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

Notas 5: Não-determinismo

Autor: João Ribeiro

Introdução

Nestas notas introduzimos "não-determinismo", um dos conceitos mais importantes na teoria da computação. Exploramos o poder deste conceito no contexto dos autómatos finitos e discutimos um pouco as suas manifestações noutras regiões da teoria da computação.

5.1 Autómatos Finitos Não-Deterministas

Nas notas anteriores explorámos o conceito de AFD, que em cada estado pode apenas escolher uma direcção baseado no símbolo que lê nesse momento. Vamos agora explorar o caso mais geral de autómatos *não-deterministas*. Em suma, estes autómatos podem escolher entre várias transições para o mesmo símbolo a partir de um estado. Enquanto que num AFD o próximo estado é completamente determinado pelo estado actual e o símbolo lido, este não é necessariamente o caso para um autómato finito não-determinista (AFN). Começamos com um exemplo de um AFN simples:

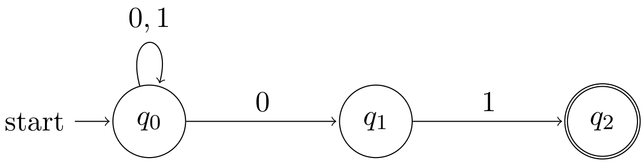

Figure 5.1: Diagrama de estados de um AFN.

Na Figura 5.1, o estado q_0 tem duas transições-0 – uma para q_0 e outra para q_1. Isto não seria permitido no modelo dos AFDs. Intuitivamente, um AFN comporta-se de forma semelhante a um AFD, excepto que quando se encontra num estado q e lê um símbolo s para o qual existe mais do que uma transição definidas, cria vários "universos paralelos" em que segue cada uma das transições possíveis. Na Figura 5.1, isto acontece quando o AFN se encontra no estado q_0 e lê um 0. O AFN aceita um dado input se este for aceite em pelo menos um destes universos paralelos. No caso da Figura 5.1, o AFN reconhece a linguagem das strings binárias que acabam em 01, pois as únicas sequências de estados válidas que terminam num estado de aceitação (o estado q_2) são da forma $(q_0, q_0, \ldots, q_0, q_1, q_2)$. Para transitarmos de q_0 para q_1 precisamos de ler um 0, e para transitarmos de q_1 para q_2 precisamos de ler um 1, após o qual não podemos ler mais símbolos. Por vezes também é útil interpretar a computação deste AFN da seguinte forma: O AFN consegue adivinhar quando faltam apenas dois símbolos para terminar o input, e nesse caso testa se estes são iguais a 01.

Outra propriedade importante dos AFNs que difere dos AFDs é que os AFNs permitem "transições-ε", isto é, transições que não consomem símbolos do input, tal como no exemplo abaixo.

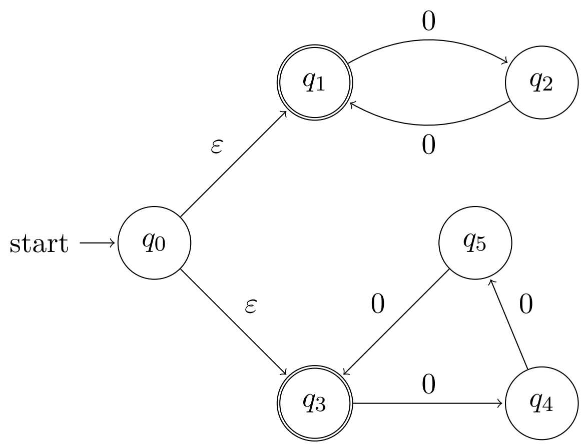

Figure 5.2: Diagrama de estados de um AFN com transições-ε.

O AFN da Figura 5.2 aceita as strings de 0s de tamanho divisível por 2 ou 3.[1] Quando a computação começa no estado q_0, o AFN divide-se em duas cópias. A primeira cópia segue a transição-ε para q_1 e a segunda cópia segue a transição-ε para q_2. Nenhuma destas transições consome símbolos do input. De seguida cada uma das cópias lê o input. Se o tamanho do input for divisível por 2, a primeira cópia aceita. Se o tamanho do input for divisível por 3, a segunda cópia aceita. Podemos também pensar de outra forma: o AFN primeiro adivinha se o input vai ter tamanho divisível por 2 ou por 3, e escolhe a direcção certa no estado inicial.

Nota 1 O conceito de AFN não é um modelo de computação realista. No entanto, veremos em breve que é um conceito extremamente versátil para raciocinar sobre linguagens regulares e outros objectos. Esta utilidade do não-determinismo estende-se a muitas outras regiões da teoria da computação.

5.1.1 Definição formal de AFN

A definição formal de um AFN é semelhante à de um AFD, com a excepção da função de transição δ. Enquanto que para um AFD a função δ mapeia um estado e um símbolo para um novo estado, no caso dos AFNs a função δ mapeia um estado e um símbolo para um *conjunto de estados*, representando as várias transições pelo mesmo símbolo, e também está definida para o novo "símbolo" ε.

Definição 5.1 (Autómato finito não-determinista) *Um AFN é um tuplo* $M = (S, \Sigma, \delta, s, F)$ *onde:*

- *S é o conjunto de estados;*

[1] É instrutivo tentar descrever um AFD que reconhece esta linguagem.

- *Σ é o alfabeto;*
- *$\delta : S \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(S)$ é a função de transição, onde relembramos que $\mathcal{P}(S)$ é o conjunto de todos os subconjuntos de S;*
- *$s \in S$ é o estado inicial;*
- *$F \subseteq S$ é o conjunto de estados finais ou de aceitação.*

Todo o AFD é também um AFN. No entanto, a implicação contrária não é verdade – existem AFNs que não são AFDs.

Definição 5.2 (Sequência de estados gerada por input e aceitação) *Seja $M = (S, \Sigma, \delta, s, F)$ um AFN e $w \in \Sigma^*$. Dizemos que $(r_0, r_1, \ldots, r_n)$ é uma sequência de estados de M gerada por w se podemos escrever $w = y_1 y_2 \ldots y_m$ para algum $m \in \mathbb{N}$ com $y_i \in \Sigma \cup \{\varepsilon\}$ para cada $i \in \{1, \ldots, m\}$ e as seguintes condições se verificam:*

1. *$r_0 = s$;*
2. *$r_{i+1} \in \delta(r_i, y_{i+1})$ para todo o $i \in \{0, 1, \ldots, m-1\}$.*

Dizemos que w é aceite *por M se existir uma sequência de estados $(r_0, r_1, \ldots, r_m)$ de M gerada por w tal que $r_m \in F$. Denotamos a linguagem das strings w aceites por M por $L(M)$.*

Ao contrário do que acontece com AFDs, podem existir várias sequências de estados de um AFN geradas pelo mesmo input w.

Vamos definir formalmente os AFNs das Figuras 5.1 and 5.2. O AFN da Figura 5.1 corresponde ao tuplo $M = (S, \Sigma, \delta, s, F)$ com $S = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1\}$, $s = q_0$, $F = \{q_2\}$, e $\delta : S \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(S)$ dada pela tabela

δ	0	1
q_0	$\{q_0, q_1\}$	q_0
q_1	$\perp$	q_2
q_2	$\perp$	$\perp$

Consideremos o input $w = 0001$. Existem várias sequências de estados de M geradas por w, incluindo uma tal sequência que termina num estado final de M, o que mostra que M aceita w. Exemplos destas sequências incluem $(q_0, q_0, q_0, q_1, q_2)$, que termina num estado final, e $(q_0, q_0, q_1, \perp)$, que aborta e rejeita pois não existe nenhuma transição-0 a partir de q_1.

O AFN da Figura 5.2 corresponde ao tuplo $N = (S, \Sigma, \delta, s, F)$ com $S = \{q_0, q_1, \ldots, q_5\}$, $\Sigma = \{0\}$, $s = q_0$, $F = \{q_1, q_3\}$, e δ definida pela tabela

δ	ε	0
q_0	$\{q_1, q_3\}$	$\perp$
q_1	$\perp$	q_2
q_2	$\perp$	q_1
q_3	$\perp$	q_4
q_4	$\perp$	q_5
q_5	$\perp$	q_3

No input $w = 000000$, existem várias sequências de estados de N geradas por w que terminam num estado final de N. Por exemplo, tanto $(q_0, q_1, q_2, q_1, q_2, q_1, q_2, q_1)$ e $(q_0, q_3, q_4, q_5, q_3, q_4, q_5, q_3)$ são sequências de estados de N geradas por w que terminam num estado final, resultantes de escrevermos $w = \varepsilon 000000$. Concluímos, em particular, que N aceita w. A primeira sequência é gerada escolhendo a transição-ε de q_0 para q_1, enquanto que a segunda sequência é gerada escolhendo a transição-ε de q_0 para q_3.

5.2 Equivalência de AFNs e AFDs

Um AFN parece, à primeira vista, ser uma máquina muito mais poderosa do que um AFD. Surpreendentemente, no contexto do reconhecimento de linguagens, AFNs e AFDs são equivalentes! Por outras palavras, para qualquer linguagem reconhecida por um AFN existe também um AFD que a reconhece. Em particular, isto quer dizer que para mostrar que uma linguagem L é regular basta descrever um AFN M tal que $L = L(M)$, uma tarefa por vezes muito mais fácil do que construir um AFD.

Teorema 5.1 *Seja M um AFN. Então, existe um AFD M' tal que $L(M') = L(M)$. Segue que uma linguagem é regular se e só se existe um AFN M tal que $L = L(M)$.*

A demonstração do Teorema 5.1 baseia-se numa construção elegante de Rabin-Scott [RS59] (também chamada de construção *powerset*) que transforma um dado AFN qualquer num AFD equivalente, que discutimos de seguida. É de realçar que Rabin e Scott foram galardoados com o conceituado prémio Turing (a mais alta distinção na ciência da computação) em 1976 por este artigo.

5.2.1 Determinização de AFNs: A construção de Rabin-Scott

Seja $L \subseteq \Sigma^*$ uma linguagem e M um AFN que reconhece L. A construção de Rabin-Scott transforma o AFN M num AFD M' que reconhece L. A ideia-chave é que os estados de M' serão *subconjuntos* de estados de M. Esta ideia permite-nos guardar os estados actuais das várias "cópias" do AFN geradas durante a sua computação apenas com um só estado. Existirá uma transição-a de um subconjunto R para outro subconjunto R' se os estados em R' forem exactamente os estados acessíveis começando em algum estado de R através de uma transição-a seguida de transições-ε.

Antes de definirmos o AFD M' formalmente, precisamos de definir algumas operações sobre conjuntos de estados do AFN $M = (S, \Sigma, \delta, s, F)$. Dado um subconjunto de estados $R \subseteq S$ e um símbolo

$a \in \Sigma$, definimos o conjunto de estados acessíveis de R por a como

$$\mathsf{reach}_a(R) = \{q \in S \mid \text{existe } r \in R \text{ tal que } q \in \delta(r, a)\}.$$

Por palavras, $\mathsf{reach}_a(R)$ é o conjunto de estados de M a que conseguimos chegar ao seguir uma transição-a a partir de um estado que pertence a R. Num AFN também existem transições-ε. Portanto, também será importante saber, para um dado conjunto de estados R, que estados são acessíveis a partir deste somente por (potencialmente) múltiplas transições-ε, que podemos tomar de graça e a que chamamos o fecho-ε de R. Mais precisamente, definimos o fecho-ε de $R \subseteq S$ como

$$\mathsf{close}_\varepsilon(R) = \{q \in S \mid \text{existe } r \in R \text{ tal que } q \text{ é acessível de } r \text{ por zero ou mais transições-}\varepsilon\}.$$

Em particular, temos sempre $R \subseteq \mathsf{close}_\varepsilon(R)$, pois os estados de R são sempre acessíveis por zero transições-ε a partir de R.

Estamos prontos para definir formalmente o AFD $M' = (S', \Sigma', \delta', s', F')$:

- Tomamos o conjunto de estados $S' = \mathcal{P}(S)$, que corresponde a todos os subconjuntos de estados de M.
- O alfabeto mantém, i.e., $\Sigma' = \Sigma$.
- O estado inicial de M' corresponde agora a todos os estados acessíveis a partir do estado inicial s de M através de transições-ε, isto é, $s' = \mathsf{close}_\varepsilon(\{s\})$.
- Um estado de M' (que corresponde a um subconjunto de estados de M) é final se este contiver pelo menos um estado final de M, isto é,

  $$F' = \{R \in \mathcal{P}(S) \mid R \cap F \neq \varnothing\}.$$

  Intuitivamente, esta definição faz sentido pois um estado de M' captura todos os estados actuais das cópias do AFN M, e sabemos que M aceita um input w quando pelo menos uma das cópias aceita w (a computação dessa cópia termina num estado final de M).
- Resta definir a função de transição $\delta' : S' \times \Sigma' \to S'$. Intuitivamente, queremos que $\delta'(R, a)$ consista em todos os estados acessíveis de um estado de R através de uma transição-a seguida de possíveis transições-ε, que são de graça. Podemos então definir $\delta'(R, a)$ à custa das operações reach e close como

  $$\delta'(R, a) = \mathsf{close}_\varepsilon(\mathsf{reach}_a(R)).$$

  Quando $\mathsf{close}_\varepsilon(\mathsf{reach}_a(R)) = \varnothing$ (o que acontece quando $\mathsf{reach}_a(R) = \varnothing$), interpretamos esta situação tal como $\delta'(R, a) = \perp$, isto é, δ' fica indefinida em (R, a).

O leitor poderá convencer-se de que o AFD M' que descrevemos é equivalente ao AFN M. Uma demonstração bastante cuidada encontra-se em [LP97, Section 2.2, Theorem 2.2.1]. Também sugerimos a leitura de [Sip13, Section 1.2, Theorem 1.39], que é menos formal mas mais intuitivo.

5.2.2 Alguns exemplos de determinização de AFNs

Usamos agora a construção de Rabin-Scott para determinizar alguns AFNs.

Começamos pelo AFN da Figura 5.1. O alfabeto do novo AFD M' é $\Sigma' = \{0,1\}$. O seu estado inicial R_0 é dado por

$$R_0 = \mathsf{close}_\varepsilon(\{q_0\}) = \{q_0\},$$

pois não existem transições-ε a partir de q_0. Determinamos agora o conjunto de estados acessíveis por transições-0 e transições-1 a partir de R_0:

- $\delta'(R_0, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(R_0)) = \mathsf{close}_\varepsilon(\{q_0, q_1\}) = \{q_0, q_1\} = R_1$.
- $\delta'(R_0, 1) = \mathsf{close}_\varepsilon(\mathsf{reach}_1(R_0)) = \mathsf{close}_\varepsilon(\{q_0\}) = \{q_0\} = R_0$.

Temos agora um novo conjunto de estados R_1 para analisar:

- $\delta'(R_1, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_0, q_1\})) = \mathsf{close}_\varepsilon(\{q_0, q_1\}) = \{q_0, q_1\} = R_1$.
- $\delta'(R_1, 1) = \mathsf{close}_\varepsilon(\mathsf{reach}_1(\{q_0, q_1\})) = \mathsf{close}_\varepsilon(\{q_0, q_2\}) = \{q_0, q_2\} = R_2$.

Passamos a analisar o novo conjunto de estados R_2:

- $\delta'(R_2, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_0, q_2\})) = \mathsf{close}_\varepsilon(\{q_0, q_1\}) = \{q_0, q_1\} = R_1$.
- $\delta'(R_2, 1) = \mathsf{close}_\varepsilon(\mathsf{reach}_1(\{q_0, q_2\})) = \mathsf{close}_\varepsilon(\{q_0\}) = \{q_0\} = R_0$.

Como todos os conjuntos a que acedemos já foram analisados, temos toda a informação necessária para descrever o AFD equivalente M'. Para escrevermos o seu diagrama de estados, resta determinar os estados finais. Como o estado final de M é q_2, os estados finais de M' correspondem a subconjuntos de estados de M que contêm q_2. O único estado final é, portanto, $R_2 = \{q_0, q_2\}$. O AFD equivalente resultante da construção de Rabin-Scott tem o diagrama de estados abaixo (podemos ignorar os estados criados por Rabin-Scott que não são acessíveis a partir do estado inicial).

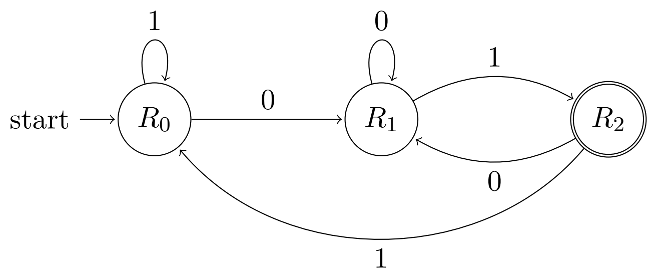

Figure 5.3: Diagrama de estados de AFD obtido por aplicação da construção de Rabin-Scott ao AFN da Figura 5.1.

Este AFD pode ser descrito formalmente pelo tuplo $M' = (S', \Sigma', \delta', s', F')$ com $S' = \{R_0, R_1, R_2\}$, $\Sigma' = \{0, 1\}$, $s' = R_0$, $F' = \{R_2\}$, e δ dada pela tabela

δ	0	1
R_0	R_1	R_0
R_1	R_1	R_2
R_2	R_1	R_0

Determinizamos agora o AFN da Figura 5.2. O alfabeto do novo AFD é $\Sigma' = \{0\}$. O seu estado inicial R_0 corresponde a

$$R_0 = \mathsf{close}_\varepsilon(\{q_0\}) = \{q_0, q_1, q_3\}.$$

Determinamos agora as várias transições. Continuamos a análise até pararmos de gerar novos conjuntos de estados:

- $\delta'(R_0, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_0, q_1, q_3\})) = \mathsf{close}_\varepsilon(\{q_2, q_4\}) = \{q_2, q_4\} = R_1$.
- $\delta'(R_1, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_2, q_4\})) = \mathsf{close}_\varepsilon(\{q_1, q_5\}) = \{q_1, q_5\} = R_2$.
- $\delta'(R_2, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_1, q_5\})) = \mathsf{close}_\varepsilon(\{q_2, q_3\}) = \{q_2, q_3\} = R_3$.
- $\delta'(R_3, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_2, q_3\})) = \mathsf{close}_\varepsilon(\{q_1, q_4\}) = \{q_1, q_4\} = R_4$.
- $\delta'(R_4, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_1, q_4\})) = \mathsf{close}_\varepsilon(\{q_2, q_5\}) = \{q_2, q_5\} = R_5$.
- $\delta'(R_5, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_2, q_5\})) = \mathsf{close}_\varepsilon(\{q_1, q_3\}) = \{q_1, q_3\} = R_6$.
- $\delta'(R_6, 0) = \mathsf{close}_\varepsilon(\mathsf{reach}_0(\{q_1, q_3\})) = \mathsf{close}_\varepsilon(\{q_2, q_4\}) = \{q_2, q_4\} = R_1$.

Os novos estados finais correspondem a conjuntos de estados do AFN que contêm estados finais – são finais, portanto, R_0, R_2, R_3, R_4, e R_6. O diagrama de estados do AFD resultante segue abaixo.

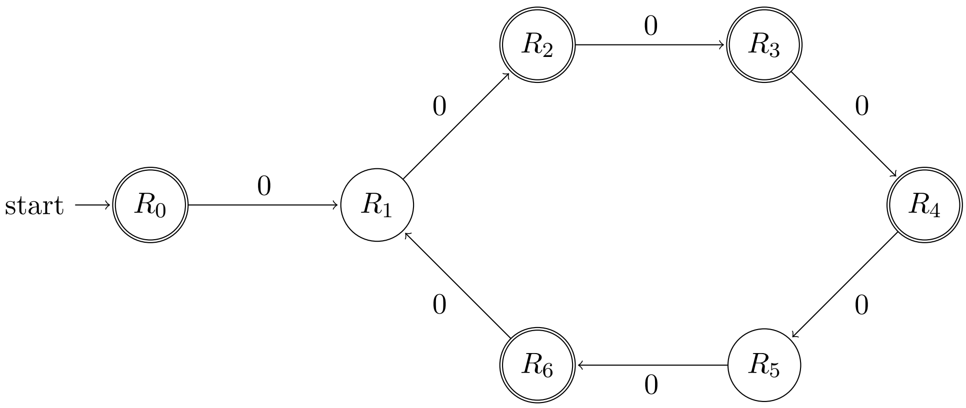

Figure 5.4: Diagrama de estados de AFD obtido por aplicação da construção de Rabin-Scott ao AFN da Figura 5.2.

5.2.3 Número de estados de AFNs e AFDs equivalentes

A construção de Rabin-Scott aplicada a um AFN com k estados resulta num AFD equivalente com 2^k estados.[2] Posto de outra forma, o AFD equivalente que obtemos requer exponencialmente mais memória para reconhecer a mesma linguagem do que o AFN original.

Enquanto teóricos da computação, é natural questionarmos se este aumento exponencial de memória é mesmo necessário. Talvez seja sempre possível construir um AFD equivalente com $k' \approx k$ estados! Infelizmente, este não é o caso. Sabemos já há muito tempo [Lup63] (o artigo relacionado de Moore [Moo71] poderá ser mais fácil de aceder) que para cada $k \in \mathbb{N}$ existe um AFN com k estados tal que qualquer AFD equivalente tem 2^k estados. Isto mostra que a construção de Rabin-Scott é óptima no pior-caso!

Um exemplo simples que fica muito próximo deste resultado é, para qualquer $k \in \mathbb{N}$, a linguagem L_k sobre $\{0,1\}$ dada por

$$L_k = \{w \in \{0,1\}^* \mid |w| \geq k \wedge w_{|w|-k+1} = 1\},$$

ou seja, a linguagem das strings binárias de tamanho pelo menos k cujo k-ésimo símbolo a contar do fim é 1. Existe um AFN com $k+1$ estados que reconhece L_k (um bom exercício para o leitor), enquanto que qualquer AFD equivalente requer pelo menos 2^k estados (intuitivamente, um para cada sufixo de tamanho k do input).

5.3 Propriedades de linguages regulares através de AFNs

Em vários casos, é muito mais fácil descrever um AFN que reconhece uma dada linguagem do que um AFD. Mais ainda, quando queremos apenas mostrar que a linguagem é regular, vimos acima (construção de Rabin-Scott) que basta desenhar um AFN para essa linguagem. Nesta secção, veremos a grande utilidade deste resultado ao estabelecermos facilmente várias propriedades de linguagens regulares com que tivemos problemas nas notas anteriores através de AFNs.

Começamos por revisitar a união de duas linguagens regulares com uma demonstração um pouco mais intuitiva.

Teorema 5.2 *Sejam A e B linguagens regulares. Então, $A \cup B$ também é regular.*

Demonstração: Sejam M_A e M_B AFDs que reconhecem A e B, respectivamente. Pelo Teorema 5.1, para mostrarmos que $A \cup B$ é regular basta construir um AFN M' que reconhece $A \cup B$.

Tal AFN M' é fácil de construir. Providenciamos uma descrição informal mas clara: O estado inicial de M' é ligado por transições-ε aos estados iniciais de M_A e M_B, e os estados finais de M' são os estados finais de M_A e M_B. Esta construção é ilustrada na Figura 5.5.

Intuitivamente, dado um input w, o AFN M' primeiro adivinha se deve testar "$w \in A$" ou "$w \in B$" seguindo a transição-ε apropriada, e depois corre o AFD relevante em w. ■

[2] Recordamos que $|\mathcal{P}(S)| = 2^{|S|}$.

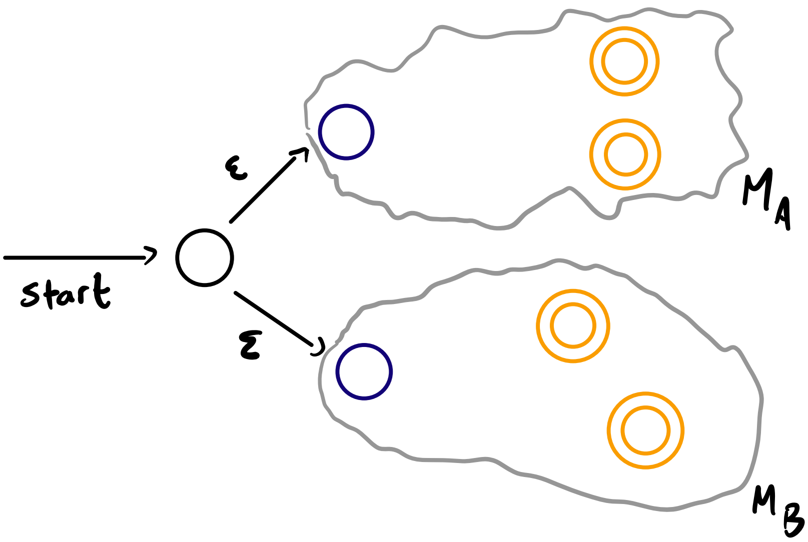

Figure 5.5: Ilustração do AFN que reconhece a união $A \cup B$ de linguagens regulares A e B. Os AFDs M_A e M_B reconhecem A e B, respectivamente. Os círculos azuis representam os estados iniciais originais de cada AFD, e os círculos laranjas os estados finais originais de cada AFD.

Passamos agora para a concatenação de linguagens regulares, algo que não tínhamos conseguido analisar anteriormente.

Teorema 5.3 *Sejam A e B linguagens regulares. Então, $A \circ B$ também é regular.*

Demonstração: Sejam M_A e M_B AFDs que reconhecem A e B, respectivamente. Pelo Teorema 5.1, para mostrarmos que $A \circ B$ é regular basta construir um AFN M' que reconhece $A \circ B$.

Providenciamos uma descrição informal mas clara de M': O estado inicial de M' corresponde ao estado inicial de M_A. A cada estado final de M_A acrescentamos uma transição-ε para o estado inicial de M_B. Os estados finais de M' são os estados finais de M_B. Esta construção é ilustrada na Figura 5.6.

Intuitivamente, dado um input w, o AFN M' adivinha o tamanho de x e y tal que $w = x \circ y$ com $x \in A$ e $y \in B$ (caso existam), corre M_A em x e depois segue a transição-ε para saltar para o estado inicial de M_B e correr y em M_B. ■

Passamos agora para o fecho de Kleene de uma linguagem regular, algo que também não tínhamos conseguido analisar.

Teorema 5.4 *Seja A uma linguagem regular. Então, A^* também é regular.*

Demonstração: Seja M_A um AFD que reconhece A. Pelo Teorema 5.1, para mostrarmos que A^* é regular basta construir um AFN M' que reconhece A^*.

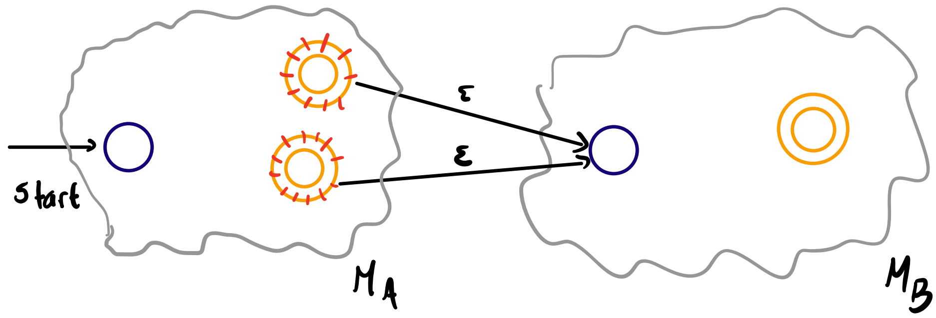

Figure 5.6: Ilustração do AFN que reconhece a concatenação $A \circ B$ de linguagens regulares A e B. Os AFDs M_A e M_B reconhecem A e B, respectivamente. Os círculos azuis representam os estados iniciais originais de cada AFD, e os círculos laranjas os estados finais originais de cada AFD. As riscas vermelhas mostram que estados deixam de ser finais no AFN.

Providenciamos uma descrição informal mas clara de M': Ligamos o novo estado inicial de M' por transição-ε ao estado inicial de M_A. Cada estado final de M_A é ligado ao novo estado inicial de M' por uma transição-ε. O único estado final de M' é o seu estado inicial. Esta construção é ilustrada na Figura 5.7.

Intuitivamente, dado um input w, o AFN M' adivinha uma partição de w como $w = x^{(1)} \circ x^{(2)} \circ \cdots \circ x^{(m)}$ com cada $x^{(i)} \in A$, para algum $m \in \mathbb{N}$ (caso exista, e notamos que podemos ter $m = 0$ quando $w = \varepsilon$). Depois, M' corre M_A $x^{(1)}$, após o qual segue as transições-ε de volta para o estado inicial de M_A para correr M_A em $x^{(2)}$, e por aí em diante. ■

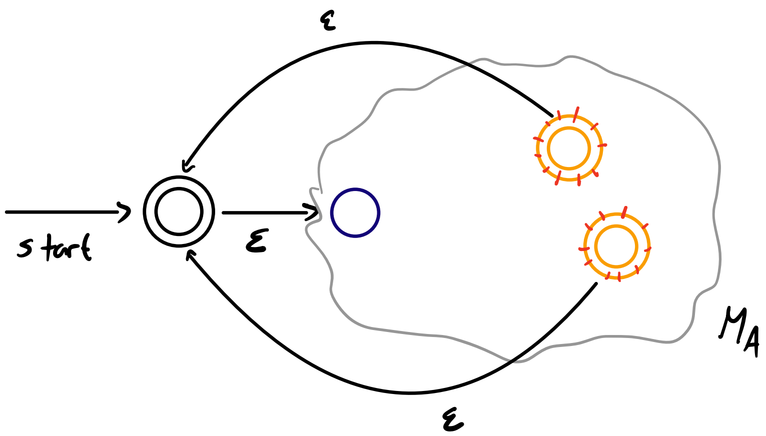

Figure 5.7: Ilustração do AFN que reconhece o fecho de Kleene A^* de uma linguagem regular A. O AFD M_A reconhece A. O círculo azul representa o estado inicial de M_A, e os círculos laranjas os estados finais de M_A. As riscas vermelhas mostram que estados deixam de ser finais no AFN.

5.4 O não-determinismo na teoria da computação

A noção de não-determinismo é extremamente importante não só no contexto da teoria de autómatos, mas também em muitos outros ramos da teoria da computação. Como mencionámos acima, Rabin e Scott receberam o prémio Turing pela introdução do não-determinismo [RS59]. Em particular, o não-determinismo aparece também de forma prominente no que é normalmente tido como o maior problema em aberto na ciência da computação: O problema P vs. NP, cuja solução acarreta um prémio de um milhão de dólares pelo Clay Mathematics Institute[3].

De forma muito informal, o problema P vs. NP prende-se com a relação entre máquinas de Turing (um importante modelo de computação que estudaremos mais tarde nesta cadeira) determinísticas e não-determinísticas. A classe P (de "polynomial time") corresponde às linguagens que são reconhecidas de forma "eficiente" por máquinas de Turing determinísticas, enquanto que a classe NP (de "non-deterministic polynomial time") corresponde às linguagens que são reconhecidas de forma "eficiente" por máquinas de Turing não-determinísticas. Estas noções correspondem à "complexidade computacional" de reconhecer uma linguagem, algo com que terão mais contacto noutras cadeiras. Na presente cadeira focamo-nos na noção de "computabilidade" – queremos saber se uma linguagem é reconhecível num dado modelo de computação, sem nos preocuparmos com a eficiência da máquina que a reconhece.

O problema P vs. NP pergunta se $\mathrm{P} = \mathrm{NP}$; a grande parte dos teóricos da computação acredita que $\mathrm{P} \neq \mathrm{NP}$. Esta conjectura é, por exemplo, um ingrediente imprescindível para muitos avanços fundamentais em criptografia. Se o leitor tiver curiosidade, recomendamos a leitura de [LP97, Chapter 6], [Sip13, Chapter 7], e do excelente livro de Arora e Barak [AB09].

5.5 Para explorar

O artigo original de Rabin e Scott [RS59] (disponível, por exemplo, aqui) é uma janela interessante para os primórdios da teoria da computação. Neste curto vídeo, Rabin fala um pouco sobre esse trabalho. Este artigo da Wikipedia colecciona muitas referências sobre o número de estados de vários tipos de autómatos equivalentes.

Recomendamos também a exploração dos Millenium Prize Problems do Clay Mathematics Institute, bem como dos recipientes dos prémios Turing e Gödel.

References

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach*. Cambridge University Press, 2009. Draft available at `https://theory.cs.princeton.edu/complexity/`.

[3] Existem, certamente, formas mais fáceis de ganhar um milhão de dólares – mas não tão divertidas!

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

[Lup63] Oleg B. Lupanov. A comparison of two types of finite sources. *Problemy Kibernetiki*, 9:321–326, 1963.

[Moo71] Frank R. Moore. On the bounds for state-set size in the proofs of equivalence between deterministic, nondeterministic, and two-way finite automata. *IEEE Transactions on Computers*, C-20(10):1211–1214, 1971.

[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.