Teoria da Computação **FCT-UNL, 2º semestre 2023-2024**

Notas 3: Autómatos Finitos Deterministas

Autor: João Ribeiro

Introdução

Nestas notas entramos, pela primeira vez, na teoria da computação. Introduzimos um modelo de computação que captura computadores simples com memória fixa, e estudamos a capacidade de tais computadores para resolverem problemas de decisão.

3.1 Representações de inputs e linguagens como problemas computacionais

Nesta cadeira vamos focar-nos no tipo mais simples de problema computacional – *problemas de decisão*. Num problema de decisão, o computador recebe um input e tem apenas de tomar uma decisão, "sim" ou "não". Primeiro, para estudarmos este problema cuidadosamente, temos de especificar com mais detalhe o formato do input. A representação de inputs como sequências finitas de caracteres de algum alfabeto é algo que nos surge naturalmente enquanto seres humanos e cientistas da computação. Mais precisamente, seja Σ um conjunto finito não-vazio, a que chamamos de *alfabeto*. Escolhas naturais para um alfabeto são, por exemplo, o nosso alfabeto latino $\{a, b, c, \ldots, z\}$, os algarismos árabes $\{0, 1, 2, \ldots, 9\}$, ou então o alfabeto binário $\{0, 1\}$. Podem convencer-se, usando a vossa própria experiência enquanto programadores, que o alfabeto binário tem a mesma "capacidade expressiva" que a língua portuguesa, ou que o conjunto de números árabes. Os inputs do nosso computador serão então elementos de Σ^*, o conjunto de todas as sequências finitas sobre Σ. Claro, alguns problemas podem ser mais fáceis de expressar usando certos alfabetos.

Chamamos *linguagem* (sobre Σ) a qualquer subconjunto $L \subseteq \Sigma^*$. Por palavras, a linguagem L é um conjunto de sequências finitas sobre o nosso alfabeto Σ. A razão pela qual definimos o conceito de linguagem L é que esta representa, implicitamente, o seguinte problema computacional: *Dada uma sequência $x \in \Sigma^*$, será que $x \in L$?* Chamamos a este problema o *problema de decisão de L*. Por exemplo, se $L = \{x \in \{0,1\}^* \mid x_1 = 1\}$ (ou seja, o conjunto de todas as sequências binárias que começam com 1), então o problema de decisão correspondente é "Dado $x \in \{0,1\}^*$, será que $x_1 = 1$?".

Estamos interessados em estudar questões do seguinte tipo:

> Para uma dada linguagem L, será que existe um "computador/algoritmo" que resolve o problema de decisão de L? (Isto é, que dado qualquer $x \in \Sigma^*$ como input decide correctamente se $x \in L$.)

Esta questão ainda não está completamente clara, pois não definimos rigorosamente o que é um "computador/algoritmo". Iremos considerar vários modelos de computação importantes que capturam computadores com diferentes capacidades. O nosso objectivo é estudar de forma rigorosa quais problemas podem ser resolvidos num dado modelo de computação. Este ramo da teoria da computação chama-se *teoria da computabilidade*. Mais concretamente, iremos estudar versões da questão acima do seguinte tipo:

- *(Agora)* Que problemas de decisão podem ser resolvidos por computadores com memória fixa?
- *(Mais tarde)* Existem problemas de decisão que estão além do alcance dos nossos poderosos computadores modernos, mesmo que tenhamos todo o tempo e memória do universo ao nosso dispor?

3.2 Autómatos Finitos Deterministas: Computadores com memória fixa

Vamos agora introduzir o nosso primeiro modelo matemático de computação, que pretende capturar computadores simples com memória limitada, chamados *Autómatos Finitos Deterministas* (AFDs). Por um lado, veremos que é possível fazer muitas coisas interessantes com estes computadores primitivos. De facto, interagimos com muitos destes computadores no nosso dia-a-dia! Por outro lado, vamos também estudar os limites fundamentais destes computadores, e desenvolver técnicas para demonstrar que certos problemas estão fora do seu alcance.

Antes de discutirmos AFDs de uma maneira mais abstracta, começamos por discutir um exemplo concreto de um sistema comum que pode ser representado como um AFD: uma porta com acesso restrito com cartão. Normalmente, a forma mais intuitiva de representar um AFD é através do seu *diagrama de estados*, como exemplificado abaixo.

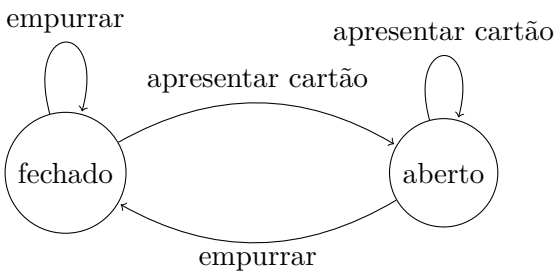

Figure 3.1: Diagrama de estados de uma porta com acesso por cartão.

A porta de acesso restrito tem dois estados, "fechado" e "aberto", representados por dois círculos. Há, também, um conjunto de acções que podemos efectuar: "empurrar" a porta e "apresentar cartão". As setas representam as transições de estado induzidas por estas acções. Se a porta estiver fechada e a tentarmos empurrar, nada acontece – esta continua fechada. Por outro lado, se apresentarmos o cartão a porta passa a estar aberta, e, após a empurrarmos para entrar, volta a fechar. Se apresentarmos cartão quando a porta está aberta, esta continua aberta.

Mais geralmente, mas ainda informalmente, um AFD consiste num único processador central com um número fixo de estados possíveis, que é inicializado num certo estado inicial. De seguida, o autómato recebe como input uma sequência (também chamada de *string*) finita de símbolos de um alfabeto Σ, que lê símbolo-a-símbolo. Cada vez que o autómato lê um símbolo da sequência, este actualiza o seu estado. Quando chega ao fim da sequência, o autómato devolve apenas um output binário ("aceito" ou "não aceito") com base no seu estado final. Podemos, então, ver tal autómato como uma máquina especializada que *decide* (ou *reconhece*) linguagens.

Segue o diagrama de estados de um AFD muito simples.

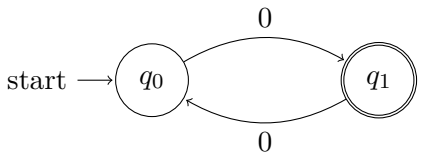

Figure 3.2: Diagrama de estados de um AFD.

Vamos interpretar este diagrama. Cada círculo representa um estado do AFD. Neste caso, há dois estados, q_0 e q_1. A seta "start $\rightarrow$" aponta para o *estado inicial* do AFD, ou seja, o estado em que o AFD se encontra antes de começar a ler o input. Neste caso, o estado inicial chama-se q_0. As setas entre círculos correspondem a *transições* de estado. Em particular, a seta que aponta de q_0 para q_1 especifica que quando o AFD se encontra no estado q_0 e lê um 0, deve actualizar o seu estado para q_1. Da mesma forma, quando se encontra no estado q_1 e lê um 0, deve actualizar o seu estado para q_0. O estado q_1 está rodeado por dois círculos. Isto representa que q_1 é um *estado de aceitação* (também chamado de *estado final*) do AFD. Se o processo de computação num dado input terminar num estado de aceitação, então o AFD "aceita" o input. Caso contrário, o AFD "não aceita" o input.

Vamos tentar perceber que strings w (sobre o alfabeto $\Sigma = \{0\}$) são aceites pelo input através de alguns exemplos. Quando $w = \varepsilon$ (a string vazia), então a sequência de estados do AFD é (q_0), isto é, somente o estado inicial. Como q_0 não é um estado de aceitação, concluímos que o AFD não aceita ε. Quando $w = 0$, a sequência de estados do AFD é (q_0, q_1). Como q_1 é um estado de aceitação, concluímos que o AFD aceita a string 0. Quando $w = 00$, a sequência de estados do AFD é (q_0, q_1, q_0). Como q_0 não é um estado de aceitação, concluímos que o AFD não aceita 00. Quando $w = 000$, a sequência de estados do AFD é (q_0, q_1, q_0, q_1). Como q_1 é um estado de aceitação, concluímos que o AFD aceita a string 000. Com base neste padrão, é fácil convencermo-nos de que o AFD aceita exactamente as strings de 0s de tamanho ímpar. Isto é, acabámos de construir um AFD (um computador, ou algoritmo, extremamente simples) que decide se uma dada string de 0s tem tamanho ímpar.

O leitor poderá questionar o interesse de estudarmos um modelo de computação tão simples. Na opinião do autor, existem várias excelentes razões para o fazermos. O estudo da capacidade dos AFDs leva ao desenvolvimento de uma teoria matemática rica e elegante, com ligações importantes a outras áreas da matemática e da teoria da computação. Dada a sua simplicidade, este modelo é especialmente apropriado para introduzir o tipo de questões que motivam a teoria da computação. Por último, AFDs também são ferramentas úteis em outras áreas da informática, como compiladores

e algoritmos em strings (por exemplo, pesquisa de padrões).

3.2.1 Definição formal de um AFD

Enquanto que a descrição de um AFD através do seu diagrama de estados é, normalmente, a mais amigável, também precisamos de uma definição mais formal de um AFD. Uma definição formal remove qualquer tipo de ambiguidades que possam existir na nossa discussão de AFDs e providencia notação standard para nos referirmos a vários componentes de um AFD. Mais ainda, existem casos onde a notação de diagrama de estados não é, de todo, apropriada (por exemplo, AFDs com muitos estados e/ou transições, ou construções de AFDs parameterizadas por variáveis).

Definição 3.1 (Autómato finito determinista) *Um AFD M é definido por um tuplo $(S, \Sigma, \delta, s, F)$, onde:*

- *S é o conjunto finito de estados de M;*
- *Σ é o alfabeto finito de M;*
- *$\delta : S \times \Sigma \to S$ é a função (talvez parcial) de transição de M;*
- *$s \in S$ é o estado inicial de M;*
- *$F \subseteq S$ é o conjunto de estados de aceitação (ou finais) de M.*

Vamos definir formalmente o AFD da Figura 3.2. Neste caso, o AFD $M = (S, \Sigma, \delta, s, F)$ é definido através de:

- o conjunto de estados $S = \{q_0, q_1\}$;
- o alfabeto $\Sigma = \{0\}$;
- o estado inicial $s = q_0$;
- o conjunto de estados de aceitação $F = \{q_1\}$;

e função de transição $\delta : S \times \Sigma \to S$ dada pela tabela

δ	0
q_1	q_2
q_2	q_1

Consideramos mais um exemplo de um AFD com diagrama de estados ilustrado na Figura 3.3.

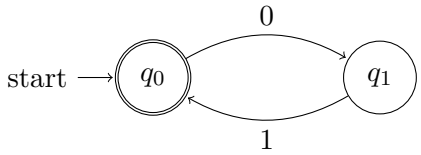

Figure 3.3: Diagrama de estados de outro AFD com função de transição parcial.

Podemos definir este AFD formalmente através do tuplo $(S, \Sigma, \delta, s, F)$ onde:

- $S = \{q_0, q_1\}$;
- $\Sigma = \{0, 1\}$;
- $s = q_0$;
- $F = \{q_0\}$.

Ao contrário do exemplo anterior, a função de transição deste AFD é parcial, pois não existe transição-1 a partir de q_0 nem transição-0 a partir de q_1. Isto quer dizer que os valores $\delta(q_0, 1)$ e $\delta(q_1, 0)$ não estão definidos, o que podemos escrever como $\delta(q_0, 1) = \perp$ e $\delta(q_1, 0) = \perp$. Geralmente, quando um AFD se encontra num estado q e lê um símbolo c para o qual não existe uma transição definida a partir de q, consideramos que o AFD termina e rejeita o input. Podemos definir a função parcial δ pela tabela

δ	0	1
q_0	q_1	$\perp$
q_1	$\perp$	q_0

Fica como exercício para o leitor convencer-se de que este AFD reconhece a linguagem $\{(01)^n \mid n \in \mathbb{N}\}$.

O uso de funções de transição parciais permite simplificar a representação de AFDs, mas é também verdade que para qualquer AFD M com função de transição parcial existe outro AFD M' com função de transição total que reconhece a mesma linguagem. Basta adicionar um novo estado (não final) a M do qual não podemos sair, para onde dirigimos todas as transições anteriormente não definidas. Capturamos esta observação no seguinte teorema.

Teorema 3.1 *Dado um AFD M qualquer, existe um AFD M' com função de transição total (também chamado de AFD* completo*) que reconhece a mesma linguagem que M.*

A Figura 3.4 exemplifica esta transformação para o AFD da Figura 3.3.

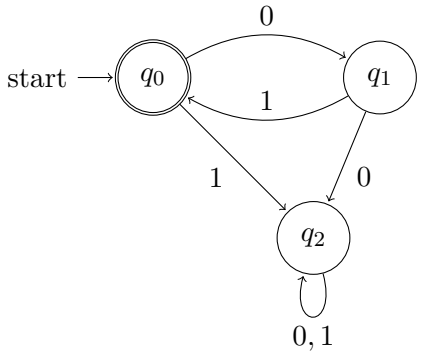

Figure 3.4: Diagrama de estados de um AFD com função de transição total que reconhece a mesma linguagem que o AFD da Figura 3.3.

Consideramos agora um exemplo de um AFD com múltiplos estados finais na Figura 3.5. Este AFD reconhece a linguagem das strings binárias que não contêm dois 0s consecutivos.

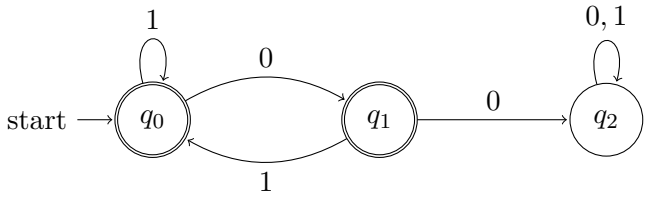

Figure 3.5: Diagrama de estados de um AFD com mais do que um estado final.

Definição de computação. Além de definirmos formalmente um AFD M, também precisamos de definir formalmente o processo de computação de M num input w. Intuitivamente, para um AFD $M = (S, \Sigma, \delta, s, F)$, a sequência de estados gerada por $w = w_1 w_2 \dots w_n$ é

$$(r_0 = s, r_1 = \delta(r_0, w_1), r_2 = \delta(r_1, w_2), \dots, r_n = \delta(r_{n-1}, w_n)).$$

Temos a seguinte definição um pouco mais geral.

Definição 3.2 (Sequência de estados gerada por input e aceitação) *Seja $M = (S, \Sigma, \delta, s, F)$ um AFD, $w = w_1 w_2 \dots w_n \in \Sigma^*$ um input, e $q \in S$ um estado de M. Dizemos que $(r_0, r_1, \dots, r_n)$ é a sequência de estados de M gerada por w a partir do estado q se as seguintes condições se verificam:*

1. *$r_0 = q$;*
2. *$\delta(r_i, w_{i+1}) = r_{i+1}$ para todo o $i \in \{0, 1, \dots, n-1\}$;*

Se for este o caso, escrevemos $\delta(q, w) = r_n$, onde r_n é o estado em que a computação termina. Escrevemos também $\delta^*(q, w) = (r_0, r_1, \dots, r_n)$. Quando q é o estado inicial de M (i.e., $q = s$), abreviamos $\delta(w) = \delta(s, w)$ e $\delta^*(w) = \delta^*(s, w)$. Dizemos que w é aceite *por M se $\delta(w) \in F$.*

Linguagem aceite por um AFD. Um AFD $M = (S, \Sigma, \delta, s, F)$ aceita algumas strings em Σ^* e rejeita as restantes. Como no exemplo acima, podemos falar da linguagem das strings aceites por M, que denotamos por $L(M)$. Mais formalmente, a linguagem $L(M) \subseteq \Sigma^*$ é definida por

$$L(M) = \{w \in \Sigma^* \mid \delta(w) \in F\},$$

onde (relembrando a Definição 3.2) $\delta(w) \in F$ significa que a computação de M no input w a partir do seu estado inicial termina num estado de aceitação de M. Dizemos também que M *aceita* ou *reconhece* a linguagem $L(M)$. Cada AFD aceita apenas uma linguagem. Por outro lado, uma linguagem pode ser aceite por mais do que um AFD (pensem em exemplos!).

No caso do AFD M da Figura 3.1, este aceita a linguagem $L(M) \subseteq \{0\}^*$ das strings de 0s de tamanho ímpar, que também podemos escrever como $L(M) = \{0^n \mid n \text{ é ímpar}\}$.

3.3 Como criar AFDs

O desenho de AFDs que reconheçam uma dada linguagem requer criatividade e experiência. Não é algo que possa ser feito de forma mecânica. É importante ler, de forma crítica, AFDs criados por outras pessoas, e praticar a resolução de muitos exercícios. Recomendamos a leitura da estratégia de Sipser [Sip13, Section 1.1], que é muito útil. Resumidamente, quando o leitor precisar de criar um novo AFD, deve colocar-se na pele do autómato e simular os seus passos.

3.4 Para explorar

Estas notas foram baseadas em [Sip13, Section 1.1] e [LP97, Section 2.1]. Leiam estas secções para conhecerem outras perspectivas sobre este tópico.

Esta aula gravada sobre codificações de inputs e problemas computacionais também poderá ser interessante.

A teoria dos autómatos teve início nos meados do século XX. Se têm um interesse histórico pelo desenvolvimento da informática, aconselhamos a leitura (por alto) de alguns trabalhos que despoletaram este estudo, tais como os artigos de McCulloch e Pitts [MP43], Mealy [Mea55], Moore [Moo56], Kleene [Kle56], e Rabin e Scott [RS59].

References

[Kle56] Stephen C. Kleene. Representation of events in nerve nets and finite automata. *Automata Studies*, (34):3, 1956. Available at https://www.rand.org/content/dam/rand/pubs/research_memoranda/2008/RM704.pdf.

[LP97] Harry R. Lewis and Christos H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall PTR, USA, 2nd edition, 1997.

[Mea55] George H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[Moo56] Edward F. Moore. Gedanken-experiments on sequential machines. *Automata Studies*, 34:129–153, 1956.

[MP43] Warren S. McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5:115–133, 1943. Available at https://www.cs.cmu.edu/~./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf.

[RS59] Michael O. Rabin and Dana Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[Sip13] Michael Sipser. *Introduction to the Theory of Computation*. CEngage Learning, 3rd edition, 2013.