só por 0's.

2. Se $f(w)$ não está definido, então a computação com input w nunca chegará a uma configuração aceitadora.

Note-se que a partir do momento em que computamos funções sobre Σ^* , podemos computar funções sobre outros domínios. Por exemplo, consideremos a função $f: \mathbb{N} \rightarrow \mathbb{N}$ tal que $g(k) = 2k$. Se tomarmos como alfabeto $\Sigma = \{0\}$, então podemos representar o número k por 0^k em Σ^* . Por exemplo, podemos representar o número 3 pela palavra $000 \in \Sigma^*$. Utilizando esta representação dos números naturais (representação unária: utilizamos um alfabeto com apenas um símbolo), concluímos que a MT da Fig. 4.5 computa g . Poderíamos também ter utilizado a representação binária, por exemplo, tomando $\Sigma = \{0, 1\}$. Aí o número 3 seria representado por 11, e teríamos de ter uma MT que desse o resultado 110 para calcular a função g , nesta representação.

Também é possível calcular funções com vários argumentos: basta codificá-los todos na mesma palavra, utilizando um símbolo especial para separar diferentes argumentos. Por exemplo, se queremos calcular a função soma $S: \mathbb{N}^2 \rightarrow \mathbb{N}$ dada por $S(x, y) = x + y$, podíamos representar cada argumento em binário, separando-os pelo símbolo #, retornando o resultado em binário. Assim, uma MT que calculasse a função S com esta representação, com o input 11#101, deveria retornar o valor 1000 (pois $3+5=8$).

4.2 Tese de Church-Turing

Uma pergunta que um engenheiro informático se pode colocar é a seguinte:

Dado um determinado problema, será que existe um algoritmo (programa) que o resolve?

A história, aliás, começa em 1900. Por ocasião da mudança do novo século, um dos mais famosos matemáticos da altura, David Hilbert, apresentou no Congresso Internacional da Matemática, em Paris, uma lista de 23 problemas matemáticos em aberto que ele julgava serem problemas fundamentais. O décimo problema consistia em saber se existia um algoritmo para resolver determinada questão. Na altura pensava-se que não existia nenhum algoritmo nessas condições, mas a grande questão era como provar essa conjectura.

Se pensarmos um pouco no assunto, verificamos que o principal problema é que não temos uma definição precisa para a noção de algoritmo. Esta noção é sobretudo intuitiva: sabemos identificar um quando o vemos, mas não sabemos defini-lo exatamente. É como tentar identificar um(a) mulher/homem bonita(o).

O que vários investigadores fizeram, sobretudo durante a década de 1930, foi formalizar o conceito de algoritmo. Vários cientistas levaram a cabo esta tarefa, obtendo modelos mais ou menos abstratos, que não convenceram toda a comunidade científica.

Mas, em 1936, com a sua máquina tão simples e convincente, Turing conseguiu impor o seu modelo.

De certo modo, o que Turing tentou capturar foi a essência do que é um ser humano executando um algoritmo (na altura não haviam computadores). Um ser humano apenas precisa de papel e lápis para executar um algoritmo. A fita modela o papel. Os símbolos modelam o que é escrito no papel. Então um ser humano executando um algoritmo estará em certo momento num “estado mental” (por exemplo, “e vai mais um”), e dependendo desse estado mental, e da porção que está atualmente a ler do papel, poderá querer alterar o que lá escreveu, ou mudar a sua atenção para o texto que está à direita, ou à esquerda.

Para além da sua simplicidade e naturalidade, uma das razões que possibilitou a aceitação generalizada da máquina de Turing como o modelo de computação que captura a essência de algoritmo, foram diversos resultados que mostraram que todos os outros modelos apresentados anteriormente eram equivalentes à máquina de Turing. Isso está refletido na seguinte afirmação, que hoje é consensualmente aceite na comunidade científica:

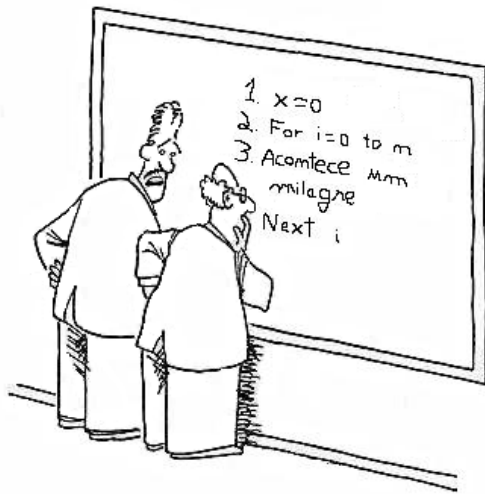
Tese de Church-Turing: *Um problema pode ser resolvido através de um algoritmo se e só se pode ser resolvido através de uma máquina de Turing.*

Note-se que esta afirmação não se pode provar, daí o nome de “tese” porque, como já vimos, a noção de algoritmo é informal. O que esta tese vem fazer é dar um sentido preciso à noção de algoritmo. Nesta cadeira iremos sempre assumir que a tese de Church-Turing é válida.

Por esta razão, a não ser que seja pedida uma descrição formal de uma MT (caso em que se deve utilizar um diagrama ou a Definição 4.1.1), iremos muitas vezes descrever uma MT através de um algoritmo que especifique os seus passos, sem entrar em detalhes de estados, etc. No entanto, a descrição deve ser suficientemente precisa para não deixar dúvidas a quem estiver a ler a descrição. Não vale dizer “Com o input, fazer os cálculos, e retornar o resultado”! (ver Fig. 4.6)

Por exemplo, para a MT da Fig. 4.5, podíamos dar o seguinte algoritmo:

1. Se o símbolo lido for B , aceitar o input sem mover a cabeça ou alterar o símbolo (o input é ε e o output será então também ε).
2. Se o símbolo lido for um 0 , marcá-lo com um X e mover a cabeça para a direita até encontrar um B .
3. Substituir o B por um X e mover a cabeça para a esquerda até encontrar um B .
4. Mover a cabeça para a direita até encontrar um 0 ou um B . Se encontrar um 0 , ir para o passo 2.
5. Mover a cabeça para a esquerda, substituindo os X 's por 0 's, até encontrar um B .



"Hum... Tenho algumas dúvidas aqui no passo 3..."

Figura 4.6: Exemplo de como **não** especificar um algoritmo.

6. Mover a cabeça uma célula para a direita e aceitar.

A tese de Church-Turing também implica que o poder computacional de uma MT com várias fitas, ou com fitas n -dimensionais, ou que possa ler a fita de forma não sequencial, etc., é igual ao do modelo apresentado na Definição 4.1.1.

Às vezes, por ser mais prático, vamos utilizar MT com 2 ou mais fitas. O esquema geral de uma MT com duas fitas é dado na Fig. 4.7, e um exemplo concreto é dado na Fig. 4.8. O que se passa em cada fita é separado pelo símbolo '|'. Por exemplo, na Fig. 4.7, a transição do estado q_0 para ele próprio deve-se ler: "se no estado q_0 , se a cabeça de leitura da 1ª fita está a ler um 0 e se a cabeça de leitura da 2ª fita está a ler um B,

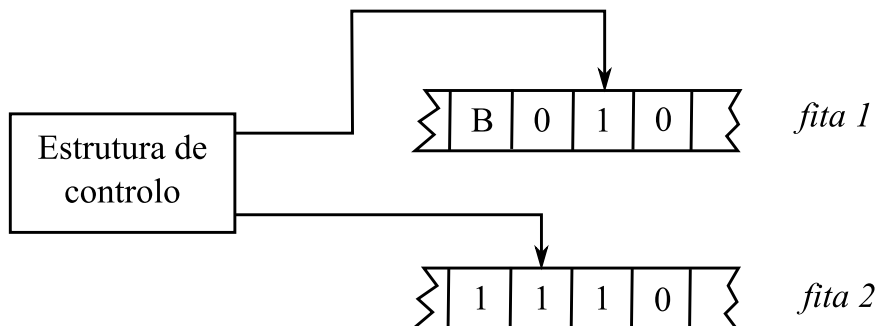


Figura 4.7: Esquema de uma máquina de Turing com duas fitas.

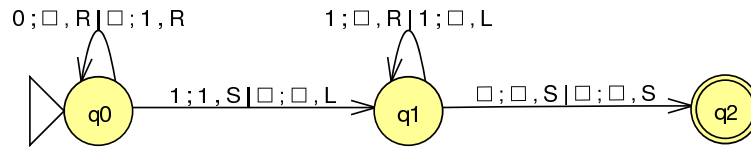


Figura 4.8: Uma máquina de Turing com duas fitas.

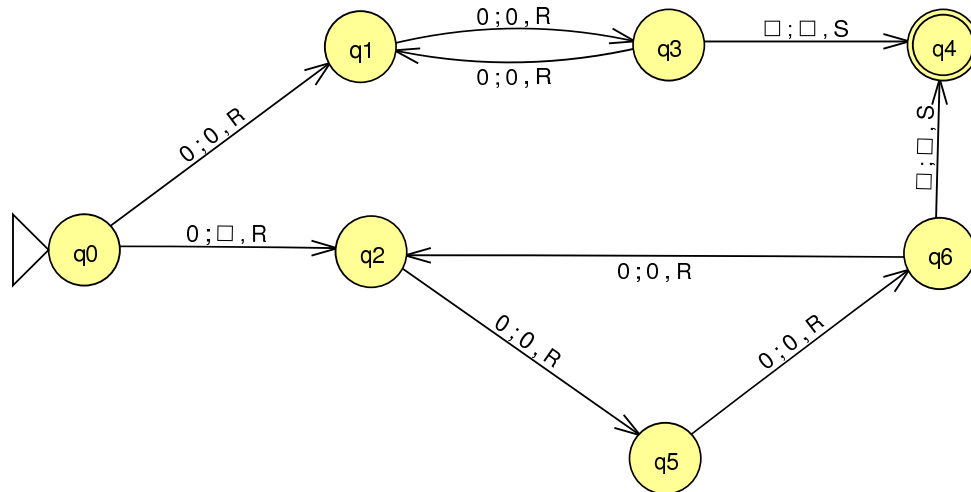


Figura 4.9: Uma máquina de Turing não-determinística.

então: manter-se no estado q_0 , substituir o 0 por um B na 1ª fita, substituir o B por um 1 na 2ª fita, mover as cabeças de leitura da 1ª e 2ª fita uma célula para a direita. Esta MT reconhece a linguagem $\{0^k 1^k \in \{0, 1\}^* | k \geq 1\}$ da seguinte forma: por cada 0 do input na 1ª fita, escreve um 1 na 2ª fita. Assim vão haver tantos 1's na 2ª fita como 0's no início do input. Quando finalmente aparecer 1's na 1ª fita, vamos ver se existem tantos quantos os da segunda fita, rejeitando o input se isso não acontecer ou aparecer 0's pelo meio.

Iremos agora analisar outra variante da MT que muitas vezes é útil. Basicamente permite não-determinismo: em cada passo da computação, pode-se ter vários resultados possíveis. A definição é semelhante à dos autómatos finitos não-determinísticos: pode haver várias transições com o mesmo símbolo. Uma diferença é que não vamos permitir transições com a palavra vazia. Um exemplo de MT não-determinística (MTND) é dado na Fig. 4.9 (note-se que do estado q_0 há 2 transições possíveis quando se está a ler o símbolo 0 com a cabeça de leitura). Esta máquina reconhece a linguagem $L = \{0^{2k} \in \{0\}^* | k \geq 1\} \cup \{0^{3k} \in \{0\}^* | k \geq 1\}$. O ramo de cima aceita as palavras de $\{0^{2k} \in \{0\}^* | k \geq 1\}$, enquanto que o ramo de baixo aceita as palavras de $\{0^{3k} \in \{0\}^* | k \geq 1\}$. Por exemplo, $0 \notin L, 00000 \notin L$, mas $00 \in L, 000 \in L$.

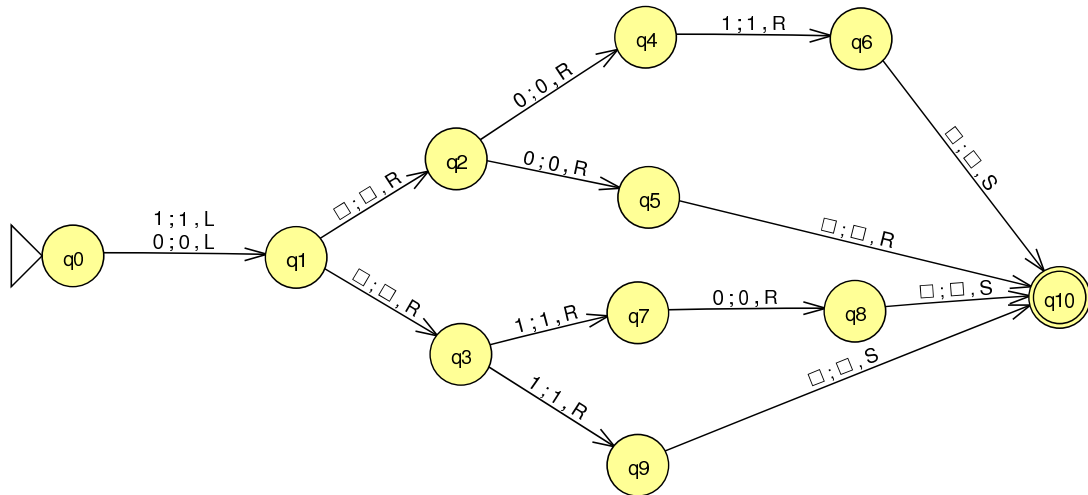


Figura 4.10: Uma máquina de Turing não-determinística.

Outro exemplo de MTND é dado na Fig. 4.10. Tem 3 estados onde há duas transições possíveis com o mesmo símbolo (estados q_1, q_2, q_3). Este AFND reconhece a linguagem $\{0, 1, 01, 10\}$. Formalmente, uma MTND é definida da seguinte forma.

Definição 4.2.1. Uma máquina de Turing não-determinística é um 7-tuplo $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$ definido como no caso da máquina de Turing, com a exceção que $\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q \times \Sigma \times \{D, E, P\})$. Repare-se que agora a uma configuração podem-se seguir várias configurações. Uma palavra $w \in \Sigma^*$ é aceite se existe uma sequência de configurações C_1, \dots, C_k , tal que C_1 é a configuração inicial para w , C_{i+1} é uma das possíveis configurações que se segue a C_i , e C_k é uma configuração aceitadora.

Obviamente que toda a MT é também não-determinística. A relação recíproca deveria ser verdadeira pela tese de Church-Turing, e é isso que o resultado seguinte mostra.

Teorema 4.2.2. Se $L(M)$ é a linguagem aceite (reconhecida) por uma máquina de Turing não-determinística M , então existe uma máquina de Turing que também aceita (reconhece) $L(M)$.

Demonstração. Por conveniência iremos utilizar uma MT com 3 fitas, que já sabemos ser equivalente a uma MT “standard” com apenas uma fita. A ideia é manter uma cópia do input na 1ª fita, utilizar a fita 2 para simular uma (de entre as várias possibilidades de caminhos de computação) computação de M , e a fita 3 é utilizada para manter um registo de todos os caminhos de computação explorados até agora. Inicialmente copia-se o input da fita 1 para a fita 2, e testa-se um caminho de computação na fita 2. Se esse caminho aceita, aceitamos o input, caso contrário, reiniciamos a computação com um novo caminho de computação e vamos repetindo o processo até que não reste mais nenhum caminho de computação por explorar.

Vamos dar os detalhes para o caso em que uma linguagem L é reconhecida por uma MTND M .

Dada a função de transição de M , δ , terá de existir um número k tal que, para cada $(q, s) \in Q \times \Sigma$, existem no máximo k escolhas para a próxima ação a realizar. Portanto em cada transição não-determinística, podemos registar um número em $\{1, \dots, k\}$ para saber qual a ação que foi tomada. O que a nossa MT vai fazer é registar esses números na 3ª fita e testar todas as possibilidades em $\{1, \dots, k\}$ i.e. vai testar todas as possíveis transições, uma de cada vez, de forma a simular todas as possíveis computações de M (que acabam sempre e são em número finito, pois estamos a reconhecer uma linguagem). Vamos agora descrever a MT procurada:

1. Inicialmente a fita 1 contém o input w e as fitas 2 e 3 estão vazias.
2. Copiar o conteúdo da fita 1 para a fita 2.
3. Utilizar a fita dois para simular a máquina M com input w . Cada vez que temos de efetuar uma nova transição não-determinística, lemos o conteúdo atual da fita 3 e fazemos a transição de acordo com o símbolo de $\{1, \dots, k\}$ que estiver lá escrito. Se o símbolo lido for um B , escrevemos um 1 por cima e fazemos a transição respetiva. Em ambos os casos, deslocamos a cabeça de leitura da fita 3 uma célula para a direita. Se chegarmos a uma configuração aceitadora, aceitar o input. Se chegarmos a uma configuração rejeitadora, ir para o passo 4.
4. Se a fita 3 está vazia, rejeitar o input. Se a fita 3 tem a palavra $s = s_1 \dots s_j \neq \varepsilon$, escrita no alfabeto $\{1, \dots, k\}$, substituir s_j pelo símbolo seguinte da ordenação dos símbolos, metendo a cabeça de leitura no símbolo s_1 do s atualizado. Se não houver símbolo seguinte, apagar s_j e repetir o passo 4.

Para o caso das linguagens aceites por M , a situação é mais complicada, pois pode haver ramos em que a computação nunca acabe, o que nos pode impedir de chegar a um ramo cuja computação aceite. Mas isso pode ser conseguido utilizando mais uma 4ª fita que mantém um registo de um contador de passos. Sempre que a computação ultrapassar o valor definido pelo contador, acabamos com essa computação (como se ela rejeitasse), e passamos para a computação seguinte. O algoritmo é o seguinte:

1. Inicialmente a fita 1 contém o input w e as fitas 2 e 3 estão vazias. A fita 4 contém o valor $i = 1$
2. While $i > 0$ do
 - (a) Copiar o conteúdo da fita 1 para a fita 2.
 - (b) Utilizar a fita dois para simular a máquina M com input w . Cada vez que temos de efetuar uma nova transição não-determinística, lemos o conteúdo atual da fita 3 e fazemos a transição de acordo com o símbolo de $\{1, \dots, k\}$

que estiver lá escrito. Se o símbolo lido for um B , escrevemos um 1 por cima e fazemos a transição respetiva. Em ambos os casos, deslocamos a cabeça de leitura da fita 3 uma célula para a direita. Se chegarmos a uma configuração aceitadora, aceitar o input. Se chegarmos a uma configuração rejeitadora, ou a computação necessitar mais do que i passos, ir para o passo d).

- (c) Se a fita 3 está vazia, ir para o passo d). Se a fita 3 tem a palavra $s = s_1 \dots s_j \neq \varepsilon$, escrita no alfabeto $\{1, \dots, k\}$, substituir s_j pelo símbolo seguinte da ordenação dos símbolos, metendo a cabeça de leitura no símbolo s_1 do s atualizado. Se não houver símbolo seguinte, apagar s_j e repetir o passo 4.
- (d) Incrementar o valor de i .

□

Vamos dar um exemplo de como esta demonstração funciona na prática (caso em que se reconhece uma linguagem). Consideremos a MT da Fig. 4.10. Temos transições não-determinísticas partindo de q_1, q_2, q_3 , que usam duas escolhas possíveis. Codifiquemos essas escolhas nos números binários 1 e 2 da seguinte forma:

$$\delta(q_1, B) = \begin{cases} (q_2, B, D) & \text{se o código é 1} \\ (q_3, B, D) & \text{se o código é 2} \end{cases} \quad \delta(q_2, B) = \begin{cases} (q_4, 0, D) & \text{se o código é 1} \\ (q_5, 0, D) & \text{se o código é 2} \end{cases}$$

$$\delta(q_3, 1) = \begin{cases} (q_7, 1, D) & \text{se o código é 1} \\ (q_9, 1, D) & \text{se o código é 2.} \end{cases}$$

Consideremos o input 00. Começamos no estado q_0 e depois vamos para o estado q_1 . Daqui vamos querer fazer nova transição, mas temos duas possibilidades. Vamos ter de registar essas possibilidades na fita 3. A fita 3 está inicialmente vazia, pelo que escrevemos um 1, e depois deslocamos a cabeça desta fita uma célula para a direita, indo para o estado q_2 . Temos novamente uma transição não-determinística no estado q_2 , pois estamos a ler um B. Escrevemos novamente um 1 na fita 3 e continuamos com a computação (a fita 3 tem o conteúdo “11”), chegando ao estado q_4 que rejeita esta computação. Agora vamos experimentar outro caminho de computação. Para isso, vamos para o passo 4 da simulação, alterando o conteúdo da fita 3 para “12”, e metendo a cabeça de leitura sobre o 1. Repetimos a computação com o input 00: começamos em q_0 , vamos para q_1 , e como estamos a ler um 1 na fita 3, vamos para o estado q_2 , movendo a cabeça desta fita uma célula para a direita. Depois temos nova transição não-determinística. Como estamos a ler um 2 na 2ª fita, vamos agora para o estado q_5 que também rejeita o input. Fazemos novamente o passo 4 (experimentar outro caminho de computação). Como já não há mais símbolos para além do 2 em $\{1, 2\}$, devemos apagar o 2 de “12” (o conteúdo da fita 3 passa a ser “1”) e aplicamos novamente o passo 4, ficando o conteúdo da fita 3 “2”. Repete-se a computação, passando pelos estados q_0, q_1, q_3 e a computação morre nesse momento. Repete-se novamente o passo 4, apagando

o “2”. Neste momento a 3ª fita fica vazia (já não há novas possibilidades de caminhos de computação), e o input é rejeitado.

Iremos frequentemente codificar objetos matemáticos em palavras de um dado alfabeto. Por exemplo, consideremos o grafo (não-orientado) G da Fig. 4.11 e o alfabeto $\Sigma = \{0, 1\}$. Então o grafo pode ser codificado na seguinte palavra de Σ^* :

$$\langle G \rangle = \underbrace{0000}_{\text{vértices}} \underbrace{11}_{1^{\text{a}} \text{ aresta}} \underbrace{0100}_{2^{\text{a}} \text{ aresta}} \underbrace{11010000}_{3^{\text{a}} \text{ aresta}} \underbrace{11001000}_{3^{\text{a}} \text{ aresta}}.$$

O primeiro bloco de zeros dá-nos o número de vértices do grafo (ou seja, 4). Depois escrevemos 11 para dizer que vem uma aresta. Como de seguida vem um 0, separado por um 1 de dois 0's, isso quer dizer que o vértice 1 está ligado ao vértice 2 por uma aresta. De seguida vêm mais dois 1's para assinalar outra aresta: a que vai do vértice 1 ao vértice 4. Finalmente vem a aresta que vai do vértice 2 ao vértice 3. Da mesma forma podemos codificar qualquer grafo G numa palavra de $\{0, 1\}^*$. Note-se (será importante para a próxima secção), que se G tem k vértices, então a descrição de cada aresta utiliza, no máximo, $2k + 3$ símbolos (dois 1's para marcar o início da aresta, $i \leq k$ zeros para marcar o primeiro vértice da aresta, mais um zero, e $j \leq k$ zeros para marcar o segundo vértice). Como não há mais do que k^2 arestas no grafo G , no máximo a descrição do grafo necessitará de $k^2(2k + 3) + k$ símbolos de $\{0, 1\}$, i.e. a descrição do grafo tem um tamanho que é polinomial no número de vértices.

Outra forma de descrever G no alfabeto $\Sigma = \{0, 1\}$ seria representá-lo numa forma mais matemática, por exemplo $G = (1, 2, 3, 4, (1, 2), (1, 4), (2, 3))$, e depois codificar cada símbolo de G em binário para escrever G em binário segundo essa codificação. Por exemplo, precisamos dos símbolos 1,2,3,4 para codificar os vértices de um grafo (mais em geral, precisaríamos dos símbolos 0,1,2,...,9, mas podemos sempre supor que o vértice é escrito em numeração de base 4), e ainda dos caracteres “(”, “,” e “)”. Necessitamos de codificar 7 símbolos em binário, e para isso precisamos de 3 bits por símbolo. Portanto, fazendo as associações

- 1 → 000
- 2 → 001
- 3 → 010
- 4 → 011
- (→ 100
- , → 101
-) → 110

a codificação do grafo $G = (1, 2, 3, 4, (1, 2), (1, 4), (2, 3))$ neste esquema seria

$$\langle G \rangle = \underbrace{100}_{(} \underbrace{000}_1 \underbrace{101}_{,} \underbrace{001}_2 \dots \underbrace{010}_3 \underbrace{110}_{)} \underbrace{110}_{)}.$$

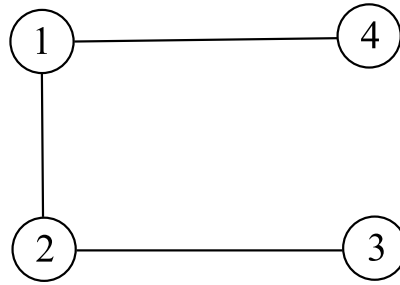


Figura 4.11: Exemplo de grafo.

Como uma MT M pode ser descrita de forma textual utilizando a notação da Definição 4.1.1, podemos também codificá-la de forma semelhante numa palavra $\langle M \rangle$ de $\{0, 1\}^*$. Em geral, qualquer objeto finito (um número, grafo, MT, um texto em português, etc.) pode ser codificado numa palavra de $\{0, 1\}^*$, desde que se utilize um procedimento adequado (i.e. deve ser possível codificar G em $\langle G \rangle$ através de um procedimento algorítmico, e $\langle G \rangle$ deve conter informação suficiente para caracterizar sem ambiguidade G). Da mesma forma podemos codificar uma MT M e o seu input w na palavra $\langle M, w \rangle$.

4.3 O problema da paragem

Dissemos que a MT era a idealização do que é efetuado por um computador. Mas, do que vimos até agora, para cada algoritmo diferente, temos de mudar o hardware da MT (i.e. estados e transições), enquanto que na prática o hardware mantém-se, e apenas se troca de programa. Não haverá aqui um problema? Não porque não é difícil conceber uma MT U que recebe como input $\langle M, w \rangle$, onde $\langle M, w \rangle$ designa a descrição da MT M e da palavra w numa palavra do alfabeto de U . Com base nessa descrição, U simula M com o input w , para quaisquer M, w , de forma semelhante ao que um interpretador de C/C++ ou Java faz. A esta máquina dá-se o nome de *máquina de Turing universal*.

Agora vamos voltar a nossa atenção para um problema prático que pode aparecer. Normalmente os ambientes de programação trazem embutidos ferramentas que permitem detetar facilmente erros de sintaxe, como por exemplo um comando “for” que foi erroneamente escrito como “fro”. Mas os erros que dão mais dores de cabeça aos programadores são os erros de semântica: o programa funciona e é sintacticamente correto, mas não faz o que é pretendido. Um caso particular é quando o programa “encrava” porque entrou numa computação infinita.

Uma pergunta que é natural fazer é porque é que ninguém desenvolveu um software que permita dizer antecipadamente se dado um programa em C/C++, por exemplo, se este vai ou não entrar num ciclo infinito de computação? A razão é simples e é uma consequência da teoria da computação: tal software não existe!

Como mostrar isso? O problema que queremos tratar, traduzido na linguagem de

teoria da computação, será: dada a linguagem

$$H_{paragem} = \{\langle M, w \rangle \mid \text{a máquina de Turing } M \text{ para para o input } w\} \quad (4.1)$$

será que ela é recursiva? Se sim o software mencionado acima existe, caso contrário o software não existe. Apesar desta linguagem ser recursivamente enumerável, ela não é recursiva, como mostra o seguinte teorema.

Teorema 4.3.1 (O problema da paragem é indecidível). *A linguagem definida por (4.1) é recursivamente enumerável, mas não é recursiva.*

Demonstração. Primeiro vamos mostrar que a linguagem $H_{paragem}$ é recursivamente enumerável. A MT U descrita pelo seguinte algoritmo aceita $H_{paragem}$:

- Com input $\langle M, w \rangle$:
 1. Simular M com input w .
 2. Se M para, então aceitar o input $\langle M, w \rangle$.

Note-se que se M para com input w , fa-lo-á em tempo finito, e logo a MT U vai aceitar $\langle M, w \rangle$ em tempo finito. Se M não para com input w , então U não para com input $\langle M, w \rangle$. Logo U aceita $H_{paragem}$, e portanto $H_{paragem}$ é recursivamente enumerável. Vamos mostrar que $H_{paragem}$ não é recursiva, utilizando uma técnica conhecida como *diagonalização*.

Por absurdo, suponhamos que a linguagem (4.1) é recursiva. Então há de existir uma MT N tal que

$$N(\langle M, w \rangle) = \begin{cases} \text{aceita} & \text{se } \langle M, w \rangle \in H_{paragem} \\ \text{rejeita} & \text{se } \langle M, w \rangle \notin H_{paragem}. \end{cases}$$

Por outras palavras,

$$N(\langle M, w \rangle) = \begin{cases} \text{aceita} & \text{se } M \text{ para com input } w \\ \text{rejeita} & \text{se } M \text{ não para com input } w. \end{cases}$$

Agora construímos outra MT P , que utiliza N como subrotina da seguinte forma:

$$P(\langle M \rangle) = \begin{cases} \text{aceita } \langle M \rangle & \text{se } N \text{ rejeita } \langle M, \langle M \rangle \rangle \\ \text{entra num ciclo infinito} & \text{se } N \text{ aceita } \langle M, \langle M \rangle \rangle \end{cases}$$

o que também pode ser lido como

$$P(\langle M \rangle) = \begin{cases} \text{aceita } \langle M \rangle & \text{se } M \text{ não para com input } \langle M \rangle \\ \text{entra num ciclo infinito} & \text{se } M \text{ para com input } \langle M \rangle. \end{cases}$$

Então temos

$$P(\langle P \rangle) = \begin{cases} \text{aceita } \langle P \rangle & \text{se } P \text{ não para com input } \langle P \rangle \\ \text{entra num ciclo infinito} & \text{se } P \text{ para com input } \langle P \rangle \end{cases}$$

o que é absurdo. Portanto a linguagem (4.1) não pode ser recursiva. □

Uma vez mostrada que esta linguagem é indecidível, é possível mostrar que muitas outras também o são, através de uma técnica chamada *reducibilidade*, que no entanto não teremos tempo de abordar (ver mais detalhes em [Sip05], [HMU06]).

Para finalizar este capítulo, e a título de curiosidade, enunciamos o teorema que mostra a existência de vírus informáticos de que falamos na Secção 1.1. Uma demonstração construtiva pode ser encontrada em [Sip05], cujo conteúdo fornece pistas suficientes para criar programas que se autorreplicam, embora chame desde já a atenção que este conhecimento é dado apenas a título pedagógico e não deve ser utilizado em aplicações que possam prejudicar outras pessoas ou com carácter criminoso.

Teorema 4.3.2 (do vírus). *Existe uma máquina de Turing M que, em qualquer input w , termina sempre a computação e no final tem na sua fita a sua própria descrição $\langle M \rangle$.*

Capítulo 5

Complexidade computacional

5.1 Introdução

No capítulo anterior vimos que há problemas, como o problema da paragem, que não podem ser resolvidos (decididos) por meio de algoritmos. Então podemos classificar os problemas computacionais como sendo decidíveis ou indecidíveis. Um problema é decidível se pode ser resolvido por um algoritmo. No entanto, a experiência diz-nos que a separação decidível versus indecidível não é suficientemente fina para capturar a classe dos problemas que se conseguem resolver na prática.

É que embora certos problemas sejam teoricamente resolúveis através de algoritmos, não o são na prática porque o algoritmo que os resolve poderá demorar biliões de anos a correr. Por isso temos necessidade de dividir a classe dos problemas decidíveis em subclasses, de acordo com a quantidade de recursos utilizados, que pode ser tempo de computação, memória utilizada, ou outros. Neste capítulo vamos apenas cingir-nos a limitações no tempo de execução.

Em geral, medir o tempo de execução de um algoritmo pode dar origem a uma expressão complexa. De forma a não nos perdermos nos detalhes, estaremos apenas interessados na ordem de grandeza dos tempos utilizados. Por exemplo, se um algoritmo pode ser executado em $\leq 5n^2 + 7n + 2$ passos para um input de tamanho n (é natural que a computação demore mais tempo para inputs maiores, não significando isso que o algoritmo seja menos eficiente), diremos simplesmente que o algoritmo demora tempo da ordem de n^2 , pois o grosso do incremento do tempo de computação vem da potência n^2 , dizendo-se que o algoritmo necessita de tempo $O(n^2)$. Vamos agora introduzir formalmente estas noções.

Definição 5.1.1. Sejam $f, g : \mathbb{N} \rightarrow \mathbb{N}$ duas funções. Dizemos que g é um *limite superior* (*assimptótico*) para f , e escreve-se $f \in O(g)$, se existem inteiros c, n_0 tais que para todo o inteiro $n \geq n_0$ se tem

$$f(n) \leq cg(n).$$

Por exemplo, $n \in O(n)$, $n^2 \in O(n^4)$, $5n \in O(n)$, mas $n^2 \notin O(n)$. Vamos agora introduzir uma definição que será bastante importante no que se segue.

Definição 5.1.2. Seja M uma máquina de Turing que para em todos os inputs. O tempo de execução de M é uma função $f : \mathbb{N} \rightarrow \mathbb{N}$, onde $f(n)$ é o maior número de passos que M utiliza para concluir uma computação com um input de tamanho n . Definimos também a seguinte classe

$$TIME(f) = \{L \mid L \text{ é uma linguagem decidida em tempo } O(f) \text{ por alguma MT}\}.$$

Por exemplo, a MT da Fig. 4.2 tem tempo de execução $O(n^2)$. De facto, dado um input $w = 0^k 1^k 0^k$ com comprimento $n (= 3k)$, a MT marca um zero do 1º bloco de zeros, e depois vai para à direita até ao final do input, marcando um 1 e um 0 do último bloco de 0's no caminho. Depois volta para trás até ao 1º bloco de zeros. Esta passagem demora tempo $\leq 2n$, ou seja demora tempo $O(n)$. No máximo vão haver $k \leq n$ passagens destas (uma por cada 0 do primeiro bloco de 0's), pelo que o tempo total de computação é limitado por $n \times O(n) = O(n^2)$. Se o input não é do tipo $0^k 1^k 0^k$, i.e. se o input é rejeitado, a computação acaba prematuramente, pelo que o tempo de execução continua a ser $O(n^2)$. Como esta máquina decide a linguagem $L_1 = \{0^k 1^k 0^k \in \{0, 1\}^* \mid k \in \mathbb{N}\}$, concluímos que $L_1 \in TIME(n^2)$.

De forma semelhante, concluímos que a MT da Fig. 4.8 decide a linguagem $L_2 = \{0^k 1^k \in \{0, 1\}^* \mid k \geq 1\}$ em tempo $O(n)$, e que $L_2 \in TIME(n)$.

Da mesma forma que definimos o tempo de execução para uma MT, podemos também fazê-lo para uma MTND.

Definição 5.1.3. Seja M uma máquina de Turing não-determinística que para em todos os ramos de computação para qualquer input. O tempo de execução de M é uma função $f : \mathbb{N} \rightarrow \mathbb{N}$, onde $f(n)$ é o maior número de passos que M utiliza para concluir um ramo de computação para um input de tamanho n . Definimos também a seguinte classe:

$$NTIME(f) = \{L \mid L \text{ é uma linguagem decidida em tempo } O(f) \text{ por alguma MTND}\}.$$

Por exemplo, a MTND da Fig. 4.9 tem tempo de execução $O(n)$ já que, com input $w \in \{0\}^*$ em que $|w| = n$, cada ramo de computação necessita de tempo $O(n)$ para ser concluído. Logo $L_3 = \{0^{2k} \in \{0\}^* \mid k \geq 1\} \cup \{0^{3k} \in \{0\}^* \mid k \geq 1\} \in NTIME(n)$. O tempo de execução da MTND da Fig. 4.10 é $O(1)$, já que todo o input pode ser decidido em tempo constante (no máximo em 5 passos).

Teorema 5.1.4. Seja $f : \mathbb{N} \rightarrow \mathbb{N}$ uma função satisfazendo $f(n) \geq n$ para todo o $n \in \mathbb{N}$. Então $NTIME(f) \subseteq TIME(2^{O(f)})$.

Demonstração. Basta seguir a demonstração do Teorema 4.2.2. Tomemos um input de comprimento n . Como as palavras da 3ª fita que registam as possibilidades de movimentos não-determinísticos não podem ter um comprimento superior ao número de passos de computação, estas palavras podem ir de

$$1 \text{ a } \underbrace{kk \dots k}_{O(f(n)) \text{ vezes}}$$

Ou seja, temos $k^{O(f(n))} = 2^{O(f(n))}$ possíveis simulações a efetuar, cada uma levando tempo $O(f(n))$. O número total de simulações determinísticas será então $O(f(n)) \times 2^{O(f(n))}$. Mas como $f(n) \geq n$, isto é tempo $2^{O(f(n))}$. \square

Por outras palavras, uma MT consegue sempre simular uma MTND, mas utilizando uma simulação que tem um custo exponencial em tempo. Por exemplo, como a MTND da Fig. 4.9 tem tempo de execução $O(n)$, concluímos que $L_3 = \{0^{2k} \in \{0\}^* | k \geq 1\} \cup \{0^{3k} \in \{0\}^* | k \geq 1\} \in TIME(2^{O(n)})$, isto é, L_3 pode ser reconhecida por uma MT em tempo $2^{O(n)}$.

O seguinte teorema mostra que adicionar várias fitas a uma MT apenas permite, no máximo, um ganho quadrático no tempo de execução relativamente a uma MT com uma só fita.

Teorema 5.1.5. *Seja $f : \mathbb{N} \rightarrow \mathbb{N}$ uma função satisfazendo $f(n) \geq n$ para todo o $n \in \mathbb{N}$. Se uma linguagem L é decidida em tempo $f(n)$ por uma máquina de Turing com k fitas, então L é decidida em tempo $O(f^2(n))$ por uma máquina de Turing com uma só fita.*

Demonstração. Antes de começar a demonstração, consideremos um exemplo. Suponhamos que temos uma MT com 3 fitas, cujo conteúdo das fitas é num dado momento

$$\begin{array}{c} \dots BB0101110BB\dots \\ \quad \quad \quad \triangle \\ \dots BBBB\dots \\ \quad \quad \quad \triangle \\ \dots BB11111BB\dots \\ \quad \quad \quad \triangle \end{array}$$

(o triângulo indicava o que a cabeça de leitura está a ler). Então o conteúdo da primeira fita pode ser codificado como $01\diamond 0110$ (os símbolos à esquerda do \diamond são os símbolos não-brancos à esquerda da cabeça de leitura, a palavra 0110 é o que está na posição da cabeça de leitura, inclusive, até à sua direita, não se contando a porção infinita de brancos que se lhe segue). Então o conteúdo das 3 fitas, assim como a posição das respetivas cabeças de leitura, pode ser registado numa única palavra

$$\#01\diamond 0110\#\diamond\#11\diamond 111\#.$$

Utilizando um procedimento semelhante, para uma MT M com k fitas, podemos registar o conteúdo das k fitas, assim como a posição das respetivas cabeças de leitura, utilizando uma única fita. Assim podemos criar uma MT M_1 , com uma única fita, que simula M . A MT simula M passo a passo da seguinte forma:

1. Lê o conteúdo da sua fita para determinar quais os símbolos lidos nas k fitas de M_1 ;
2. Com base nesses símbolos e no estado atual, atualiza o estado e o conteúdo das k fitas (ou melhor dizendo, a sua codificação numa única fita).

Note-se que às vezes vamos ter de “empurrar” (fazer um shift) quando efetuamos os movimentos virtuais das cabeças das fitas de M . Por exemplo, se na configuração

$$\# \blacklozenge 010110 \# \dots \#$$

o próximo passo será manter o símbolo 0 atualmente lido na fita 1 e deslocar a cabeça de leitura uma casa para a direita, o conteúdo virtual das 3 fitas será codificado como

$$\# \blacklozenge B010110 \# \dots \#.$$

Por outras palavras, fizemos um “shift” no conteúdo da fita de M_1 .

Vamos agora ver quanto tempo se demora a simular M com a máquina M_1 para um input de tamanho n . Como M utiliza tempo $f(n) \geq n$, cada uma das fitas de M nunca vai ter mais de $f(n)$ símbolos (mais precisamente $O(f(n))$). Logo a descrição das k fitas de M na única fita de M_1 precisa de $k \times O(f(n)) = O(f(n))$ símbolos.

A simulação de um passo de M pela MT M_1 necessita de $2O(f(n))$ passos (ler a fita e voltar atrás – passo 1 indicado atrás) + $k \times O(f(n))$ (para atualizar o conteúdo da fita. Pode haver até k shifts, necessitando cada um deles de tempo $O(f(n))$ para ser executado). Resumindo, cada passo de M pode ser executado em tempo $O(f(n))$ por M_1 .

Mas, com input de tamanho n , M utiliza no máximo $O(f(n))$ passos de computação. Logo M_1 necessita, no total da simulação, de tempo $O(f(n)) \times O(f(n)) = O(f^2(n))$. \square

Por exemplo, já sabemos que a MT da Fig. 4.8, que utiliza duas fitas, decide a linguagem $L_2 = \{0^k 1^k \in \{0, 1\}^* | k \geq 1\}$ em tempo $O(n)$. Então o teorema anterior permite-nos concluir que a linguagem L_2 pode ser decidida por uma MT com uma só fita em tempo $O(n^2)$.

5.2 As classes P e NP

A partir das definições da secção anterior, podemos definir algumas classes de complexidade.

Definição 5.2.1. Definimos as seguintes classes

$$\begin{aligned}
 P &= \bigcup_{k \in \mathbb{N}} TIME(n^k) \\
 NP &= \bigcup_{k \in \mathbb{N}} NTIME(n^k) \\
 EXPTIME &= \bigcup_{k \in \mathbb{N}} TIME(2^{n^k})
 \end{aligned}$$

Por outras palavras, a classe P (NP , respetivamente) é a classe das linguagens decididas por uma MT (MTND, respetivamente) em tempo polinomial. A classe $EXPTIME$ é a classe das linguagens decididas por uma MT em tempo exponencial. Note-se que, na definição de P ou de $EXPTIME$, não importa se utilizamos uma MT com uma só fita, ou com várias, devido ao Teorema 5.1.5.

Como exemplos,

$$\begin{aligned} L_1 &= \{0^k 1^k 0^k \in \{0, 1\}^* \mid k \in \mathbb{N}\} \in P \\ L_2 &= \{0^k 1^k \in \{0, 1\}^* \mid k \geq 1\} \in P \\ L_3 &= \{0^{2k} \in \{0\}^* \mid k \geq 1\} \cup \{0^{3k} \in \{0\}^* \mid k \geq 1\} \in NP \\ L_4 &= \{0, 1, 01, 10\} \in NP \end{aligned}$$

pois L_1 pode ser decidida em tempo $O(n^2)$ pela MT da Fig. 4.2, L_2 pode ser decidida em tempo $O(n)$ pela MT da Fig. 4.8, L_3 pode ser decidida em tempo $O(n)$ pela MTND da Fig. 4.9, e L_4 pode ser decidida em tempo $O(1)$ pela MTND da Fig. 4.10. A experiência mostra que os problemas que podem ser resolvidos em tempo razoável são aqueles que estão em P . É claro que pode dizer que um problema que necessita de tempo $n^{1000000}$ para ser resolvido não é resolúvel na prática, mas normalmente os problemas “úteis” em P podem ser resolvidos em tempo com expoentes baixos, tipicamente n ou n^2 .

Que relações existem entre estas classes? Sabe-se que

$$P \subseteq NP \subseteq EXPTIME.$$

A relação $P \subseteq NP$ é óbvia, e a inclusão $NP \subseteq EXPTIME$ vem do Teorema 5.1.4 (note-se que $O(2^{cn^k}) \subseteq O(2^{n^{k+1}})$). Sabe-se ainda que $P \neq EXPTIME$, mas não se conhece mais nenhuma relação entre estas 3 classes. Por exemplo, não se sabe se $NP = EXPTIME$ ou não, nem se $P = NP$ ou não.

Quando estivermos a tentar determinar a complexidade de algoritmos que envolvam cálculos aritméticos, utilizaremos as seguintes hipóteses:

- Adição: dados dois inputs de tamanho $\leq n$, a sua adição pode ser calculada em tempo $O(n)$.
- Multiplicação, divisão inteira (div), resto da divisão inteira (mod): dados dois inputs de tamanho $\leq n$, estas operações podem ser calculadas em tempo $O(n^2)$.

(assumimos que os números são escritos em notação unária, ou em base $d \geq 2$. Os tempos das operações podem ser obtidos em qualquer livro de Análise Numérica, ou inspecionado o tempo necessário para executar os cálculos “à mão”). Existem formas alternativas de caracterizar NP . Uma delas é utilizando *verificadores*. Suponhamos que temos uma linguagem L em NP , reconhecida em tempo $O(n^k)$ por uma MTND N_1 . Presentemente a teoria apenas garante existir uma MT determinística que reconhece L em tempo $2^{O(n^k)}$. No entanto, se alguém nos disser que $w \in L$, e além do mais nos fornecer

o caminho c de uma computação aceitadora (na demonstração do Teorema 4.2.2, este c é a palavra da fita 3 que leva a uma configuração aceitadora. Por exemplo, na MTND da Fig. 4.10 – ver comentário a seguir à demonstração do Teorema 4.2.2 – a palavra $c = 12$ dá-nos o caminho de uma computação que aceita o input 0), conseguimos verificar a veracidade dessa informação com uma MT determinística em tempo polinomial. Esta informação adicional c costuma ser designada de certificado, pois permite-nos certificar que o input é aceite, com uma MT determinística, em tempo polinomial. Esta é a ideia base dos certificados de segurança: não podem ser criados em tempo polinomial por um utilizador qualquer, i.e. não se pode determinar em tempo polinomial um caminho aceitador com uma MT determinística, mas pode-se verificar essa informação (caminho) em tempo polinomial, garantindo a segurança do comércio eletrónico (desde que as premissas indicadas atrás sejam satisfeitas).

Na prática, esta descrição de NP com certificados é mais útil do que considerar MTND's.

Teorema 5.2.2. $L \in NP$ se e só se existe uma máquina de Turing M cujo tempo de execução é polinomial (em w) tal que

$$L = \{w \in \Sigma^* \mid M \text{ aceita } \langle w, c \rangle, \text{ para algum } c \in \Sigma^*\}. \quad (5.1)$$

Demonstração. Demonstração da direção \implies do “se e só se”. Se $L \in NP$, então existe uma MTND N que aceita L . Podemos criar uma MT M que recebe como input $\langle w, c \rangle$, que basicamente simula N , com input w , para o caminho de computação indicado por c (ver a demonstração do Teorema 4.2.2 e os comentários que se lhe seguem). Se esse caminho aceitar, M aceita $\langle w, c \rangle$, caso contrário rejeita $\langle w, c \rangle$. Ora $w \in L$ sse existe uma computação de N que aceita w sse M aceita $\langle w, c \rangle$ para algum $c \in \Sigma^*$. Por outras palavras (5.1) é satisfeita.

Para mostrar a implicação inversa, suponhamos que (5.1) é satisfeita. Suponhamos ainda que M corre em tempo n^k , onde $n = |w|$. Então M nunca vai ler mais do que n^k símbolos de c . Então podemos criar uma MTND N_1 que faz o seguinte (com input w):

1. Criar não-deterministicamente uma palavra $c \in \Sigma^*$ de comprimento n^k (tempo n^k).
2. Correr M com input $\langle w, c \rangle$ (tempo n^k).
3. Se M aceitar $\langle w, c \rangle$, aceitar w , caso contrário rejeitar (tempo 1).

O que é isso de criar não-deterministicamente uma palavra $c \in \Sigma^*$? Suponhamos, por exemplo, que $\Sigma = \{0, 1\}$. Então podemos criar não-deterministicamente uma palavra de comprimento 3 em Σ^* através da MTND da Fig. 5.1

Por outras palavras, por cada caminho de computação dessa MTND, é gerada uma palavra de $\{0, 1\}^*$ com comprimentos 3 e vice-versa. Portanto existe um caminho de computação que escreve 000 na fita, outro que escreve 010, outro que escreve 101, etc. Note-se que cada caminho necessita só de 3 passos. Em geral, criar não-deterministicamente palavras de Σ^* de comprimento n^k demora tempo n^k .

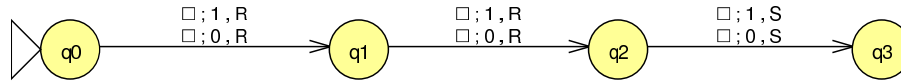


Figura 5.1: Exemplo de máquina de Turing não-determinística que gera números não-deterministicamente.

Portanto N_1 corre em tempo $O(n^k)$ e w é aceite por N_1 sse existe um caminho de computação aceitador sse existe um $c \in \Sigma^*$ de comprimento n^k tal que M aceita $\langle w, c \rangle$ sse $w \in L$. Logo, N_1 reconhece L , donde $L \in NP$. \square

Por outras palavras, a classe NP é a classe dos problemas em que, dada uma solução, é fácil verifica-la, mas é muito difícil determina-la, exceto se o problema também pertence a P .

Existem muitos outros exemplos onde o certificado parece ser fundamental, por exemplo em problemas envolvendo redes (grafos). Obviamente, quando considerarmos problemas com grafos, o input não será um grafo G , mas a sua codificação $\langle G \rangle$ utilizando, por exemplo, as codificações dadas no final da Secção 4.2. Note-se que se G tem n vértices, então para as codificações indicadas atrás, tem-se $n \leq |\langle G \rangle|$. Além do mais, existe um $j \in \mathbb{N}$ tal que $|\langle G \rangle| \leq n^j$. Portanto, se o nosso input for um grafo G com n vértices, e se mostrarmos que existe um algoritmo que é executado em tempo polinomial em n , esse algoritmo também será executado em tempo polinomial relativamente ao tamanho do input $\langle G \rangle$. Assim, para mostrar que um problema com grafos pertence a P (ou NP), basta mostrar que existe uma MT (MTND) que resolve o dado problema em tempo $O(n^k)$, onde n é o número de vértices do grafo G dado como input (em vez de mostrar que o tempo é $O(|\langle G \rangle|^k)$, já que o input é $w = \langle G \rangle$).

Considere, por exemplo, o seguinte problema:

$$L_2 = \{ \langle G \rangle \text{ onde } G \text{ é um grafo não-orientado} \mid \text{existe um caminho que passa exatamente uma vez por cada vértice de } G \}. \quad (5.2)$$

A seguinte MT determinística permite decidir L_2 (supomos que G tem n vértices):

1. De entre os n vértices de G , $1, \dots, n$, seleccionar uma sequência de n vértices distintos i_1, \dots, i_n .
2. Testar se $i_1 \rightarrow \dots \rightarrow i_n$ é um caminho em G . Por outras palavras, testar se $(i_1, i_2), \dots, (i_{n-1}, i_n)$ são arestas de G (pode-se ver se a aresta (i_1, i_2) pertence a G “scannado” o input $\langle G \rangle$. Cada teste destes demora $O(|\langle G \rangle|) = O(n^j)$ passos). Se sim, aceitar, se não voltar ao passo 1 com outra sequência de vértices não testada (se não restarem mais possibilidades, rejeitar).

A execução da rotina para cada possível sequência de vértices demora tempo $O(n^{j+1})$ (temos $\sim n$ arestas a testar, demorando cada uma delas $O(n^j)$ passos a testar). Mas há $n!$

possibilidades de escolhas para os vértices no passo 1, pelo que esta MT não é executada em tempo polinomial, mas somente em tempo exponencial. Logo $L_2 \in EXPTIME$. No entanto, se utilizarmos uma MTND, o tempo de computação passa a ser polinomial. Para isso basta utilizar o seguinte algoritmo:

1. De entre os n vértices de G , $1, \dots, n$, selecionar **não-deterministicamente** uma sequência de n vértices distintos i_1, \dots, i_n .
2. Testar se $i_1 \rightarrow \dots \rightarrow i_n$ é um caminho em G . Se sim, aceitar, se não rejeitar.

Esta MTND também tem $n!$ caminhos de computação (uma para cada possível caminho de G), mas cada caminho pode ser executado em tempo $O(n^{j+1})$. Logo o tempo de execução da MTND é $O(n^{j+1})$, donde $L_2 \in NP$.

Outra forma de mostrar que $L_2 \in NP$ é utilizar certificados: dado um grafo G e um certificado (solução) c , a seguinte MT permite verificar em tempo polinomial se $G \in L_2$ (i.e. permite verificar em tempo polinomial a solução):

1. Dado o input $\langle G, c \rangle$, testar se c codifica uma sequência i_1, \dots, i_n de n vértices de G .
2. Testar se $i_1 \rightarrow \dots \rightarrow i_n$ é um caminho em G .
3. Se ambos os testes deram uma resposta afirmativa, aceitar $\langle G, c \rangle$, caso contrário, rejeitá-lo.

Como já dissemos, o certificado é basicamente a codificação do caminho aceitador (i.e. da solução) que mostra que $G \in L_2$. Outra forma de o ver, e que normalmente é utilizada na prática (foi utilizado no exemplo em cima) é assumir que se c é um certificado, então codifica uma peça de informação que garante que o input pertence à linguagem. No exemplo anterior, a propriedade que G tem de ter para pertencer a L_2 é que exista um caminho com n vértices distintos em G . Então para demonstrar que $G \in L_2$ basta exibir esse caminho, ou seja, é natural que o certificado seja a codificação desse caminho. Essa ideia é utilizada nas aulas práticas (para pensar no que é o certificado basta pensar na seguinte pergunta: “qual a peça de informação que eu tenho de exibir para garantir que o input pertença à linguagem?”. Em geral, o certificado não será nada mais do que uma codificação dessa peça de informação).

5.3 Problemas NP-completos

Em 1971 Stephen Cook introduziu a noção de problema NP -completo como ferramenta para abordar o problema “ $P = NP?$ ” Esta noção é importante não só pela contribuição que poderá ter na resolução do problema “ $P = NP?$ ”, mas também porque permite concluir que certos problemas em NP são especialmente difíceis. Antes de introduzir esta noção, precisamos de algumas definições.

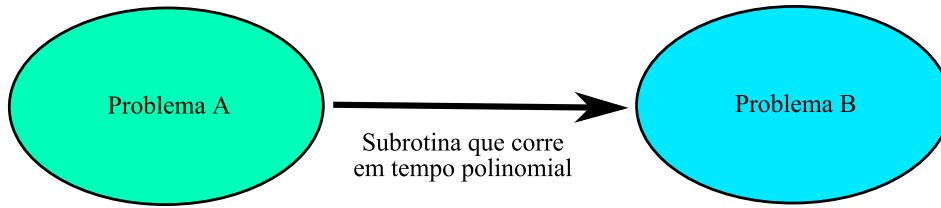


Figura 5.2: Redutibilidade de linguagens.

Definição 5.3.1. Uma função $f : \Sigma^* \rightarrow \Sigma^*$ diz-se *computável em tempo polinomial* se existe uma máquina de Turing que computa f em tempo polinomial.

Por exemplo, a função $f : \{0\}^* \rightarrow \{0\}^*$ dada por $f(0^n) = 0^{2n}$ pode ser calculada em tempo $O(n^2)$ pela MT da Fig. 4.5, donde f é computável em tempo polinomial.

Definição 5.3.2. Uma linguagem A é *reduzível em tempo polinomial* a uma linguagem B , escrito como $A \leq_P B$, se existe uma função $f : \Sigma^* \rightarrow \Sigma^*$ computável em tempo polinomial, tal que para todo o $w \in \Sigma^*$,

$$w \in A \text{ se e só se } f(w) \in B.$$

Por outras palavras, existe um subrotina que permite converter, em tempo polinomial, o problema “ $x \in A$?” para o problema “ $x \in B$?”, como sugere a Fig. 5.2. Assim, se for fácil decidir a linguagem B , também será fácil decidir A , utilizando a subrotina de conversão. Por exemplo, considere-se a linguagem

$$HAMPATH = \{ \langle G, s, t \rangle \text{ onde } G \text{ é um grafo e } s, t \text{ são vértices} \mid \text{existe um caminho hamiltoniano de } s \text{ a } t \}, \quad (5.3)$$

onde um caminho hamiltoniano é um caminho que passa exatamente uma vez por cada vértice de G . Então, dada a linguagem L_2 definida pela Equação (5.2), tem-se $HAMPATH \leq_P L_2$. Para mostrar isso, temos de definir a função f que faz a redução polinomial. O que fazemos é dado o grafo G , construir um novo grafo G' da seguinte forma: G' é uma cópia de G , em que se adicionaram dois novos vértices v_1, v_2 , e duas novas arestas $(v_1, s), (v_2, t)$. Então a única forma de existir um caminho que passa por todos os vértices de G' uma única vez é ter um caminho $v_1 \rightarrow s \rightarrow \dots \rightarrow t \rightarrow v_2$ onde $s \rightarrow \dots \rightarrow t$ é um caminho hamiltoniano (repare que se v_1 não surgir num extremo do caminho, ele aparece no meio da sequência de vértices. Então teremos $\dots \rightarrow s \rightarrow v_1 \rightarrow s \rightarrow \dots$ i.e. s aparece 2 vezes no caminho, o que não pode ser. Um raciocínio idêntico aplica-se a v_2). Assim, $\langle G, s, t \rangle \in HAMPATH$ se e só se $\langle G' \rangle \in L_2$. Portanto, definindo $f(\langle G, s, t \rangle) = \langle G' \rangle$, é fácil ver que f pode ser computado em tempo polinomial (basta acrescentar 2 vértices e duas arestas na descrição de G , apagando os vértices s, t na descrição $\langle G, s, t \rangle$) e que

$$\langle G, s, t \rangle \in HAMPATH \text{ se e só se } f(\langle G, s, t \rangle) \in L_2.$$

Por outras palavras, $HAMPATH \leq_P L_2$. Podemos interpretar esta relação da seguinte forma: se temos um algoritmo para decidir L_2 , então podemos adaptar este algoritmo, utilizando pouco mais esforço computacional, para decidir $HAMPATH$. Vamos agora ver várias consequências da redutibilidade em tempo polinomial.

Teorema 5.3.3. *Se $A \leq_P B$ e $B \in P$, então $A \in P$.*

Demonstração. Seja M a MT que decide B em tempo polinomial e seja a f a redução polinomial de A para B . Então o seguinte algoritmo decide A em tempo polinomial:

1. Para um input w calcule $f(w)$;
2. De seguida utilize $f(w)$ como input para B . Se B aceita $f(w)$, aceitar w , caso contrário rejeitar w . □

Definição 5.3.4. *Uma linguagem B é NP -completa se satisfaz as seguintes definições:*

1. $B \in NP$;
2. Todo o $A \in NP$ é redutível em tempo polinomial para B .

Teorema 5.3.5. *Se B é NP -completo e $B \in P$, então $P = NP$.*

Demonstração. Imediata a partir do Teorema 5.3.3. □

Isto mostra que se conseguirmos arranjar um algoritmo que resolva um problema NP -completo em tempo polinomial, então mostramos que $P = NP$. Como é normalmente conjecturado que $P \neq NP$, esse resultado pode ser visto de outra forma: os problemas de tipo NP -completo são os mais “difíceis” da classe NP , e se nos depararmos com um, é muito possível que não exista um algoritmo que o resolva em tempo útil (polinomial) pelo que, na prática, teremos de recorrer a heurísticas ou algoritmos probabilísticos. Mas como identificar problemas NP -completos? Os seguintes dois teoremas dão-nos informações valiosas.

Teorema 5.3.6. *Seja \mathcal{U} o conjunto das fórmulas booleanas. Então a seguinte linguagem é NP -completa*

$$SAT = \{ \langle \phi \rangle \text{ onde } \phi \in \mathcal{U} \mid \phi \text{ é satisfazível} \}.$$

A linguagem SAT foi a primeira linguagem que se demonstrou ser NP -completa. A partir deste conhecimento, torna-se muito mais fácil mostrar que outras linguagens são NP -completas, através do seguinte teorema.

Teorema 5.3.7. *Se B é NP -completo e $B \leq_P C$, onde $C \in NP$, então C é NP -completo.*

Demonstração. Seja $A \in NP$. Como B é NP -completo, $A \leq_P B$. Mas como $B \leq_P C$, conclui-se que $A \leq_P C$. Além do mais $C \in NP$, donde C será NP -completo. □

Em particular, utilizando o teorema anterior em conjugação com a linguagem SAT , é possível mostrar que a linguagem $HAMPATH$ definida pela Equação (5.3) é NP -completa (ver, por exemplo, [Sip05]).

Vimos também que a linguagem L_2 da Equação (5.2) tem as seguintes propriedades: $L_2 \in NP$ e $HAMPATH \leq_P L_2$, pelo que podemos concluir, utilizando o teorema anterior, que L_2 é também NP -completa. Utilizando este procedimento podemos provar que muitas linguagens são NP -completas. Por exemplo, o livro [GJ79] tem mais de 300 linguagens NP -completas.

Finalmente introduzimos uma definição que é utilizada várias vezes em aplicações.

Definição 5.3.8. Uma linguagem L diz-se *NP-difícil* (*NP-hard* em inglês) se todo o $A \in NP$ é redutível em tempo polinomial para L (a diferença em relação a linguagens NP -completas é que não tem necessariamente de ser $L \in NP$).

Em particular, toda a linguagem NP -completa é NP -difícil. Na Fig. 5.3 é apresentado um diagrama que relaciona as principais classes de linguagens estudadas nesta cadeira. Há dois resultados que não foram provados nestes apontamentos, mas que se verificam: classe das linguagens livres de contexto $\subsetneq P$ (ver [Sip05] para uma demonstração); classe das linguagens r.e. \subsetneq classe de todas as linguagens (pois o complemento da linguagem $H_{paragem}$ define uma linguagem que não é r.e. – ver exercício 82 das folhas teórico-práticas). A fronteira entre P e NP está a tracejado, uma vez que não se sabe se $P \neq NP$.

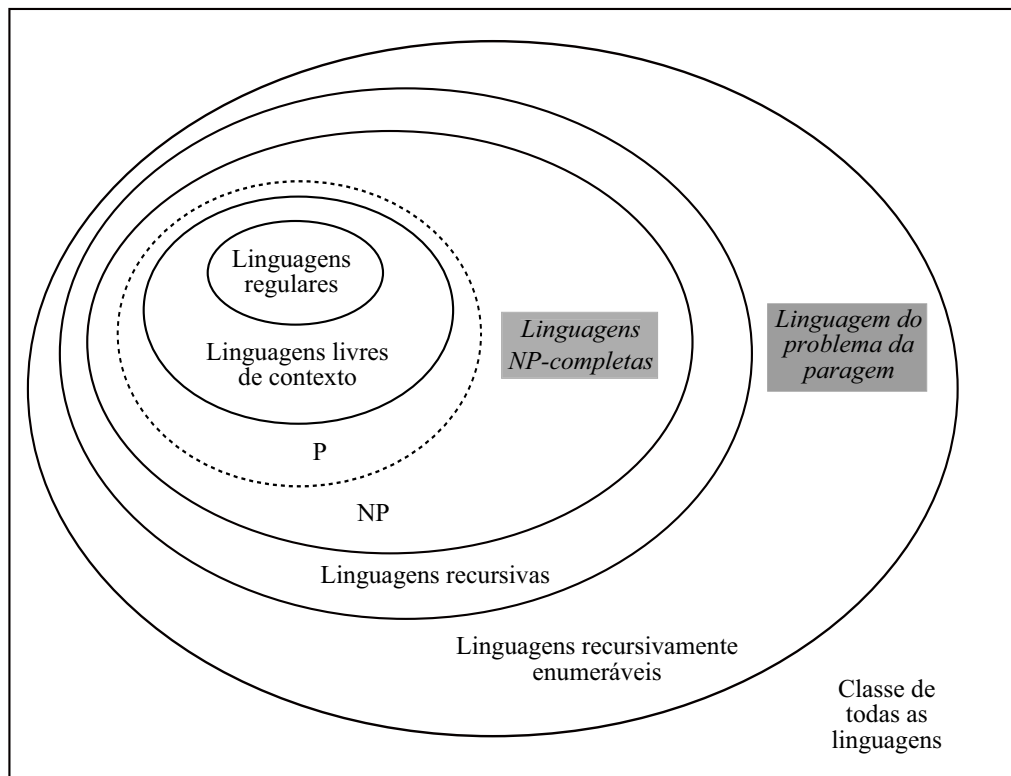


Figura 5.3: Relação entre as principais linguagens estudadas nesta cadeira.

Bibliografia

- [GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: a Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.
- [HMU06] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 3rd edition, 2006.
- [Sip05] M. Sipser. *Introduction to the Theory of Computation*. Course Technology, 2nd edition, 2005.