Introdução à Teoria da Computação

Márcio Moretto Ribeiro

1 de Dezembro de 2020

Conteúdo

Capítulo 1

Introdução

1.1 Apresentação

O curso de Teoria da computação procura responder duas perguntas centrais da área de Ciência da Computação:

1. Que problemas são resolvíveis de forma automática? (computabilidade)
2. Que problemas são resolvíveis de maneira eficiente? (complexidade)

Para responder à primeira pergunta primeiro devemos esclarecer alguns pontos:

- O que estamos chamando de *problema* no sentido computacional do termo?
- O que é um *método automático* de resolução?

Extencionalmente, um *problema computacional* é uma espécie de função que descreve os valores aceitos como *entrada* (domínio) e os valores esperados como *saída*. A solução de um problema computacional, como estamos acostumados, é uma sequência de instruções inequívocas – um algoritmo – que dado um elemento válido de entrada produz a saída esperada.

Exemplo 1.1.1:

O *problema da ordenação* pode ser descrito da seguinte maneira:

Entrada: Uma sequência de n inteiros $\langle a_1, \ldots, a_n \rangle$.

Saída: Uma permutação da entrada $\langle a'_1, \ldots, a'_n \rangle$ em que $a'_i \leq a'_j$ para todo $i < j$.

Uma *instância* desse problema seria dada por uma entrada específica (e.g. $\langle 4, 2, 42, 24 \rangle$) e sua saída ($\langle 2, 4, 24, 42 \rangle$). E a solução do problema é qualquer dos algoritmos de ordenação que estudamos no curso de análise de algoritmos.

Nosso foco neste curso será nos problemas ditos de *decisão*, ou seja, naqueles cuja saída deve ser `SIM` ou `NÃO`

Exemplo 1.1.2:

O seguinte é um problema clássico de decisão:

Entrada: Um inteiro $n > 1$.

Saída: `SIM` se `n` é primo e `NÃO` caso contrário.

Como veremos na próxima seção, existe uma relação íntima entre problemas de decisão e linguagens formais, a saber, para cada problema de decisão existe uma linguagem formal equivalente e vice-versa. De fato, em grande parte do curso estudaremos classes de linguagens formais.

Voltemos então para a pergunta: o que é um método automático de resolução?

Para responder a essa pergunta, que não tem nada de trivial, precisamos de um *modelo* do funcionamento de dispositivos eletrônicos. Como qualquer outro modelo, faremos abstrações, simplificaremos e desprezaremos variáveis.

Podemos refrasear nossa pergunta, de maneira agora um pouco mais precisa:

- Que linguagens são reconhecíveis por certo modelo de computação?

Durante o curso estudaremos uma sequência de modelos de computação com *expressividade* crescente. Começaremos com modelos simples, adequados para representar dispositivos simples e capazes de reconhecer uma certa classe de linguagens. Conforme avançarmos no curso estudaremos modelos mais sofisticados capazes de representar classes cada vez maiores de linguagens.

O diagrama abaixo apresenta um resumo dos três primeiros capítulos do livro. No Capítulo 2 estudaremos Autômatos Finitos Determinísticos e

não-Determinísticos, mostraremos que eles são equivalentes e são capazes de reconhecer exatamente a classe das linguagens regulares e terminaremos mostrando que nem toda linguagem é regular. No Capítulo 3 começaremos estudando as Linguagens Livres de Contexto e os Autômatos de Pilha, mostraremos a equivalência entre os dois e terminaremos mostrando que nem toda linguagem é livre de contexto. No Capítulo 4 veremos Máquinas de Turing Determinísticas e não-Determinísticas, Linguagens Recursivas e Recursivamente Enumeráveis.

Modelos de Computação		**Classes de Linguagens**	
AFD AFN	$\Leftrightarrow$	Linguagens Regulares	*expressividade* $\downarrow$
AP	$\Leftrightarrow$	Linguagens Livres de Contexto	
MT	$\Leftrightarrow$	Linguagens Recursivas Ling. Recursivamente Enumeráveis	

Tabela 1.1: Resumo da apostila

As Máquinas de Turing (MTs) são o modelo mais completo que veremos. Na verdade a Tese de Church afirma que, de fato, as MTs são o modelo mais completo possível. Em outras palavras: se algo pode ser computado, então ele pode ser computado por uma MT.

Por outro lado, veremos que existem linguagens que não são reconhecíveis por MTs, ou equivalentemente, existem problemas de decisão que não admitem solução computacional (*indecidíveis*).

$$Regulares \subset LLCs \subset Recursivas \subset REs \subset Linguagens$$

Para finalizar o curso, no último capítulo nos voltaremos para a segunda pergunta central:

- Que problemas podem ser resolvidos de maneira eficiente?

A resposta dessa pergunta certamente depende do modelo de computação que estamos considerando e o que entendemos por *eficiência*. Vamos convencionar que estamos falando de MTs e que por eficientes queremos dizer soluções que consomem tempo polinomial em relação ao tamanho da entrada. Assim, podemos refrasear a questão central da seguinte forma:

- Para quais problemas existe uma MT que o resolve em tempo polinomial?

Chamaremos essa classe de problemas de P. Note que, diferente do curso de análise de algoritmos em que o foco está nos algoritmos e sua eficiência, aqui o foco está nos problemas. A pergunta não é quão eficiente é uma determinada solução de um problema, mas que problemas estão em cada classe.

Se substituirmos as MT Determinísticas por não-Determinísticas, teremos outra classe de problemas, a classe NP.

O maior problema em aberto hoje na computação, e quiçá na matemática, é saber se essas classes coincidem:

$$P \stackrel{?}{=} NP$$

No fim do curso procuraremos definir o problema de maneira formal e apresentar os poucos resultados mais simples sobre esse assunto.

1.2 Problemas de Decisão

Vamos começar estudando um problema de decisão de grande importância na Teoria da Computação por se tratar do primeiro problema demonstradamente NP-completo (veremos isso no Capítulo 5). O problema que nos referimos é o de decidir se uma fórmula proposicional é ou não satisfatível. Recordemos do curso de Matemática Discreta que uma fórmula proposicional sempre pode ser escrita na Forma Normal Conjuntiva (FNC), ou seja, como uma conjunção de disjunção de literais. Vamos então definir a linguagens das fórmulas supondo que ela esteja na FNC para facilitar.

Partimos de um conjuntos finito cujos elementos são chamados *variáveis proposicionais* $\mathbb{P} = \{p_1 \dots p_n\}$.

Um *literal* é qualquer elemento de $\mathbb{L} = \mathbb{P} \cup \overline{\mathbb{P}}$ onde $\overline{\mathbb{P}} = \{\bar{p_i} : p_i \in \mathbb{P}\}$. Ou seja, um literal é uma variável ou sua negação que representamos pelo mesmo símbolo com um traço em cima.

Uma sequência de literais é chamado de *cláusula* e a representaremos como $l_1 l_2 \dots l_n$ onde $l_i \in \mathbb{L}$.

Exemplo 1.2.1:

Seja $\mathbb{P} = \{p_1, p_2, p_3\}$ então as seguintes são cláusulas:

$p_1 p_2 p_3 \bar{p_1}$

$p_2 \bar{p_2} p_1$

p_1

$p_1p_2p_3$

Uma sequência de cláusulas é chamada de *fórmula* e será representada como $c_1; c_2; \ldots; c_n$.

Exemplo 1.2.2:

Seja $\mathbb{P} = \{p_1, p_2, p_3\}$ então as seguintes são fórmulas:

$p_1p_2; p_2p_3; \bar{p_3}$

$p_1\bar{p_1}; p_2\bar{p_2}$

$p_1p_2p_3$

Interpretaremos uma sequência de literais de maneira disjuntiva e uma sequência de cláusulas de maneira conjuntiva da seguinte forma. Uma função $v : \mathbb{L} \to \{0, 1\}$ é uma *valoração* se para todo $p \in \mathbb{L}$ temos:

$$\begin{array}{lll} v(p) = 1 & \text{sse} & v(\bar{p}) = 0 \\ v(p) = 0 & \text{sse} & v(\bar{p}) = 1 \end{array}$$

Dizemos que uma valoração v *satisfaz uma cláusula* $l_1 \ldots l_n$ se $v(l_i) = 1$ para algum i. Em outras palavras, a valoração satisfaz a cláusula se ela atribuiu o valor verdade para algum literal da cláusula. Uma valoração v *satisfaz uma fórmula na FNC* $c_1; \ldots; c_n$ ela satisfaz cada uma das cláusulas da fórmula.

Exemplo 1.2.3:

Seja $\mathbb{P} = \{p_1, p_2, p_3\}$. A valoração v tal que $v(p_1) = v(p_2) = 1$ e $v(p_3) = 0$ satisfaz as seguintes fórmulas:

$p_1\bar{p_2}; p_2; p_3p_1$

$p_1\bar{p_1}; p_2\bar{p_2}$

$\bar{p_3}$

Por outro lado, v não satisfaz as seguintes fórmulas:

p_3

$p_3; p_1p_2; p_1$

$p_3p_1; \bar{p_1}\bar{p_2}$

Uma fórmula é dita *satisfatível* se existe uma valoração v que a satisfaça.

Exemplo 1.2.4:

São exemplos de fórmulas satisfatíveis:

$p_1p_2; \bar{p_1}\bar{p_2}$

p_1p_2

p_1

São exemplos de fórmulas *não* satisfatíveis:

$p_1; \bar{p_1}$

$p_1\bar{p_2}; p_2; \bar{p_1}$

O problema da satisfatibilidade, ou simplesmente SAT, é um problema de decisão que pode ser enunciado da seguinte forma:

Entrada: Uma fórmula α na FNC qualquer sobre $\mathbb{P}$.

Saída: `SIM` se α é satisfatível e `NÃO` caso contrário.

1.3 Linguagens Formais

Um *alfabeto* é um conjunto finito qualquer Σ cujos elementos são chamados *símbolos*.

Exemplo 1.3.1:

São exemplos de alfabeto:

$\Sigma = \{p_1, p_2, p_3\}$

$\overline{\Sigma} = \{\bar{p_1}, \bar{p_2}, \bar{p_3}\}$

$\Sigma \cup \overline{\Sigma} = \{p_1, p_2, p_3, \bar{p_1}, \bar{p_2}, \bar{p_3}\}$

Uma sequência de símbolos de um alfabeto Σ é chamada de uma *string* ou *palavra* sobre esse alfabeto. A string vazia, que representa a sequencia de zero símbolos, será representa por ε. O comprimento de uma string s é representado por $|s|$.

Exemplo 1.3.2:

São string sobre $\Sigma = \{0, 1\}$ os seguintes:

01110

11

1

ε

Além disso temos que:

$|01110| = 5$

$|11| = 2$

$|1| = 1$

$|\varepsilon| = 0$

Sejam $x = a_1 \dots a_n$ e $y = b_1 \dots b_m$ duas strings. A *concatenação* de x com y será representada por $x \cdot y = xy = a_1 \dots a_n b_1 \dots b_n$. Note que nem sempre $x \cdot y = y \cdot x$. Além disso, para todo x temos que $\varepsilon \cdot x = x \cdot \varepsilon = x$.

O conjunto de todas as strings sobre um alfabeto Σ será representada por Σ^*. Um conjunto de strings A sobre Σ é chamado de uma *linguagem sobre* Σ i.e. $A \subseteq \Sigma^*$ é uma linguagem.

Exemplo 1.3.3:

São linguagens sobre $\Sigma = \{p_1, p_2\}$:

$A = \{p_1, p_2, p_1p_2, p_1p_1\}$

$B = \{p_1\}$

$C = \emptyset$

$D = \{\varepsilon, p_1, p_1p_1, p_1p_1p_1, \dots\}$

Note que, como no último exemplo, uma linguagem pode ser infinita. Existe um problema de decisão naturalmente associado a cada linguagem L, o *problema do reconhecimento*:

Entrada: $x \in \Sigma^*$

Saída: `SIM` se $x \in L$ e `NÃO` caso contrário.

Conversamente, todo problema de decisão possui uma linguagem formal naturalmente associada da seguinte forma. Seja A a linguagem das entradas aceitas como válidas para o problema. Considere agora todas as strings x para as quais o problema de decisão deve responder `SIM`. O conjunto dessas strings é a linguagem associada ao problema.

Exemplo 1.3.4:

O problema SAT induz uma linguagem formal $A \subseteq \Sigma^*$ aonde $\Sigma = \{p_1, \ldots, p_n, \bar{p_1}, \ldots, \bar{p_n}\}$. A linguagem das fórmulas satisfatíveis.

1.4 Bibliografia

- Introdução à Teoria da Computação - Michael Sipser
- Elementos da Teoria da Computação - Lewis e Papadimitrius
- Computabilidade, Funções Computáveis, Lógica e os Fundamentos da Matemática - Carnielli e Epstein
- Computational Complexity - Christos H. Papadimitriou
- Lógica para Computação - Flávio Soares, Marcelo Finger e Ana Cristina Vieira de Melo

Capítulo 2

Autômatos Finitos

Neste capítulo estudaremos um modelo simples de computação, os autômatos finitos, e a classe das linguagens regulares.

2.1 Linguagens Regulares

Voltaremos nossa atenção um instante para conjuntos (classes) de linguagens, ou seja, conjuntos de conjuntos de strings.

Exemplo 2.1.1:

Ł $= \{A : A \subseteq \Sigma^*\}$ é o conjunto de todas as linguagens sobre Σ

$\emptyset$ é a classe vazia.

$\{\emptyset\}$ é a classe que contém apenas a linguagem vazia.

$\{\{\varepsilon\}, \emptyset\}$ é a classe que contém a linguagem vazia e a linguagem que possui apenas a string vazia.

Podemos aplicar operações sobre linguagens. Como linguagens são conjuntos de strings, podemos tomar a *união* de duas linguagens:

$$A \cup B = \{x \in \Sigma^* : x \in A \text{ ou } y \in B\}$$

Exemplo 2.1.2:

$$\begin{aligned} A &= \{p_1p_2, p_1, p_2p_1\} \\ B &= \{p_1p_1, p_3, p_1\} \\ A \cup B &= \{p_1p_2, p_1, p_2p_1, p_1p_1, p_3\} \end{aligned}$$

Outra operação sobre linguagens é *concatenação* que consiste na concatenação de cada combinação de strings da linguagem:

$$A \circ B = \{x \cdot y \in \Sigma^* : x \in A \text{ e } x \in B\}$$

Exemplo 2.1.3:

$$\begin{aligned} A &= \{p_1p_2, p_1\} \\ B &= \{p_1p_1, p_3\} \\ A \circ B &= \{p_1p_2p_1p_1, p_1p_2p_3, p_1p_1p_1, p_1p_3\} \\ B \circ A &= \{p_1p_1p_1p_2, p_1p_1p_1, p_3p_1p_2, p_3p_1\} \\ A \circ A &= \{p_1p_2p_1p_2, p_1p_2p_1, p_1p_1p_2, p_1p_1\} \end{aligned}$$

Podemos, por fim, aplica a *estrela de Kleene* sobre uma linguagem para produzir todas as possíveis concatenações dos elementos:

$$A^* = \{x_1 \dots x_k \in \Sigma^* : x_i \in A\}$$

Exemplo 2.1.4:

$$\begin{aligned} A &= \{a, b\} \\ A^* &= \{\varepsilon, a, b, aa, ab, bb, ba, aaa, aab, aba, abb, \dots\} \end{aligned}$$

Repare que a notação Σ^* é consistente com a definição de estrela de Kleene.

As operações de união, concatenação e estrela de Kleene sobre linguagens são chamadas *operações regulares*.

Exemplo 2.1.5:

$$\begin{aligned} A &= \{a\} \\ B &= \{aa, b\} \\ A \circ B &= \{aaa, ab\} \\ A \cup (A \circ B) &= \{a, aaa, ab\} \\ (A \cup (A \circ B))^* &= \{\varepsilon, a, aaa, ab, aa, aaaa, aab, aaaaaa, aaaab, \dots\} \end{aligned}$$

Uma classe de linguagens Ł é *fechada por união* quando temos que:

$$\text{se } A, B \in \text{Ł então } A \cup B \in \text{Ł}$$

Analogamente, uma classe de linguagens Ł é *fechada por concatenação* quando temos que:

$$\text{se } A, B \in \text{Ł então } A \circ B \in \text{Ł}$$

Por fim, Ł é *fechada pela estrela de Kleene* quando temos que:

$$\text{se } A \in \text{Ł então } A^* \in \text{Ł}$$

Exemplo 2.1.6:

$$\begin{aligned} \text{Ł}_1 &= \{\{a\}, \{b\}\} \\ \text{Ł}_2 &= \{\{a\}, \{b\}, \{a, b\}\} \\ \text{Ł}_3 &= \{\{a\}, \{aa\}, \{aaa\}, \{aaaa\} \dots\} \\ \text{Ł}_4 &= \{\{a\}, \{\varepsilon, a, aa, aaa, \dots\}\} \end{aligned}$$

Ł_1 não é fechada por união, mas Ł_2 é. Ł_3 é fechada por concatenação e Ł_4 é fechada pela estrela de Kleene.

A classe das *linguagens regulares* é a menor classe de linguagens fechada por união, concatenação e estrela de Kleene que contém a seguinte linguagem:

$$\{\{a\} : a \in \Sigma\}$$

Uma foram alternativa de definir linguagens regulares é por meio de expressões regulares. Uma *expressão regular* pode ser definida da seguinte forma:

se $r \in \Sigma$ então r é uma expressão regular,

ϵ é uma expressão regular,

$\emptyset$ é uma expressão regular,

se r_1 e r_2 são expressões regulares então $r_1 \cup r_2$ é uma expressão regular,

se r_1 e r_2 são expressões regulares então $r_1 r_2$ é uma expressão regular e

se r é uma expressão regular então $r^\star$ é uma expressão regular.

Exemplo 2.1.7:

São expressões regulares:

$\emptyset$

01

$01^\star \cup 1$

$\epsilon \cup \emptyset$

Denotaremos $L(r)$ a linguagem *expressa* pela expressão regular r:

$$\begin{aligned} L(a) &= \{a\} \text{ para todo } a \in \Sigma \\ L(\epsilon) &= \{\varepsilon\} \\ L(\emptyset) &= \emptyset \\ L(r_1 \cup r_2) &= L(r_1) \cup L(r_2) \\ L(r_1 r_2) &= L(r_1) \circ L(r_2) \\ L(r^\star) &= L(r)^* \end{aligned}$$

$$
\begin{aligned}
L(\emptyset) &= \emptyset \\
L(01) &= L(0) \circ L(1) \\
&= \{0\} \circ \{1\} \\
&= \{01\} \\
L(01^\star \cup 1) &= L(01^\star) \cup L(1) \\
&= L(0) \circ L(1^\star) \cup \{1\} \\
&= \{0\} \circ \{1\}^* \cup \{1\} \\
&= \{1, 0, 01, 011, 0111 \dots\} \\
L(\epsilon \cup \emptyset) &= L(\epsilon) \cup L(\emptyset) \\
&= \{\varepsilon\} \cup \emptyset \\
&= \{\varepsilon\}
\end{aligned}
$$

Podemos definir a classe das linguagens regulares como a classe das linguagens expressíveis por meio de expressões regulares.

2.2 Autômatos Finitos Determinísticos

Um *Autômato Finito Determinístico* (AFD) é um modelo de computação, o mais simples que estudaremos, adequado para representar sistemas computacionais simples como portas automáticas, elevadores e termostatos.

Um AFD é definido formalmente como uma 5-upla $M = \langle Q, \Sigma, \delta, q_0, F \rangle$ em que:

Q é um conjunto finito cujos elementos são chamados *estados*,

Σ é uma *alfabeto*,

$\delta : Q \times \Sigma \to Q$ é uma função de estados e símbolos em estados chamada *função de transição*,

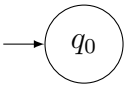
$q_0 \in Q$ é um estado chamado *inicial* e

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

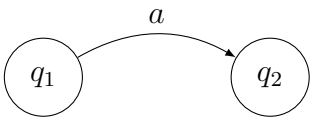
Representaremos um AFD pictoricamente por meio de um *diagrama de estados*. Nesse tipo de diagrama, cada estado $q \in Q$ é representado por uma circunferência:

Os estados finais $q \in F$ são representados por uma circunferência dupla:

O estado inicial é destacado com uma seta:



A função de transição é representada por uma seta entre os estado com uma etiqueta:



$$\delta(q_1, a) = q_2$$

Exemplo 2.2.1:

Considere o seguinte AFD:

$$\begin{aligned} M &= \langle Q, \Sigma, \delta, q_0, F \rangle \\ Q &= \{q_0, q_1, q_2\} \\ \Sigma &= \{0, 1\} \\ F &= \{q_1\} \end{aligned}$$

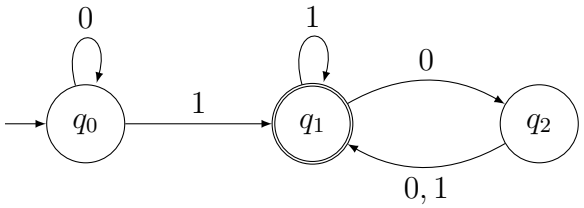
Para simplificar, normalmente escreveremos a função δ como uma tabela:

δ	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

Essa tabela indica que:

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

O seguinte diagrama de estados representa esse AFD M:



Dizemos que um AFD $M = \langle Q, \Sigma, \delta, q_o, F \rangle$ *aceita*, ou *reconhece*, uma string $\omega = a_0 a_1 \dots a_n$ se existe uma sequência de estadps $r_0, r_1, \dots, r_m$ tal que:

1. $r_0 = q_0$
2. $\delta(r_i, a_{i+1}) = r_{i+1}$
3. $r_m \in F$

Dizemos que M *consome* a string conforme passa de um estado para outro. Assim, começando pelo estado inicial, a cada passo a função de transição indica qual o próximo estado conforme consome um símbolo da string. Ao

final do processo, quando todos os símbolos foram consumidos, a string é aceita se o estado atual for final.

Escrevemos $L(M)$ para a linguagem das strings aceitas por M.

$$L(M) = \{\omega \in \Sigma^* : M \text{ aceita } \omega\}$$

Exemplo 2.2.2:

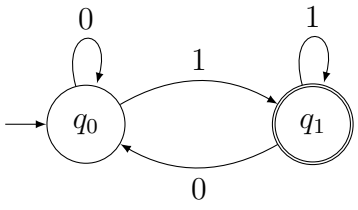
O AFD M do exemplo anterior aceita a string 1101.

1. $r_0 = q_0$
2. $r_1 = q_1$ pois $\delta(q_0, 1) = q_1$
3. $r_2 = q_1$ pois $\delta(q_1, 1) = q_1$
4. $r_3 = q_2$ pois $\delta(q_1, 0) = q_2$
5. $r_4 = q_1$ pois $\delta(q_2, 1) = q_1$
6. a string é aceita, pois $r_4 = q_1 \in F$

Exemplo 2.2.3:

$$M_1 = \langle \{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\} \rangle$$

δ	0	1
q_0	q_0	q_1
q_1	q_0	q_1

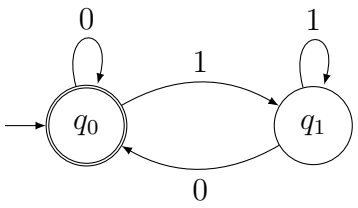


$$L(M_1) = \{\omega \in \{0, 1\}^* : \omega \text{ termina com } 1\}$$

Exemplo 2.2.4:

$$M_2 = \langle \{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_0\} \rangle$$

δ	0	1
q_0	q_0	q_1
q_1	q_0	q_1

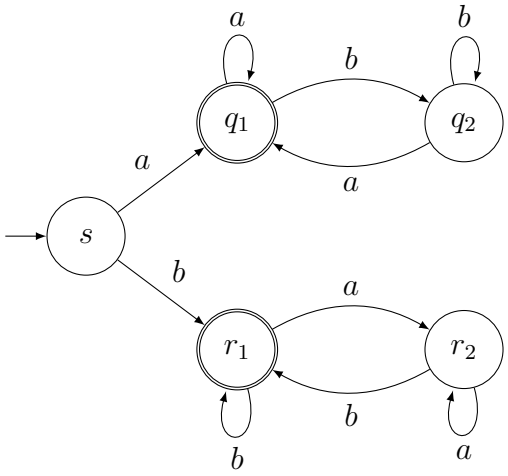


$$L(M_2) = \{\omega \in \{0,1\}^* : \omega = \varepsilon \text{ ou } \omega \text{ termina com } 0\}$$

Exemplo 2.2.5:

$$M_3 = \langle \{s, q_1, q_2, r_1, r_2\}, \{a, b\}, \delta, s, \{q_1, r_1\} \rangle$$

δ	a	b
s	q_1	r_1
q_1	q_1	q_2
q_2	q_1	q_2
r_1	r_2	r_1
r_2	r_2	r_1

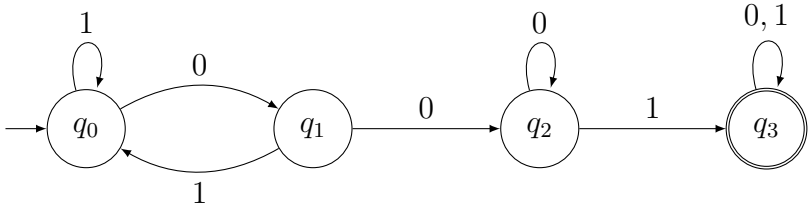


$$L(M_3) = \{\omega \in \{a,b\}^* : \omega = \text{ começa e termina com o mesmo símbolo}\}$$

Exemplo 2.2.6:

$$M_4 = \langle \{q_0, q_1, q_2, q_3\}, \{0,1\}, \delta, q_0, \{q_3\} \rangle$$

δ	0	1
q_0	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_3
q_3	q_3	q_3

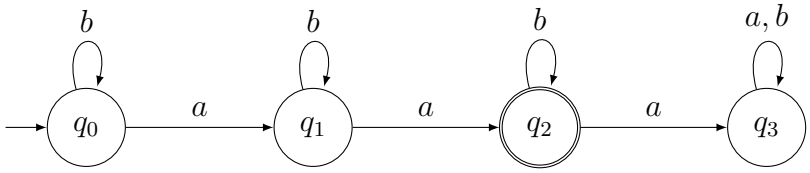


$$L(M_4) = \{\omega \in \{0,1\}^* : \omega \text{ contém a substring } 001\}$$

Exemplo 2.2.7:

$$M_5 = \langle \{q_0, q_1, q_2, q_3\}, \{a, b\}, \delta, q_0, \{q_2\} \rangle$$

δ	a	b
q_0	q_1	q_0
q_1	q_2	q_1
q_2	q_3	q_2
q_3	q_3	q_3

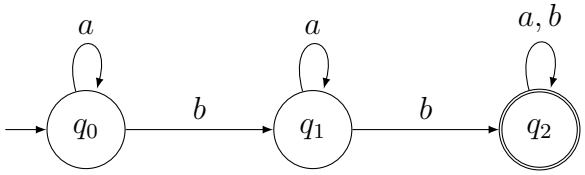


$$L(M_5) = \{\omega \in \{a, b\}^* : \omega \text{ contém exatamente dois } a\}$$

Exemplo 2.2.8:

$$M_6 = \langle \{q_0, q_1, q_2\}, \{a, b\}, \delta, q_0, \{q_2\} \rangle$$

δ	a	b
q_0	q_0	q_1
q_1	q_1	q_2
q_2	q_2	q_2



$$L(M_6) = \{\omega \in \{a, b\}^* : \omega \text{ contém pelo menos dois } b\}$$

2.3 Autômatos Finitos Não-Determinísticos

Um AFD ao ler um símbolo a em um estado q tem uma única possíbilidade de próximo estado (por isso determinístico). Na definição isso é garantido pelo fato de δ ser uma função. No diagrama de estados isso se reflete no fato de que de cada estado sai uma e uma única seta com cada símbolo do alfabeto.

Os *autômatos finitos não-determinísticos* (AFN) extendem os determinísticos em dois aspectos:

1. ao ler um símbolo em um estado o AFN possui um conjunto (possivelmente vazio) de possiblidades de próximos estados e

2. é possível mudar de estado sem consumir nenhum símbolo da entrada.

Definimos formalmente um AFN é também definido como uma 5-upla $N = \langle Q, \Sigma, \Delta, q_0, F\rangle$, mas agora $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to 2^Q$. Ou seja, a entrada da função pode ser a string vazia ϵ e sua saída é um conjunto de estados.

Representamos o diagrama de estados da mesma forma que fizemos com os AFDs, mas agora é possível que de um mesmo estado partam mais de uma seta com o mesmo síbolo, existem setas com ε e pode haver estados em que não haja seta com determinado símbolo.

Uma string é *aceita* por um AFN se existir *alguma* possibilidade de execução do autômato que consuma toda string e termine em um estado final.

Formalmente, $N = \langle Q, \Sigma, \Delta, q_0, F\rangle$ aceita uma string $\omega = y_1 \dots y_n$ onde $y_i \in \Sigma \cup \{\varepsilon\}$ se *existe* uma sequência de estados $r_0, \dots, r_m$ tal que:

1. $r_0 = q_0$

2. $r_{i+1} \in \Delta(r_i, y_{i+1})$

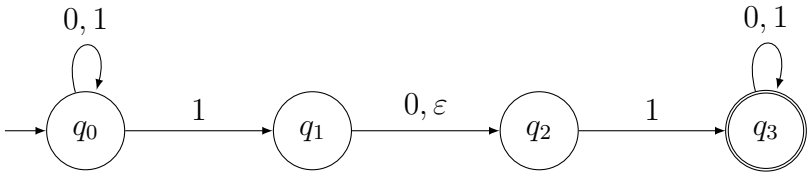
3. $r_m \in F$

Novamente, escrevemos $L(N)$ para a linguagem formada pelas strings aceitas por N.

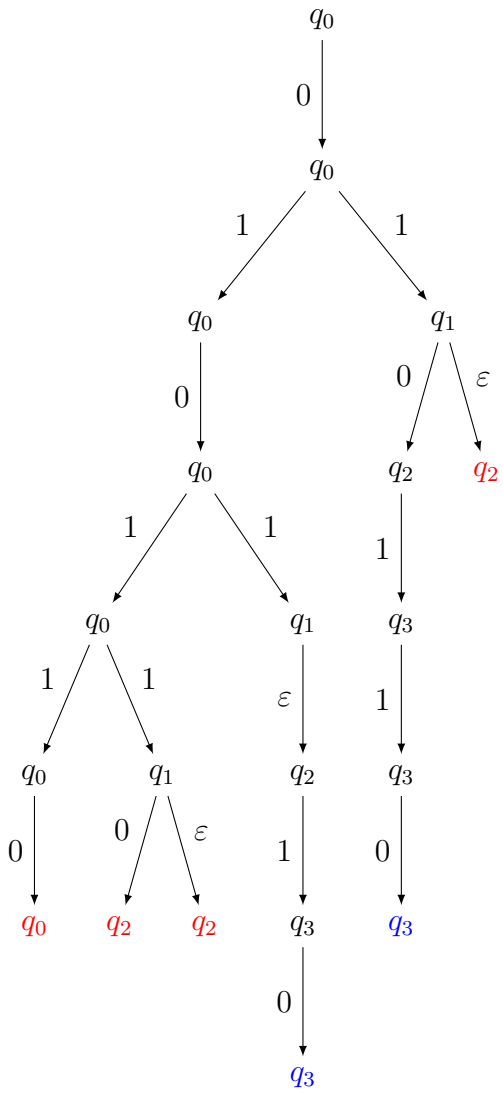
Exemplo 2.3.1:

$$N_1 = \langle \{q_0, q_1, q_2, q_3\}, \{0, 1\}, \Delta, q_0, \{q_3\}\rangle$$

Δ	0	1	ε
q_0	$\{q_0\}$	$\{q_0, q_1\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\{q_3\}$	$\emptyset$
q_3	$\{q_3\}$	$\{q_3\}$	$\emptyset$



Vamos simular as possíveis execuções desse autômato para a entrada 010110:



Cada ramo dessa árvore representa uma possível execução do autômato para a entrada dada. O autômato para apenas quando toda a entrada foi consumida. Note que nos três ramos mais a esquerda quando isso ocorre não estamos em um estado final, mas nos dois ramos seguintes sim. Basta que exita um ramo, uma possibilidade de execução, para que a string seja aceita. Assim, neste caso a string de fato é aceita, basta escolher um caminha que termine em um estado final. Por exemplo: $q_0, q_0, q_1, q_1, q_2, q_3, q_3, q_3$

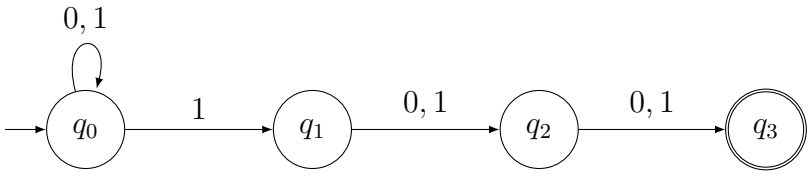
satisfaz a definição para a string $0101\varepsilon 10 = 010110$.

$$L(N_1) = \{\omega \in \{0,1\}^* : \omega \text{ contém 101 ou 11 como substring}\}$$

Exemplo 2.3.2:

$$N_2 = \langle \{q_0, q_1, q_2, q_3\}, \{0,1\}, \Delta, q_0, \{q_3\} \rangle$$

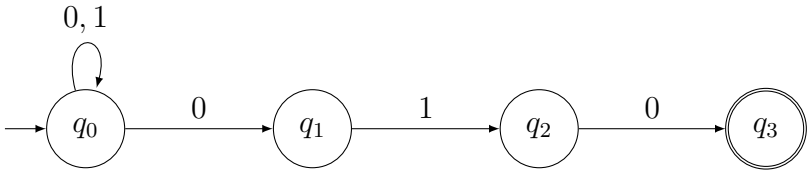
Δ	0	1	ε
q_0	$\{q_0\}$	$\{q_0, q_1\}$	$\emptyset$
q_1	$\{q_2\}$	$\{q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\{q_3\}$	$\emptyset$
q_3	$\emptyset$	$\emptyset$	$\emptyset$



$$L(N_2) = \{\omega \in \{0,1\}^* : \omega \text{ contém 1 na antepenúltima posição}\}$$

Exemplo 2.3.3:

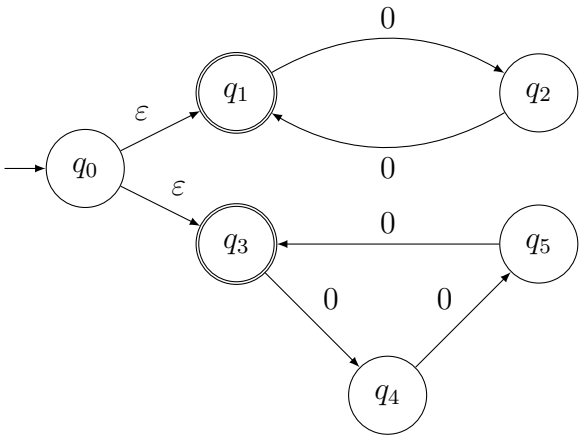
A partir daqui vamos apresentar os autômatos apenas por seu diagrama. O seguinte é o diagrama de N_3:



$$L(N_3) = \{\omega \in \{0,1\}^* : |\omega| \text{ termina com } 010\}$$

Exemplo 2.3.4:

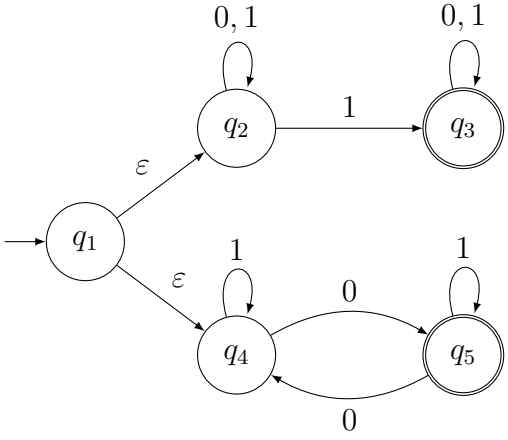
O seguinte é o diagrama de N_4:



$$L(N_4) = \{\omega \in \{0\}^* : |\omega| \text{ é múltiplo de 2 ou de 3}\}$$

Exemplo 2.3.5:

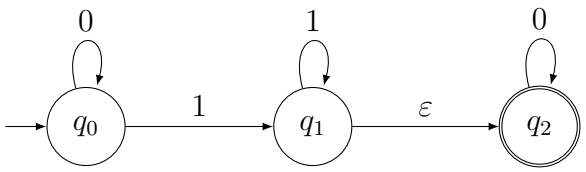
O seguinte é o diagrama de N_5:



$$L(N_5) = \{\omega \in \{0,1\}^* : \omega \text{ contém 1 ou um número ímpar de 0s}\}$$

Exemplo 2.3.6:

O seguinte é o diagrama de N_6:



$$L(N_6) = L(0^\star 11^\star 0^\star)$$

2.4 AFD ≡ AFN

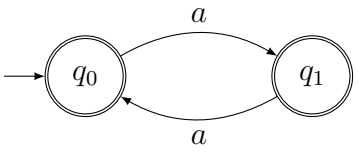
Vimos nas últimas seções dois modelos computacionais. O primeiro é mais próximo da descrição de dispositivos simples, mas sua descrição em termos de diagramas possui diversas limitações. O segundo modelo é mais flexível e mais fácil de descrever pictoricamente.

A pergunta que procuraremos responder nessa seção é se algum desses modelos é mais expressivo que o outro. Ou seja, será que algum deles é capaz de resolver problemas, reconhecer linguagens, que o outro não consegue. A resposta será negativa. De fato, ambos os modelos são equivalentes em um sentido bastante preciso. Comecemos então com essa definição.

Dois autômatos M_1 e M_2 são ditos *equivalentes* (escrevemos $M_1 \sim M_2$) se reconhecem a mesma linguagem, ou seja, se $L(M_1) = L(M_2)$.

Exemplo 2.4.1:

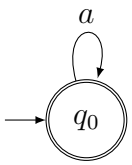
$$M_1 = \langle \{q_0, q_1\}, \{a\}, \delta, q_0, \{q_0, q_1\} \rangle$$



δ	a
q_0	q_1
q_1	q_0

$$L(M_1) = L(a^\star)$$

$$M_2 = \langle \{q_0\}, \{a\}, \delta, q_0, \{q_0\} \rangle$$



δ	a
q_0	q_0

$$L(M_2) = L(a^\star)$$

$$M_1 \sim M_2$$

Primeiramente devemos mostrar que AFNs são uma extensão dos AFDs. Essa parte coincide com nossa intuição uma vez que todo diagrama de um AFD é também um diagrama para um AFN (o contrário não vale!). Vamos formalizar essa ideia no seguinte teorema:

Teorema 2.4.2. *Se M é um AFD então existe um AFN N tal que $M \sim N$*

Demonstração. Seja $M = \langle Q, \Sigma, \delta, q_o, F \rangle$ um AFD qualquer. Considere o AFN $N = \langle Q, \Sigma, \Delta, q_0, F \rangle$ em que:

$$\Delta(a, q) = \begin{cases} \{q'\} & \text{se } a \in \Sigma \text{ e } \delta(a, q) = q' \\ \emptyset & \text{se } a = \varepsilon \end{cases}$$

Note que os diagramas de M e de N são idênticos e segue trivialmente que $M \sim N$ □

A demonstração da outra equivalência exige mais cuidado e faremos em duas partes. Primeiro vamos supor que não fosse permitido mudar de estado sem consumir símbolos em um AFN. Ou seja, suponhamos que não seja permitido usar ε nas setas no diagrama de estados. Vamos mostrar que é possível construir um AFD equivalente a esse AFN. A ideia da construção é que cada estado no AFD simula um conjunto de estados no AFN. Conforme consumimos a string nesse AFD o estado atual representa o conjunto de todos os estados possíveis no AFN ao consumir os mesmos símbolos.

Lema 2.4.3. *Seja N um AFN em que não é permitido mudar de estados sem consumir símbolos, então existe um AFD M tal que $N \sim M$.*

Demonstração. Não faremos a demonstração completa, apenas apresentaremos a construção e posteriormente mostraremos alguns exemplos.

Seja $N = \langle Q, \Sigma, \Delta, q_0, F\rangle$, construiremos $M = \langle Q', \Sigma', \delta, q_0', F'\rangle$ da seguinte forma:

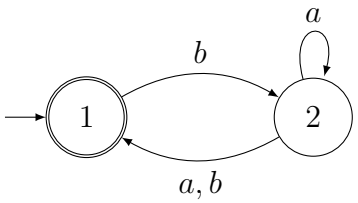
1. $Q' = 2^Q$
2. $\Sigma' = \Sigma$
3. $q_0' = \{q_0\}$
4. $F' = \{R \in Q' : R \cap F \neq \emptyset\}$
5. $\delta(R, a) = \bigcup_{r \in R} \Delta(r, a)$

□

Exemplo 2.4.4:

$$N_1 = \langle \{1, 2\}, \{a, b\}, \Delta, 1, \{1\}\rangle$$

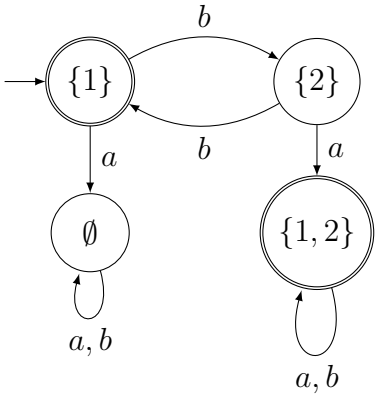
Δ	a	b
1	$\emptyset$	$\{2\}$
2	$\{1, 2\}$	$\{1\}$



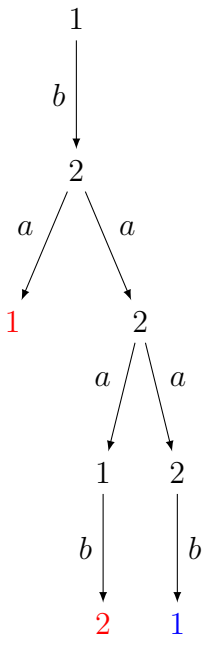
Seguindo a construção que vimos no teorema anterior:

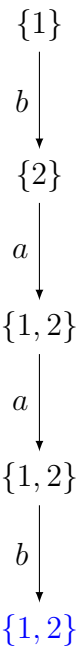
$$M_1 = \langle \{\emptyset, \{1\}, \{2\}, \{1, 2\}\}, \{a, b\}, \delta, \{1\}, \{\{1\}, \{1, 2\}\}\rangle$$

δ	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{1\}$	$\emptyset$	$\{2\}$
$\{2\}$	$\{1,2\}$	$\{1\}$
$\{1,2\}$	$\{1,2\}$	$\{1,2\}$



Para terminar vamos simular nos dois a leitura da string *baab*:





Para completar nossa prova precisamos lidar com as setas rotuladas por ε. O que faremos nesse caso será incluir no estado atual todos os estados que conseguimos alcançar sem precisar consumir símbolos.

Teorema 2.4.5. *Para todo AFN N existe um AFD M tal que $N \sim M$.*

Demonstração. Sejam N e M como definidos na demonstração do lema anterior Considere a seguinte função $E : Q' \to Q'$:

$$E(R) = \{q \in Q : \exists q' \in R(q' \overset{\varepsilon}{\rightsquigarrow} q)\}$$

Ou seja, existe um caminho de algum $q' \in R$ até q passando apenas por setas com etiqueta ε.

Vamos agora atualizar a definição de M em dois pontos:

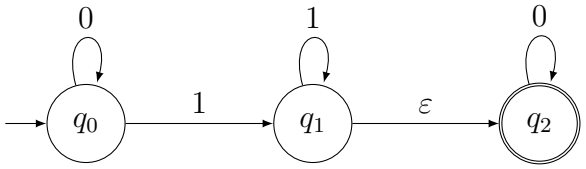
3'. $q_0 = E(\{q_0\})$

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

□

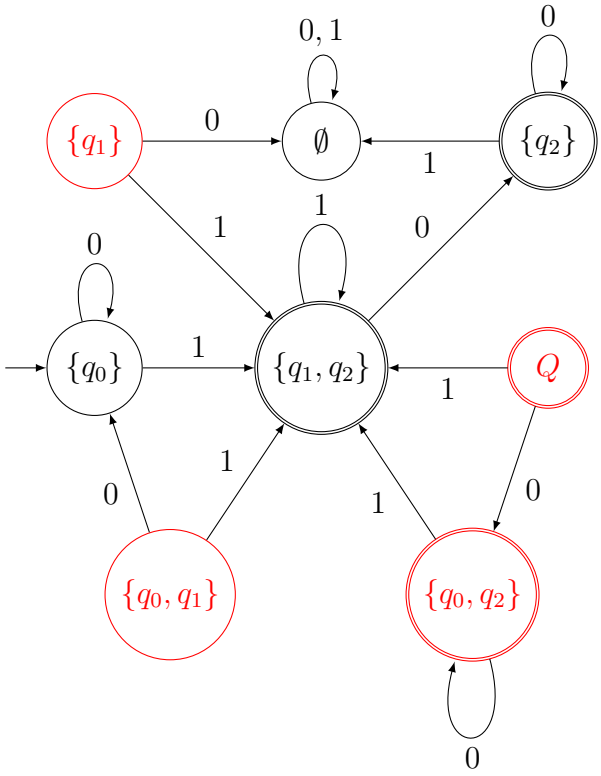
Exemplo 2.4.6:

Considere o AFN N_2 representado pelo seguinte diagrama de estados:

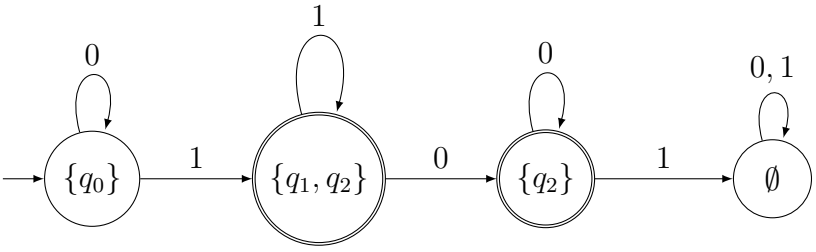


$$L(N_2) = L(0^\star 1 1^\star 0^\star)$$

Usando a construção dos teoremas anteriores produzimos o seguinte AFD M_2:



A construção que vimos possui sempre uma quantidade exponencialmente maior de estados, porém, alguns deles podem ser supérfluos. Note que os estados em vermelho não são alcançáveis a partir do estado inicial, logo, podem ser omitidos (formalmente, o autômato gerado ao se omitir esses estados é equivalente a esse).



Removendo os estados supérfluos é fácil ver que:

$$L(M_2) = L(N_2) = L(0^\star 11^\star 0^\star)$$

Portanto $M_2 \sim N_2$.

2.5 Linguagens Regulares são Reconhecíveis por AFNs

Na seção anterior vimos que AFDs e AFNs são capazes de reconhecer a mesma classe de linguagens. Nesta seção começaremos a investigar que essa classe é exatamente a classe das linguagens regulares.

Mostraremos nessa seção que toda linguagem regular é reconhecível por algum AFN e, consequentemente, por algum AFD. Para tanto, temos que mostrar que a classe das linguagens reconhecíveis por AFNs é fechada por $*$, $\cup$, $\circ$ e que contém $\{\{a\} : a \in \Sigma\}$.

Lema 2.5.1. *Se $A, B \subseteq \Sigma^*$ são reconhecíveis por AFNs então $A \cup B$ também é. Ou seja, a classe das linguagens reconhecíveis por AFNs é fechada por união.*

Demonstração. A hipótese garante que existem AFNs N_1 e N_2 tal que $L(N_1) = A$ e $L(N_2) = B$. Vamos construir a partir de N_1 e N_2 um AFN N tal que $L(N) = L(N_1) \cup L(N_2)$

Sejam:

$$\begin{aligned} N_1 &= \langle Q_1, \Sigma, \Delta_1, q_1, F_1 \rangle \\ N_2 &= \langle Q_2, \Sigma, \Delta_2, q_1', F_2 \rangle \end{aligned}$$

Vamos construir $N = \langle Q, \Sigma, \Delta, q_0, F \rangle$:

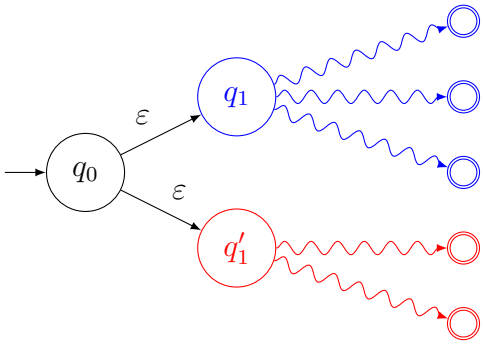
$Q = Q_1 \cup Q_2 \cup \{q_0\}$

$F = F_1 \cup F_2$

$$\Delta(q,a) = \begin{cases} \Delta_1(q,a) & \text{se} \quad q \in Q_1 \\ \Delta_2(q,a) & \text{se} \quad q \in Q_2 \\ \{q_1, q_1'\} & \text{se} \quad q = q_0 \text{ e } a = \varepsilon \\ \emptyset & \text{se} \quad q = q_0 \text{ e } a \neq \varepsilon \end{cases}$$

N_1 N_2

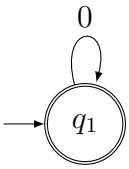
N

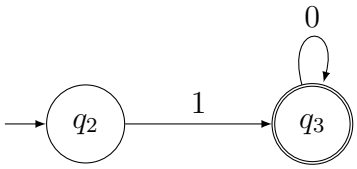


□

Exemplo 2.5.2:

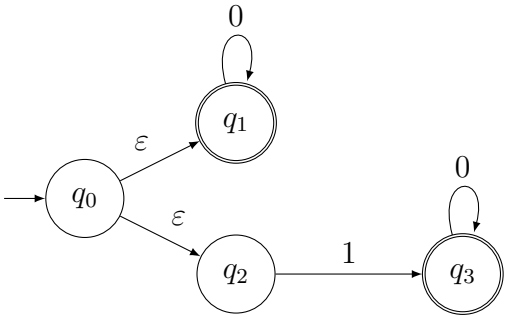
N_1 N_2





$L(N_1) = L(0^\star)$ $L(N_2) = L(10^\star)$

N



$$L(N) = L(N_1) \cup L(N_2) = L(0^\star) \cup L(10^\star) = L(0^\star \cup 10^\star)$$

Lema 2.5.3. *Se $A, B \subseteq \Sigma^*$ são reconhecíveis por AFNs então $A \circ B$ também é. Ou seja, a classe das linguagens reconhecíveis por AFNs é fechada por concatenação.*

Demonstração. A hipótese garante que existem AFNs N_1 e N_2 tal que $L(N_1) = A$ e $L(N_2) = B$. Vamos construir a partir de N_1 e N_2 um AFN N tal que $L(N) = L(N_1) \circ L(N_2)$

Sejam:

$$\begin{aligned} N_1 &= \langle Q_1, \Sigma, \Delta_1, q_1, F_1 \rangle \\ N_2 &= \langle Q_2, \Sigma, \Delta_2, q_1', F_2 \rangle \end{aligned}$$

Vamos construir $N = \langle Q, \Sigma, \Delta, q_0, F \rangle$:

$Q = Q_1 \cup Q_2$

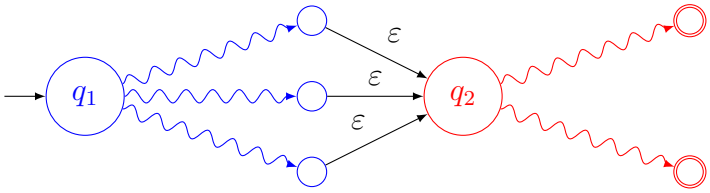
$q_0 = q_1$

$F = F_2$

$$\Delta(q,a) = \begin{cases} \Delta_1(q,a) & \text{se} \quad q \in Q_1 \text{ e } q \notin F_1 \\ \Delta_1(q,a) & \text{se} \quad q \in F_1 \text{ e } a \neq \varepsilon \\ \Delta_1(q,a) \cup \{q_1'\} & \text{se} \quad q \in F_1 \text{ e } a = \varepsilon \\ \Delta_2(q,a) & \text{se} \quad q \in Q_2 \end{cases}$$

N_1 N_2

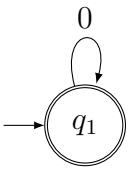
N

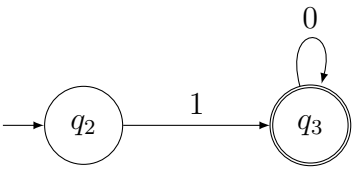


□

Exemplo 2.5.4:

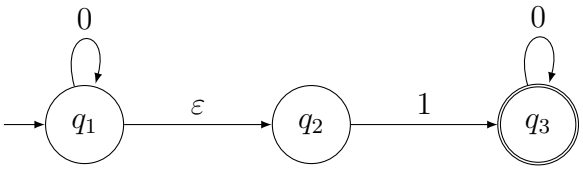
N_1 N_2





$L(N_1) = L(0^\star)$ $L(N_2) = L(10^\star)$

N



$$L(N) = L(N_1) \circ L(N_2) = L(0^\star) \circ L(10^\star) = L(0^\star 10^\star)$$

Lema 2.5.5. *Se $A \subseteq \Sigma^*$ é reconhecível por um AFN então A^* também é. Ou seja, a classe das linguagens reconhecíveis por AFNs é fechada por estrela de Kleene.*

Demonstração. A hipótese garante que existe AFNs N_1 tal que $L(N_1) = A$. Vamos construir a partir de N_1 um AFN N tal que $L(N) = L(N_1)^*$

Seja:

$$N_1 \quad = \quad \langle Q_1, \Sigma, \Delta_1, q_1, F_1 \rangle$$

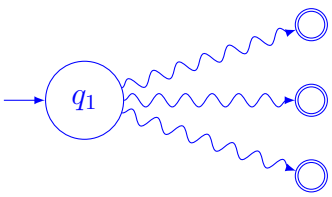
Vamos construir $N = \langle Q, \Sigma, \Delta, q_0, F \rangle$:

$Q = Q_1 \cup \{q_0\}$

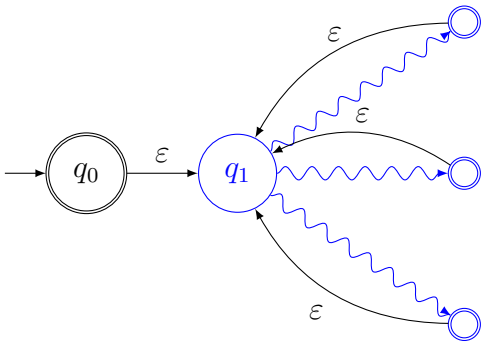
$F = F_1 \cup \{q_0\}$

$$\Delta(q,a) = \begin{cases} \Delta_1(q,a) & \text{se} \quad q \in Q_1 \text{ e } q \notin F_1 \\ \Delta_1(q,a) & \text{se} \quad q \in F_1 \text{ e } a \neq \varepsilon \\ \Delta_1(q,a) \cup \{q_1\} & \text{se} \quad q \in F_1 \text{ e } a = \varepsilon \\ \{q_1\} & \text{se} \quad q = q_0 \text{ e } a = \varepsilon \\ \emptyset & \text{se} \quad q = q_0 \text{ e } a \neq \varepsilon \end{cases}$$

N_1



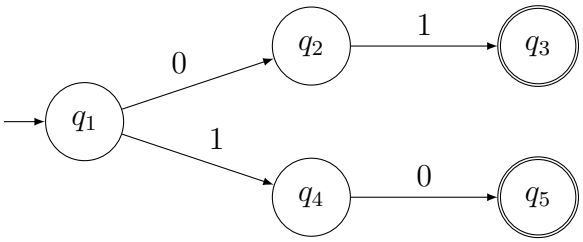
N



□

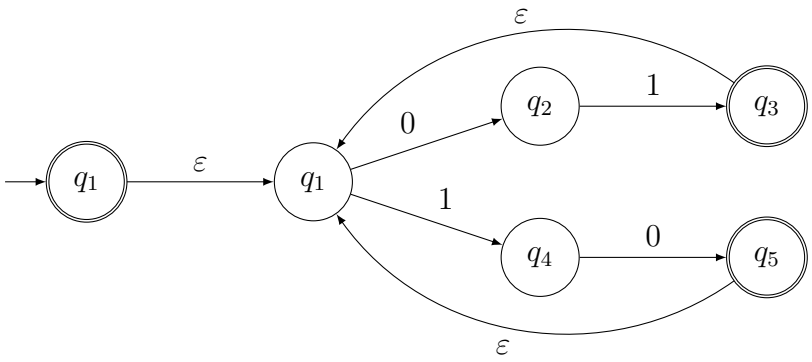
Exemplo 2.5.6:

N_1



$$L(N_1) = L(01 \cup 10)$$

N

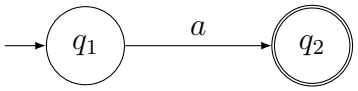


$$L(N) = L(N_1)^* = L((01 \cup 10)^\star)$$

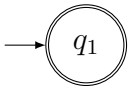
Teorema 2.5.7. *Toda linguagem regular é reconhecível por um AFN. Ou seja, a classe das linguagens reconhecíveis por AFNs contém a classe das linguagem regulares.*

Demonstração. Pelos lemas anteriores sabemos que a classe das linguagens reconhecíveis por AFNs é fechada por união, concatenação e estrela de Kleene. Para completar a prova mostraremos que a classe contém as linguagens $\{a\}$ para todo $a \in \Sigma$, $\{\varepsilon\}$ e $\emptyset$. Os seguintes AFNs fazem exatamente isso:

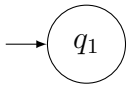
N_a



N_ε



$N_\emptyset$

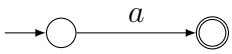


□

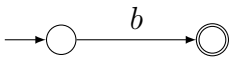
Exemplo 2.5.8:

Construiremos o autômato que reconhece $L((ab \cup a)^\star)$ usando o método visto neste capítulo:

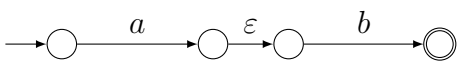
a



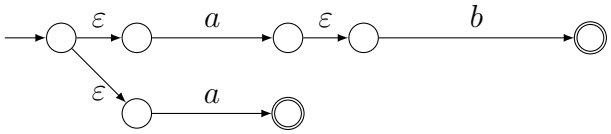
b



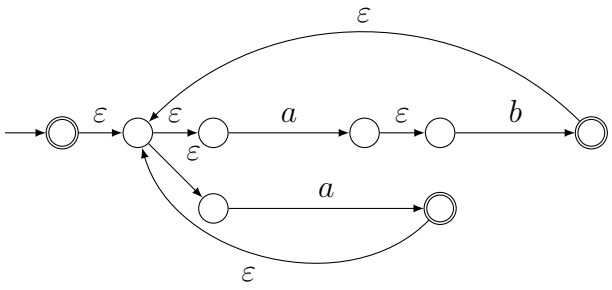
ab



$ab \cup a$



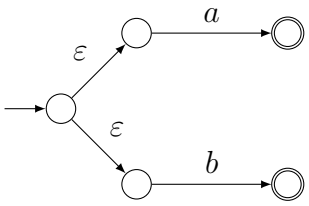
$(ab \cup a)^\star$



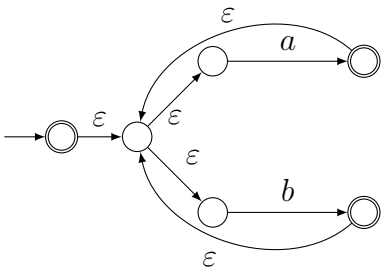
Exemplo 2.5.9:

Para concluir, construiremos o autômato que reconhece $L((a \cup b)^\star aba)$ usando o método visto neste capítulo:

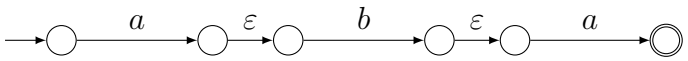
$a \cup b$



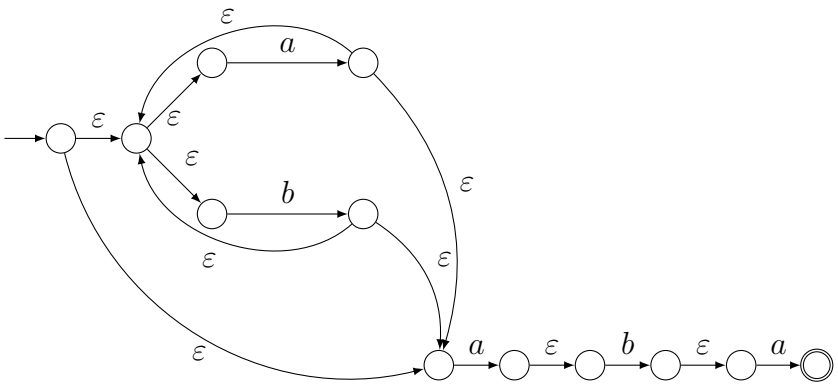
$(a \cup b)^\star$



aba



$(a \cup b)^\star aba$



2.6 Linguagens Reconhecíveis por AFNs são Regulares

Na última seção vimos que toda linguagem regular pode ser reconhecida por um autômato finito. Veremos agora a relação recíproca, a saber, que toda linguagem reconhecível por um autômato é regular. Para isso comecemos com a seguinte definição. Um *Autômato Finito Generalizado* (AFG) é uma 5-upla $\langle Q, \Sigma, \delta, q_i, q_f \rangle$ em que:

Q é um conjunto de estados,

Σ é um alfabeto,

$q_i \in Q$ é um estado chamado de *inicial*,

$q_f \in Q$ é um estado chamado *final* e

$\delta : (Q - \{q_f\}) \times (Q - \{q_i\}) \to R$ em que R é o conjunto das expressões regulares sobre Σ.

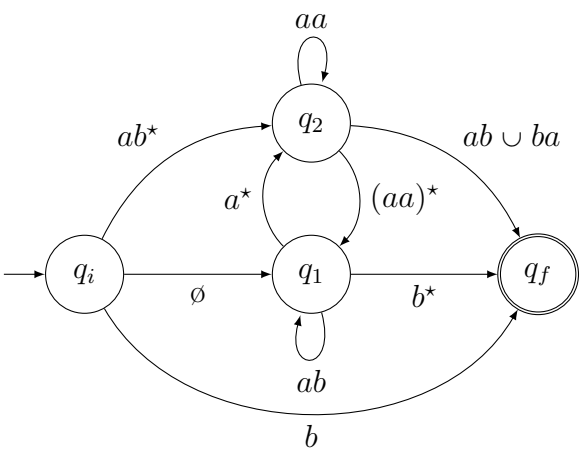
Em outras palavras, um AFG é como um AFN com um único estado final e aonde as transições são etiquetadas não com um símbolo, mas com uma expressão regular. Um AFG *aceita* uma string $\omega \in \Sigma^*$ se $\omega = \omega_1 \cdot \omega_2 \ldots \omega_k$ e existe uma sequência de estados $q_0, \ldots, q_k$ tal que:

1. $q_0 = q_i$
2. $q_k = q_f$
3. $\omega_j \in L(R_j)$ onde $R_j = \delta(q_{j-1}, q_j)$ para $0 < j \leq k$

Exemplo 2.6.1:

$$G = \langle \{q_i, q_1, q_2, q_f\}, \{a, b\}, \delta, q_i, q_f \rangle$$

δ	q_i	q_1	q_2	q_f
q_i	$\times$	$\emptyset$	$ab^\star$	b
q_1	$\times$	ab	$a^\star$	$b^\star$
q_2	$\times$	$(aa)^\star$	aa	$ab \cup ba$
q_f	$\times$	$\times$	$\times$	$\times$



$$aba, aab, abbbab, aaab, b \in L(G)$$

Omitiremos a partir de agora as setas com etiqueta ∅.

Lema 2.6.2. *Para todo AFD M existe um AFG G equivalente.*

Demonstração. Seja $M = \langle Q, \Sigma, \delta, q_o, F\rangle$. Construímos $G = \langle Q \cup \{q_i, q_f\}, \Sigma, \delta', q_i, q_f\rangle$ e para todo $q_j \in Q - \{q_f\}$ e $q_k \in Q - \{q_i\}$ temos:

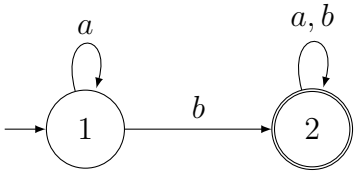
$$\delta'(q_j, q_k) = \begin{cases} \epsilon & \text{se} \quad q_j = q_i \text{ e } q_k = q_0 \\ \epsilon & \text{se} \quad q_j \in F \text{ e } q_k = q_f \\ \bigcup a_i & \text{se} \quad \delta(a_i, q_j) = q_k \\ \emptyset & \text{se} \quad \nexists a \in \Sigma \text{ com } \delta(a, q_j) = q_k \end{cases}$$

□

Exemplo 2.6.3:

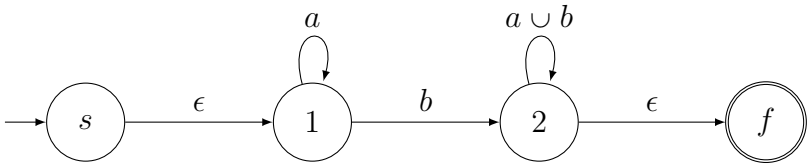
$$M = \langle \{1, 2\}, \{a, b\}, \delta, 1, \{2\}\rangle$$

δ	a	b
1	1	2
2	2	2



$$G = \langle \{1, 2, s, f\}, \{a, b\}, \delta', s, f\rangle$$

δ'	s	1	2	f
s	×	ϵ	$\emptyset$	$\emptyset$
1	×	a	b	$\emptyset$
2	×	$\emptyset$	$a \cup b$	ϵ
f	×	×	×	×



Lema 2.6.4. *Se G é um AFG com $k > 2$ estados, então existe um AFG G' com $k - 1$ estados tal que $G \sim G'$.*

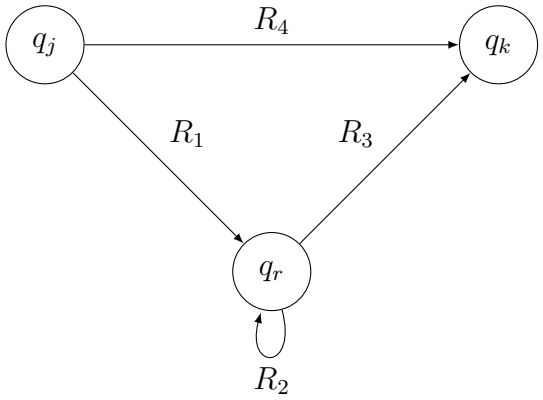
Demonstração. Seja $G = \langle Q, \Sigma, \delta, q_i, q_f \rangle$ um AFG e $q_r \in Q - \{q_i, q_f\}$, Construiremos $G' = \langle Q', \Sigma, \delta', q_i, q_f \rangle$ da seguinte forma:

$$Q' = Q - \{q_r\}$$

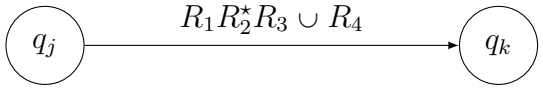
$\delta'(q_j, q_k) = R_1 R_2^\star R_3 \cup R_4$ aonde:

$$R_1 = \delta(q_j, q_r)$$
$$R_2 = \delta(q_r, q_r)$$
$$R_3 = \delta(q_r, q_k)$$
$$R_4 = \delta(q_j, q_k)$$

Diagramaticamente, partimos de um diagrama com o seguinte formato:



E depois de remover q_r chegamos nos seguinte:



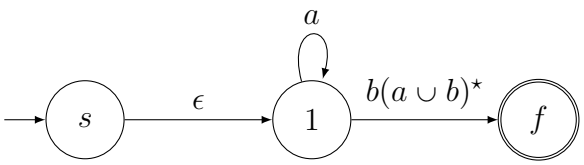
□

Exemplo 2.6.5:

Considere o AFG G do Exemplo 2.6.3. Vamos remover o estado 2 seguindo a construção do lema anterior.

$$G' = \langle \{s, 1, f\}, \{a, b\}, \delta', s, f \rangle$$

δ'	s	1	f
s	$\times$	ϵ	$\emptyset$
1	$\times$	a	$b(a \cup b)^\star$
f	$\times$	$\times$	$\times$



Teorema 2.6.6. *Toda linguagem reconhecível por AFD é regular.*

Demonstração. Seja A uma linguagem reconhecível por um AFD M i.e. $L(M) = A$. Pelo Lema 2.6.2 existe um AFG G tal que $M \sim G$. Além disso, examinando a construção do Lema 2.6.4 temos que se o número de estados de M é k então o número de estado de G é $k + 2$.

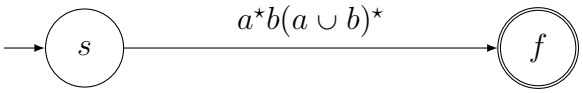
Pelo lema 2.6.4 existe G_1 com um estado a menos ($k + 1$ estados) tal que $G_1 \sim G$. Aplicando o lema k vezes obtemos $G_k \sim G$ com exatamente 2 estados: q_i e q_f. É claro que $L(G_k) = L(R)$ aonde $\delta_{G_k}(q_i, q_f) = R$. Como $G_k \sim G$, temos $L(G) = L(H) = L(R)$.

$$L(M) = L(G) = L(G_1) = \cdots = L(G_k) = L(R)$$

□

Exemplo 2.6.7:

Considere o AFD M do Exemplo 2.6.3. Mostramos que ele é equivalente a um AFG e, em no Exemplo 2.6.5 mostramos um AFG equivalente com 3 estados. Removendo mais um estado ficamos com o seguinte diagrama:

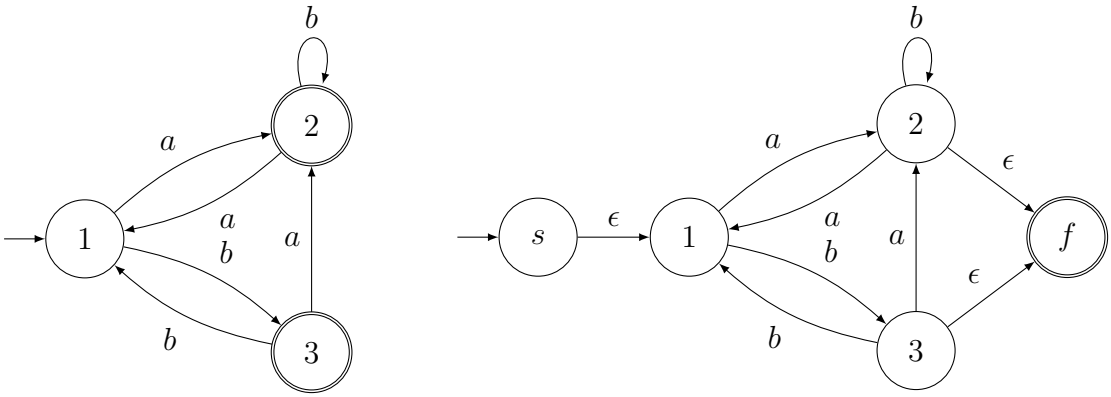


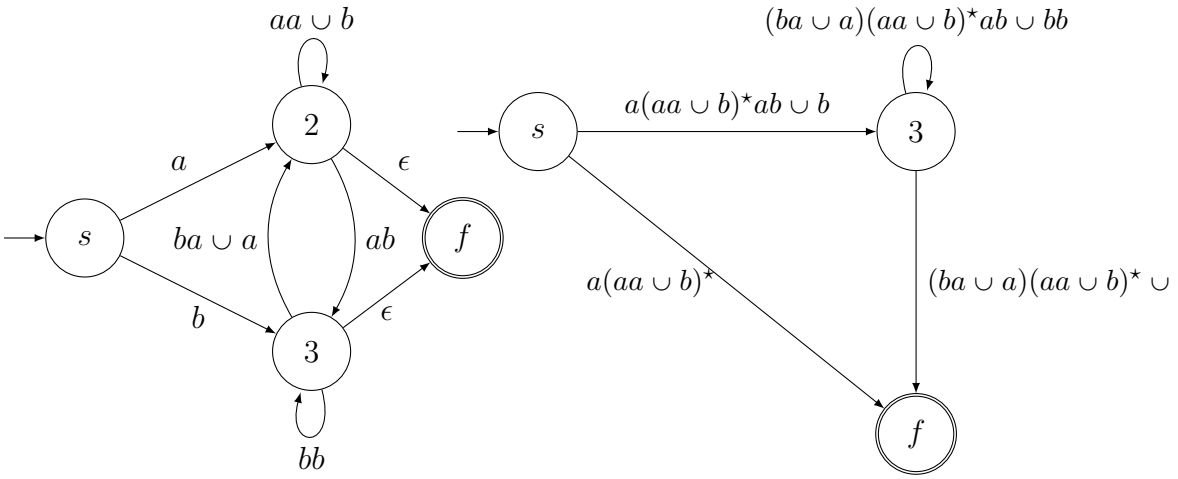
Portanto temos que:

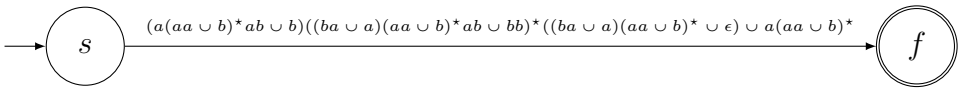
$$L(M) = L(a^\star b(a \cup b)^\star)$$

Exemplo 2.6.8:

Considere o seguinte AFD:







2.7 Linguagens Não-Regulares

Neste capítulo vimos dois modelos computacionais: AFDs e AFNs. Ambos reconhecem a mesma classe de linguagens, a saber, as linguagens regulares. Para completar este capítulo veremos exemplos de linguagens que não são regulares.

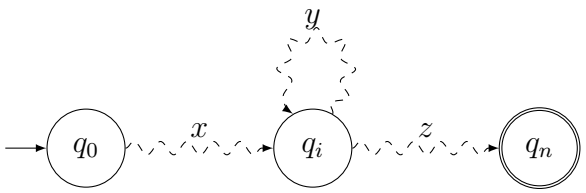
Lema 2.7.1 (Lema do Bombeamento)**.** *Se A é uma linguagem regular então exite um número p (chamado* comprimento do bombeamento*) tal que se $\omega \in A$ e $|\omega| \geq p$ então $\omega = x \cdot y \cdot z$ onde:*

1. *$x \cdot y^i \cdot z \in A$ para todo $i \leq 0$,*
2. *$|y| > 0$ e*
3. *$|x \cdot y| \leq p$*

Demonstração. Como A é regular, existe AFD $M = \langle Q, \Sigma, \delta, q_0, F\rangle$ tal que $L(M) = A$. Seja $p = |Q| - 1$ e seja $\omega \in A$ uma string tal que $|\omega| \geq p$.

Como $\omega = a_1 a_2 \ldots a_n \in A = L(M)$ existe uma sequência de estados $q_0, \ldots, q_n$ onde $q_n \in F$ e $q_{i+1} = \delta(q_i, a_{i+1})$. Como $|\omega| = n \geq p$, pelo *princípio da casa dos pombos*[1] existe pelo menos um estado que se repete na sequência.

Seja q_i o primeiro estado que se repete na sequência. Temos então a seguinte situação:



[1]O princípio da casa dos pombos garante que se há n pombos e p casas com $n > p$ então com certeza pelo menos uma casa terá mais de um pombo.

Ou seja, a sequência $q_0, \ldots, q_i$ reconhece a string x, a sequência $q_i, \ldots, q_i$ reconhece y e a sequência $q_i, \ldots, q_n$ reconhece z.

Portanto, ω pode ser dividido em três partes $\omega = x \cdot y \cdot z$ em que $|y| > 0$. Além disso, como q_i é o primeiro estado que se repete, temos que $|x \cdot y| \leq p$. Por fim, como ilustrado, o autômato reconhece y um número arbitrário de vezes. Ou seja, $x \cdot y^i \cdot z \in A$ para todo $i = 0, 1, 2, \ldots$ □

Exemplo 2.7.2:

Vamos mostrar que $A = \{0^n 1^n : n \geq 0\}$ não é regular.

Seja p o comprimento do bombeamento e $\omega = 0^p 1^p \in A$. Se A fosse regular, pelo lema do bombeamento, ω poderia ser dividida em três partes $\omega = xyz$ e para todo $i = 0, 1, \ldots$ teríamos $xy^i z \in A$.

Se y é formada só por 0s ou só por 1s então $xyyz$ possuiria um número diferente de 0s e 1s e, portanto, $xyyz \notin A$. Se y possuir 0s e 1s então $xyyz$ conterá pelo menos um 0 entre dois 1s e, portanto, $xyyz \notin A$.

Logo A não pode ser regular.

Exemplo 2.7.3:

Mostraremos que $A = \{\omega : \omega$ tem o mesmo número de 0s e 1s$\}$ não é regular.

Seja p o comprimento do bombeamento e $\omega = 0^p 1^p \in A$. Se A fosse regular poderíamos escrever $\omega = xyz$ com $|xy| \leq p$ e $xy^i z \in A$ para todo $i \geq 0$.

Como $|xy| \leq p$, por definição, xy contém apenas 0s. Neste caso, $xyyz$ deve conter mais 0 do que 1s. Logo $xyyz \notin A$.

Exemplo 2.7.4:

Vamos mostrar que $A = \{\omega\omega : \omega \in \{0, 1\}^*\}$ não é regular.

Seja p o comprimento do bombeamento e $\omega = 0^p 1 0^p 1 \in A$. Dividimos $\omega = xyz$ com $|xy| \leq p$ então y contém apenas 0s e $xyyz = 0^p 0 \ldots 0 1 0^p 1 \notin A$.

Exemplo 2.7.5:

Por fim, vamos mostrar que $A = \{0^i 1^j : i > j\}$ não é regular.

Seja p o comprimento do bombeamento e $\omega = 0^{p+1}1^p \in A$. Tomamos $\omega = xyz$ com $|xy| \leq p$ e $|y| > 0$. Portanto, y contém apenas 0s e $xy^0z = xz \notin A$.

Neste capítulo estudamos a classe das linguagens regulares e dois modelos de computação, autômatos finitos determinísticos e não-determinísticos. Provamos que esses dois modelos são equivalentes e que a classe das linguagens que eles reconhecem coincide com a classe das linguagens regulares. Terminamos o capítulo vendo exemplos de linguagens que não são regulares.

Capítulo 3

Autômatos de Pilha

Estudamos até exaustivamente uma classe de linguagens, as regulares. Apresentamos essa classe de maneira declarativa por meio de expressões regulares e imperativa com dois modelos de computação: autômatos finitos determinísticos e não-determinísticos que vimos serem equivalentes. No fim do capítulo mostramos que nem toda linguagem é regular.

Neste capítulo estudaremos uma classe de linguagens mais completa, as linguagens livres de contexto. Como as linguagens regulares, apresentaremos tais linguagens de maneira declarativa por meio de gramáticas livres de contexto e imperativa por meio dos autômatos com pilha.

3.1 Linguagens Livres de Contexto

Uma *Gramática Livre de Contexto* (GLC) é uma 4-upla $\langle V, \Sigma, R, S\rangle$ em que:

> V é um conjunto finito cujos elementos são chamados *variáveis*,
>
> Σ é um conjunto finito disjunto de V (i.e. $\Sigma \cap V \neq \emptyset$) cujos elementos são chamados *terminais*,
>
> R é um conjunto finito de *regras* e cada regra é da forma $v_1 \to v_2 \dots v_n$ onde $v_1 \in V$ e $v_i \in V \cup \Sigma$ para $i = 2, \dots, n$ e
>
> $S \in V$ é uma variável chamada *inicial*.

Se u, v e w strings sobre o alfabeto $V \cup \Sigma$ e $A \to w$ é uma regra da gramática, dizemos que uAv *origina* uwv (escrevemos $uAv \Rightarrow uwv$). Dizemos

que u *deriva* v (escrevemos $u \Rightarrow^* v$) se $u = v$ ou existe uma sequência $u_1, \ldots, u_k$ para $k \geq 0$ em que:

$$u \Rightarrow u_1 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$$

A *linguagem associada à gramatica* $G = \langle V, \Sigma, R, S \rangle$, ou simplesmente a linguagem de G é $\{\omega \in \Sigma^* : S \Rightarrow^* \omega\}$.

Se uma linguagem A possui uma gramatica livre de contexto associada a ele então A é chamada *linguagem livre de contexto*.

Exemplo 3.1.1:

Considere a seguinte GLC $G = \langle V, \Sigma, R, S \rangle$:

- $V = \{S\}$
- $\Sigma = \{0, 1\}$
- $R = \{S \to \varepsilon, S \to 0S1\}$

01 pertence a linguagem dessa gramática:

$$\begin{aligned} S &\Rightarrow 0S1 \\ &\Rightarrow 0\varepsilon 1 = 01 \end{aligned}$$

000111 pertence a linguagem dessa gramática:

$$\begin{aligned} S &\Rightarrow 0S1 \\ &\Rightarrow 00S11 \\ &\Rightarrow 000S111 \\ &\Rightarrow 000\varepsilon 111 \\ &\Rightarrow 000111 \end{aligned}$$

Não é difícil notar que a linguagem dessa gramática é:

$$\{0^n 1^n : n \geq 0\}$$

Para apresentar as próximas gramáticas, usaremos a seguinte abreviação:

$$\{A \to w, A \to u, A \to v\}$$

Será substituído simplesmente por:

$$A \to w|u|v$$

Exemplo 3.1.2:

$G = \langle \{S\}, \{0,1\}, R, S \rangle$ em que R é:

$$S \to SS|0|1$$

01 pertence à linguagem de G:

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow 0S \\ &\Rightarrow 01 \end{aligned}$$

0101 pertence à linguagem de G:

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow 0S \\ &\Rightarrow 0SS \\ &\Rightarrow 0S0 \\ &\Rightarrow 010 \end{aligned}$$

Exemplo 3.1.3:

$$G = \langle \{S\}, \{0,1,\star,\cup,\epsilon,\emptyset\}, R, S \rangle$$
$$S \to 0|1|\epsilon|\emptyset|SS|S \cup S|S^\star$$

Vamos mostrar que $10 \cup 1^\star \in L(G)$

$$\begin{aligned} S &\Rightarrow S \cup S \\ &\Rightarrow SS \cup S \\ &\Rightarrow SS \cup S^\star \\ &\Rightarrow 1S \cup S^\star \\ &\Rightarrow 10 \cup S^\star \\ &\Rightarrow 10 \cup 1^\star \end{aligned}$$

Exemplo 3.1.4:

$$G = \langle V, \Sigma, R, Expr \rangle$$

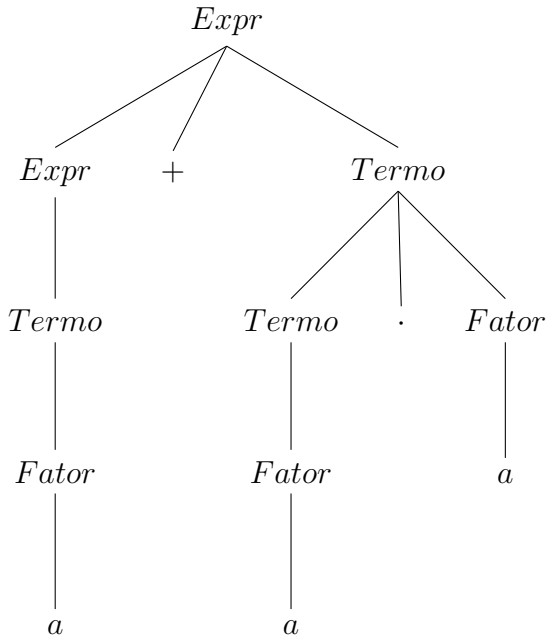
- $V = \{Expr, Termo, Fator\}$
- $\Sigma = \{a, +, \cdot, (,)\}$

$$\begin{aligned} Expr &\rightarrow Expr + Termo | Termo \\ Termo &\rightarrow Termo \cdot Fator | Fator \\ Fator &\rightarrow (Expr) | a \end{aligned}$$

Vamos mostrar que $a + a \cdot a \in L(G)$.

$$\begin{aligned} Expr &\Rightarrow Expr + Termo \\ &\Rightarrow Expr + Termo \cdot Fator \\ &\Rightarrow Termo + Termo \cdot Fator \\ &\Rightarrow Fator + Fator \cdot Fator \\ &\Rightarrow a + a \cdot a \end{aligned}$$

Podemos representar a derivação do último exemplo por meio de uma *árvore sintática*:



Note que uma mesma string pode ser derivada de uma mesma gramática por diferentes árvores sintáticas. Esse fenômeno é chamado *ambiguidade*.

Uma derivação de uma string ω em uma gramática G é uma *derivação mais a esquerda* se a cada passo a variável remanescente mais a esquerda é aquela que será substituída no próximo passo. Uma string é *derivada de maneira ambígua* na gramática G se ela tem mais de uma derivação à esquerda. Uma GLC é dita *ambígua* se ela gera alguma string de maneira ambígua.

Exemplo 3.1.5:

$$G = \langle \{S\}, \{+, \cdot, a\}, R, S \rangle$$

$$S \to S + S | S \cdot S | a$$

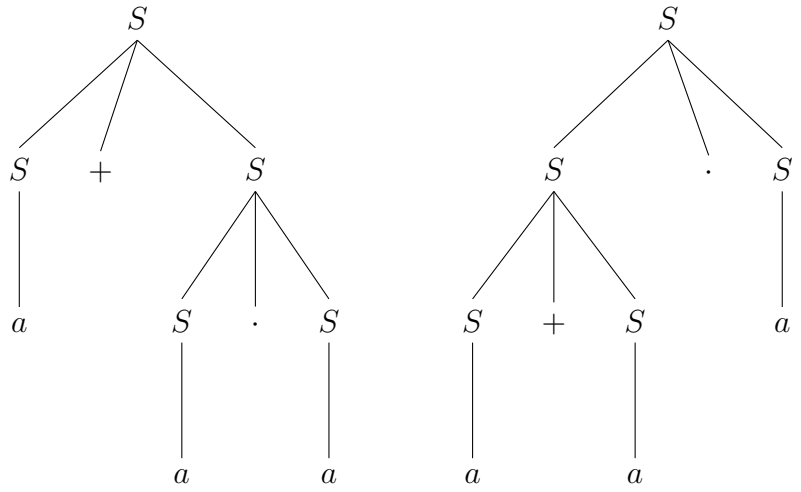
Vamos derivar a esquerda a expressão $a + a \cdot a$:

$$\begin{aligned} S &\Rightarrow S + S \\ &\Rightarrow a + S \\ &\Rightarrow a + S \cdot S \\ &\Rightarrow a + a \cdot a \end{aligned}$$

Alternativamente podemos derivar a mesma expressão à esquerda da seguinte maneira:

$$\begin{aligned} S &\Rightarrow S \cdot S \\ &\Rightarrow S + S \cdot S \\ &\Rightarrow a + S \cdot S \\ &\Rightarrow a + a \cdot a \end{aligned}$$

Essas derivações são representadas pelas seguintes árvores sintáticas.



Uma GLC está na *Forma Normal de Chomsky* (FNC) se toda regra é de uma das seguintes formas:

$S \to \varepsilon$

$A \to BC|a$

Onde $a \in \Sigma$, $A \in V$ e $B, C \in V - \{S\}$.

Teorema 3.1.6. *Toda linguagem livre de contexto é gerada por uma GLC na FNC.*

3.2 Autômatos de Pilha

Em um dos exemplos da seção anterior vimos que a linguagem não-regular $\{0^n 1^n : n \geq 0\}$ é livre de contexto. Portanto, os autômatos finitos não são adequados para reconhecer LLC. Nesta seção veremos um novo modelo de computação chamado autômato com pilha e mais para frente mostraremos sua relação íntima com as LLCs. Um autômato com pilha acrescenta aos autômatos finitos a capacidade de armazenar um mínimo de informação – uma mémoria muito simples. As informações armazenadas, porém, só podem ser acessadas na ordem inversa a que são inseridas como uma pilha (o úlitmo símbolo inserido é o primeiro a ser extraído).

Um *autômato com pilha* (AP) é uma 6-upla $\langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ onde:

- Q é um conjunto finito cujos elementos são chamados *estados*,
- Σ é um alfabeto chamado *alfabeto de entrada*,
- Γ é um alfabeto chamado *alfabeto da pilha*,
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times (\Gamma \cup \{\varepsilon\}) \to 2^{Q \times (\Gamma \cup \{\varepsilon\})}$ é a *função de transição*,
- $q_0 \in Q$ é o *estado inicial* e
- $F \subseteq Q$ é o conjunto dos *estados finais*.

Um AP $M = \langle Q, \Sigma, \Gamma, \delta, q_0, F \rangle$ *aceita* uma string ω se $\omega = \omega_1 \omega_2 \ldots \omega_n$ onde $\omega_i \in \Sigma \cup \{\varepsilon\}$ e existe uma sequência de estados $r_0, r_1, \ldots, r_m \in Q$ e uma sequência de strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ tal que:

1. $r_0 = q_0$ e $s_0 = \varepsilon$ (a pilha começa vazia),
2. $\langle r_{i+1}, b \rangle \in \delta(r_i, \omega_{i+1}, a)$ onde $s_i = at$ e $s_{i+1} = bt$ para $t \in \Gamma^*$ (lê um símbolo, vai para o próximo estado e atualiza a pilha) e

3. $r_m \in F$ (termina em um estado final).

A cada passo o AP lê um símbolo $\omega_{i+1} \in \Sigma \cup \{\varepsilon\}$, desempilha um símbolo $a \in \Gamma \cup \{\varepsilon\}$ da pilha, empilha outro $b \in \Gamma \cup \{\varepsilon\}$ e vai para o novo estado r_{i+1}.

Exemplo 3.2.1:

$$\begin{aligned} M &= \langle Q, \Sigma, \Gamma, \delta, q_1, F \rangle \\ Q &= \{q_1, q_2, q_3, q_4\} \\ \Sigma &= \{0, 1\} \\ \Gamma &= \{0, \$\} \\ F &= \{q_1, q_4\} \end{aligned}$$

δ é dado pela seguinte tabela:

	0			1			ε		
	0	\$	ε	0	\$	ε	0	\$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$	$\{(q_3, \varepsilon)\}$					
q_3				$\{(q_3, \varepsilon)\}$				$\{(q_4, \varepsilon)\}$	
q_4									

Na tabela omitimos as células que deveriam ser preenchidas por $\emptyset$ deixando-as vazias.

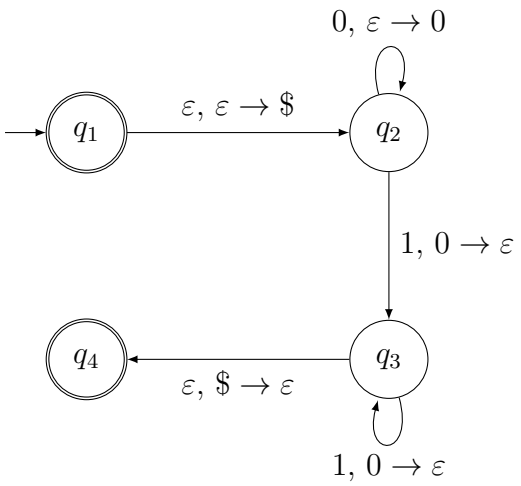
M reconhece a string 01:

1. Começo no estado q_1, empilho \$ e vou para q_2 (pilha: \$).
2. Leio 0, empilho 0 e fico em q_2 (pilha: 0\$).
3. Leio 1, desempilho 0 e vou para q_3 (pilha: \$).
4. Desempilho \$ e vou para $q_4 \in F$ (pilha: ε).

Podemos representar um AP usando um diagrama de estados. O diagrama de estados de um AP é como um diagrama de AFNs, mas em cada transição, além do símbolo a ser lido, indicamos os símbolos a serem desempilhados e empilhados (exemplo $a \to b$ indica que deve-se desempilhar a e empilhar b).

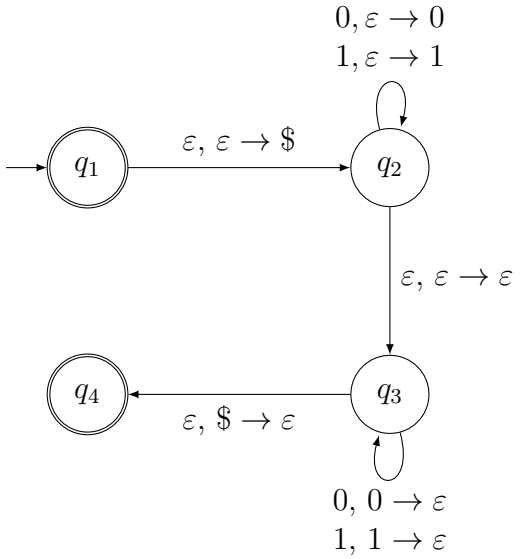
Exemplo 3.2.2:

Vamos ilustrar o autômato do último exemplo:



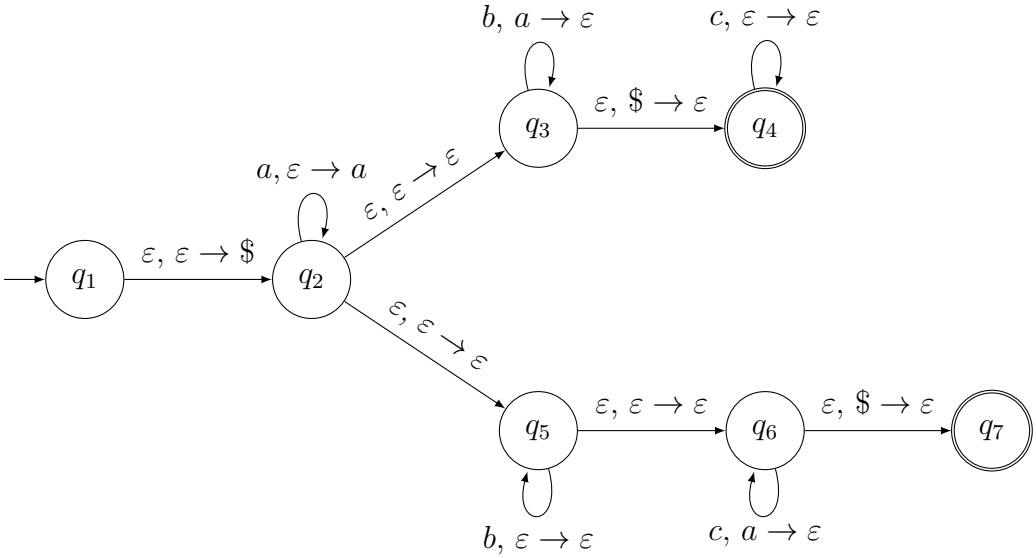
A pilha garante que será reconhecida a mesma quantidade de 0s e de 1s. Portanto $L(G) = \{0^n 1^n : n \geq 0\}$

Exemplo 3.2.3:



Procure verificar com alguns exemplos que $L(G) = \{\omega\omega^R : \omega \in \{0,1\}^*\}$ aonde ω^R é ω escrito de trás para frente.

Exemplo 3.2.4:



Procure verificar com alguns exemplos que:

$$L(G) = \{a^i b^j c^k : i = j \text{ ou } i = k\}$$

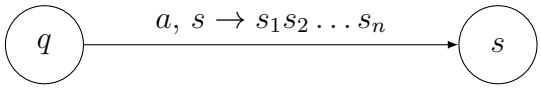
3.3 LLCs são Reconhecíveis por APs

Vimos que há linguagens que são representadas por GLCs que não são regulares e, portanto, não são reconhecíveis por autômatos finitos. Nesta seção veremos que toda linguagem livre de contexto é reconhecível por algum autômato de pilha.

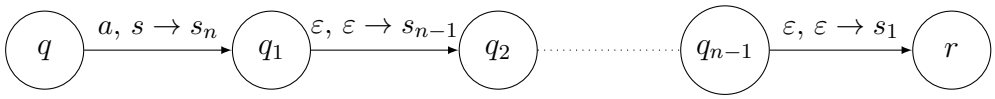
Para esta seção usaremos uma abreviação para descrever o empilhamento de uma sequência de símbolos. Seja $\omega \in \Gamma^*$, $r, q \in Q$, $a \in \Sigma$ e $s \in \Gamma$, escrevemos $\langle r, \omega \rangle \in \Delta(q, a, s)$ para indicar que ao ler a no estado q, desempilhamos s e empilhamos cada um dos símbolos de ω antes de ir para r. Ou seja, se $\omega = s_1 s_2 \ldots s_n \in \Gamma^*$, então $\langle r, \omega \rangle \in \Delta(q, a, s)$ é uma abreviação para:

$$\begin{aligned} \langle q_1, s_n \rangle &\in \Delta(q, a, s) \\ \{\langle q_2, s_{n-1} \rangle\} &= \Delta(q_1, \varepsilon, \varepsilon) \\ \{\langle q_3, s_{n-2} \rangle\} &= \Delta(q_2, \varepsilon, \varepsilon) \\ &\dots \\ \{\langle r, s_1 \rangle\} &= \Delta(q_{n-1}, \varepsilon, \varepsilon) \end{aligned}$$

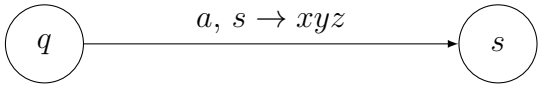
No diagrama de estados, escrevemos:



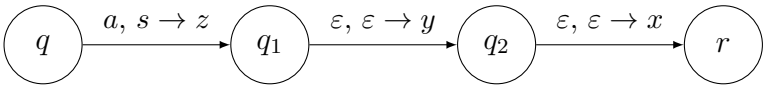
Para abreviar o seguinte:



Exemplo 3.3.1:



É uma abreviação de:



Teorema 3.3.2. *Toda linguagem livre de contexto é reconhecida por um Autômato com Pilha.*

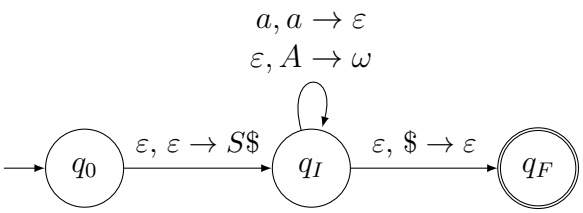
Demonstração. Se A é uma LLC, por definição, existe uma GLC $G = \langle V, \Sigma, R, S \rangle$ associada a A i.e. $L(G) = A$.

Construiremos um AP $P = \langle Q, \Sigma, \Gamma, \Delta, q_0, F \rangle$ que reconhece A i.e. $L(P) = A$.

- $Q = \{q_0, q_I, q_F\} \cup E$ onde E é o conjunto de estados necessários para abreviação que vimos acima.
- $F = \{q_F\}$
- $\Gamma = V \cup \Sigma \cup \{\$\}$
- Δ é apresentado abaixo.

$$\begin{aligned}
\Delta(q_0, \varepsilon, \varepsilon) &= \{\langle q_I, S\$\rangle\} \\
\Delta(q_I, \varepsilon, \$) &= \{\langle q_F, \varepsilon\rangle\} \\
\Delta(q_I, a, a) &= \{\langle q_I, \varepsilon\rangle\} \text{ para todo } a \in \Sigma \\
\Delta(q_I, \varepsilon, A) &= \{\langle q_I, \omega\rangle\} \text{ para todo } A \to \omega \in R
\end{aligned}$$

Diagramaticamente temos:



Em palavras, primeiro inserimos \$ para marcar o fim da pilha e em seguida inserimos a variável inicial S na pilha e seguimos para o estado intermediário q_I. Então, não-deterministicamente empilhamos o corpo de alguma das regras $A \to \omega$ ou desempilhamos um símbolo terminal a e o reconhecemos. Quando a pilha chega no fim (no símbolo \$), desempilhamos e vamos para o estado final. □

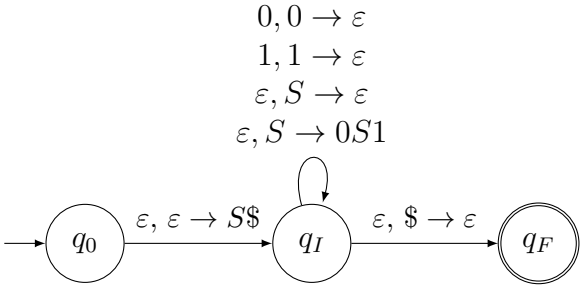
Exemplo 3.3.3:

Considere a gramática $G = \langle\{S\}, \{0, 1\}, R, S\rangle$ aonde R possui as seguintes regras:

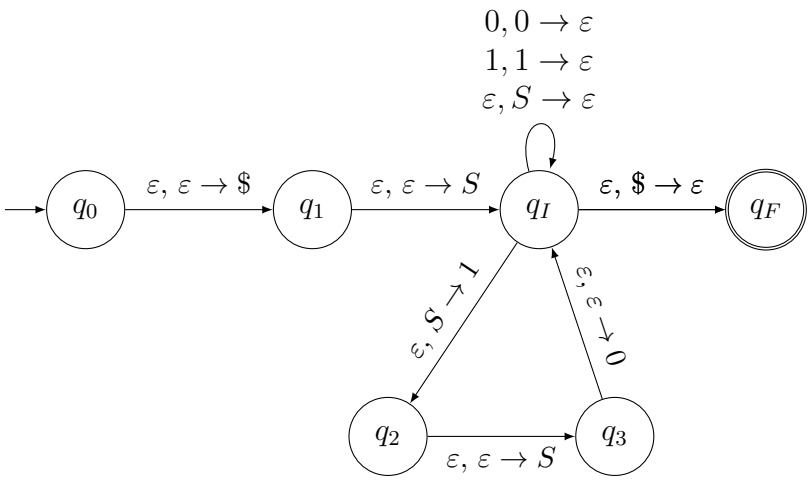
$$S \to 0S1|\varepsilon$$

Como já vimos, $L(G) = \{0^n 1^n : n \geq 0\}$.

Usando a construção do teorema anterior, temos que o seguinte AP reconhece essa linguagem:



O diagrama acima é uma abreviação para o seguinte diagrama:

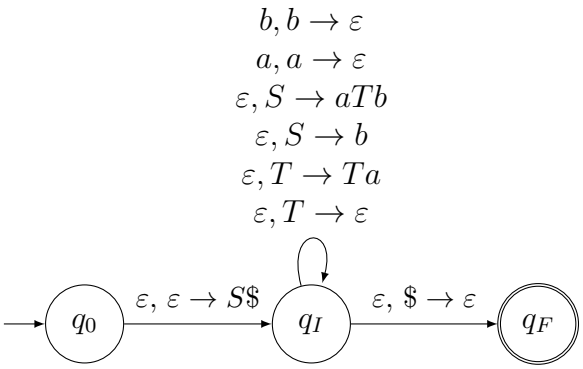


Exemplo 3.3.4:

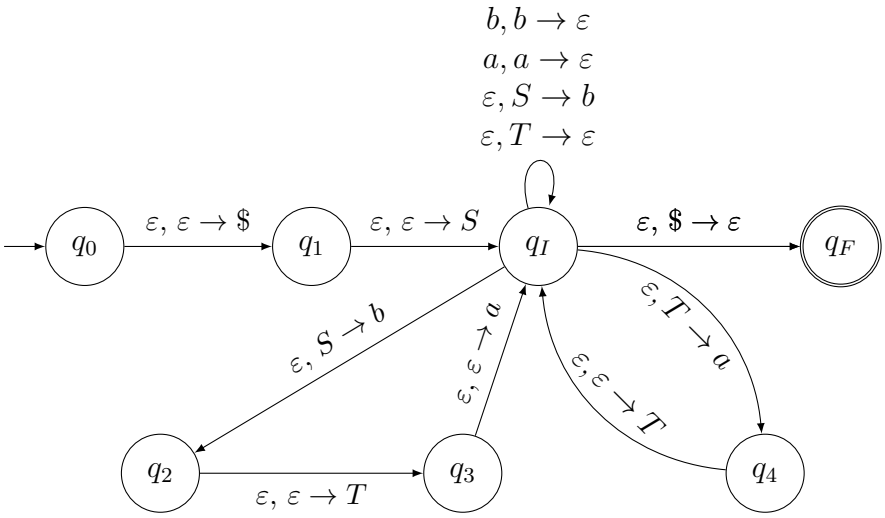
Considere a gramática $G = \langle \{S, T\}, \{a, b\}, R, S \rangle$ com as seguintes regras:

$$\begin{aligned} S &\rightarrow aTb|b \\ T &\rightarrow Ta|\varepsilon \end{aligned}$$

Usando a construção do teorema anterior o seguinte AP reconhece a linguagem representada por essa gramática:



Esse diagrama é uma abreviação do seguinte:

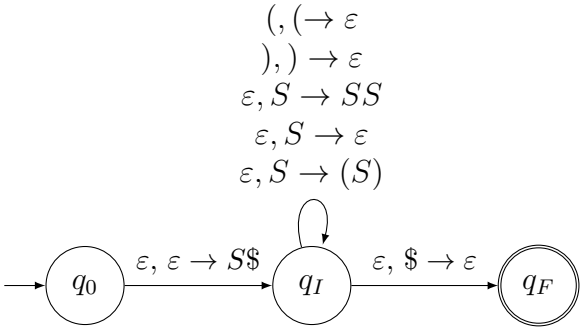


Exemplo 3.3.5:

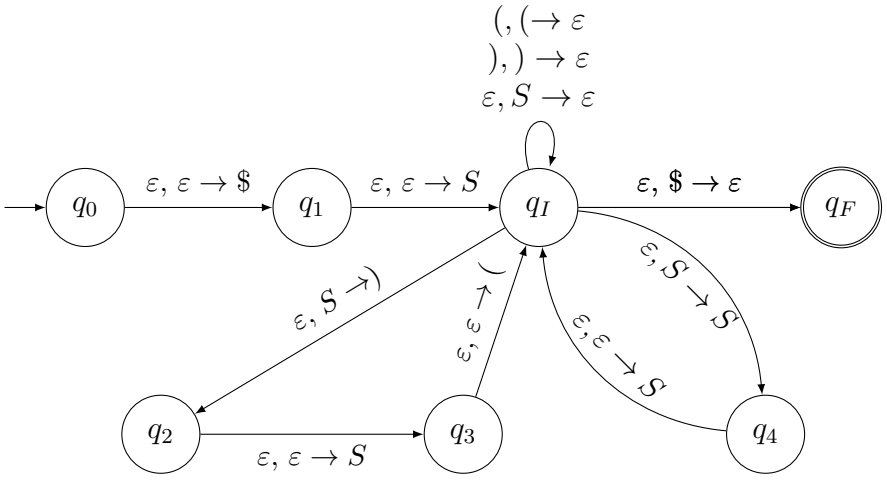
Considere outra gramática agora que possui um único não-terminal S e dois dois símbolos terminais '(' e ')':

$$S \rightarrow (S)|SS|\varepsilon$$

Novamente usaremos a construção do teorema para construir um autômato de pilha que reconhece a mesma linguagem que essa gramática:



Esse diagrama é uma abreviação do seguinte:



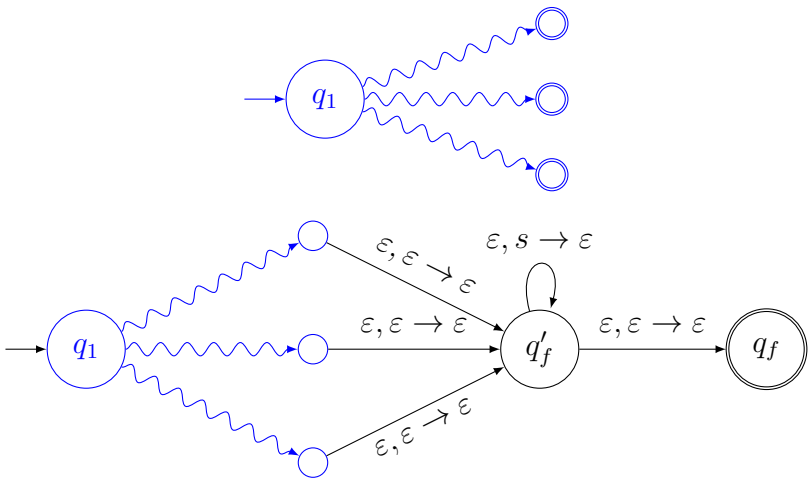
3.4 Linguagens Reconhecíveis por APs são Livres e Contexto

Na seção anterior vimos que toda linguagem livre de contexto é reconhecível por um autômato de pilha. Nesta seção faremos o inverso, a saber, mostraremos que toda linguagem reconhecída por autômatos de pilha é livre de contexto.

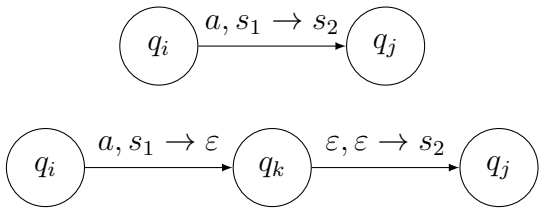
Lema 3.4.1. *Todo AP P é equivalente a outro AP P' em que:*

1. *o conjunto de estados finais possui um único elemento q_f,*
2. *as transações só empilham ou desempilham, mas nunca ambas ao mesmo tempo e*
3. *chega ao estado final com a pilha vazia.*

Demonstração. Para garantir os itens 1 e 3 criamos transições de cada estado final de P para q'_f que não leem nada e não empilham nem desempilham nada (setas com etiqueta $\varepsilon, \varepsilon \to \varepsilon$). Uma transição de q'_f para si mesmo que não le nada e desempilha s para cada $s \in \Gamma$ e uma transição que não lê nada e não mexe na pilha que vai de q'_f para q_f.



Para garantir a condição 2, substituimos toda transição que empilha e desempilha ao mesmo tempo por uma que desempilha seguida por outra que empilha.



□

Teorema 3.4.2. *Toda linguagem reconhecida por APs é livre de contexto.*

Demonstração. Seja $P = \langle Q, \Sigma, \Gamma, \Delta, p_0, F\rangle$. Pelo lema anterior existe P' equivalente a P satisfazendo as três proriedades. Criaremos uma gramática $G = \langle \Sigma, V, R, S\rangle$ que reconhece $L(P) = L(P')$.

- $V = \{A_{pq} : p, q \in Q\}$
- $S = A_{p_0 p_f}$
- R é formado por três tipos de regras:
  1. $A_{pq} \to A_{pr}A_{rq} \in R$ para todo $p, r, q \in Q$
  2. se $\langle r, t\rangle \in \Delta(p, a, \varepsilon)$ e $\langle q, \varepsilon\rangle \in \Delta(s, b, t)$ então $A_{pq} \to aA_{rs}b \in R$
  3. $A_{pp} \to \varepsilon \in R$ para todo $p \in Q$

Primeiro demonstraremos por indução no tamanho da derivação o seguinte:

Hipótese de Indução: Se $A_{pq} \Rightarrow^k x$ então P começa no estado p, reconhece x e chega no estado q com a pilha vazia.

Base: As únicas derivações de tamanho 1 são da forma $A_{pp} \to \varepsilon$ e claro que o autômato que começa e termina em q reconhece ε.

Passo de Indução: Precisamos mostrar que se $A_{pq} \Rightarrow^{k+1} x$ então P começa em p, reconhece x e chega em q com a pilha vazia. O primeiro passo dessa derivação deve ser

1. $A_{pq} \Rightarrow aA_{rs}b$ ou
2. $A_{pq} \Rightarrow A_{pr}A_{rq}$

No primeiro caso temos que $x = ayb$ e, portanto, $A_{rs} \Rightarrow^k y$. Pela H.I. P reconhece y indo de r até s e terminando com a pilha vazia. Como a $A_{pq} \to aA_{rs}b \in R$ então $\langle r, t\rangle \in \Delta(p, a, \varepsilon)$ e $\langle q, \varepsilon\rangle \in \Delta(s, b, t)$. Então p vai para r e empilha t ao ler a e desempilha t ao ler b e ir para q.

No segundo caso, temos que $x = yz$ e $A_{pr} \Rightarrow^* y$ e $A_{rq} \Rightarrow z$. Ambas derivações devem ter comprimento menor que $k + 1$ e, logo, pela H.I. P reconhece y indo de p para r e reconhece z indo de r até q.

Por fim, resta provar por indução no número de passos de computação de P que:

Hipótese de Indução: Se P reconhece x indo de p para q em k passos então $A_{pq} \Rightarrow^* x$

Base: Em 0 passos não sai do estado p e reconhece ε. Pela regra $A_{pp} \to \varepsilon$ geramos ε.

Passo de Indução: Suponha que P vai de p até q em $k + 1$ passos e reconhece x.

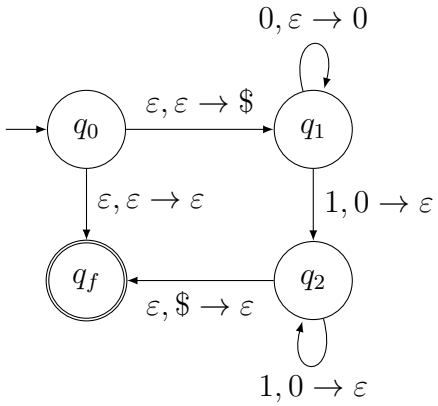
Suponhamos primeiro que em nenhum momento no processo a pilha fique vazia. Neste caso, o símbolo t empilhado no começo é desempilhado no fim. Se a é o símbolo lido no começo, b o símbolo lido no fim, então r o estado seguinte a p e s é o anterior a q. Ou seja, se $\langle r, t\rangle \in \Delta(p, a, \varepsilon)$ e $\langle q, \varepsilon\rangle \in \Delta(s, b, t)$ então $A_{pq} \to aA_{rs}b \in R$. Seja $x = ayb$, pela H.I., $A_{rs} \Rightarrow^* y$ e logo $A_{pq} \Rightarrow^* x$.

Por outro lado, se a pilha chega esvaziar então ela reconhece uma string y até ficar vazia e z até q e $x = yz$. Seja r o estado em P quando a pilha está vazia. Pela H.I. $A_{pr} \Rightarrow^* y$ e $A_{rp} \Rightarrow^* z$. Como $A_{pq} \to A_{pr}A_{rq} \in R$ então $A_{pq} \Rightarrow^* yz = x$. □

Corolario 3.4.3. *Toda linguagem regular é livre de contexto.*

Demonstração. Não é difícil notar que todo AFD é um AP aonde a pilha nunca é usada. Vimos que toda linguagem regular é reconhecida por um AFD, portanto toda linguagem regular é reconhecida por um AP e portanto é livre de contexto. □

Exemplo 3.4.4:



$$\left.\begin{array}{rcl} \langle q_1, 0\rangle & \in & \Delta(q_1, 0, \varepsilon) \\ \langle q_2, \varepsilon\rangle & \in & \Delta(q_2, 1, 0) \end{array}\right\} A_{q_1q_2} \to 0A_{q_1q_2}1$$

$$\left.\begin{array}{rcl} \langle q_1, 0\rangle & \in & \Delta(q_1, 0, \varepsilon) \\ \langle q_2, \varepsilon\rangle & \in & \Delta(q_1, 1, 0) \end{array}\right\} A_{q_1q_2} \to 0A_{q_1q_1}1$$

$$\left.\begin{array}{rcl} \langle q_1, \$\rangle & \in & \Delta(q_0, \varepsilon, \varepsilon) \\ \langle q_f, \varepsilon\rangle & \in & \Delta(q_2, \varepsilon, \$) \end{array}\right\} A_{q_0q_f} \to \varepsilon A_{q_1q_2}\varepsilon$$

P reconhece 0011, portanto essa string deve estar em $L(G)$ onde G é a gramática acima.

$$\begin{array}{rcl} A_{q_0q_f} & \Rightarrow & \varepsilon A_{q_1q_2}\varepsilon = A_{q_1q_2} \\ & \Rightarrow & 0A_{q_1q_2}1 \\ & \Rightarrow & 00A_{q_1q_1}11 \\ & \Rightarrow & 00\varepsilon 11 = 0011 \end{array}$$

3.5 Linguagens que não são Livres de Contexto

Para concluir nosso estudo do modelo dos autômatos de pilha, vamos mostrar exemplos de linguagens que não são livres de contexto. Para isso seguiremos

passos análogos aos do capítulo anterior: mostraremos uma condição necessária para uma linguagem ser livre de contexto então mostraremos exemplos de linguagens que não satisfaçam essa condição e, portanto, não são livres de contexto.

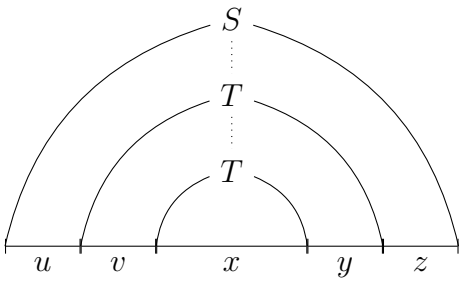
Lema 3.5.1 (Bombeamento para LLCs)**.** *Se A é uma LLC então existe p (comprimento do bombeamento) tal que se $\omega \in A$ e $|\omega| \geq p$ então $\omega = uvxyz$ e:*

1. *$uv^ixy^iz \in A$ para todo $i \geq 0$*
2. *$|vy| > 0$ e*
3. *$|vxy| \leq p$*

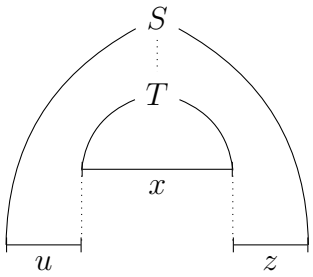
Demonstração. Se A é uma LLC, então por definição existe uma GLC $G = \langle \Sigma, V, R, S \rangle$ tal que $L(G) = A$. Seja b o número máximo de símbolos a direita em uma regra em R. Se partirmos de uma varíavel qualquer em G, em h passos o comprimento máximo da string que é possível produzir é b^h (se desenharmos a árvore sintática da string produzida desta forma, h é a altura desta árvore).

O comprimento do bombeamento será $p = b^{|V|+1}$. Se $\omega \in A$ e $|\omega| \geq p$, como na hipótese, existe k tal que $S \Rightarrow^k \omega$. Vamos supor que k seja o menor valor em que S deriva ω. Note que necessariamente $k \geq |V| + 1$. É claro que esse caminho possui $|V| + 1$ símbolos não-terminais, logo, pelo princípio da casa dos pombos, pelo menos uma variável ocorre mais de uma de uma vez neste caminho. Seja $T \in V$ a última variável que ocorre mais de uma vez.

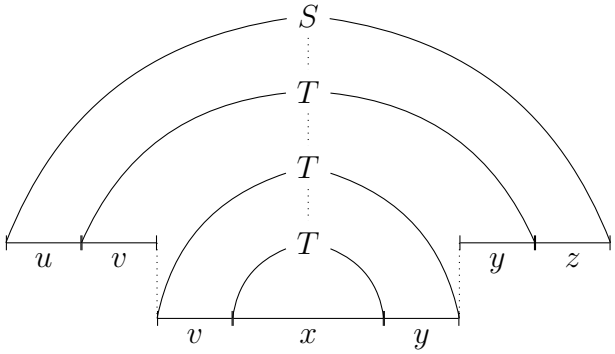
Dividimos ω em 5 partes $\omega = uvxyz$ de forma que a penúltima ocorrência de T gera vTy e a última gera x (i.e. $S \Rightarrow^* uTz \Rightarrow^* uvTyz \Rightarrow uvxyz$).



Note que $S \Rightarrow^* uxz$ se substituirmos a penúltima ocorrência de T pela última (i.e. $S \Rightarrow^* uTz \Rightarrow^* uxz$).



Da mesma forma, $S \Rightarrow^* uv^ixy^iz$ para qualquer $i > 1$ bastando repetir i vezes a última ocorrência de T pela penúltima.



Se $|vy| = 0$ então $v = y = \varepsilon$ e, portanto, $S \Rightarrow^l uxz = \omega$ como substituindo a penúltima ocorrência de T pela última e $l < k$ contrariando a suposição.

Se $|vxy| > p$ então, pelo princípio da casa dos pombos, na derivação da penúltima ocorrência de T até vxy alguma variável deveria repetir contrariando a suposição de que T era a última variável que se repetia. □

Exemplo 3.5.2:

$B = \{a^nb^nc^n : n \geq 0\}$ não é livre de contexto.

Seja p o comprimento do bombeamento, e $\omega = a^pb^pc^p \in B$. Se B fosse uma LLC então, pelo lema, $\omega = uvxyz$, $|vy| > 0$ e $uv^ixy^iz \in B$ para todo $i \geq 0$. Temos duas possibilidades:

1. se v e y um único tipo de símbolo cada, então uv^2xy^2z não conterá a mesma quantidade de as, bs e cs e, portanto, $uv^2xy^2z \notin B$.

2. se v ou y contém mais de um símbolo distinto então uv^2xy^2z contém símbolos na ordem errada e, portanto, $uv^2xy^2z \notin B$.

Concluímos que B não é livre de contexto.

Exemplo 3.5.3:

$C = \{a^i b^j c^k : 0 \leq i \leq j \leq k\}$ não é livre de contexto.

Seja p o comprimento do bombeamento e $\omega = a^p b^p c^p \in C$. Pelo lema, se C fosse livre de contexto, teríamos $\omega = uvxyz$ com $|vy| > 0$ e $uv^ixy^iz \in C$ para todo $i > 0$. Considere os dois possíveis casos:

1. y e v só contém um tipo de símbolo cada:
   - se a não ocorre em vy então $uxz \notin C$;
   - se b não ocorre em vy, mas a ocorre, então $uv^2xy^2z \notin C$, pois possui mais as do que bs e se c ocorre $uv^2xy^2z \notin C$ por motivo análogo;
   - se c não ocorre em vy então $uv^2xy^2z \notin C$, pois a string possuiria mais as ou mais bs do que c.
2. se y ou v possuem mais de um tipo de símbolo então $uv^2xy^2z \notin C$, pois possui símbolos na ordem errada.

Concluímos que C não é livre de contexto.

Na seção anterior, vimos que todas as linguagens regulares são livres de contexto, mas anteriormente havíamos mostrado que exitem linguagens livres de contexto ($\{0^n1^n : n \geq 0\}$ por exemplo) que não são regulares.

$$\text{Ling. Reg.} \subset \text{LLCs}$$

Nesta seção vimos que esxistem linguagens formais que não são livres de contexto:

$$\text{LLCs} \subset \text{Ling. Formais}$$

Além disso, vimos que linguagens regulares coincidem com as linguagens reconhecíveis por autômatos finitos e que as livres de contexto coincidem com as reconhecíveis por autômatos com pilha.

No próximo capítulo passaremos à questão central do curso, a saber, a existência de problemas que não possuem solução computacional. Para tanto, precisamos de um modelo de computação capaz de dar conta de qualquer dispositivo mecânico.

Capítulo 4

Máquinas de Turing

Estudamos até aqui modelos de computação de expressividade crescente. Começamos com autômatos finitos, vimos que existem linguagens que não conseguimos reconhecer com esse tipo de autômatos. Identificamos exatamente a classe de linguagens que esse tipo de modelo é capaz de reconhecer, a saber, as linguagens regulares. Passamos então para os autômatos com pilha que são mais expressivos, reconhecem todas as liguagens regulares e mais algumas não-regulares. Novamente encontramos limitações, linguagens que não são reconhecíveis por autômatos com pilha.

Neste capítulo estudaremos um modelo ainda mais expressivo, as Máquinas de Turing (MTs). Temos dois grandes objetivos neste capítulo. O primeiro é convencer que este é o modelo definitivo de computação, ou seja, que não existe modelo de computação mais expressivo que as Máquinas de Turing. Esse resultado, que não é nem pode ser um teorema, é chamado de Tese de Church-Turing. O argumento para esta tese serão três: provaremos que as MTs são mais expressivas que os autômatos com pilha, em seguida mostraremos uma serie de variantes das MTs e provaremos que todas são equivalentes (i.e. tem a mesma expressividade) e, por fim, provaremos que toda MT pode ser simulada por uma MT específica chamada de Máquina de Turing Universal. O segundo objetivo deste capítulo é provar que, mesmo sendo o modelo de computação mais completo, as MTs possuem limitações. Ou seja, existem problemas computacionais que não podem ser resolvidos por MTs.

4.1 Máquinas de Turing Determinísticas

Uma MT consiste de uma fita formada por células em sequência, potencialmente infinita em ambas as direções e uma cabeça que lê o conteúdo de cada célula e guarda o estado atual. Uma função de transição indica, dado o estado atual e o símbolo sendo lido qual é a próxima operação: ir para esquerda ou ir para a direita e qual o novo símbolo na célula atual.

Formalmente temos que uma *Máquina de Turing Determinística*, ou simplesmente uma MT, é uma 7-upla $\langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r \rangle$ em que:

Q é um conjunto finito de *estados*,

Σ é o *alfabeto da entrada*,

Γ é o *alfabeto da fita* e $\Sigma \cup \{\llcorner\} \subseteq \Gamma$,

$\delta : Q \times \Gamma \to Q \times \Gamma \times \{E, D\}$ é a *função de transição*,

$q_0 \in Q$ é o *estado inicial*,

$q_a \in Q$ é o *estado de aceitação* e

$q_r \in Q$ é o *estado de rejeição*.

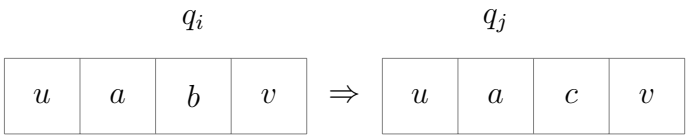
A cada passo, a MT está e, uma certa *configuração*. A configuração indica a sequência de símbolos antes da cabeça na fita e a sequência de símbolos depois da cabeça. Uma configuração pode ser representada por uma string da seguinte forma:

$$C = \omega_1 q \omega_2$$

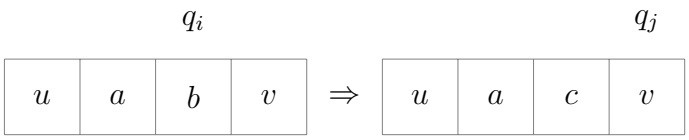
As strings $\omega_1 \in \Gamma^*$ e $\omega_2 \in \Gamma^*$ indicam as sequências antes e depois da cabeça. O estado q indica o estado atual e o primeiro símbolo de ω dois é o símbolo sendo lido. Uma configuração em que $q = q_a$ é dita de *aceitação* e em que $q = q_r$ é de *rejeição*. Configurações de aceitação ou de rejeição são ditas *configurações de parada*. A *função de transição* define para cada configuração C_i qual o próxima configuração C_{i+1}.

Exemplo 4.1.1:

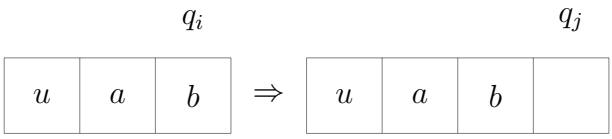
1. $uaq_ibv \Rightarrow uq_jacv$ se $\delta(q_i, b) = \langle q_j, c, E \rangle$



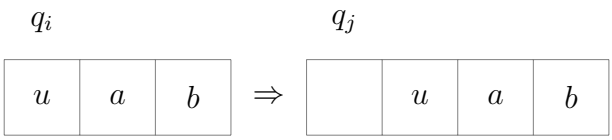
2. $uaq_ibv \Rightarrow uacq_jv$ se $\delta(q_i, b) = \langle q_j, c, D\rangle$



3. $uaq_ib \Rightarrow uabq_j\sqcup$ se $\delta(q_i, b) = \langle q_j, b, D\rangle$



4. $q_iuab \Rightarrow q_j\sqcup uab$ se $\delta(q_i, u) = \langle q_j, u, E\rangle$



Uma MT *aceita* uma string $\omega \in \Sigma^*$ se existe uma sequência de configurações $C_1, C_2, \ldots, C_k$ em que:

1. $C_1 = q_0\omega$ (configuração inicial),

2. $C_i \Rightarrow C_{i+1}$ para $i < k$ e

3. $C_k = \omega_1 q_a \omega_2$ (configuração de aceitação)

Uma MT *rejeita* uma string $\omega \in \Sigma^*$ se existe uma sequência de configurações que satisfaz os dois primeiros itens e o seguinte:

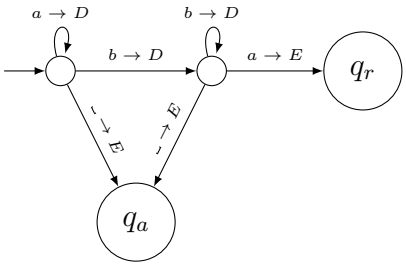
3' $C_k = \omega_1 q_r \omega_2$ (configuração de rejeição)

Note que para rejeitar uma string não basta não aceitá-la.

Uma linguagem A é *Turing-reconhecível* ou *recursivamente enumerável* (r.e.) se existe uma MT que aceita todas as strings de A. Um linguagem B é *Turing-decidível* ou *recursiva* se existe uma MT que aceita todas as strings em B e rejeita todas as strings em $\bar{B}$.

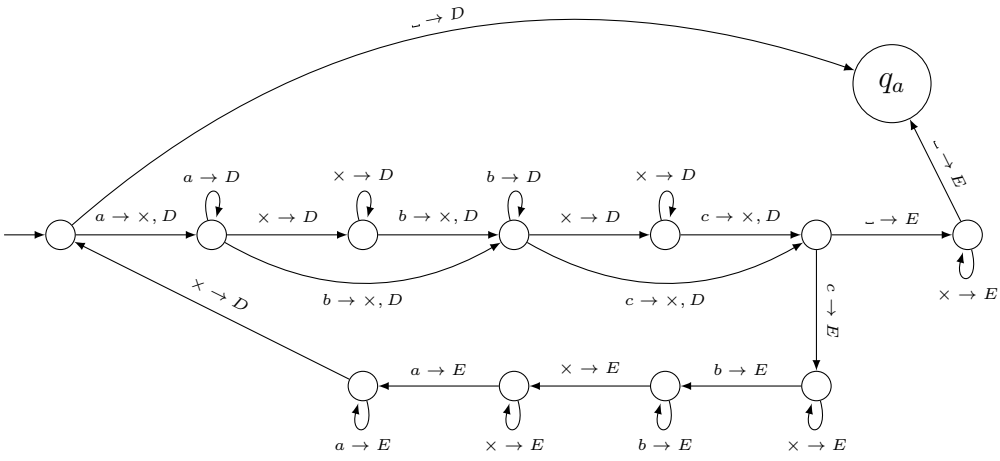
Exemplo 4.1.2:

A linguagem $L(a^*b^*)$ é recursiva.



Exemplo 4.1.3:

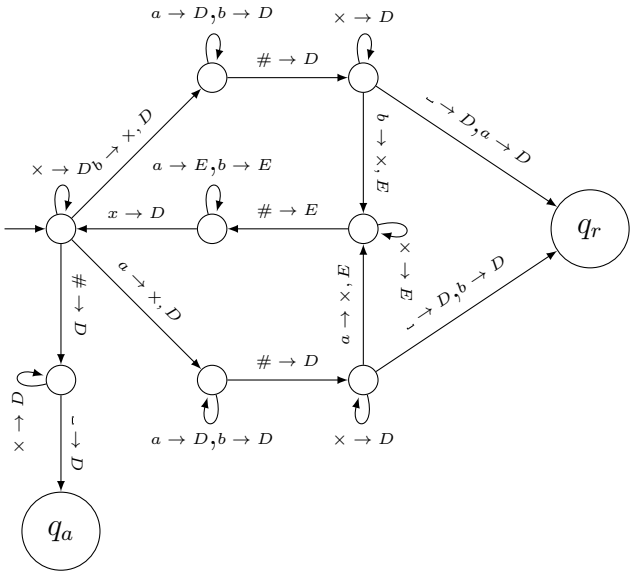
A linguagem $\{a^n b^n c^n : n \geq 0\}$ é recursiva[1].



Exemplo 4.1.4:

[1]Para não poluir o diagrama omitimos as transições para q_r

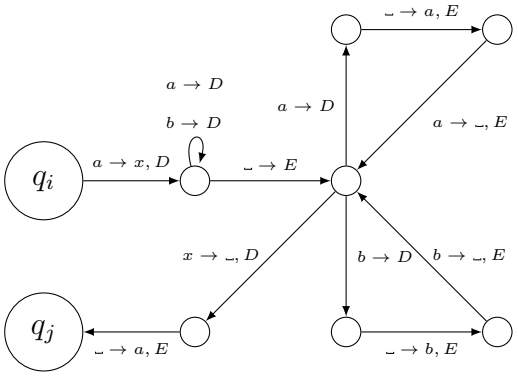
A linguagem $\{\omega\#\omega : \omega \in \{a,b\}^*\}$ é recursiva.



Exemplo 4.1.5:

A Máquina de Turing a seguir tem o seguinte efeito:

$$\omega_1 q_i a \omega_2 \Rightarrow^* \omega_1 q_j \sqcup a \omega_2$$



4.2 Máquinas de Turing Multifitas

Uma variante das Máquinas de Turing são aquelas com múltiplas fitas. Nesse caso, a cada passo temos k símbolos sendos lidos e a função de transição indica o que fazer em cada uma das fitas dado o estado atual e os k símbolos que estão sendo lidos. Formalmente, como numa MT tradicional temas a seguinte tupla:

$$M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r \rangle$$

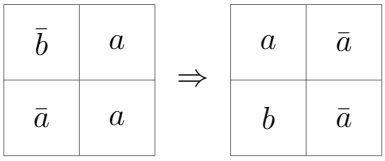
Neste caso, porém, temos que:

$$\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{E, D\}^k$$

Ou seja, a função de transição leva um estado e k símbolos em um novo estado, k novos símbolos e k direções.

Exemplo 4.2.1:

$$\delta(q_0, \langle a, b \rangle) = \langle q_1, \langle b, a \rangle, \langle D, D \rangle \rangle$$



Teorema 4.2.2. *Para toda MT multifita existe uma MT simples equivalente.*

Demonstração. Faremos aqui apenas o esboço da prova. Simulamos as k fitas em uma única fita com delimitadores indicados pelo símbolo #.

- a entrada $a_1 \dots a_n$ será representada em uma única fita como $\#\bar{a_1} \dots a_n \# \sqcup \# \dots \#$
- varre a entrada para verificar os símbolos sendo lidos
- varre novamente efetuando as transições em cada um dos trechos da fita

- se em algum ponto estivermos em # e a instrução for D devemos abrir um espaço em branco antes de # (Exemplo 4.1.5).

□

Exemplo 4.2.3:

Considere o seguinte estado em uma MT multifita:

1	0	$\bar{1}$	
$\bar{0}$	0		
0	$\bar{1}$	1	1

Ela seria representada em uma MT simples da seguinte forma:

#	1	0	$\bar{1}$	#	$\bar{0}$	0	#	0	$\bar{1}$	1	1	#

Teorema 4.2.4. *Uma linguagem é recursiva se e somente se existem MTs que reconhecem A e $\bar{A}$*

Demonstração. Se A é recursivo então, por defineção existe uma MT M que decide A e, portanto, existe uma MT que reconhece A.

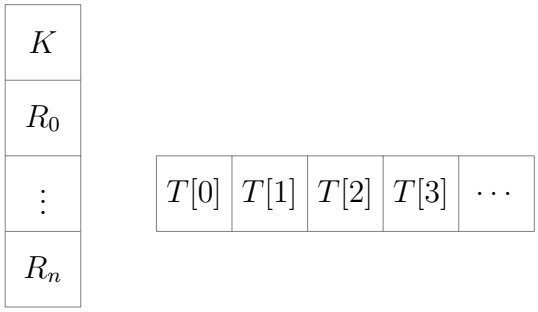
Seja M' uma MT igual a M exceto que em M' trocamos q_a por q_r. A máquina M' aceita tudo que M rejeita e rejeita tudo que M aceita. Portanto M' reconhece $\bar{A}$.

Agora sejam M_1 e M_2 MTs que reconhecem A e $\bar{A}$ respectivamente. Construímos uma MT com duas fitas que simula M_1 e M_2 em paralelo. Ou seja, simula M_1 na primeira fita e M_2 na segunda. Essa MT deve aceitar ω se M_1 aceita ω e deve rejeitar ω se M_2 aceita ω. Pelo Teorema 4.2.2 temos que existe uma MT simples equivalente a essa de fita dupla e esta MT decide A. □

4.3 Máquinas de Acesso Aleatório (RAM)

Vamos considerar agora uma máquina que a princípio parece bem diferente de uma MT, muito mais parecida com um computador moderno. Uma *Máquina de Acesso Aleatório (RAM)* tem a capacidade de acessar um elemento qualquer em um único passo desde que ele esteja devidamente endereçado.

Em uma RAM temos um número de *registradores* capazes de armazenar e manipular *endereços* das células de memória. Um *programa* em uma RAM é uma sequência de instruções que manipulam o conteúdo dos registradores e da memória. O primeiro registrador tem uma função especial e é chamado *acumulador*. Além disso, o programa mantém um contador K que indica a instrução a ser executada.



Uma *Máquina de Acesso Aleatório* (RAM) é um par $M = \langle k, \Pi \rangle$ em que $k > 0$ indica o número de registradores e $\Pi = \pi_0, \pi_1, \ldots, \pi_n$ é uma sequência de instruções (programa) da Tabela 4.3 admitindo que $\pi_n = halt$.

Formalmente um *configuração* de uma RAM é uma k + 2-upla $\langle K, R_0, \ldots, R_{k-1}, T \rangle$ em que:

1. $K \in \mathbb{Z}_p$ é o *contador de instruções*
2. uma *configuração de parada* é tal que $K = 0$
3. R_j é o *valor do registrador* j
4. $T : \mathbb{N} \to \mathbb{N}$ leva um natural i (*endereço*) em seu *conteúdo* m.

Dizemos que a configuração $C = \langle K, R_0, \ldots, R_{k-1}, T \rangle$ de uma RAM $M = \langle k, \Pi \rangle$ *produz em um passo* $C' = \langle K', R'_0, \ldots, R'_{k-1}, T' \rangle$ (escrevemos $C \Rightarrow_M C'$) se C' reflete o resultado da aplicação da instrução π_K em C. Se partindo

$read(j)$	$R_0 \leftarrow T[R_j]$
$write(j)$	$T[R_j] \leftarrow R_0$
$store(j)$	$R_j \leftarrow R_0$
$load(j)$	$R_0 \leftarrow R_j$
$load(= c)$	$R_0 \leftarrow c$
$add(j)$	$R_0 \leftarrow R_0 + R_j$
$add(= c)$	$R_0 \leftarrow R_0 + c$
$sub(j)$	$R_0 \leftarrow R_0 - R_j$
$sub(= c)$	$R_0 \leftarrow R_0 - c$
$half$	$R_0 \leftarrow \lfloor R_0/2 \rfloor$
$jump(s)$	$K \leftarrow s$
$jpos(s)$	$R_0 > 0 \Rightarrow K \leftarrow s$
$jzero(s)$	$R_0 = 0 \Rightarrow K \leftarrow s$
$halt$	$K \leftarrow 0$

Tabela 4.1: Catálogo de instruções de uma RAM

de C produzimos C' em um número finito de passos, então escrevemos $C \Rightarrow_M^* C'$.

Exemplo 4.3.1:

Considere a seguinte márquina $\langle \Pi, 4 \rangle$:

1. $store(2)$
2. $jzero(10)$
3. $load(3)$
4. $add(1)$
5. $store(3)$
6. $load(2)$
7. $sub(= 1)$
8. $store(2)$
9. $jump(1)$
10. $load(3)$
11. $halt$

Essa máquina começa com valores n e x nos registradores 0 e 1 e termina com $n \cdot x$ no acumulador. Ou seja, a maquina calcula a multiplicação.

Simulando com entrada 2 e 3 podemos conferir que:

$$\langle 1; 2, 3, 0, 0; \emptyset \rangle \Rightarrow_M^* \langle 11; 6, 3, 0, 6; \emptyset \rangle$$

Para faciliar a leitura e a escrita de programas podemos usar a abreviação $R_3 \leftarrow R_3 + R_1$ para a sequência comum de instruções $load(3)$, $add(1)$, $store(3)$ e $R_2 \leftarrow R_2 - 1$ para $load(2)$, $sub(= 1)$, $store(2)$. Além disso, podemos dar nomes como x, y e z para R_1, R_2 e R_3. Por fim, as instruções 2 e 9 normalmente são expressas com um loop `while` contendo as instruções a serem repetidas.

O programa anterior pode, então ser reescrito da seguinte forma:

```
z = x
enquanto y > 0
   z = z + x
   y = y - 1
x = z
```

Considere um alfabeto finite Σ. Podemos enumerar seus elementos $E : \Sigma \to \mathbb{N}$. A *configuração inicial* de uma RAM $M = \langle K, \Pi \rangle$ cuja entrada é $\omega = a_1 \dots a_n$ é $\langle 1; 0, 0, \dots; T \rangle$ em que $T[1] = E(a_1)$, $T[2] = E(a_2)$ ... $T[n] = E(a_n)$.

Dizemos que M *aceita* $x \in \Sigma^*$ se a configuração inicial de M para entrada x produz uma configuração de parada em que $R_0 = 1$ e *reiejta* x se produz uma configuração de parada em que $R_0 = 0$. Dizemos que M *decide* uma linguagem A se M aceita todo $x \in A$ e rejeita todo $x \notin A$.

Exemplo 4.3.2:

O seguinte programa decide a linguagem $\{a^n b^n b^n : n \geq 0\}$ considerando que $E(a) = 1$, $E(b) = 2$ e $E(c) = 3$:

1	$load(= 0)$	
2	$store(1)$	$R_1 = 0$
3	$store(2)$	$R_2 = 0$
4	$store(3)$	$R_3 = 0$
5	$load(= 1)$	
6	$store(4)$	$R_4 = 1$

$$
\left.\begin{array}{lll}
7 & read(4) & \\
8 & sub(=1) & T[R_4]-1 \\
9 & jzero(17) & \\
10 & load(4) & \\
11 & add(=1) & \\
12 & store(4) & R_4+=1 \\
13 & load(1) & \\
14 & add(=1) & \\
15 & store(1) & R_1+=1 \\
16 & jump(7) &
\end{array}\right\} \text{enquanto } T[R_4]=1
$$

$$
\left.\begin{array}{lll}
17 & read(4) & \\
18 & sub(=2) & T[R_4]-2 \\
19 & jzero(27) & \\
20 & load(4) & \\
21 & add(=1) & \\
22 & store(4) & R_4+=1 \\
23 & load(2) & \\
24 & add(=1) & \\
25 & store(2) & R_2+=1 \\
26 & jump(17) &
\end{array}\right\} \text{enquanto } T[R_4]=2
$$

$$
\left.\begin{array}{lll}
27 & read(4) & \\
28 & sub(=3) & T[R_4]-3 \\
29 & jzero(37) & \\
30 & load(4) & \\
31 & add(=1) & \\
32 & store(4) & R_4+=1 \\
33 & load(3) & \\
34 & add(=1) & \\
35 & store(3) & R_3+=1 \\
36 & jump(27) &
\end{array}\right\} \text{enquanto } T[R_4]=3
$$

$$
\left.\begin{array}{lll}
37 & load(1) & \\
38 & sub(2) & R_1-R_2 \\
39 & jzero(42) & \\
40 & load(=0) & \\
41 & halt & rejeita
\end{array}\right\} \text{rejeita se } R_1 \neq R_2
$$

$$
\left.\begin{array}{lll}
42 & load(2) & \\
43 & sub(3) & R_2 - R_3 \\
44 & jzero(47) & \\
45 & load(=0) & \\
46 & halt & rejeita
\end{array}\right\} \text{rejeita se } R_2 \neq R_3
$$

$$
\left.\begin{array}{lll}
47 & read(4) & \\
48 & jzero(51) & \\
49 & load(=0) & \\
50 & halt & rejeita
\end{array}\right\} \text{rejeita se } T[R_4] \neq 0
$$

$$
\left.\begin{array}{ll}
51 & load(=1) \\
52 & halt
\end{array}\right\} \text{aceita se tiver chegado aqui}
$$

O seguinte é uma abreviação desse programa:

```
a = b = c = 0
n = 1
enquanto T[n] == 1
   n = n + 1
   a = a + 1
enquanto T[n] == 2
   n = n + 1
   b = b + 1
enquanto T[n] == 3
   n = n + 1
   c = c + 1
se a != b || b != c || T[n] != 0
   rejeita
senão
   aceita
```

Teorema 4.3.3. *Para toda RAM existe uma MT equivalente.*

Demonstração. Construir uma RAM que simula uma MT é relativamente simples, mas pedante. Deixaremos essa parte em aberto.

Construir uma MT que simula uma RAM é mais complicado, mas também possível. Para tanto precisaríamos de uma MT com 7 fitas:

1. guarda a entrada
2. guarda o conteúdos dos registradores
3. guarda o valor atual de K
4. guarda o valor do registrador sendo lido

5 - 7 executam as operações (no caso das operações aritméticas duas fitas guardam os fatores e uma o resultado)

□

4.4 Máquinas de Turing Não-determinísticas

Em uma Máquina de Turing não-determinística, cada configuração pode levar a uma um mais configurações. Uma string é *aceita* se partindo da configuração inicial *existe* uma sequência de configurações que chega a uma configuração de aceitação.

Formalmente temos que:

$$N = \langle Q, \Sigma, \Gamma, \Delta, q_0, q_a, q_r \rangle$$

Em que Δ não é uma função de transição que recebe um estado e um símbolo e leva a um conjunto de configurações:

$$\Delta : Q \times \Gamma \to 2^{Q \times \Gamma \times \{E,D\}}$$

Assim, partindo da configuração inicial, a cada passo temos um conjunto de próximas configurações possíveis. Podemos imaignar se formando uma árvore de configurações. A entrada é aceita se algum dos ramos da árvore chega a uma configuração de aceitação. A entrada é rejeitada apenas se cada um dos ramos chega a uma configuração de rejeição.

Máquinas não-determinísticas são ideias. Não temos pretenção de construí-las. Porém, por mais que pareçam muito mais poderosas, assim com as outras variantes de MT que vimos até aqui, essas máquinas também computam o mesmo que uma MT simples.

Teorema 4.4.1. *Para toda MT não determinísitca existe uma MT simples equivalente.*

Demonstração. Seja $N = \langle Q, \Sigma, \Gamma, \Delta, q_0, q_a, q_r \rangle$ uma MT não-determinística e seja b o tamanho máximo de uma ramificação em N pode chegar – ou seja, $b = max_{(a,q)\in\Sigma\times Q}(|\Delta(a,q)|)$ – e seja então $\Sigma_b = \{1, 2, \ldots, b\}$.

Simularemos N em uma MT com três fitas. A primeira fita contém a entrada ω. A segunda fita fará a simulação e a terceira contém uma string $s \in \Sigma_b^*$ que indica as escolhas não determinísticas a serem feitas.

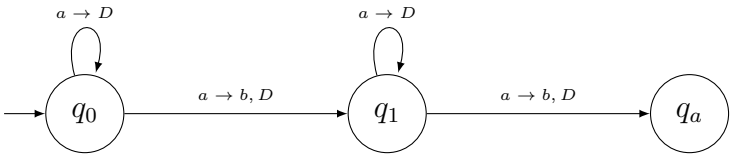
1. Copiamos ω para a fita 2.

2. Simulamos N na fita 2. Antes de cada passo, verificamos o próximo símbolo da fita 3 para escolher qual das transições válidas seguir. Se processamos todos os símbolos da fita 3, não há mais instruções válidas ou encontramos um estado de rejeição, vá para o próximo item. Se encontramos um estado de aceitação, pare e aceite a entrada.

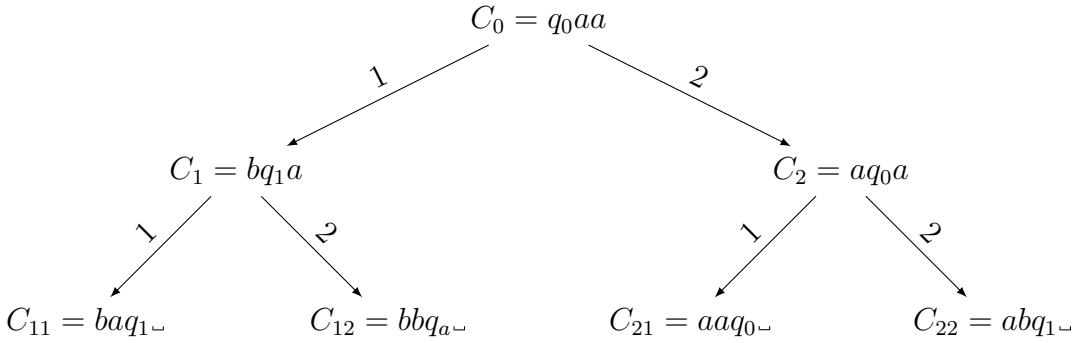
3. Apagamos todo o conteúdo da fita 3 e escrevemos a próxima string em Σ_b^* na ordem lexicográfica, apagamos a fita 2 e voltamos para o passo 1.

Em poucas palavras, estamos fazendo uma busca em largura nas configurações de N. □

Exemplo 4.4.2:



Vamos simular essa máquina com entrada $\omega = aa$. Nesse caso $b = 2$



	fita 2	fita 3
1	$\bar{a}a$	
2	$b\bar{a}$	$\bar{1}$
3	$\bar{a}a$	
4	$a\bar{a}$	$\bar{2}$
5	$\bar{a}a$	
6	$b\bar{a}$	$\bar{1}1$
7	$ba\bar{\sqcup}$	$1\bar{1}$
8	$\bar{a}a$	
9	$b\bar{a}$	$\bar{1}2$
10	$bb\bar{\sqcup}$	$1\bar{2}$

4.5 Hierarquia de Chomsky

A teoria das linguagens formais tem origem na convergência de diversas áreas: lógica e teoria das funções recursivas, circuitos booleanos, modelagem de sistemas biológicos, linguística computacional e projeto de linguagens de programação. Os modelos de Turing e Post são de meados dos anos 30 e 40 e deram origem à Ciência da Computação. O que se costumava se chamar de *sistemas generativos* (aqui chamados de gramáticas) começaram a ser estudados nos anos 40. As linguagens regulares aparecem nos trabalhos de Kleene e gramáticas livres de contexto aparecem nos trabalhos de linguistas como Bloch nessa época. Os trabalhos que organizaram o campo, porém, são da segunda metade dos anos 50. Em uma série de trabalhos seminais, Chomsky introduz o que hoje é conhecido como *Hierarquia de Chomsky* que apresentamos aqui. Para completar essa micro-retrospectiva histórica, os anos 60 foram marcados pelo desenvolvimento da área de complexidade computacional que trataremos no próximo capítulo. Nos anos 70 e 80, a complexidade computacional estabeleceu as bases dos fundamentos teóricos da criptografia. Desde os anos 90 uma dos temas mais quentes na área é o da computação quântica, seus modelos de computação e sua complexidade.

Uma *gramática*, no sentido mais geral, é uma tupla $\langle V, \Sigma, R, S\rangle$ cujas regras são da forma:

$$uxv \to uyv$$

Em que $x \in V \cup \Sigma$ e $u, v, y \in (\Sigma \cup V)^*$. Dizemos que de uma string $\omega_1 uxv\omega_2$ derivamos outra string $\omega_1 uyv\omega_2$ a partir da gramática G se a gramática possui a regra $uxv \to uyv$. Nesse caso escrevemos $\omega_1 uxv\omega_2 \Rightarrow_G \omega_1 uyv\omega_2$ ou simplesmente $\omega_1 uxv\omega_2 \Rightarrow \omega_1 uyv\omega_2$ se estiver claro pelo contexto sobre qual gramática nos referimos. As strings em Σ^* que são derivadas a partir do variável inicial S em um número finito de passos formam uma linguagem que chamamos de $L(G)$. O *Tipo 0* é a classe de linguagens produzidas por quaisquer gramáticas. Essa classe coincide com o que chamamos de *recursivamente enumeráveis*. A demonstração desse fato foge ao escopo dessas notas.

O *Tipo 1* é o que hoje chamamos de *Linguagens Dependentes do Contexto* (LDC). Essas linguagens são produzidas a partir de gramáticas cujas regras são do tipo $uAv \to uyv$ em que $A \in V$ e $y \neq \varepsilon$.

Exemplo 4.5.1:

Seja $G = \langle \{S, B, C, H\}, \{a, b, c\}, R, S \rangle$ tal que R é formado pelas seguintes regras:

$$\begin{aligned} S &\to aSBC|aBC \\ CB &\to HB \\ HB &\to HC \\ HC &\to BC \\ aB &\to ab \\ bB &\to bb \\ bC &\to bc \\ cC &\to cc \end{aligned}$$

Vamos mostrar que $aabbcc \in L(G)$

$$
\begin{aligned}
S &\Rightarrow aSBC \\
&\Rightarrow aaBCBC \\
&\Rightarrow aaBHBC \\
&\Rightarrow aaBHCC \\
&\Rightarrow aaBBCC \\
&\Rightarrow aabBCC \\
&\Rightarrow aabbCC \\
&\Rightarrow aabbcC \\
&\Rightarrow aabbcc
\end{aligned}
$$

Essa gramática dependente do contexto reconhece a linguagem $\{a^n b^n c^n : n \leq 0\}$ que não é livre de contexto.

A classe das linguagens dependentes de contexto coincide com a classe das linguagens reconhecíveis pelos chamados *Autômatos Linearmente Limitados* (ALL). Um ALL é uma Máquina de Turing não determinística cuja fita é limitada linermente pelo tamanho da entrada. Pela definição de MT que vimos, o cabeçote tem liberdade para se deslocar indefinidamente para a esquerda ou para a direita. Esse modelo pressupõe uma quantidade ilimitade de espaço de memória. Nos ALLs se o tamanho da entrada é n, o tamanho da fita não pode ultrapassar $O(n)$

É possível mostrar, embora essa seja uma tarefa difícil, que existem linguagens recursivas que não estão em ALL. Não faremos essa demostração. Como todo ALL é um MT com restrições, a classe das LDCs estão propriamente contida na classe das linguagens recursivamente enumeráveis.

As LLCs são obtidas por meio de gramáticas cujas regras estão restritas aquelas com um único símbolo não terminal na cabeça (Capítulo 3). Vimos que a classe das LLCs coincide com a classe das linguagens reconhecidas por autômatos de pilha. É evidente que toda LLC é uma LDC, assim tem que ser possível simular qualquer AP usando um ALL. Isso é o que o teorema a seguir prova:

Teorema 4.5.2. *Linguagens livres de contexto são reconhecíveis por Autômatos Lineramente Limitados.*

Demonstração. Se A é uma LLC, por definição, existe uma GLC G associada a A. Vimos que deve existir G' na forma normal de Chomsky equivalente a G. Escrevemos, então, uma MT não-determinística que faz o seguinte:

1. Começa na configuração inicial $q_0\omega$ e põe $\#S$ depois de ω sendo S o estado inicial de G'.

2. Repetidamente substitui não deterministicamente a primeira variável depois de # pelo corpo das regras em G'.

3. Testa para ver se o lado esquerdo de # é igual do ao direito (Exemplo 4.1.4) e aceita a string em caso afirmativo.

Como G' está na forma normal, se $\omega \in L(G')$ então ω será reconhecida usando $2 \cdot |\omega| - 1$ derivações de regras. □

As LLCs é o *Tipo 2* da Hierarquia de Chomsky e está contida no Tipo 1. Como a linguagem $\{a^n b^n c^n : n \geq 0\}$ é dependente de contexto, mas não é livre do contexto, as LLCs estão propriamente contidas nas LDCs.

O *Tipo 3* é obtido restringindo as regras às fomas $A \to aB$ ou $A \to a$ em que $a \in \Sigma$ e $A, B \in V$. É possível mostrar que esse tipo de gramática gera exatamente as linguagens regulares. No Capítulo 2 mostramos que a classe das linguagens regulares coincide com a classe das linguagens reconhecíveis por autômatos finitos. Vimos também um exemplo de LLC que não pertence ao Tipo 3 (a linguagem $\{a^n b^n : n \leq 0\}$). Portanto, o Tipo 3 está estritamente contido no 2.

Com isso chegamos à seguinte hierarquia que resume bem os principais resultados do campo:

Regulares (Tipo 3)	$\subset$	LLC (Tipo 2)	$\subset$	LDC (Tipo 1)	$\subset$	RE (Tipo 0)
⫽ ⫽		⫼		⫼		⫼
AFD AFN		AP		ALL		MT

4.6 O Problema da Parada

As diversas variantes de Máquinas de Turing (multifitas, RAM e mesmo as não-determinísticas) são equivalentes a MTs simples. Além disso vimos que todas as Línguagens Livres de Contexto são recursivas (Teorema **??**), ou seja, toda LLC podem ser reconhecidas por uma MT. Vimos também linguagens

que não são livres de contexto e são reconhecidas por MTs. Assim, parece que chegamos em um modelo que é mais expressivo do que os que vimos até aqui e parece que chegamos em uma espécie de limite – todas as tentativas de tornar o modelo mais expressivo falharam. Nesta seção veremos que mesmo esse modelo super-expressivo tem limitações.

Para tanto precisamos fazer uma digressão sobre o conceito de infinito

O infinito de Cantor

Dizemos que dois conjuntos A e B tem a mesma *cardinalidade* se existe uma bijeção entre eles. Ou seja, se existe alguma função $f : A \to B$ que leva cada elemento de A em um elemento distinto de B (*injetora*) de forma que não sobre elementos em B (*sobrejetora*).

Note que se A e B são finitos, nossa definição garante que eles possuem a mesma quantidade de elementos (se A possui mais elementos não há como a função f ser injetora e se B possui mais elementos ela não pode ser sobrejetora). A cardinalidade representa a forma mais primitiva de contagem: uma pedra (do conjunto A das pedras) para cada carneiro (do conjunto B de carneiros). Quando extrapolamos essa definição para os conjuntos infinitos, temos alguns resultados um pouco contra-intuitivos.

Exemplo 4.6.1:

O conjunto dos naturais $\mathbb{N}$ tem a mesma cardinalidade do conjunto dos números pares.

Para mostrar isso, basta achar uma função bijetora que leve naturais em pares. A função $f(n) = 2n$ faz isso.

O exemplo anterior mostra dois conjuntos infinitos que tem a mesma cardinalidade (mesma quantidade de elementos por assim dizer). Poderíamos levantar a hipótese de que todo conjunto infinito possui a mesma cardinalidade, mas o teorema a seguir provado por Cantor no final do século XIX mostra que esse não é o caso.

Teorema 4.6.2 (Cantor)**.** *Seja A um conjunto qualquer, o conjunto $2^A := \{B : B \subseteq A\}$ (chamado conjunto das partes de A) tem cardinalidade extritamente maior do que A.*

Demonstração. Suponha por absurdo que exista uma bijeção entre $f : A \to 2^A$ e considere o conjunto $B := \{x \in A : x \notin f(x)\}$. Como $B \subseteq A$, por definição $B \in 2^A$. Se f fosse bijetora, deveria existir $x \in A$ tal que $f(x) = B$.

Se $x \in f(x)$, então $x \in B = f(x)$ e nesse caso $x \notin f(x)$ pela definição de B, o que seria uma contradição. Por outro lado, se $x \notin f(x) = B$ então, pela defição de B, temos que $x \in f(x)$, o que também seria uma contradição.

Concluímos que não existe uma função bijetora entre A e 2^A. □

Voltemos agora às MTs. Podemos representar uma MT é descrita como uma sequência de instruções com o seguinte formato:

$$q_0 a \to q_1 b D$$

Ou seja, podemos representar uma sequência de instruçõe como uma string sobre o alfabeto $\Sigma_{MT} = Q \cup \Sigma \cap \{E, D, \to, \#\}$ (o símbolo # é usado para separar as instruções). Existem infinitas MTs, ou equivalentemente, infinitas strings em Σ^*_{MT}. Pelo teorema de Cantor vimos que o conjunto $2^{\Sigma^*_{MT}}$ tem cardinalidade maior do que Σ^*_{MT}. Ou seja, existem mais linguagens do que MTs. Concluímos que deve haver linguagens que não são reconhecidas por Máquinas de Turing. Antes de mostrar um exemplo disso, vamos explorar uma consquência importante do fato de que qualquer MT pode ser descrita como uma string.

Máquina de Turing Universal

Uma MT universal U recebe $\langle M \rangle \in \Sigma^*_{MT}$ – a representação de uma MT M – e uma entrada x. A máquina U e aceita essa entrada se M aceita x e rejeita a entrada se M rejeita x[2]. Em outras palavras U reconhece a seguinte linguagem:

$$A_{MT} := \{\langle M, x \rangle : M \text{ aceita } x\}$$

A existência de uma MT universal nos mostra que se codificarmos uma única MT, a saber uma MT universal, em um harware, podemos *simular* qualquer MT como um software. Essa descoberta do começo dos anos 30 dá origem ao que hoje chamamos de *computação*.

[2]Para uma descrição compreensível da construção de uma MT universal tal qual descrita por Turing, veja `https://plato.stanford.edu/entries/turing-machine/#TuriUnivMach` (Consultado em outubro de 2020)

O Problema da Parada

Note que não dicemos que U decide A_{MT}. Se M aceita x, então, por definição U aceita $\langle M, x\rangle$, mas não estabelecemos o que ocorre se M não aceita x. Neste caso, a máquina U não pode aceitar a entrada $\langle M, x\rangle$. Ela pode rejeitar $\langle M, x\rangle$, mas podem ocorrer outras coisas. U pode, por exemplo, entrar em loop infinito. O teorema a seguir mostra que não é possível construir uma MT que decida para a entrada $\langle M, x\rangle$ se a MT M reconhece x:

Teorema 4.6.3. *A_{MT} não é recursiva.*

Demonstração. Suponha por absurdo A_{MT} seja recursiva. Por definição, deve existir uma MT H tal que:

$$H(\langle M, x\rangle) = \begin{cases} \text{aceita} & \text{se } M \text{ aceita } x \\ \text{rejeita} & \text{se } M \text{ não aceita } x \end{cases}$$

Se essa MT existisse, poderíamos trivialmente construir uma MT D que faz o seguinte:

$$D(\langle M\rangle) = \begin{cases} \text{aceita} & \text{se } M \text{ não aceita } \langle M\rangle \\ \text{rejeita} & \text{se } M \text{ aceita } \langle M\rangle \end{cases}$$

Temos então que:

$$D(\langle D\rangle) = \begin{cases} \text{aceita} & \text{se } D \text{ não aceita } \langle D\rangle \\ \text{rejeita} & \text{se } D \text{ aceita } \langle D\rangle \end{cases}$$

Contrariando a definição de D. Logo, não podem existir uma MT D e, portanto, não pode existir H que decide A_{MT}. □

A prova do teorema acima segue uma lógica similar a demonstração do Teorema de Cantor. Ambas utilizam uma técnica chamada de "diagonalização".

Corolario 4.6.4. *$\overline{A_{MT}}$ não é recursivamente enumerável.*

A_{MT} é um exemplo de linguagem recursivamente enumerável que não é recursiva.

Tese de Church-Turing

Nas seções anteriores vimos o quão expressivas são as MTs. Nesta vimos algumas limitações.

A pergunta que resta é se existe algum modelo de computação mais expressivo do que as Máquinas de Turing. Ou seja, algum modelo que reconhece um conjunto ainda maior de linguagens.

Nos anos 30 o matemático Alonzo Church levantou a hipótese de que não. A *Tese de Church-Turing*, como ficou conhecida, estabelece que não existem modelos de computação mais expressivos do que as MTs. Temos três motivos para crer que a hipótese seja válida:

1. a equivalência entre muitos modelos distintos (não só os que vimos em aqui, mas principalmente as funções recursivas e o cálcula lambda)

2. a existência de uma MT universal

3. a propria simplicidade e generalidade do modelo de Turing

Podemos manter, porém, uma postura cética e aceitar a tese enquanto não se apresenta nenhum outro modelo mais expressivo.

4.7 Redutibilidade

Na seção anterior vimos um exemplo de problema indecidível. Para provar que outros problemas também são indecidíveis usaremos uma técnica chamada redução. Uma *redução* é uma maneira de converter um problema em outro. Assim se sabemos como resolver um problema A podemos resolver outros problemas reduzindo-os a ele. Conversamente se sabemos que A não pode ser resolvido e reduzimos A a outro problema B então descobrimos que B também não pode ser resolvido.

Exemplo 4.7.1:

Considere a linguagem V_{MT} das representações de Máquinas de Turing que reconhecem a linguagem vazia, ou seja, que não reconhecem nenhuma string.

$$V_{MT} := \{\langle M \rangle : L(M) = \emptyset\}$$

Vamos reduzir o problema A_{MT} a esse problema. Considere as seguintes MTs:

$$O_\omega(x) = \begin{cases} \text{rejeita} & \text{se } x \neq \omega \\ \text{aceita} & \text{se } M \text{ aceita } \omega \end{cases}$$

Note que a máquina O_ω simula a máquina M. Além disso, já vimos no Exemplo 4.1.4 que é possível construir uma MT que verifica se duas strings são iguais.

Agora suponha por absurdo que exista uma MT R que decide V_{MT}. Poderíamos, portanto, construir a seguinta máquina:

$$S(\langle M, \omega \rangle) = \begin{cases} \text{rejeita} & \text{se } R \text{ aceita } \langle O_\omega \rangle \\ \text{aceita} & \text{se } R \text{ rejeita } \langle O_\omega \rangle \end{cases}$$

Se R aceita $\langle O_\omega \rangle$ então $L(O_\omega) = \emptyset$ e, portanto, M não aceita ω (caso contrário $\omega \in L(O_\omega)$). Ou seja, se S rejeita $\langle M, \omega \rangle$ então M não aceita ω.

Por outro lado, se R rejeita $\langle O_\omega \rangle$ então $L(O_\omega) \neq \emptyset$ e, portanto, M aceita ω. Ou seja, se S aceita $\langle M, \omega \rangle$ se e somente se M aceita ω.

Em outras palavras $L(S) = A_{MT}$.

Pelo Teorema 4.6.3 sabemos que A_{MT} é indecidível. Portanto, S não pode existir. Vimos, porém, que a existência de R implica que somos capazes de construir S. Concluímos que R, uma máquina que decide V_{MT}, não pode existir. Ou seja, V_{MT} é indecidível.

Existem várias maneiras de definir formalmente o conceito de *redução* de um problema A para um problema B. Focaremos em um tipo. A *redução por mapeamento* determina que existe uma *função computável* f que converte instâncias do problema A em instâncias de B.

Definição 4.7.2 (Função Computável)**.** *Uma função $f : \Sigma^* \to \Sigma^*$ é* computável *se existe alguma MT M que para toda entrada ω para exatamente com $f(\omega)$ na fita.*

Definição 4.7.3 (Redução por mapeamento)**.** *A linguagem A é* redutível por mapeamento *à linguagem B (escrevemos $A \leq_m B$) se existe um função computável $f : \Sigma^* \to \Sigma^*$ tal que:*

$$\omega \in A \text{ se e somente se } f(\omega) \in B$$

Derivamos dois resultados diretos dessa definição:

Corolario 4.7.4. *Se $A \leq_m B$ e B é decidível, então A é decidível.*

Demonstração. Se B é decidível então existe uma MT M_B que decide B – ou seja, que aceita todas as strings $\omega \in B$ e rejeita todas as strings $\omega \notin B$. Como $A \leq_m B$, por definição, existe f computável que reduz de A para B. Podemos então construir uma MT M_A que decide A da seguinte forma:

$$M_A(\omega) = \begin{cases} \text{aceita} & \text{se } M_B \text{ aceita } f(\omega) \\ \text{rejeita} & \text{se } M_B \text{ rejeita } f(\omega) \end{cases}$$

□

Corolario 4.7.5. *Se $A \leq_m B$ e A é indecidível, então B é indecidível.*

Exemplo 4.7.6:

Vamos usar a redução por mapeamento para mostrar que a seguinte linguagem é indecidível:

$$EQ_{MT} := \{\langle M_1, M_2 \rangle : L(M_1) = L(M_2)\}$$

Considere a função f que recebe como entrada $\langle M \rangle$ e produz como saída $\langle M, M_\perp \rangle$ em que $M_\perp$ é uma MT que rejeita qualquer entrada. É faćil notar que f é computável, basta criar uma MT que concatena na entrada a descrição da MT $M_\perp$, que é fácil de construir. Essa função reduz o problema V_{MT} ao problema EQ_{MT}, portanto, $V_{MT} \leq_m EQ_{MT}$. No Exemplo 4.7.1 vimos que V_{MT} não é decidível e concluímos que EQ_{MT} também não é.

Vimos até aqui três problemas indecidíveis – A_{MT}, V_{MT} e EQ_{MT}. Os três recebem como entrada a codificação de uma ou mais MTs. Para concluir o capítulo vamos apresentar um exemplo de problema indecidível que recebe outro tipo de entrada. O problema a seguir foi concebido por Emil

Post um cientista russo contemporâneo ao Turing que concebeu um modelo de computação muito similar ao que vimos neste capítulo. A prova da indecidibilidade desse problema é uma redução de A_{MT} a ele, mas será omitida por conter muitos detalhes técnicos pouco interessantes.

Exemplo 4.7.7:

Considere um conjunto de pares de strings $\{\langle t_1, b_1\rangle, \langle t_2, b_2\rangle, \ldots, \langle t_k, b_k\rangle\}$ em que $t_i, b_i \in \Sigma^*$. Um *emparelhamento* é uma sequência desses pares de strings de forma que a concatenação dos ts seja idêntica a concatenação dos bs. (Note que não exigimos que cada par $\langle t, b\rangle$ ocorra uma única vez.)

Se preferirem, podemos imaginar um par $\langle t, b\rangle$ como uma espécie de pedra de dominó. No exemplo a seguir temos um emparelhamento válido com o alfabeto $\Sigma = \{a, b, c\}$. A entrada é dada pelo seguinte conjunto:

$$\left\{ \left[\frac{ca}{a}\right], \left[\frac{b}{ca}\right], \left[\frac{abc}{c}\right], \left[\frac{a}{ab}\right] \right\}$$

O seguinte é um emparelhamento válido:

$$\left[\frac{a}{ab}\right] \left[\frac{b}{ca}\right] \left[\frac{ca}{a}\right] \left[\frac{a}{ab}\right] \left[\frac{abc}{c}\right]$$

Esse emparelhamento é válido pois a concatenação das strings da parte de cima do dominó coincide com a concatenação de baixo:

$$abcaaabc$$

O *Problema da Correspondência de Post* (PCP) é o seguinte. Dado um conjunto de pares de strings (peças de dominó), determinar se existe um emparelhamento válido para esse conjunto. Esse problema pode ser descrito por meio da seguinte linguagem:

$$PCP := \{\langle P\rangle : P \text{ é uma instância que possui emparelhamento }\}$$

Como adiantamos, a linguagem PCP é indecidível.

Capítulo 5

Complexidade Computacional

Até aqui nos ocupamos principalmente do problema da expressivdade de modelos de computação. Ou seja, o que é possível computar com cada modelo. Terminamos o último capítulo com um modelo bastante expressivo das Máquinas de Turing. Vimos que mesmo nesse modelo há problemas que não são computáveis, como o problema da parada.

Neste último capítulo nos voltaremos para outra questão: que problemas computacionais são resolvíveis de maneira eficiente? Por efeciente entendemo que há algum recurso escasso consumido pelo algoritmo que resolve o problema, por exemplo tempo ou espaço de memória.

5.1 Complexidade de Tempo

O *tempo de execução* de uma MT M é uma função $f : \mathbb{N} \to \mathbb{N}$ em que $f(n)$ é o número máximo de passos de derivação para uma entrada ω qualquer de tamanho n.

$$TIME(t(n)) = \{A \subseteq \Sigma^* : \exists \text{ MT simples que decide } A \text{ em tempo } O(t(n))\}$$

Exemplo 5.1.1:

$TIME(n)$ é a classe dos problemas resolvíveis em tempos *linear* no pior caso.

$TIME(n^2)$ é a classe dos problemas resolvíveis em tempo *quadrático* no pior caso.

Teorema 5.1.2. *Se $t(n) \geq n$ então toda MT multifita que consome tempo $t(n)$ é equivalente a uma MT simples que consome tempo $O(t^2(n))$.*

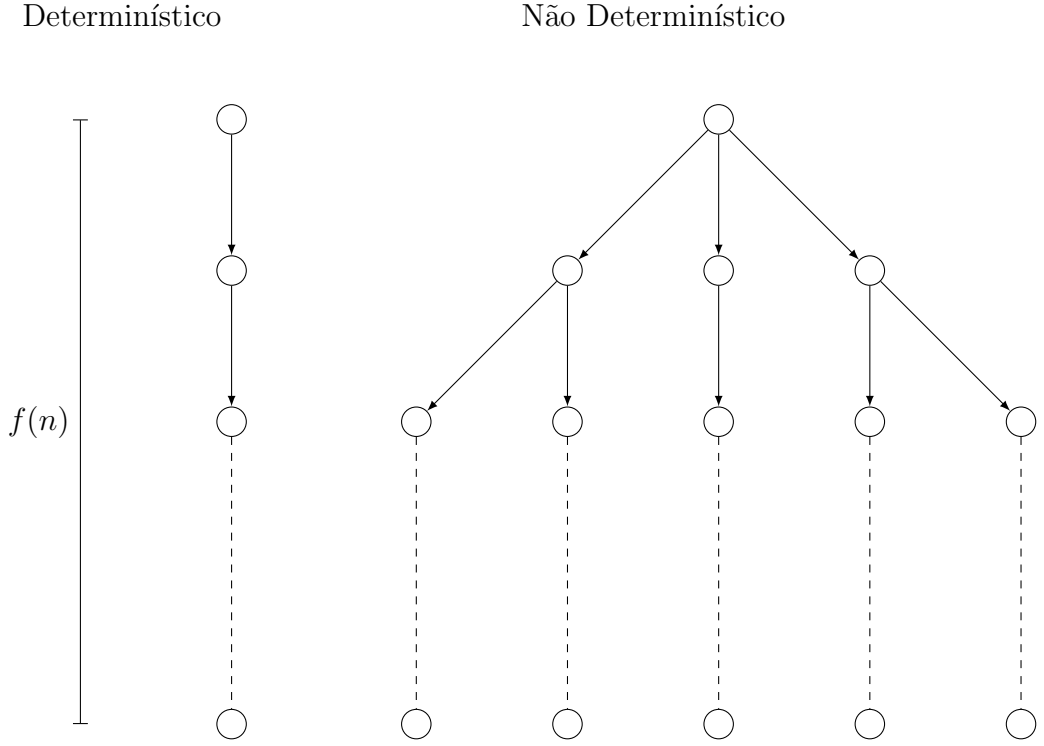
Demonstração. Considere a simulação de uma MT com k fitas que vimos no Teorema **??**.

M varre a fita em tempo $O(n)$ para obter as informação necessárias para o próximo passo.

Para executar um passo M no pior precisamos abrir um espaço em branco na fita e para isso deslocamos todo conteúdo uma posição para a direita. Nesse caso como o tamanho máximo da fita é $O(t(n))$, precisaríamos de $O(t(n))$ passos para esse deslocamento.

Assim, o tempo total de excecução é $t(n).O(t(n)) + O(n)$. Se $t(n) \geq n$ então $t(n).O(t(n)) + O(n) = O(t^2(n))$. □

O tempo de execução de uma MT não-determinística N é uma função $f : \mathbb{N} \to \mathbb{N}$ em que $f(n)$ é o número máximo de passos de *alguma* derivação de N para a entrada ω de tamanho n.



Teorema 5.1.3. *Se $t(n) \geq n$ então toda MT não-determinística que consome tempo $t(n)$ é equivalente a uma MT simples que consome tempo $2^{O(t(n))}$.*

Demonstração. Vimos no Teorema **??** como simular uma MT não-determinística N usando uma MT com 3 fitas usando uma busca em largura.

Seja b o número máximo de ramificações de na excecuçaõ N. O número total de nós da árvore é $O(b^{t(n)})$ e a excecução de cada nó toma tempo $O(t(n))$ no pior caso.

Assim, o tempo total de excecução dessa simulação é $O(t(n).b^{t(n)}) = 2^{O(t(n))}$ se $t(n) > n$.

Por fim, essa MT de três fitas pode ser simulada por uma MT simples que consome tempo $2^{O(t^2(n))} = 2^{2O(t(n))} = 2^{O(t(n))}$. □

$NTIME(t(n)) = \{A \subseteq \Sigma^* : \exists \text{ MT não-det. que decide } A \text{ em tempo } O(t(n))\}$

Vamos definir duas classes de complexidade de tempo. A classe P contém todas as linguagens decidíveis por MT simples em tempo polinomial e a classe NP que contém todas as linguágens decidíveis por MTs não-determinísticas em tempo polinomial:

$$P = \bigcup_k TIME(n^k)$$

$$NP = \bigcup_k NTIME(n^k)$$

É evidente que toda linguagem em P pertence a NP. Ou seja, $P \subseteq NP$. Não sabemos, porém, se é verdade que $NP \subseteq P$. Em outra palavras, se existem soluções polinomiais em MTs simples para os problemas em que possuem solução em MTs não-determinísticas. Esse é o principal problema em aberto na computação.

Uma forma alternativa de apresentar a classe de problemas NP é por meio de um *oráculo*. Um *oráculo* (ou verificador) para uma linguagem A é um algotimo V tal que:

$$A = \{\omega : V \text{ aceita } \langle \omega, o \rangle \text{ para alguma string } o\}$$

A string o na descrição acima é chamada de *certificado*.

Exemplo 5.1.4:

Seja $L = \{p_1, \ldots, p_n, \bar{p_1} \ldots \bar{p_n}\}$ uma alfabeto. Uma *cláusula* sobre L é uma string $c \in L^*$ e uma *fórmula* é uma string $f \in (L \cup \{;\})^*$. Uma *valoração* é uma função $v : L \to \{0, 1\}$ tal que $v(p) = 1$ sse $v(\bar{p}) = 0$. Uma valoração v *satisfaz uma cláusula* c se $v(l) = 1$ para *algum* l em c e v *satisfaz uma fórmula* $f = c_1; c_2; \ldots; c_m$ se ele satisfaz *todas* as cláusulas $c_1, \ldots, c_n$.

Definimos o *problema da satisfatibilidade* da seguinte forma:

$$SAT = \{f \in (L \cup \{;\})^* : \text{existe } v \text{ que satisfaz } f\}$$

Uma valoração pode ser descrita como uma string $o \in \{0, 1\}^*$. (Por exemplo, a string 101 indica que $v(p_1) = 1$, $v(p_2) = 0$ e $v(p_3) = 1$).

É fácil construir uma MT V que recebe uma fórmula $f \in (L \cup \{;\})^*$ e um string $o \in \{0, 1\}^*$ e aceita se a valoração v representada por o satisfaz f e rejeita caso contrário. Essa verificação pode ser feita em tempo polinomial em relação a $|f|$. Note que podemos descrever o problema SAT da seguinte forma:

$$SAT = \{f \in (L \cup \{;\})^* : V \text{ aceita } \langle f, o\rangle \text{ para algum } o \in \{0, 1\}^*\}$$

Dizemos, portanto, que V é um *verificador polinomial* para SAT.

Teorema 5.1.5. *Uma linguagem $A \in NP$ sse exsite um verificador polinomial para A.*

Demonstração. Se $A \in NP$ então, por definição, existe uma MT não-determinística N que decide A em tempo polinomial. Considere uma string ω qualquer. Se $\omega \in A$ então N aceita, senão rejeita. De qualquer forma existe um ramo da excecução de N que termina em menos de $O(n^k)$ passos. Seja o a codificação desse ramo (a string que indica a cada passo qual o caminho que foi seguido). Simulando N como uma MT com três fitas, e colocando o na terceira, decidimos se ω é aceito ou não em tempo polinomial.

Agora considere o outro lado. Seja V ium verificador para A que decide se a entrada é aceita em tempo $O(n^k)$. Escolhemos não deterministicamente uma string o com tamanho máximo n^k. Em cada ramo e excecutamos V sobre $\langle \omega, o\rangle$ para um o distinto e aceitamos ω se V aceitar $\langle \omega, o\rangle$ para algum o. Se nenhum ramo V aceitar a entrada então rejeitamos ω. □

Temos, portanto, que um problema está na classe NP se existe um verificador polinomial para ele. Tal verificador estabelece com auxílio de um certificado, se a entrada é aceita. Podemos definir uma outra classe de problemas que possuem um verificador que decide em tempo polinomial se a entrada é *rejeitada* com auxílio e uma string chamda de *desqualificador*. Esse classe é chamada coNP.

Exemplo 5.1.6:

Uma fórmula proposicional f na Forma Normal Conjuntiva é *válida* ou uma *tautologia* se para toda valoração v temos que $v(f) = 1$.

$TAUT = \{f \in (L \cup \{;\})^* : V \text{ rejeita } \langle f, o \rangle \text{ para algum } o \in \{0,1\}^*\}$

O problema TAUT é, portanto, um problema CoNP.

5.2 NP-completude

Na última seção definimos as classes P e NP e mencionamos que a pergunta se $P \stackrel{?}{=} NP$ é um problema em aberto na computação. O que faremos então será tentar classificar que problemas são mais "fáceis" ou mais "difíceis" do que outros.

Dizemos que uma função $f : \Sigma^* \to \Sigma^*$ é *computável em tempo polinomial* se existe um polinômio p e uma MT que ao receber $\omega \in \Sigma^*$ para depois de $p(|\omega|)$ passos e devolve $f(\omega)$.

Uma linguagem A é *polinomialmente redutível* a B (escrevemos $A \leq_p B$) se existe $f : \Sigma^* \to \Sigma^*$ que seja computável em tempo polinomial e tal que $\omega \in A$ sse $f(\omega) \in B$.

O teorema a seguir mostra que a redutibilidade polinonimal preserva o pertencimento na classe P:

Teorema 5.2.1. *Se $A \leq_P B$ e $B \in P$ então $A \in P$.*

Demonstração. Seja M uma MT que decide B em tempo polinomial e seja f a redução polinomial de A em B. Construímos uma MT N da seguinte forma: N recebe ω e computa $f(\omega)$ então roda M sobre $f(\omega)$.

Pela definição de f, M aceita $f(\omega)$ sse $\omega \in A$ e, portanto, N aceita ω. Além disso, N é polinomial pois cada passo é polinomial e polinômios são fechados por composição. □

Exemplo 5.2.2:

Considere o seguinte problema de decisão, uma restrição do problema SAT.

$3SAT = \{f \in SAT : \text{cada clásula de } f \text{ tem tamanho exatamente } 3\}$

Vamos mostrar que $SAT \leq_P 3SAT$.

A transformação vai substituir cada cláusula $c_i = l_1 \dots l_n$ de cada fórmula $f = c_1; c_2; \dots; c_m$ pela seguinte sequência de cláusulas: $l_1 l_2 m_1; \overline{m_1} l_3 m_2; \overline{m_2} l_4 m_3; \dots; \overline{m_{n-3}} l_{n-1} l_n$. Essa transformação é claramente polinomial e é possível mostrar que $f \in SAT$ sse essa nova fórmula também for satisfatível.

Uma linguagem A é *NP-completa* se:

- $A \in NP$ e
- para todo $B \in NP$ temos que $B \leq_P A$

Os seguintes são corolários da definição de NP-completude:

Corolario 5.2.3. *Seja A uma linguagem NP-completa, se $A \in P$ então $P = NP$.*

Corolario 5.2.4. *Se A é NP-completa e $A \leq_P B$ então B também é NP-completa.*

Ou seja, intuitivamente as linguagens NP-completas são as mais difíceis dentro da classe NP. Além disso, se conhecemos uma linguagem NP-completa, então podemos inferir que outras linguagens também o são por redução polinomail.

Resta mostrar que pelo menos uma linguagem é NP-completa.

Teorema 5.2.5 (Cook-Levin)**.** *A linguagem SAT é NP-completa.*

Demonstração. Mostramos na última seção que $SAT \in NP$. Temos que mostrar que $B \leq_P SAT$ para todo $B \in NP$. Partimos da constatação de que se $B \in NP$, então existe uma MT não-determinística N que decide B em tempo polinomial n^k.

Um *tableau* para N sobre a entrada ω é uma tabela $n^k \times n^k$ cujas linhas são configurações de um ramo de N com entrada ω. Assim, a primeira linha contém a configuração inicial e deve haver um tableau que contém uma configuração de aceitação para cada $\omega \in B$.

Vamos representar o tableuau como um fórmula f que é satisfatível sse existe um tableau que aceita ω.

Seja $C = Q \cup \Gamma \cup \{\#\}$, temos uma variável $x_{i,j,s}$ para cada $i, j \in \{1, \ldots, n^k\}$ e cada $s \in C$. A ideia é que uma valoração v satisfaz $x_{i,j,s}$ se a célula $\langle i, j \rangle$ no tableau contém o símbolo s. Projetaremos a fórmula f de modo que uma valoração que satisfaz f corresponde a um tableau que reconhece ω.

$$f_c = x_{1,1,s_1} x_{1,1,s_2} \ldots x_{1,1,s_n}; \overline{x_{1,1,s_1} x_{1,1,s_2}}; \overline{x_{1,1,s_1} x_{1,1,s_3}} \ldots; x_{1,2,s_1} x_{1,2,s_2} \ldots$$

A fórmula $f_c \in SAT$ sse cada célula contém exatamente um símbolo.

Escrevemos a fórmula f_i de forma que $f_i \in SAT$ sse a primeira linha do tableau contém a configuração inicial de N.

$$f_a = x_{1,1,q_a} x_{1,2,q_a} \ldots x_{n^k,n^k,q_a}$$

A fórmula $f_a \in SAT$ sse alguma linha é uma configuração de aceitação.

Uma *janela* 2×3 no tableua é *legal* se não viola as ações especificadas pela função de transição de N (Exemplo 5.2.6). Escrevemos f_m como a conjunção de todas as janelas legais. Ou seja, f_m é tal que $f_m \in SAT$ sse a configuração da linha i segue da configurção da linha $i - 1$ em N.

Assim, a fórmula $f = f_c; f_i; f_a; f_m \in SAT$ sse $\omega \in B$ para algum $B \in NP$. □

Exemplo 5.2.6:

Considere que $\Delta(q_1, b) = \{\langle q_2, c, E \rangle, \langle q_2, a, D \rangle\}$, as seguintes janelas são legais:

a	q_1	b
a	a	q_2

a	q_1	b
q_2	a	c

Corolario 5.2.7. *3SAT é NP-completa*

5.3 Problemas NP-completos

Na seção anterior vimos que há uma conjunto de problemas chamados NP-completos. Qualquer problema NP pode ser reduzido a um problema NP-completo. Assim, esses são os mais difíceis entre os problemas em NP.

Vimos também que para provar que um problema é NP-completo podemos usar a técnica da redução polinomial. Se mostrarmos que é possível reduzir um problema NP-completo A a nosso problema B, então B e deve ser pelo menos tão difícil quanto A. Portanto, B deve também ser um problema NP-completo. Mostrar que um problema é NP-completo não é uma prova de que ele não pode ser resolvido em tempo polinomial, mas indica que a dificuldade em encontrar uma solução polinomial não é uma incapacidade do programador, mas uma questão em aberto na ciência.

Os problemas NP-completos ocorrem em diversas áreas distintas da computação. Nesta seção apresentaremos sem as provas de redução uma lista de problemas NP-completos.

Exemplo 5.3.1:

O primeiro problema que trataremos é as vezes chamado de *problema do caixeiro viajante*. Um caixeiro viagente, ou um mascate, é um vendedor que viaja de cidade em cidade levando suas mercadorias. Imagino um caixeiro que precisa passar por um conjunto de cidades ligadas por uma malha de estradas. Ele precisa passar por todas as cidades, mas quer evitar de passar duas vezes por uma mesma cidade, visto que isso seria ineficiente. O problema do mascate é saber se existe uma forma de passar pelas cidades todas sem repetir.

O problema do caixeiro viajante poder ser modelado como um problema de grafos.

Um *grafo* é uma estrutura formada por um conjunto V cujos elementos são chamados de *vértices* e um conjunto de pares de elementos $E \subseteq \{\{v, w\} : v, w \in V\}$ chamado de *arestas*. Se $\{v, w\} \in E$ então dizemos que os vértices v e w são *adjacentes*. Um *caminho* em um grafo é uma sequencia de nós distintos $v_1, v_2, \ldots, v_n \in V$ tal que para todo $i \in 1, \ldots n-1$ temos que v_i e v_{i+1} são adjacentes. Um *ciclo* em um grafo é um caminho $v_1, v_2, \ldots, v_n$ tal que v_n é adjacente a v_1 . Um *ciclo hamiltoniano* é um ciclo em um grafo que contém todos os vértices em V.

Podemos representar então as cidades como nós em um grafo $G = \langle V, E \rangle$ e as estradas como arestas. O problema do caixeiro viajante se resume então ao de decidir se existe um ciclo hamiltonia em G.

Note que se conhecemos um ciclo hamiltoniano, podemos conferi-lo em tempo polinimial. Esse ciclo é um certificado e, portanto, esse problema está em NP. Além disso, é possível, embora não iremos fazê-lo, reduzir o problema 3-SAT ao problema dos ciclos hamiltonianos. Portanto, esse problema é NP-completo.

Exemplo 5.3.2:

Imagine agora que você está organizando uma festa. Cada convidado conhece outros convidados, mas não necessariamente todos. Alguns amigos você conheceu em um mesmo contexto, eles fazem parte de uma mesma comunidade. Nesse grupo todos conhecem todos. Você se pergunta então qual será que a maior comunidade entre nesse conjunto de convidados.

Mais uma vez podemos modelar esse como um problema de grafos. Os convidados são os vértices do seu grafo e uma aresta ocorre se eles se conhecem. Uma conjunto de nós em que todos são adjacentes a todos os demais é chamado e um *clique*. Dado um grafo, o *problema do clique* consiste em decidir se existe um clique no grafo com um certo tamanho K.

Se conhecemos um conjunto que resolve o problema, podemos verificá-lo em tepo polinomial. Portanto, temos um cetificado e o problema está em NP. É possível mostrar também que o problema do clique é NP-completo.

Exemplo 5.3.3:

Imagina que você possui um mapa e um estojo com k lápis de cores diferentes. Sua tarefa é colorir o mapa de forma que nunca dois países sejam coloridos com a mesma cor.

Novamente esse problema pode ser modelado como um problema de grafos.

Neste caso, cada país representa um nó e países que fazem fronteira são ligados por uma aresta. O *problema da coloração* em um grafo é exatamente o de pintar os vértices com k cores distintas de forma que vértices adjacentes não sejam pintados da mesma cor.

Uma instância desse problema ocorre quando $k = 3$, ou seja, quando temos 3 cores. Novamente, se nos é dada uma mapeamento de cores – que nó está pintado de que cor – podemos verificá-lo em tempo polinomial. Portanto, temos um certificado polinomial e o problema está em NP. Além disso, é também mostrar que, no caso em que $k = 3$ esse problema é NP-completo.

Exemplo 5.3.4:

Suponha que você possui um conjunto de pedaços de rodapé de diferentes tamanhos e uma parede com um tamano determinado na qual você gostaria de aplicá-lo. Qualquer pedaço desses rodapés pode ser aplicado em qualquer ordem, mas você gostaria de que ao final o comprimento total coincida exatamente com o comprimento da parede.

Podemos modelar esse problema da seguinte forma. Temos um conjunto de número inteiros $c_1, \ldots, c_n$ que representam os comprimentos dos rodapés e o comprimento da parede l. Desejamos selecionar selecionar um subonjunto $S \subseteq \{1, \ldots n\}$ tal que $\sum_{i \in S} c_i = l$.

Se temos o subconjunto S basta somar os elementos para verificar se a solução é válida. Isso certamente pode ser feito em tempo polinomial e, poranto, o *problema da soma dos subconjuntos* está em NP. É mais dificil, mas é possível mostrar que esse problema é em NP-completo.

Exemplo 5.3.5:

Suponha que você está em uma sala cheia de itens preciosos e uma mochila. Você sabe o valor dos itens e sabe o peso de cada um. Sua mochila tem um limite de capacidade de peso que você também conhece. Seu objetivo é determinar se é possível guardar na mochila uma quantidade de itens que ultrapasse um certo valor K, mas não estoure a capacidade W da mochila.

Podemos modelar esse problema da seguinte forma. Para cada item i temos seu peso que é dado por um inteiro w_i e seu valor dado por outro inteiro v_i. Existe um subconjunto $S \subseteq \{1, \ldots, n\}$ tal que $\sum_{i \in S} w_i \leq W$ e $\sum_{i \in S} v_i \geq K$?

Mais uma vez, se nos for dado o S podemos verificar se ele satisfaz as condições em tempo polinomial. Portanto, o *problema da mochila* está em NP. Além disso, é possível mostrar que este também é um problema NP-completo.

Transição de fase

Até aqui vimos resultados teóricos sobre NP-completude. Para completar nosso estudo mostraremos aqui uma constatação empírica.

O problema SAT, além de ter sido o primeiro problema demostradamente NP-completo, é um dos problemas mais estudados na classe NP. Importantes competições para avaliara os melhores algoritmos e implementações de para esse problema são organizadas anualmente desde 2002. Esses programas se tornaram tão eficientes que muitas vezes compensa traduzir um problema NP qualquer para SAT e então resolvê-lo usando um desses programas.

Aqueles que estudam o programa SAT notaram em meados dos anos 90 que as boas implementações de algoritmos para resolver o problema tem uma característica em comum. Esses algoritmos são eficientes para decidir a satisfatibilidade de fórmulas que possuem poucas cláusulas e muitas variáveis. Nesse caso a imensa maior parte das instâncias é satisfatível e isso é fácil de aferir computacionalmente. Esses algoritmos são bastante bons também para decidir a satisfatibilidade de fórmulas que possuem muitas cláusulas e poucas variáveis. Nesses caso a situação é inversa. A maior parte das fórmulas é insatisfatível e embora aferir isso seja um pouco mais difícil, o tempo de

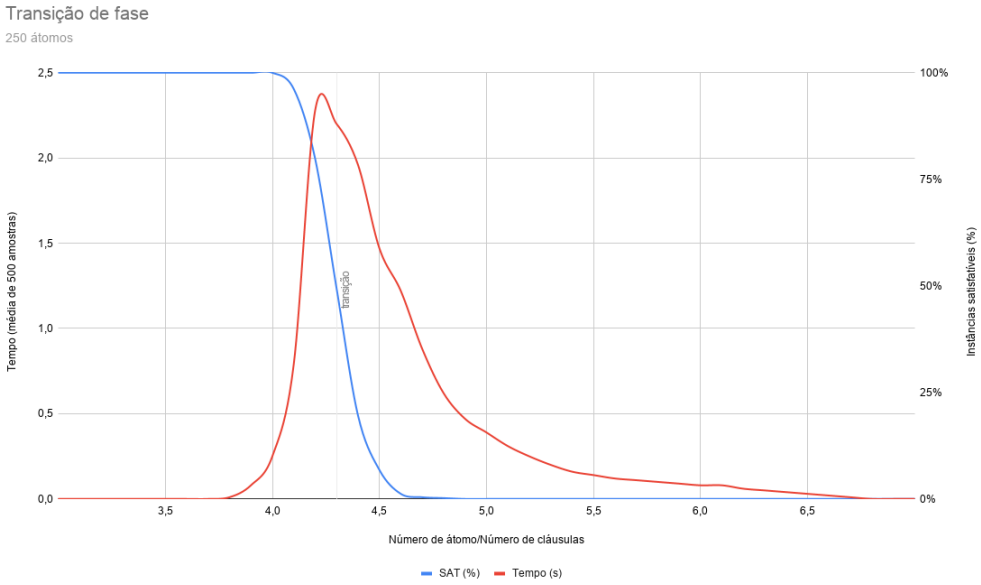


Figura 5.1: Transição de fase. A difículdade do problema 3-SAT ocorre no ponto em que há a mesma quatidade de instâncias satisfatíveis e insatisfatíveis.

processamento é ainda bastante baixo. Os problemas que são realmente difíceis de processar são aqueles em que há uma proporção equilibrada de fórmulas satisfatíveis e insatisfatíveis.

A Figura 5.3 plotamos de processamento do problema 3-SAT. O eixo y desse gráfico representa o tempo médio de processamento de instâncias aleatórias desse problema. O eixo x representa a razão $\frac{L}{N}$ em que L é o número de cláusulas das instâncias e N o número de variáveis proposicionais. O gráfico apresenta um pico quando essa fração é $4,3$ o que coincide com o ponto em que o número de instâncias satisfatíveis é igual ao número de instâncias insatisfatíveis.

Esse fenômeno é chamado *transição de fase* em analogia a fenômenos físicos com a temperatura da água que passa por um situação singular na transição entre os estados.

5.4 Relação entre as classes de complexidade de tempo

A discussão da seção anterior indica o pouco que conhecemos sobre a dificuldade de um problema. A classe dos problemas NP-completos é a classe dos mais difíceis dentre os problemas NP. Porém, não sabemos qual é relação entre a classe P e a classe NP. Acreditamos que essas classes sejam distintas, mas isso nunca foi provado.

A esta altura talvez seja interessante dar um passo atrás e nos perguntar uma coisa mais básica. Conseguimos garantir que dado mais tempo somos capazes de resolver mais problemas? A intuição que construímos no curso de Introdução à Analise de Algoritmo é de que sim. Existem problemas para os quais existem solução quadrática, mas não existe solução linear.

O *Teorema da Hierarquia* foi possivelmente o primeiro resultado importante da teoria da complexidade.

Teorema 5.4.1. *Hierarquia Para qualquer função* $t : \mathbb{N} \to \mathbb{N}$ *onde* $t(n) > n$

$$TIME(t(n)) \subset TIME(O(t^3(n))$$

Demonstração. A afirmação $TIME(t(n)) \subseteq TIME(t(n)log^2(t(n)))$ é trivial. O difícil é mostrar que essas classes são diferntes. Ou seja, que existe uma lingaguem que está em $TIME(t(n)log^2(t(n)))$, mas não está em $TIME(t(n))$.

Vamos seguir um argumento de diagonalização similar ao da prova da indecidibilidade do problema da parada. Primeiro considere a seguinte linguagem:

$$H_t = \{\langle M, \omega\rangle : \text{ M aceita } \omega \text{ em no máximo } t(|\omega|) \text{ passos } \}$$

Para mostrar que lingaugem $H_t \in O(t^3(n))$ precisaríamos mostrar que é possível construir uma Máquina Universal de Turinal que simula M em $O(t^3(n))$ passos. Essa é uma demonstração construtiva não muito interessante. Antes de passar para a próxima parte da demonstração, apenas comentaremos que é simples construir uma simulação de M usando um MT com 3-fitas: uma que guarda a entrada ω, uma que produz a saída e outra que processa a simulação é relativamente fácil de construir. Como vimos na Seção **??** é possível então transformar essa MT com 3-fitas em uma MT simples. Esse caminho resvole o problema em tempo proporcional a $O(t^3(n))$ que é suficiente para o que pretendemos mostrar a seguir. Cabe aqui comentar que

é possível construir uma MT universal bem mais eficiente – $O(t(n)log^2(t(n)))$ –, mas isso não é necessário para os resultados dessa seção.

A parte interessante da demostração é provar que $H_t \notin TIME(t(\lfloor \frac{n}{2} \rfloor))$. Suponha por absurdo o contrário. Neste caso, seria possível construir a seguinte MT:

$$D_t(\langle M \rangle) = \begin{cases} \text{aceita} & \text{se } M_{H_t} \text{ não aceita } \langle M, M \rangle \\ \text{rejeita} & \text{se } M_{H_t} \text{ aceita } \langle M, M \rangle \end{cases}$$

Note que D_t processa $\langle M, M \rangle$ no mesmo tempo $t(\lfloor \frac{2n+1}{2} \rfloor) = t(n)$ que M_{H_t}.

Podemos então repetir o mesmo argumento do problema da parada: O que ocorre se passarmos a descrição $\langle D_t \rangle$ como entrada para D_t? Se a entrada é aceita então M_{H_t} não aceita $\langle D_t, D_t \rangle$, mas pela definião de H_t isso significa que D_t não aceita $\langle D_t \rangle$ o que é uma contradição. Se a entrada não é aceita então M_{H_t} aceita $\langle D_t, D_t \rangle$ e também chegamos em uma contradição. Concluímos que $H_t \notin TIME(t(\lfloor \frac{n}{2} \rfloor))$.

Juntando as duas partes existe um problema que não está em $TIME(t(n))$, mas está em $TIME(t(2n+1)^3)$. □

Vamos introduzir agora mais uma classe de complexidade. A classe $EXPTIME$ contém todos os problemas que podem ser decididos por uma Máquina de Turing determinísitica em tempo exponencial em relacão ao tamanho da entrada. O teorema da hierarquia nos mostra que essa classe está propriamente contida na classe P

Corolario 5.4.2.

$$P \subset EXPTIME$$

Demonstração. Partimos do fato conhecido que $P \subseteq TIME(2^n)$, ou seja, qualquer polinômio eventualemente se torna menor do que 2^n. Mas pelo Teorema da Hierarquia temos que $TIME(2^n) \subset TIME(2^{O(n^3)}) \subseteq EXPTIME$. Portanto, $P \subset EXPTIME$. □

Vamos resumir o que sabemos até agora sobre as classes de complexidade. Apresentamos quatro classes:

1. P: a classe dos problemas decidíveis em tempo polinomial por uma MT determinística.

2. NP: a classe dos problemas que possuem certificado polinomial.

3. $coNP$: a classe dos problemas que possuem desqualificador polinomial.
4. $EXPTIME$: a classe dos problemas decidívei e tempo exponencial por um MT determinística.

Sabemos que $P \subseteq NP$ e que $P \subseteq coNP$. Além disso, quando introduzimos as MT não determinísticas, vimos que é possível simular qualquer uma delas em uma MT determinística. Essa simulação toma tempo exponencial e, portanto, $NP \subseteq EXPTIME$. Não é difícil perceber que da mesma forma temos que $coNP \subseteq EXPTIME$. Por fim, acabamos de demonstrar que $P \neq EXPTIME$. Sabemos, portanto que existem problemas que podemos resolver em tempo exponencial, mas que não são resolvíveis em tempo polinomial. Não sabemos, de fato esse é o maior problemas em aberto na computação (!), se $P \neq NP$. Na verdade não sabemos praticamente mais nada sobre as relações entre essas classes do que foi aqui exposto.

5.5 Complexidade de Espaço

Até aqui nos ocupamos em estudar problemas que podem ser resolvidos com Máquinas de Turing que possuem uma limitação no tempo de processamento. Para completar nosso estudo sobre complexidade computacional, investigaremos classes de complexidade de problemas que podem ser resolvidos por MTs com espaço de memória limitado.

No modelo das MTs, o espaço é avaliado pelo número de células da fita que foram preenchidos. Como no caso da complexidade de tempo, mediremos a complexidade de espaço de uma MT como uma função $f : \mathbb{N} \to \mathbb{N}$ em que n é o tamanho da entrada e $f(n)$ o número máximo de células preenchidas.

$$SPACE(f(n)) = \{L : \exists MT \text{ det. que decide } L \text{ usando espaço } O(f(n))\}$$

No caso de uma MT não-determinística, a complexidade de espaço é medida no pior caso. Ou seja, $f(n)$ é o maior número de células preenchidas sobre qualquer um dos ramos.

$$NSPACE(f(n)) = \{L : \exists MT \text{ não-det. que decide } L \text{ usando espaço } O(f(n))\}$$

Nos interessa particularmente a classe $PSPACE$ que é a classe de todas as linguagens decidíveis usando espaço polinomial.

$$PSPACE = \bigcup_k SPACE(n^k)$$

Analogamente a classe $NPSPACE$ é a classe das linguagems decidíveis usando espaço polinomial em MTs não determinísticas. O teorema a seguir nos ajuda a entender a relação entre essas duas classes:

Teorema 5.5.1 (Savitch)**.** *Para qualquer função $f : \mathbb{N} \to \mathbb{N}$ onde $f(n) > n$:*

$$NPSPACE(f(n)) = SPACE(f^2(n))$$

Demonstração. Para verificar se uma MT não-determinística N aceita uma entrada ω, vamos simulá-la usando uma MT determinística M, como fizemos na Capítulo **??**. Desta vez, porém, precisamos cuidar de medir o espaço que estamos ocupando com essa simulação.

A primeira conta que precisamos fazer é calcular o número de possíveis de configurações de uma MT. Definimos uma configuração como uma string da forma $\omega_1 q \omega_2$ em que q é o estado atual e ω_1 e ω_2 são strings que representam o que está na fita antes e depois da cabeça de leitura. O tamanho da string $\omega_1 \cdot \omega_2$ é no máximo $f(n)$ por definição. O número de configurações possíveis é, portanto, no máximo $|Q|.|\Sigma|^{f(n)}$. Os valores $|Q|$ e $|\Sigma|$ são constantes que dependem apenas da MT. Portanto o número de configurações possíveis é $c^{f(n)}$ para uma constante c.

Podemos então imaginar um grafo em que os nós são as configurações e existe uma aresta entre duas configurações C_i e C_j se $C_i \Rightarrow C_j$. O que precisamos testar é se existe um caminho da configuração inicial C_0 para alguma das possíveis configurações de aceitação.

Poderíamos resolver isso por meio de uma busca em profundidade. Se recordarem da busca em profundidade em um grafo, ela usa uma chamada recursiva ou, equivalentemente, uma pilha. Essa pilha deve armazenar os nós que foram visitados. No pior caso, todos os nós são visitados até chegar em uma configuração de aceitação, portanto, seria necessário armazenar $c^{f(n)}$ nós.

A solução de Savitch, embora muito ineficiente do ponto de vista do tempo de execução, resolve o problema com um gasto muito modesto de espaço.

PATH(x, y, i)
1 ▷ Recebe dois nós x e y e um inteiro i
2 ▷ Verifica se existe caminho de x até y em G de tamanho máximo 2^i
3 **if** $i = 0$
4 **then return** x é adjacente a y?
5 **for** todos nós z
6 **then return** PATH$(x, z, i-1)$ e PATH$(z, y, i-1)$
7 **return** falso

Esse algoritmo é recursivo e, portanto, precisa manter uma pilha de recursão. O tamanho máximo dessa pilha é i.

Começamos da representação da configuração inicial C_0. Para cada uma das possíveis configurações finais usamos o algoritmo acima para verificar se há um caminho até ela. Como vimos, o tamanho máximo desse caminho é $c^{f(n)}$. No pior caso, portanto, $2^i = c^{f(n)}$ o que daria $i = lg(c^{f(n)}) = O(f(n))$.

Portanto temos que armazenar no máximo $O(f(n))$ configurações cada qual com tamanho máximo $O(f(n))$. Concluímos que nossa simulação ocupa espaço $O(f^2(n))$. □

Corolario 5.5.2.

$$PSPACE = NPSPACE$$

Demonstração. Segue do teorema anterior e o fato de que o quadrado de qualquer polinômio é um polinômio. □

Algumas relações entre a classe $PSPACE$ e as classes de complexidade de tempos são simples. Primeiro note que em tempo polínomial o máximo que somos capazes de preencher é uma quantidade polinomial de células. Portanto temos que $P \subseteq PSPACE$ e $NP \subseteq NPSPACE$, mas acabamos de ver que $PSPACE = NPSPACE$. Concluímos então que $NP \subseteq PSPACE$.

Além disso, como vimos na prova do Teorema **??**, uma máquina que usa espaço limitado por uma função $f(n)$ pode ter no máximo $c^{f(n)}$ estados diferentes. Como c elevado a qualquer polinômio é uma função exponencial, temos que $PSPACE \subseteq EXPTIME$.

O diagrama abaixo resume todas as relações entre classes de complexidade que vimos:

$$\begin{array}{ccccccc} & & coNP & & & & \\ & \subsetneq & & \subsetneq & & & \\ P & & & & PSPACE & \subseteq & EXPTIME \\ & \subsetneq & & \subsetneq & & & \\ & & NP & & & & \end{array}$$

Apêndice A

Exercícios

A.1 Exercícios do Capítulo 2

Exercício 1: Para cada uma das seguintes expressões regulares dê uma string na linguagem representada por ela e uma string que não está nessa linguagem.

a) $(ab \cup \epsilon)b^*$

b) $(ab)^\star bb$

c) $(a \cup b)ba^\star$

d) $(aa)^\star(bb)^\star bb$

Exercício 2: Dê o diagrama de estado **e** a descrição formal de AFDs que reconheçam as seguintes linguagens:

a) $\{\omega \in \{0,1\}^* : \omega$ começa com 1 e termina com $0\}$

b) $\{\omega \in \{0,1\}^* : \omega$ contém a substring $000\}$

c) $\{0,1\}^* - \{\varepsilon\}$

d) $\{\omega \in \{0,1\}^* : \omega$ começa com 1 e tem comprimento par $\}$

Exercício 3: Dê o diagrama de estados de AFNs que reconheçam a linguagem:

a) $0^\star 1^\star$ com dois estados.

b) $(01)^\star$ com três estados.

c) $(0 \cup 1)$ com três estados.

d) $\{\omega \in \{0,1\}^\star : \omega$ começa com 0 e tem comprimento par ou começa com 1 e tem comprimento ímpar $\}$

Exercício 4: Seja $A = \{\omega \in \{0,1\}^* : \omega$ começa com 1 e termina com $0\}$ e $B = \{\omega \in \{0,1\}^* : \omega$ começa com 0 e tem comprimento par ou começa com 1 e tem comprimento ímpar $\}$. Desenhe o diagrama de estados para AFN que reconheça:

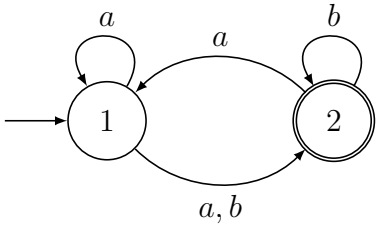
a) $A \circ B$

b) $B \circ A$

c) $A \cup B$

d) B^*

Exercício 5: Use o método visto em sala para desenhar o diagrama de estados AFD que reconheça a mesma linguagem que o seguinte diagrama AFN reconhece. Em seguida desenhe o mesmo AFD omitindo os estados supérfluos.



Exercício 6: Use o método visto em aula para encontrar uma expressão regular que reconhece a linguagem reconhecida pelo segundo AFD desenhado acima.

A.2 Exercícios do Capítulo 3

Exercício 7: O que é uma gramática ambígüa? A seguinte gramática $G = \langle V, \Sigma, R, E \rangle$, cujas regras R estão descritas a seguir, é ambígüa?

$$E \rightarrow E \wedge E | E \vee E | p | \neg p$$

Exercício 8: Desenhe o diagrama de estados de um autômato com pilha que reconhece a seguinte linguagem[1]:

$$A = \{\omega.\omega^R : \omega \in \{0,1\}^*\}$$

Exercício 9: Mostre que a linguagem do exercício anterior não é regular.

Exercício 10: Mostre uma GLC associada a cada uma das linguagens abaixo:

a) $\{\omega \in \{0,1\}^* : \omega$ possui pelo menos dois 1s$\}$

b) $\{\omega.\omega^R : \omega \in \{0,1\}^*\}$

c) $\{0^n 1^n : n \geq 0\}$

Exercício 11: Use o teorema visto em aula para construir um autômato com pilha a partir da gramática $G = \langle V, \Sigma, R, E \rangle$, cujas regras R estão descritas a seguir:

$$\begin{aligned} E &\rightarrow C \wedge C | C \\ C &\rightarrow L \vee L | L \\ L &\rightarrow p | \neg p \end{aligned}$$

Exercício 12: Mostre que a seguinte linguagem não é livre de contexto:

$$\{0^n 1^n 0^n 1^n : n \geq 0\}$$

[1] Lembre-se que ω^R é ω com os símbolos invertidos

A.3 Exercícios do Capítulo 4

Exercício 13: Considere a Máquina de Turing $M = \langle Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r \rangle$ em que $Q = \{q_0, q_1, q_a, q_r\}$, $\Sigma = \{a, b\}$, $\Gamma = \{a, b, \sqcup\}$, e δ é o seguinte:

$$\begin{aligned}
\delta(q_0, a) &= \langle q_0, a, D \rangle \\
\delta(q_0, b) &= \langle q_0, b, D \rangle \\
\delta(q_0, \sqcup) &= \langle q_1, \sqcup, E \rangle \\
\delta(q_1, a) &= \langle q_a, a, D \rangle \\
\delta(q_1, b) &= \langle q_r, b, D \rangle \\
\delta(q_1, \sqcup) &= \langle q_r, \sqcup, D \rangle
\end{aligned}$$

Para cada uma das seguintes strings, escreva as configurações da máquina, da inicial até a final e indique se a string é aceita ou rejeitada:

1. aaa
2. aba
3. aab
4. bbb

Exercício 14: Construa uma MT que decide se a string $\omega \in \{a, b\}^*$ começa com a e termina com b.

Exercício 15: Construa uma MT que decide se a string $\omega \in \{0\}^*$ tem comprimento par.

Exercício 16: Explique com suas palavras o que é uma linguagem recursiva e o que é uma linguagem recursivamente enumerável. Dê um exemplo de linguagem recursivamente enumerável que não seja recursiva.

A.4 Exercícios do Capítulo 5

Exercício 17: Explique com suas palavras o que é um problema NP e o que é um problema NP-completo. Dê um exemplo de problema NP-completo.

Exercício 18:

- O que é um verificador polinomial?
- Dê um exemplo de problema que possui um verificador polinomial e indique qual seria esse verificador.
- O que podemos dizer sobre a complexidade computacional deste problema?

Exercício 19: Sabemos que o problema SAT é NP-completo, mostre que 3SAT é NP-completo.