Capítulo 12: Autômatos

O nosso modelo de referência para computação mecânica é a máquina de Turing. Uma máquina Turing tem apenas dois componentes, uma CPU e uma fita. Vamos agora tirar a fita e estudar a CPU sozinha.

Dito de outra maneira, enquanto uma máquina de Turing tem memória ilimitada, os dispositivos que usamos todos os dias não têm. Podemos perguntar que tarefas podem ser feitas por uma máquina com memória limitada.

12.1. Máquinas de estados finitos

Produzimos um novo modelo de computação, modificando a definição da Máquina de Turing. Eliminaremos a capacidade de escrever, alterando a cabeça da fita de leitura/escrita para somente leitura. Isto dar-nos-á uma visão do que pode ser feito apenas com estados. Descobriremos que este tipo de máquina pode fazer muitas coisas, mas não tantas como uma máquina Turing.

No nosso novo modelo, utilizaremos o mesmo tipo de tabelas de transição e grafos de transição que fizemos com as máquinas de Turing.

Exemplo 12.1.1

Um interruptor de energia tem dois estados, q_{off} e q_{on} e o seu alfabeto de entrada tem um símbolo, `toggle`. (Figura 12.1)

Figura 12.1: Um interruptor de energia.

Exemplo 12.1.2

Opere esta catraca, colocando duas fichas (token) e depois passando (push) por ela. Ela possui três estados e o seu alfabeto de entrada é Σ = { `token`, `push` }.

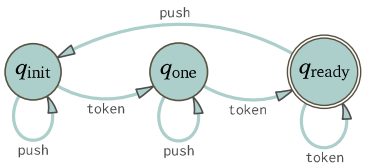


Figura 12.2: Uma catraca que deve receber duas fichas antes de liberar a passagem.

Como vimos com as máquinas de Turing, os estados são uma forma limitada de memória. Por exemplo, q_{one} é, no exemplo acima, como a catraca "se lembra" de ter recebido até agora uma ficha.

Exemplo 12.1.3

Esta máquina de venda automática dispensa artigos que custam 30 centavos. Ela aceita apenas moedas de 5, 10 e 25 centavos. A figura é complexa, por isso vamos mostrá-la em três camadas. A primeira é das setas para as moedas de 5 centavos (que chamamos de *n*, de *nickel* em inglês) e o apertar do botão de distribuição.

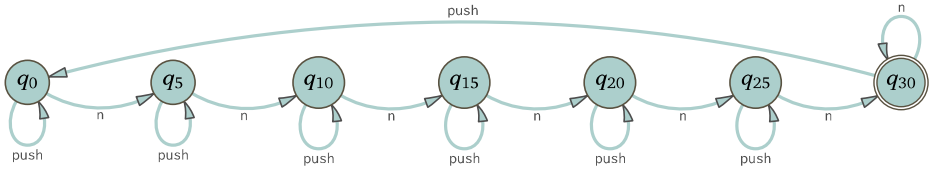


Figura 12.3: Máquina de vendas. Camada para as moedas de 5 centavos.

Após receber 30 centavos e receber outra moeda de 5 centavos, esta máquina faz algo não muito sensato: fica em q_{30}. Na prática, uma máquina teria mais estados para acompanhar os excessos, para que pudéssemos dar troco, mas aqui ignoramos isso. A seguir vêm as setas para as moedas de 10 centavos (que chamamos de *d*, de *dime* em inglês)

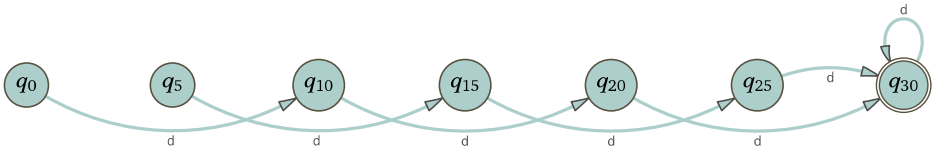


Figura 12.4: Máquina de vendas. Camada para as moedas de 10 centavos.

e para as moedas de 25 centavos (que chamamos de *q*, de *quarter* em inglês).

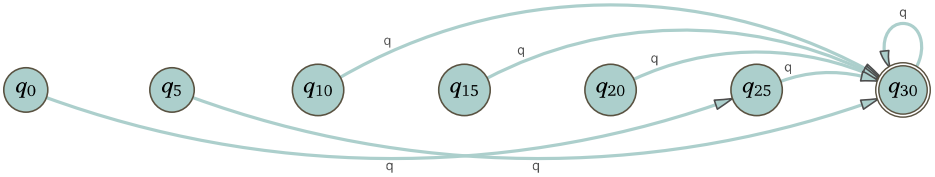


Figura 12.5: Máquina de vendas. Camada para as moedas de 25 centavos.

Exemplo 12.1.4

Esta máquina, quando iniciada no estado q_0 e alimentada com cadeias de bits, manterá o registro do resto módulo 4 da quantidade de 1's.

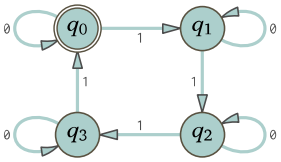


Figura 12.6: Quantidade de 1's módulo 4

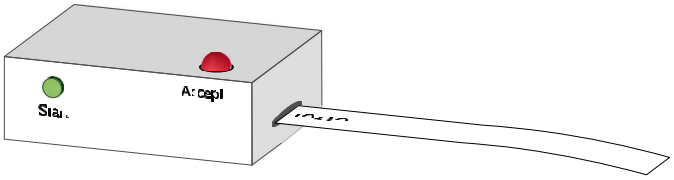
Definição

Uma **máquina de estados finitos**, ou **autômato de estados finitos**, é composta por cinco coisas, $\mathcal{M} = \langle Q, q_{\text{start}}, F, \Sigma, \Delta \rangle$. São elas: um **conjunto finito de estados** Q , um dos quais é o **estado inicial** q_{start}, um subconjunto $F \subseteq Q$ de **estados de aceitação** ou **estados finais**, um conjunto **alfabeto de entrada** finito Σ, e uma **função do próximo estado** ou **função de transição** $\Delta : Q \times \Sigma \to Q$.

Isto pode não parecer imediatamente como a nossa definição de uma Máquina de Turing. Parte disso deve-se ao fato de já termos definido os termos 'alfabeto' e 'função de transição'. As outras diferenças decorrem do fato de que as máquinas de estados finitos não podem escrever. Em primeiro lugar, por não poderem escrever, as máquinas de estados finitos não precisam mover a fita para o trabalho de rascunho, por isso abandonamos os símbolos de ações de fita L e R.

A outra diferença entre as máquinas de estados finitos e as máquinas de Turing é a presença dos estados de aceitação. Considere, na máquina de venda automática do Exemplo 12.1.3, o estado q_{30}. Ele é um estado de aceitação, o que significa que a máquina viu na entrada o que procura. O mesmo se aplica ao estado q_{ready} do Exemplo 12.1.2 e ao estado q_{on} do interruptor de alimentação do Exemplo 12.1.1. Embora possamos conceber uma Máquina de Turing para indicar uma escolha, organizando de modo que, para cada entrada, a máquina parará e a única coisa na fita será um 1 ou 0, uma máquina de estados finitos fornece uma decisão, terminando num destes estados designados. Podemos imaginar que os estados de aceitação são ligados a uma luz vermelha para que saibamos quando uma computação é bem-sucedida. Nos grafos de transição denota-se os estados finais com círculos duplos e nas tabelas de funções de transição marcamo-los com um '+'.

Para trabalhar com uma máquina de estados finitos, colocamos a entrada de comprimento finito na fita e pressionamos Start.



A máquina consome a entrada, em cada passo apagando o caractere anterior da fita e depois lendo o seguinte. Podemos rastrear os passos quando a máquina de módulo 4 do Exemplo 12.1.4 recebe a entrada `10110`.

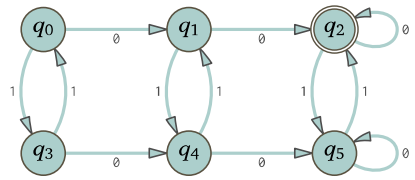
Passo	Configuração
0	1 0 1 1 0 q_0
1	0 1 1 0 q_1
2	1 1 0 q_1
3	1 0 q_2
4	0 q_3
5	q_3

Consequentemente, não há problema da parada para as máquinas de estados finitos — elas sempre param após um número de passos igual ao comprimento da entrada.

No final, ou a luz `Accept` está acesa ou não está. Se estiver acesa, então dizemos que a máquina **aceita** a cadeia de entrada, caso contrário **rejeita** a cadeia.

Exemplo 12.1.5

Esta máquina aceita uma string se e somente se contiver pelo menos dois `0`'s, bem como um número par de `1`'s. (O '+' ao lado de q_2 marca-o como um estado de aceitação).

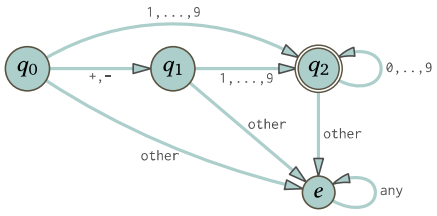


Δ	0	1
q_0	q_1	q_3
q_1	q_2	q_4
+ q_2	q_2	q_5
q_3	q_4	q_0
q_4	q_5	q_1
q_5	q_5	q_2

Esta máquina ilustra o segredo para conceber máquinas de estado finito, que cada estado tenha um significado intuitivo. O estado q_4 significa "até agora, a máquina viu um 0 e um número ímpar de 1's". E q_5 significa "até agora, a máquina viu dois 0's mas um número ímpar de 1's". O grafo traz à tona este princípio. A sua primeira linha tem estados que até agora têm visto um número par de 1's, enquanto os estados da segunda linha viram um número ímpar. A sua primeira coluna contém estados que não viram 0's, a segunda coluna contém estados que viram um zero, e a terceira coluna tem estados que viram dois 0's.

Exemplo 12.1.6

Esta máquina aceita strings que são válidas como representações decimais de números inteiros. Assim, ela aceita '21' e '−707' mas não aceita '501−'. Tanto o grafo de transição como a tabela agrupam algumas entradas quando resultam na mesma ação. Por exemplo, quando no estado q_0 esta máquina faz a mesma coisa, quer a entrada seja + ou −, ou seja, passa para q_1.



Δ	+, -	0, ... 9	*else*
q_0	q_1	q_2	e
q_1	e	q_2	e
+ q_2	e	q_2	e
e	e	e	e

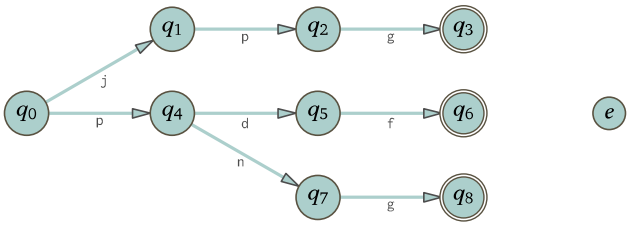
Qualquer caractere de entrada incorreto envia a máquina para o estado de erro, e, que é um estado de sumidouro, o que significa que a máquina nunca sai desse estado.

As nossas descrições de máquina de estado finito assumirão geralmente que o alfabeto é claro a partir do contexto. Por exemplo, o exemplo anterior apenas diz "else" (ou no grafo, "other"). Na prática, tomamos o alfabeto como sendo o conjunto de caracteres que alguém poderia possivelmente digitar, incluindo letras como `a` e `A` ou caracteres como ponto de exclamação ou abre parênteses. Assim, a concepção de uma

máquina de estados finitos de acordo com um padrão moderno poderia utilizar todo o Unicode. Mas para os exemplos e exercícios aqui, vamos utilizar pequenos alfabetos.

Exemplo 12.1.7

Esta máquina aceita cadeias que são membros do conjunto { `jpg`, `pdf`, `png` } de extensões de nomes de arquivo. Note que ela tem mais de um estado final.

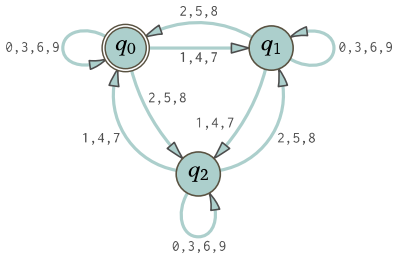


Esse desenho omite muitas arestas, as que envolvem o estado de erro e. Por exemplo, do estado q_0 qualquer caractere de entrada que não seja `j` ou `p` é um erro. (Colocar todas as arestas faria uma bagunça. Casos como este são aqueles em que a tabela de transição é melhor do que a figura do grafo. Mas a maioria das nossas máquinas são pequenas e têm apenas alguns estados e arestas, por isso, normalmente preferimos a figura).

Este exemplo ilustra que para qualquer linguagem finita existe uma máquina de estado finito que aceita uma cadeia se e somente se for um membro da linguagem. A ideia é colocar as cadeias em ordem alfabética, e para aquelas que têm prefixos comuns, a máquina passa pelas partes partilhadas em conjunto, como aqui com `pdf` e `png`.

Exemplo 12.1.8

Embora não tenham memória de rascunho, as máquinas de estados finitos podem realizar trabalhos úteis, como alguns tipos de aritmética. Esta máquina aceita cadeias representando um número natural que é um múltiplo de três, tais como `15` e `5013`.

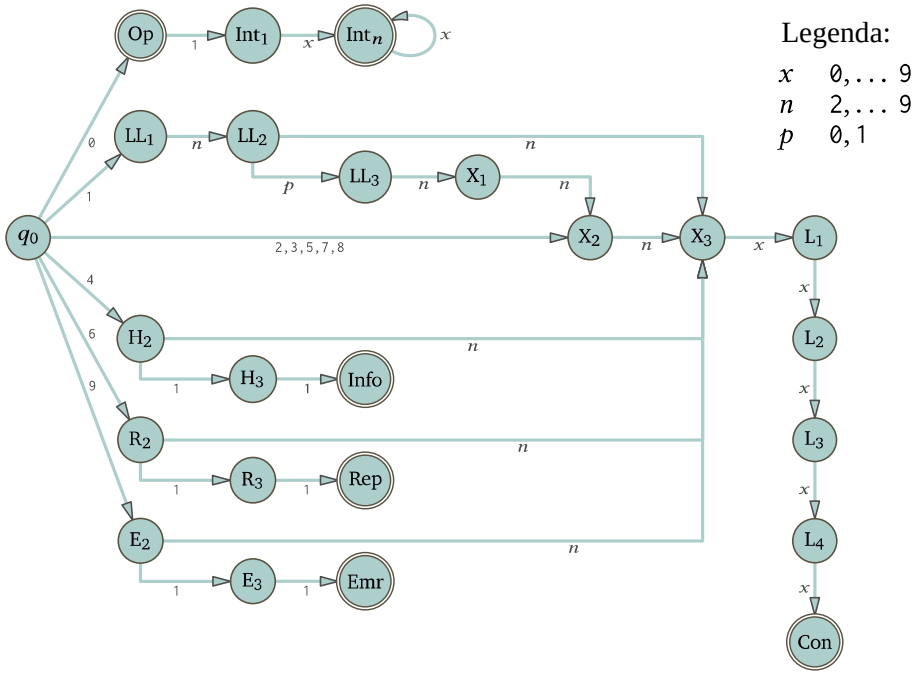


Como q_0 é um estado de aceitação, esta máquina aceita a cadeia vazia.

Exemplo 12.1.9

Esta é uma versão simplificada de como os números de telefone costumavam ser tratados na América do Norte. Considere o número 1-802-555-0101. O 1 inicial significa que a chamada deve deixar a central local para ir para as linhas de longa distância. O 802 é um código de área; o sistema pode perceber isso porque o seu segundo dígito é 0 ou 1, portanto não é uma central local da mesma área. A seguir o sistema processa o número de central local, o 555, encaminhando a chamada para uma central

local física específica. Essa central processa o número de linha 0101, e faz a conexão.



Hoje em dia, o quadro é muito mais complicado. Por exemplo, já não é necessário que os códigos de área tenham um dígito central 0 ou 1. Esta complicação adicional é possível porque em vez de comutar com dispositivos físicos, fazemo-lo agora em software.

Após a definição da máquina de Turing, demos uma descrição formal da ação dessas máquinas. A seguir, fazemos o mesmo aqui.

Uma **configuração** de uma máquina de estados finitos é um par $C = \langle q, \tau \rangle$, onde q é um estado, $q \in Q$, e τ é uma cadeia (possivelmente vazia), $\tau \in \Sigma^*$. Iniciamos uma máquina com alguma cadeia de **entrada** τ_0 e dizemos que a configuração inicial é $C_0 = \langle q_0 , \tau_0 \rangle$.

Uma máquina de estados finitos atua por uma sequência de **transições** de uma configuração para outra. Para $s \in \mathbb{N}^+$ a configuração da máquina após a s-ésima transição é a sua configuração no **passo** s, C_s .

Esta é a regra para fazer uma transição (dizemos por vezes que é uma transição **permitida** ou **legal**, para enfatizar). Suponha que a máquina está na configuração $C_s = \langle q, \tau_s \rangle$. No caso de τ_s não ser vazia, apague o símbolo inicial da cadeia, c. Ou seja, onde $c = \tau_s[0]$, pegue $\tau_{s+1} = \langle \tau_s[1], \ldots \tau_s[k] \rangle$ para $k = |\tau_s| - 1$. Então, o próximo estado da máquina é $\hat{q} = \Delta(q, c)$ e a sua próxima configuração é $C_{s+1} = \langle \hat{q}, \tau_{s+1} \rangle$. Denote esta relação antes-depois entre as configurações por $C_s \vdash C_{s+1}$.[1]

O outro caso é quando a string τ_s é vazia. Esta é a **configuração de parada** C_h. Nenhuma transição se segue a partir da configuração de parada.

1 Assim como para as Máquinas de Turing, leia o símbolo $\vdash$ como “deriva em um passo”.

A cada transição, o comprimento da string da fita diminui de um, de modo que cada computação eventualmente atinge uma configuração de parada $C_h = \langle q, \varepsilon \rangle$. Uma computação de máquina de estados finitos é uma sequência $C_0 \vdash C_1 \vdash C_2 \vdash \ldots \vdash C_h$. Podemos abreviar essa sequência por $\vdash^*$, como em $C_0 \vdash^* C_h$.[2]

Se o estado ao se chegar na configuração de parada for um estado final, $q \in F$, então a máquina aceita a entrada τ_0, caso contrário ela rejeita τ_0.

Note que, tal como no formalismo para as máquinas de Turing, o cerne das definições é a função de transição Δ. Ela faz a máquina mover-se passo a passo, de configuração em configuração, em resposta à entrada.

Exemplo 12.1.10

A máquina de múltiplos de três do Exemplo 12.1.8 fornece a computação. $\langle q_0$, 5013$\rangle$ $\vdash \langle q_2$, 013$\rangle \vdash \langle q_2$, 13$\rangle \vdash \langle q_0$, 3$\rangle \vdash \langle q_0, \varepsilon \rangle$. Uma vez que q_0 é um estado de aceitação, a máquina aceita 5013.

Definição

O conjunto de cadeias aceitas por uma máquina de estados finitos $\mathcal{M}$ é a **linguagem dessa máquina, $\mathcal{L}(\mathcal{M})$**, ou a linguagem **reconhecida**, ou **decidida**, (ou **aceita**), pela máquina.

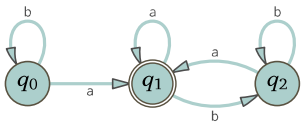
Para as máquinas de estados finitos, decidir uma linguagem é equivalente a reconhecê-la. 'Reconhecer' é o termo mais comum do que 'Decidir' aqui.

Definição

Para qualquer máquina de estados finitos com função de transição $\Delta : Q \times \Sigma \to Q$, a **função de transição estendida** $\hat{\Delta} : \Sigma^* \to Q$ fornece o estado em que a máquina terminará após ter começado no estado inicial e consumido a cadeia dada.

Exemplo 12.1.11

A função de transição estendida $\hat{\Delta}$ desta máquina



Δ	a	b
q_0	q_1	q_0
+ q_1	q_1	q_2
q_2	q_1	q_2

estende a sua função de transição ordinária Δ na medida em que repete a primeira linha da tabela de Δ.

$$\hat{\Delta}(\mathsf{a}) = q_1 \quad \hat{\Delta}(\mathsf{b}) = q_0$$

2 Leia o símbolo $\vdash^*$ como "deriva eventualmente", ou simplesmente "deriva".

(Ignoramos a diferença entre os caracteres de entrada Δ e as cadeias de caracteres de comprimento um de $\hat{\Delta}$). Aqui está o efeito de $\hat{\Delta}$ nas cadeias de caractere de comprimento dois.

$$\hat{\Delta}(\mathtt{aa}) = q_1 \quad \hat{\Delta}(\mathtt{ab}) = q_2 \quad \hat{\Delta}(\mathtt{ba}) = q_1 \quad \hat{\Delta}(\mathtt{bb}) = q_0$$

Note que isso requer determinismo, sem o qual $\hat{\Delta}$ não estaria bem definida; Δ tem um estado seguinte para todas as configurações de entrada e assim, por indução, para todas as cadeias de entrada, $\hat{\Delta}$ tem um estado de saída ao final.

Finalmente, note a semelhança entre $\hat{\Delta}$ e ϕ_e, a função computada pela máquina de Turing $\mathcal{P}_e$. Ambas tomam como entrada o conteúdo da fita inicial da sua máquina, e ambas fornecem como saída o resultado da sua máquina.

12.2. Não-determinismo

Tanto as máquinas de Turing quanto as máquinas de estados finitos têm a propriedade de que o estado seguinte é completamente determinado pelo estado e caractere atuais. Uma vez que se estabelece uma fita inicial e se aperta o `Start`, então simplesmente se caminha através dos passos como, bem… como um autômato. Consideramos agora máquinas não determinísticas, nas quais, a partir de qualquer configuração, a máquina poderia mover-se para mais de um estado seguinte, ou apenas para um, ou mesmo para nenhum estado.

Motivação. Imagine uma gramática com algumas regras e símbolo de início. É-nos fornecida uma cadeia e nos é perguntado se existe uma derivação. O desafio para estes problemas é que por vezes é preciso adivinhar qual o caminho que a derivação deve tomar. Por exemplo, se tivermos $S \to \mathtt{a}S \mid \mathtt{b}A$ então a partir de S podemos fazer duas coisas diferentes; qual delas funcionará?

Nos exemplos de derivações com gramáticas, esperamos que uma pessoa inteligente tivesse o discernimento para adivinhar o caminho correto. No entanto, se em vez disso você estivesse escrevendo um programa, então poderia tentar cada caso; poderia fazer uma busca em largura na árvore de todas as derivações, até obter um sucesso.

O filósofo estadunidense e famoso jogador de basebol Y Berra disse: “Quando chegar a uma bifurcação na estrada, pegue-a”. Essa é uma forma natural de atacar este problema: quando se deparar com múltiplas possibilidades, bifurque um processo filho para cada uma. Assim, a rotina pode começar com o estado inicial S e para cada regra que possa ser aplicada ela desencadeia um processo filho, derivando uma cadeia que é uma substituição desde o início. Depois disso, cada filho encontra cada regra que poderia aplicar à sua cadeia e gera os seus próprios filhos, cada um dos quais possui agora uma cadeia que é duas substituições desde o início. Continue até aparecer a cadeia desejada σ, se é que alguma vez ela vai aparecer.

O exemplo prototípico é o famoso problema do Caixeiro Viajante, o de encontrar o circuito mais curto para todas as cidades numa lista. Comece com um mapa das estradas nos vinte e seis estados do Brasil. Queremos saber se existe uma viagem que visite cada capital de estado e regresse ao local onde começou em, digamos, menos de 16.000 quilômetros. Vamos começar em Recife, a capital de Pernambuco. A partir

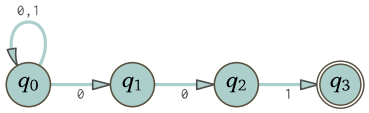
daí, para cada potencial próxima capital, podemos bifurcar um processo, fazendo vinte e cinco novos processos. O processo que é atribuído a João Pessoa, Paraíba, por exemplo, saberia que a viagem até agora é de 122 quilômetros. Na iteração seguinte, cada filho bifurcaria os seus próprios processos filhos, vinte e quatro deles. Por exemplo, o processo que, depois de Recife foi atribuído a João Pessoa, teria um filho atribuído a Natal, Rio Grande do Norte, e saberia que até agora a viagem é de 288 quilômetros. No final, se houver uma viagem de menos de 16.000 quilômetros, então algum processo saberá disso. Haverá muitos processos e muitos deles terão falhado em encontrar uma viagem curta, mas se mesmo um apenas for bem-sucedido, então consideramos a busca global como um sucesso.

Esta computação é não determinística, na medida em que enquanto está acontecendo a máquina está simultaneamente em muitos estados diferentes. Imagine uma máquina sem limites de paralelismo, onde sempre que houver um trabalho para um agente computacional adicional, uma CPU, você pode alocar uma.[3] Pense numa tal máquina como angelical, na medida em que sempre que ela quiser mais recursos computacionais, tais como a possibilidade de alocar novos filhos, esses recursos simplesmente aparecem.

Esta seção considera as máquinas de estados finitas não determinísticas (máquinas de Turing não determinísticas apareceram no capítulo dez). Teremos duas formas de pensar sobre o não-determinismo, dois modelos mentais.[4] O primeiro foi introduzido acima: quando uma máquina deste tipo é apresentada com múltiplos estados seguintes possíveis, então a máquina bifurca, de modo a estar em todos eles simultaneamente. O exemplo seguinte ilustra isso.

Exemplo 12.2.1

A máquina de estados finitos abaixo é não determinística porque saindo de q_0 há duas setas rotuladas como `0`. Ela também tem estados com um deficit de arestas; por exemplo, nenhuma seta `1` sai de q_1, portanto se ela estiver nesse estado e ler essa entrada então não passa para nenhum estado.



A figura a seguir mostra o que acontece com a entrada 00001. Representamos o rastro da computação como uma árvore. Por exemplo, no primeiro 0, a computação divide-se em duas.

3 Isto ecoa a nossa experiência com computadores quotidianos, quando estamos escrevendo um e-mail numa janela enquanto vemos um vídeo noutra. A máquina parece estar em múltiplos estados simultaneamente, embora possa, na realidade, estar "fatiando o tempo", ou seja, executando cada processo em sucessão por alguns tiques.

4 Embora estes modelos sejam úteis para aprender e pensar sobre o não-determinismo, eles não fazem parte das definições e provas formais.

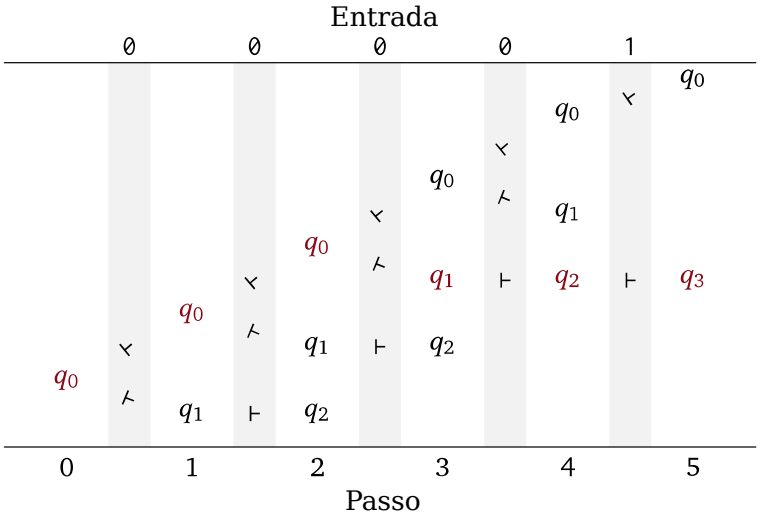


Figura 12.7: Árvore para a computação da máquina (não-determinística).

Quando consideramos a abordagem de bifurcação para derivações de strings ou para o Caixeiro Viajante, observamos que se existisse uma solução, então algum processo filho a encontraria. O mesmo acontece aqui; há um ramo da árvore da computação que aceita a cadeia de entrada. Há também ramos que não são bem-sucedidos. O da parte inferior morre após o passo 2 porque quando o estado atual é q_2 e a entrada é 0, esta máquina passa para o não-estado.[5] Outro é o ramo da parte superior, que nunca morre mas também não aceita a entrada. No entanto, não nos importamos com ramos mal sucedidos, apenas nos importamos que haja um ramo bem-sucedido. Por isso, vamos definir que uma máquina não determinística aceita uma entrada se houver pelo menos um ramo na árvore da computação que aceite a entrada.

A máquina no exemplo acima aceita uma cadeia se ela terminar em dois `0`'s e um `1`. Quando lhe fornecemos a entrada `00001`, o problema que a máquina enfrenta é: quando deve parar de percorrer o loop de q_0's e partir para a direita? A nossa definição diz que a máquina aceita esta entrada de modo que a máquina resolveu este problema — visto de fora, poderíamos dizer, talvez um pouco fantasiosamente, que a máquina adivinhou corretamente. Este é o nosso segundo modelo para o não-determinismo. Vamos imaginar a programação chamando uma função, alguma amb(S, R_0, R_1, ..., R_{n-1}), e o computador conseguindo, de alguma forma, adivinhar uma sequência bem-sucedida.

Dizer que a máquina está adivinhando é chocante. Com base nas aulas de programação, a intuição de uma pessoa pode ser que "adivinhar" não é mecanicamente realizável. Em vez disso, podemos imaginar que a resposta é fornecida à máquina ("dê a volta duas vezes, depois vá para a direita") e só temos que verificá-la. Este modelo mental de não determinismo parece demoníaco porque o fornecedor da resposta assemelha-se a um ser sobrenatural, um demônio, que de alguma forma conhece respostas que de outra forma não podem ser encontradas, mas não é confiável e portanto deve-se verificar que a resposta não é um truque. Sob este modelo, um cálculo não-determinístico aceita a entrada se existir um ramo da árvore de cálculo que uma máquina determinística, se lhe for dito que ramo tomar, poderia verificar.

5 O não-estado não pode ser um estado de aceitação, uma vez que não é sequer um estado.

Abaixo descreveremos o não determinismo usando ambos os paradigmas: como uma máquina que está em múltiplos estados ao mesmo tempo, e como uma máquina que adivinha. Como mencionado acima, aqui faremos isso para as máquinas de estados finitos, mas isso também poderia ser feito no contexto das máquinas de Turing.

Definição

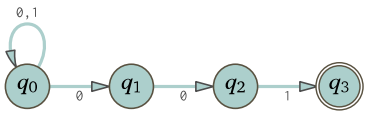
A função do próximo estado de uma máquina de estados finitos não determinística não produz estados únicos, mas sim conjuntos de estados.

Uma **máquina de estados finitos não determinística** $\mathcal{M} = \langle Q, q_{\text{start}}, F, \Sigma, \Delta \rangle$ consiste num conjunto finito de estados Q , um dos quais é o estado inicial q_{start}, um subconjunto $F \subseteq Q$ de estados de aceitação ou estados finais, um conjunto alfabeto de entrada finito Σ, e uma função do próximo estado $\Delta : Q \times \Sigma \to \mathbb{P}(Q)$.

Vamos utilizar estas máquinas de três maneiras. Primeiro, com elas descobrimos o não-determinismo, que é de extrema importância na análise de complexidade computacional. Em segundo lugar, elas também são úteis na prática; nos exemplos a seguir veremos tarefas que são mais facilmente resolvidas desta forma. Finalmente, vamos usá-los para demonstrar o Teorema de Kleene.

Exemplo 12.2.2

Esta é a máquina de estados finitos não determinística do Exemplo 12.2.1, juntamente com a sua função de transição.



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

Note que numa máquina não determinística, os valores das células da tabela de transição não são estados, são conjuntos de estados.

As máquinas não determinísticas podem parecer conceitualmente confusas, por isso as formalidades são uma ajuda. Contraste estas definições com as de máquinas determinísticas.

Uma **configuração** é um par $C = \langle q, \tau \rangle$, onde q $\in Q$ e $\tau \in \Sigma^*$. Uma máquina começa com uma configuração inicial $C_0 = \langle q_0, \tau_0 \rangle$. A string τ_0 é a **entrada**.

Dada a configuração inicial, pode haver uma ou mais sequências de **transições**. Suponha que existe uma configuração de máquina $C_s = \langle q, \tau_s \rangle$. Para $s \in \mathbb{N}^+$, no caso em que τ_s não é a cadeia de caracteres vazia, uma transição remove o primeiro símbolo c da cadeia de caracteres para obter τ_{s+1}, toma um membro $\hat{q}$ do conjunto $\Delta(q, c)$ como próximo estado da máquina e depois assume uma configuração subsequente $C_{s+1} = \langle \hat{q}, \tau_{s+1} \rangle$. Denote que duas configurações estão conectadas por uma transição por $C_s \vdash C_{s+1}$.

O outro caso é quando τ_s é a string vazia. Esta é uma **configuração de parada**, C_h. Depois de C_h, não se seguem transições.

Uma **computação não determinística de uma máquina de estados finitos** é uma sequência de transições que termina numa configuração de parada, $C_0 = \langle q_0, \tau_0\rangle \vdash C_1 \vdash C_2 \vdash \ldots C_h = \langle q, \varepsilon\rangle$. A partir de uma configuração inicial pode haver muitas dessas sequências. Se pelo menos uma terminar em um estado de parada com $q \in F$, então a máquina **aceita** a entrada τ_0, caso contrário ela **rejeita** τ_0.

Exemplo 12.2.3

Para a máquina não determinística do Exemplo 12.2.1, a figura Figura 12.7 mostra a seguinte computação:

$$\langle q_0, \texttt{00001}\rangle \vdash \langle q_0, \texttt{0001}\rangle \vdash \langle q_0, \texttt{001}\rangle \vdash \langle q_1, \texttt{01}\rangle \vdash \langle q_2, \texttt{1}\rangle \vdash \langle q_3\ , \varepsilon\rangle$$

Como termina num estado de aceitação, a máquina aceita a cadeia inicial, `00001`.

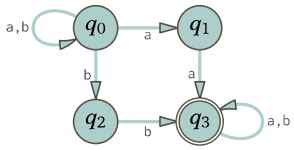
Definição

Para uma máquina de estados finitos não determinística $\mathcal{M}$, o conjunto de cadeias aceitas é a **linguagem da máquina $\mathcal{L}(\mathcal{M})$**, ou a linguagem **reconhecida**, (ou **aceita**), por essa máquina.[6]

Também vamos adaptar a definição da **função de transição estendida** $\hat{\Delta} : \Sigma^* \to Q$. Seja $\mathcal{M}$ uma máquina não-determinística com função de transição $\Delta : Q \times \Sigma \to Q$. Comece com $\hat{\Delta}(\varepsilon) = \{q_0\}$. Onde $\hat{\Delta}(\tau) = \{q_{i0}, q_{i1}, \ldots, q_{ik}\}$ para $\tau \in \Sigma^*$, defina $\hat{\Delta}(\tau \hat{}\ t) = \Delta(q_{i0},t) \cup \Delta(q_{i1},t) \cup \ldots \cup \Delta(q_{ik},t)$ para todo $t \in \Sigma$. Então a máquina aceita $\sigma \in \Sigma^*$ se e somente se qualquer elemento de $\hat{\Delta}(\sigma)$ for um estado final.

Exemplo 12.2.4

A linguagem reconhecida por esta máquina não determinística



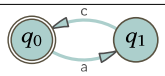
é o conjunto de cadeias que contém a substring `aa` ou `bb`. Por exemplo, a máquina aceita `abaaba` porque existe uma sequência de transições permitidas terminando num estado de aceitação, a saber, este:

$$\langle q_0, \texttt{abaaba}\rangle \vdash \langle q_0, \texttt{baaba}\rangle \vdash \langle q_0, \texttt{aaba}\rangle \vdash \langle q_1, \texttt{aba}\rangle \vdash \langle q_2, \texttt{ba}\rangle \vdash \langle q_2, \texttt{a}\rangle \vdash \langle q_2, \varepsilon\rangle$$

Exemplo 12.2.5

Com $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$ esta máquina não determinística.

6 Abaixo definiremos algo chamado ε transições que tornam 'reconhecida' a ideia adequada aqui, ao invés de 'decidida'.

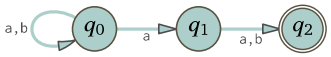


reconhece a linguagem $\{ (\texttt{ac})^n \ n \in \mathbb{N} \} = \{\varepsilon, \texttt{ac}, \texttt{acac}, \texttt{acac}, \ldots \}$. O símbolo `b` não aparece em nenhuma seta, portanto não desempenhará um papel em nenhuma cadeia aceitável.

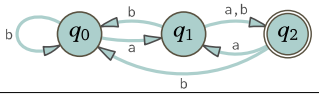
Muitas vezes uma máquina não determinística de estados finitos é mais fácil de escrever do que uma máquina determinística que faz o mesmo trabalho.

Exemplo 12.2.6

Esta é uma máquina não-determinística que aceita qualquer cadeia de caracteres cujo penúltimo caractere é `a`

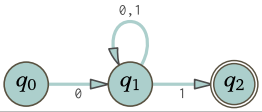


Ela é mais simples do que a máquina determinística:



Exemplo 12.2.7

Esta máquina aceita $\{\sigma \in \mathbb{B}^* \mid \sigma = 0 \hat{} \tau \hat{} 1 \text{ onde } \tau \in \mathbb{B}^*\}$.



Exemplo 12.2.8

Este é um receptor de abertura de porta de garagem que espera ouvir o controle remoto enviar o sinal `0101110`. Ou seja, reconhece a linguagem $\{\sigma \hat{} \texttt{0101110} \mid \sigma \in \mathbb{B}^*\}$.

Observação. Tendo visto alguns exemplos, paramos para reconhecer novamente, como fizemos quando discutimos o anjo e o demônio, que algo sobre o não-determinismo é inquietante. Se alimentarmos $\tau =$ `010101110` ao receptor do exemplo anterior, então ele a aceita.

$$\langle q_0, \texttt{010101110}\rangle \vdash \langle q_0, \texttt{10101110}\rangle \vdash \langle q_0, \texttt{0101110}\rangle \vdash \langle q_1, \texttt{101110}\rangle \vdash \langle q_2, \texttt{01110}\rangle \vdash \langle q_3, \texttt{1110}\rangle \vdash \langle q_4, \texttt{110}\rangle \vdash \langle q_5, \texttt{10}\rangle \vdash \langle q_6\ , \texttt{0}\rangle \vdash \langle q_7\ , \varepsilon\rangle$$

Mas a sequência de estados da máquina é concebida para uma string, `0101110`, que começa com dois conjuntos de `01`'s, enquanto que τ começa com três. Como ela adivinha que deve ignorar o primeiro `01` mas agir sobre o segundo? Claro, em matemática podemos considerar tudo o que pudermos definir com precisão. No entanto, estudamos até agora o que pode ser feito por dispositivos que, em princípio, são fisicamente realizáveis, de modo que isto pode parecer ser uma mudança de rumo.

No entanto, a seguir mostraremos como converter qualquer máquina de estados finitos não determinística em determinística que faz o mesmo trabalho. Assim, podemos pensar numa máquina de estados finitos não determinística como uma abreviatura, uma conveniência. Isto evita pelo menos uma parte do paradoxo de adivinhar, pelo menos para as máquinas de estados finitos.

Transições ε. Outra extensão, para além do não-determinismo, é permitir transições ε, ou movimentos ε. Alteramos a definição de uma máquina de estados finitos não determinística, de modo a que em vez de $\Delta : Q \times \Sigma \to \mathbb{P}(Q)$, a assinatura da função de transição é $\Delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathbb{P}(Q)$.[7] O comportamento associado é que a máquina pode transitar espontaneamente, sem consumir qualquer entrada.[8]

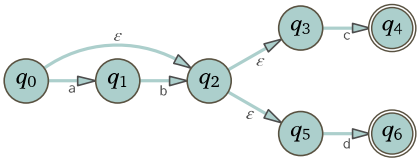
Exemplo 12.2.9

Esta máquina reconhece representações inteiras válidas. Note o ε entre q_0 e q_1.

Por causa do ε ela pode aceitar cadeias que não comecem com um sinal + ou −. Por exemplo, com a entrada 123 a máquina pode começar seguindo a transição ε para o estado q_1, depois ler o 1 e fazer a transição para q_2, e ficar lá enquanto processa o 2 e 3. Este é um ramo da árvore da computação que aceita a entrada, e por isso a cadeia 123 está na linguagem da máquina.

Exemplo 12.2.10

Uma máquina pode seguir duas ou mais transições ε. A partir de q_0 esta máquina pode permanecer nesse estado, ou fazer a transição para q_2, ou q_3, ou q_5, tudo sem consumir qualquer entrada.



Ou seja, a linguagem desta máquina é o conjunto de quatro elementos L = {`abc`, `abd`, `c`, `d`}.

Podemos fornecer uma definição precisa da ação de uma máquina de estados finitos não-determinística com transições ε.

Primeiro definimos a coleção de estados alcançáveis por movimentos ε a partir de um determinado estado. Para tal, utilizamos $E : Q \times \mathbb{N} \to \mathbb{P}(Q)$ onde $E(q, i)$ é o conjunto de estados alcançáveis a partir de q dentro de, no máximo, uma quantidade i de transições ε. Ou seja, defina $E(q, 0) = \{q\}$ e sendo $E(q, i) = \{q_{i0}, \ldots, q_{ik}\}$, temos que $E(q, i + 1) = E(q, i) \cup \Delta(q_{i0}, \varepsilon) \cup \ldots \cup \Delta(q_{ik}, \varepsilon)$. Observe que estes conjuntos são aninhados, $E(q, 0) \subseteq E(q, 1) \subseteq \ldots$ e que cada um deles é um subconjunto de Q.

7 Assuma que $\varepsilon \notin \Sigma$.
8 Ou, pense nisso como fazer uma transição ao consumir a cadeia de caracteres vazia, ε.

Mas Q tem apenas um número finito de estados, então deve haver um $\hat{\imath} \in \mathbb{N}$ onde a sequência de conjuntos pára de crescer, $E(q, \hat{\imath}) = E(q, \hat{\imath} + 1) = \ldots$. Defina a função de fechamento ε, $\hat{E} : \mathrm{Q} \to \mathcal{P}(Q)$ por $\hat{E}(q) = E(q, \hat{\imath})$.

Com isso, estamos prontos para descrever a ação da máquina. Tal como antes, uma **configuração** é um par $C = \langle q, \tau \rangle$, onde $q \in Q$ e $\tau \in \Sigma^*$. Uma máquina começa com alguma **configuração inicial** $C_0 = \langle q_0, \tau_0 \rangle$, em que a string τ_0 é a entrada.

A descrição chave é a de uma **transição**. Considere uma configuração $C_s = \langle q, \tau_s \rangle$ para $s \in \mathbb{N}$ e suponha que τ_s não é a cadeia de caracteres vazia. Vamos descrever uma configuração $C_{s+1} = \langle \hat{q}, \tau_{s+1} \rangle$ que está relacionada com a configuração dada por $C_s \vdash C_{s+1}$ (tal como na descrição anterior de máquinas não determinísticas sem transições ε, pode haver mais do que uma configuração relacionada desta forma com C_s). A cadeia de caracteres é fácil; basta retirar o primeiro caractere para obter $\tau_s = t \frown \tau_{s+1}$ onde $t \in \Sigma$. Para obter um estado legal $\hat{q}$: (i) encontre o fecho ε $\hat{E}(q) = \{q_{s0}, \ldots, q_{sk}\}$ (ii) faça $\bar{q}$ um elemento do conjunto $\Delta(q_{s0}, t) \cup \Delta(q_{s1}, t) \cup \ldots \cup \Delta(q_{sk}, t)$, e (iii) tome $\hat{q}$ como um elemento do fecho ε $\hat{E}(\bar{q})$.

Se τ_s é a string vazia, então esta é uma **configuração de parada**, C_h. Nenhuma transição segue C_h.

Uma **computação de uma máquina de estados finitos não determinística** é uma sequência de transições que termina numa configuração de parada, $C_0 = \langle q_0, \tau_0 \rangle \vdash C_1 \vdash C_2 \vdash \ldots C_h = \langle q, \varepsilon \rangle$. A partir de um dado C_0 pode haver muitas dessas sequências. Se pelo menos uma terminar com um estado de parada, tendo $q \in F$, então a máquina **aceita** a entrada τ_0, caso contrário **rejeita** τ_0.

Com isso, modificaremos a definição da função de transição estendida $\hat{\Delta} : \Sigma^* \to Q$ que fornecemos anteriormente. Comece definindo $\hat{\Delta}(\varepsilon) = \hat{E}(q_0)$. Depois a regra para passar de uma string para a sua extensão é a de $\tau \in \Sigma^*$ e onde $\hat{\Delta}(\tau) = \{q_{i0}, q_{i1}, \ldots q_{ik}\}$. $\Delta(\tau \frown t) = \hat{E}(\Delta(q_{i0}, t)) \cup \ldots \cup \hat{E}(\Delta(q_{ik}, t))$ para $t \in \Sigma$.

Observe que esta máquina não determinística com transições ε aceita uma cadeia $\sigma \in \Sigma^*$ se qualquer um dos estados em $\hat{\Delta}(\sigma)$ for um estado final.

Nota. É certo que estas constituem um conjunto intrincado de definições, mas elas estão aqui para demonstrar algo. Nos exemplos usamos frequentemente termos informais tais como “adivinhar” e “demônio”. No entanto, não tome esta linguagem evocativa e informal como uma incapacidade de seguir a ortodoxia matemática. Podemos perfeitamente dar definições e resultados com total precisão.

Exemplo 12.2.11

Para a máquina do Exemplo 12.2.10, esta sequência mostra que ela aceita `abc`:

$$\langle q_0, \mathsf{abc} \rangle \vdash \langle q_1, \mathsf{bc} \rangle \vdash \langle q_2 , \mathsf{c} \rangle \vdash \langle q_3, \mathsf{c} \rangle \vdash \langle q_4 , \varepsilon \rangle$$

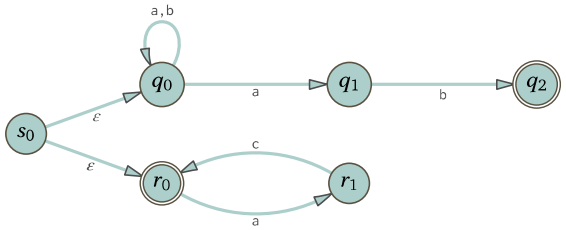
(note a transição ε entre q_2 e q_3). Esta sequência mostra que ela também aceita a cadeia de entrada d:

$$\langle q_0, d \rangle \vdash \langle q_5, d \rangle \vdash \langle q_6, \varepsilon \rangle.$$

Tal como com as máquinas não-determinísticas, uma das razões que nos leva a utilizar transições ε é que elas podem tornar a resolução de um trabalho complexo muito mais fácil.

Exemplo 12.2.12

Uma transição ε pode juntar duas máquinas com uma conexão paralela. Vejamos uma máquina cujos estados são nomeados com q's combinados com uma cujos estados são nomeados com r's

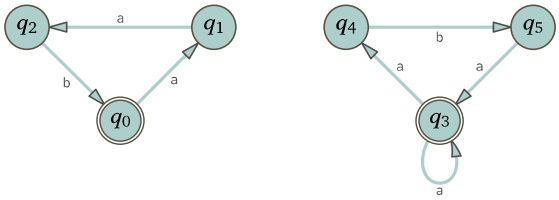


A linguagem da máquina não determinista superior é $\{\sigma \in \Sigma^* \mid \sigma \text{ termina em } \mathtt{ab}\}$ e a linguagem da máquina inferior é $\{\sigma \in \Sigma^* \mid \sigma = (\mathtt{ac})^n \text{ para algum } n \in \mathbb{N}\}$, onde $\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. A linguagem para a máquina inteira é a união:

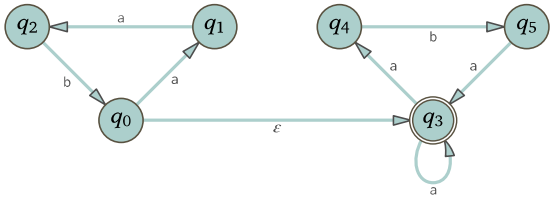
$$\{\sigma \in \Sigma^* \mid \sigma \text{ termina em } \mathtt{ab} \text{ ou } \sigma = (\mathtt{ac})^n \text{ para } n \in \mathbb{N}\}$$

Exemplo 12.2.13

Uma transição ε pode também fazer uma ligação em série entre máquinas. A máquina da esquerda abaixo reconhece a linguagem $\{(\mathtt{aab})^m \mid m \in \mathbb{N}\}$ e a máquina da direita reconhece $\{(\mathtt{a}|\mathtt{aba})^n \mid n \in \mathbb{N}\}$.



Se inserirmos uma ponte ε a partir de cada um dos estados finais do lado esquerdo (neste exemplo existe apenas um desses estados) para o estado inicial do lado direito



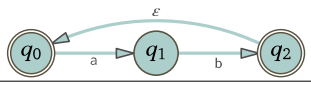
então a máquina combinada aceita cadeias na concatenação dessas linguagens.

$$\mathcal{L}(\mathcal{M}) = \{\sigma \in \{\mathtt{a}, \mathtt{b}\}^* \mid \sigma = (\mathtt{aab})^m\,(\mathtt{a}|\mathtt{aba})^n \text{ para } m, n \in \mathbb{N}\}$$

Por exemplo, ela aceita aabaababa e aabaabaaaa.

Exemplo 12.2.14

Uma aresta de transição ε pode também produzir o fecho de Kleene de uma máquina não-determinística. Por exemplo, sem a aresta ε a linguagem desta máquina é {ε, ab}, enquanto que com ela a linguagem é $\{(\mathsf{ab})^n \mid n \in \mathbb{N}\}$.



A seguir veremos que o não-determinismo não muda o que podemos fazer com as máquinas de estados finitos.

Teorema: Equivalência dos tipos de máquinas

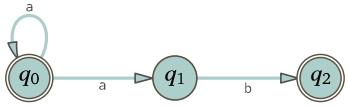
A classe de linguagens reconhecidas por máquinas de estados finitos não determinísticas é igual à classe de linguagens reconhecidas por máquinas de estados finitos determinísticas. Isto permanece verdadeiro se permitirmos que as máquinas não determinísticas tenham transições ε.

Podemos mostrar que as duas classes são iguais, mostrando que são subconjuntos uma da outra. Uma direção é fácil; qualquer máquina determinística é, basicamente, uma máquina não determinística. Ou seja, numa máquina determinística, a função do próximo estado produz estados únicos e para transformá-la numa máquina não determinística basta converter esses estados em conjuntos unitários. Assim, o conjunto de linguagens reconhecidas pelas máquinas determinísticas é um subconjunto do conjunto reconhecido pelas máquinas não determinísticas.

Vamos demonstrar a inclusão na outra direção de forma construtiva. Mostraremos como começar com uma máquina não determinística com transições ε e construir uma máquina determinística que reconhece a mesma linguagem. Não apresentaremos uma prova completa (embora certamente uma seja possível) simplesmente porque uma prova é relativamente complexa e os exemplos abaixo são inteiramente convincentes.

Exemplo 12.2.15

Considere esta máquina não determinística, $\mathcal{M}_N$, sem transições ε.



O que diferencia uma máquina não determinística é que ela pode estar em múltiplos estados ao mesmo tempo. Assim, para a máquina determinística associada $\mathcal{M}_D$, cada linha da tabela da função de transição abaixo é um conjunto $s_i = \{q_{i1}, \ldots q_{ik}\}$ dos estados $\mathcal{M}_N$'s.

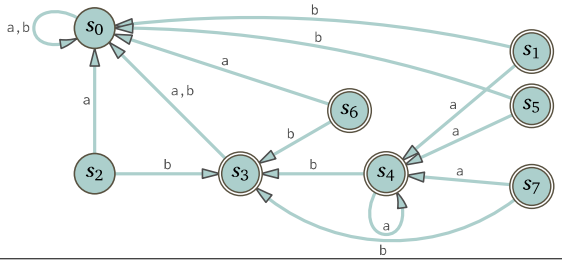
Como ilustração do processo de construção da tabela, suponha que a máquina acima está em $s_5 = \{q_0, q_2\}$ e está lendo a. Combine os estados seguintes devido a q_0, o conjunto $\Delta_N(q_0, \mathsf{a}) = \{q_0, q_1\}$, com os estados seguintes devido a q_2, o conjunto $\Delta_N(q_2, \mathsf{a}) = \{\ \}$. A união desses dois conjuntos dá $\Delta_D(s_5, \mathsf{a}) = \{q_0, q_1\}$, que abaixo é o estado s_4.

O estado inicial da $\mathcal{M}_D$ é $s_1 = \{q_0\}$. Um estado da máquina determinística $\mathcal{M}_D$ é de aceitação se algum dos seus elementos q's é um estado de aceitação em $\mathcal{M}_N$.

Em geral, compute a função de transição da máquina determinística com $\Delta_D(s_i, x) = \Delta_N(q_{i0}, x) \cup \ldots \cup \Delta_N(q_{ik}, x)$, onde $s_i = \{q_{i0}, \ldots q_{ik}\}$ e $x \in \Sigma$. Um exemplo é $\Delta_D(s_5, \mathsf{a}) = \Delta_N(q_0, \mathsf{a}) \cup \Delta_N(q_2, \mathsf{a})$, que é igual a $\{q_0, q_1\} \cup \{\ \} = \{q_0, q_1\} = s_4$.

Δ_D	a	b
$s_0 = \{\ \}$	s_0	s_0
$+\ s_1 = \{q_0\}$	s_4	s_0
$s_2 = \{q_1\}$	s_0	s_3
$+\ s_3 = \{q_2\}$	s_0	s_0
$+\ s_4 = \{q_0, q_1\}$	s_4	s_3
$+\ s_5 = \{q_0, q_2\}$	s_4	s_0
$+\ s_6 = \{q_1, q_2\}$	s_0	s_3
$+\ s_7 = \{q_0, q_1, q_2\}$	s_4	s_3

Além da conveniência da notação, nomear os conjuntos de estados como s_i's torna claro que $\mathcal{M}_D$ é uma máquina de estados finitos determinística. Tal como o seu grafo de transição.

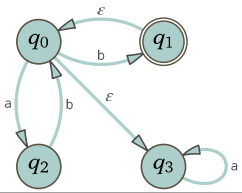


Se a máquina não determinística tem k estados, então sob esta construção a máquina determinista tem 2^k estados. Tipicamente, muitos deles podem ser eliminados. Por exemplo, na máquina acima referida, o estado s_6 é claramente inalcançável, uma vez que não há setas chegando nele.

A seguir expandimos essa construção para abranger transições ε. Basicamente, seguimos essas transições. Por exemplo, o estado inicial da máquina determinística é o fecho ε de $\{q_0\}$, o conjunto dos estados de $\mathcal{M}_N$ que são alcançáveis por uma sequência de transições ε a partir de q_0. Além disso, suponha que temos uma máquina não determinística e que estamos construindo a função do próximo estado da máquina determinística associada Δ_D, que a configuração atual é $s_i = \{q_{i1}, q_{i2}, \ldots\}$ e que a máquina está lendo a. Se houver uma transição ε de q_{ij} para algum q, então para o conjunto dos estados seguintes adicione $\Delta_N(\{q\}, \mathsf{a})$, e de fato adicione todo o fecho ε.

Exemplo 12.2.16

Considere esta máquina não determinística:



Para encontrar o conjunto dos próximos estados, siga as transições ε. Por exemplo, suponha que esta máquina está em q_0 e que o próximo caractere da fita é a. A seta à esquerda leva a máquina de q_0 para q_2. Alternativamente, seguindo a transição ε de q_0 para q_3 e depois lendo o a nos dá q_3. Portanto, a máquina estará em seguida nos dois estados q_0 e q_3. Estes são os fechos ε.

Estado q	q_0	q_1	q_2	q_3
Fecho ε $\hat{E}(q)$	$\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_2\}$	$\{q_3\}$

A máquina determinística completa está a seguir. O estado inicial é o fecho ε de $\{q_0\}$, o estado $s_7 = \{q_0, q_3\}$. Um estado é de aceitação se contiver algum elemento do fecho ε de q_1.

Δ_D	a	b
$s_0 = \{\}$	s_0	s_0
$+ s_1 = \{q_0\}$	s_{10}	s_{12}
$+ s_2 = \{q_1\}$	s_{10}	s_{12}
$s_3 = \{q_2\}$	s_0	s_7
$+ s_4 = \{q_3\}$	s_4	s_0
$+ s_5 = \{q_0, q_1\}$	s_{10}	s_{12}
$+ s_6 = \{q_0, q_2\}$	s_{10}	s_{12}
$+ s_7 = \{q_0, q_3\}$	s_{10}	s_{12}
$+ s_8 = \{q_1, q_2\}$	s_{10}	s_{12}
$+ s_9 = \{q_1, q_3\}$	s_{10}	s_{12}
$+ s_{10} = \{q_2, q_3\}$	s_4	s_7
$+ s_{11} = \{q_0, q_1, q_2\}$	s_{10}	s_{12}
$+ s_{12} = \{q_0, q_1, q_3\}$	s_{10}	s_{12}
$+ s_{13} = \{q_0, q_2, q_3\}$	s_{10}	s_{12}
$+ s_{14} = \{q_1, q_2, q_3\}$	s_{10}	s_{12}
$+ s_{15} = \{q_0, q_1, q_2, q_3\}$	s_{10}	s_{12}

Em geral, para um estado s e caractere de fita x, para computar $\Delta_D(s, x)$: (i) encontre o fecho ε de todos os q's do estado s, dando $\cup_{q \in s} \hat{E}(q) = \{q_{i0}, \ldots q_{ik}\}$, depois (ii) pegue a união dos próximos estados do q_{ij} para obter $T = \Delta_N(qi_0, x) \cup \ldots \cup \Delta_N(q_{i0}, x)$ e finalmente (iii) encontre o fecho ε de cada elemento de T, e tome a união de todos eles, $\cup_{t \in T} \hat{E}(t)$.

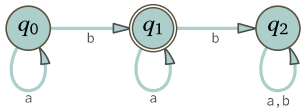
Como exemplo, tome $s = s_5 = \{q_0, q_1\}$ e $x = a$. Então (i) nos dá $\{q_0, q_3\} \cup \{q_0, q_1, q_3\} = \{q_0, q_1, q_3\}$. Os estados seguintes para (ii) são $\Delta_N(q_0, \text{a}) = \{q_2\}$, $\Delta_N(q_1, \text{a}) = \{\}$ e

$\Delta_N(q_3, \texttt{a}) = \{q_3\}$, portanto $T = \{q_2, q_3\}$. Finalmente, para (iii) a união dos fechos ε nos dá $\{q_2\} \cup \{q_3\} = \{q_2, q_3\} = s_{10}$.

12.3. Expressões regulares

Em 1951, S Kleene[9] estava estudando um modelo matemático de neurônios. Estes são como máquinas de estados finitos, na medida em que não têm memória de rascunho. Ele observou padrões para as linguagens que são reconhecidas por tais dispositivos.

Por exemplo, esta máquina de estados finitos



aceita cadeias que têm alguma quantidade de `a`'s, seguidas de um `b`, seguidas de mais alguns `a`'s e um `b`, e depois seguidas por mais caracteres. Kleene introduziu uma forma conveniente, chamada expressões regulares, para denotar construções tais como "qualquer número de" e "seguido por". Ele forneceu uma definição como veremos em breve, e apoiou-a com o teorema que veremos posteriormente.

Uma expressão regular é uma string que descreve uma linguagem. Iremos introduzi-las por alguns exemplos. Estes utilizam o alfabeto $\Sigma = \{\texttt{a}, \ldots \texttt{z}\}$.

Exemplo 12.3.1

A cadeia `h(a|e|i|o|u)t` é uma expressão regular que descreve cadeias que começam com `h`, têm uma vogal no meio, e terminam com `t`. Ou seja, esta expressão regular descreve a linguagem constituída por cinco palavras de três letras cada, $\mathcal{L} = \{$ `hat`, `het`, `hit`, `hot`, `hut`$\}$.

A barra vertical '|', que é uma espécie de 'ou', e os parênteses, que fornecem o agrupamento, não fazem parte das cadeias descritas; eles são **metacaracteres**.

Além da barra vertical e dos parênteses, a expressão regular também usa concatenação, uma vez que o `h` inicial é concatenado com `(a|e|i|o|u)`, que por sua vez é concatenado com `t`.

Exemplo 12.3.2

A expressão regular `ab*c` descreve a língua cujas palavras começam por `a`, seguidas por qualquer número de `b`'s (incluindo possivelmente zero `b`'s), e terminam com um `c`. Assim, '$*$' significa 'repetir a coisa anterior qualquer número de vezes'. Esta expressão regular descreve a linguagem L = { `ac`, `abc`, `abbc`, ...}.

Exemplo 12.3.3

9 Ele foi aluno de Church.

Há uma interação entre a barra vertical e a estrela. Considere a expressão regular (b|c)*. Ela poderia significar "qualquer número de repetições de escolhas entre b ou c" ou "escolha um b ou c e repita esse caractere qualquer número de vezes".

A definição adota o primeiro significado. Assim, a linguagem descrita por a(b|c)* consiste em palavras que começam com a e terminam com qualquer mistura de b's e c's, de modo que $\mathcal{L}$ = {a, ab, ac, abb, abc, acb, acc, ...}.

Em contraste, para descrever a linguagem cujos membros começam com a e terminam com qualquer número de b's ou qualquer número de c's, $\hat{L}$ = { a, ab, abb , ..., ac, acc, ...}, utilize a expressão regular a(b*|c*).

Ou seja, as regras para precedência do operador são: a estrela liga-se com mais força, depois a concatentação, depois o operador de alternância (barra vertical), |. Para obter outra ordem, usar parênteses.

Definição: Expressões regulares

Seja Σ um alfabeto que não contém nenhum dos metacaracteres), (,|, ou *. Uma **expressão regular sobre Σ** é uma cadeia que pode ser derivada desta gramática

⟨*regex*⟩ → ⟨*concat*⟩
| ⟨*regex*⟩ '|' ⟨*concat*⟩

⟨concat⟩ → ⟨*simple*⟩
| ⟨*concat*⟩ ⟨*simple*⟩

⟨*simple*⟩ → ⟨*char*⟩
| ⟨*simple*⟩*
| (⟨*regex*⟩)

⟨*char*⟩ → ∅ | ε | x_0 | x_1 | ...

onde os caracteres x_i são membros do alfabeto Σ .[10]

Quanto à sua semântica, o que significam as expressões regulares, definiremos isso recursivamente. Começamos com a última linha, as expressões regulares com um único caractere, e forneceremos a linguagem que cada opção ali descreve. Trataremos então da formação obtida nas outras linhas, considerando cada uma das possibilidades como a descrição de uma linguagem.

A linguagem descrita pela expressão regular de um único caractere ∅ é o conjunto vazio, $\mathcal{L}(\emptyset) = \emptyset$. A linguagem descrita pela expressão regular que consiste apenas no caractere ε é a linguagem de um só elemento que consiste apenas na cadeia vazia, L(ε) = {ε}. Se a expressão regular consiste em apenas um caractere do alfabeto Σ então a linguagem que descreve contém apenas uma cadeia de caracteres e essa cadeia tem apenas esse único caractere, como em $\mathcal{L}$(a) = {a}.

Terminamos com a definição da semântica das operações. Começamos com as expressões regulares R_0 e R_1 que descrevem as linguagens $\mathcal{L}(R_0)$ e $\mathcal{L}(R_1)$. Assim, o símbolo

10 Tal como fizemos com outras gramáticas, aqui usamos o símbolo da barra vertical | como um metacaractere, para colapsar regras com o mesmo lado esquerdo. Mas a barra também aparece em expressões regulares. Para esse uso, envolvemo-la em aspas simples, como '|'.

da barra vertical descreve a união das linguagens, de modo a que $\mathcal{L}(R_0|R_1) = \mathcal{L}(R_0) \cup \mathcal{L}(R_1)$. A concatenação das expressões regulares descreve a concatenação das linguagens, $\mathcal{L}(R_0 \hat{} R_1) = \mathcal{L}(R_0) \hat{} \mathcal{L}(R_1)$. E, o fecho de Kleene da expressão regular descreve o fecho da linguagem, $\mathcal{L}(R_0^*) = \mathcal{L}(R_0)^*$.

Exemplo 12.3.4

Considere a expressão regular aba* sobre $\Sigma = \{a, b\}$. Trata-se da concatenação de a, b, e a*. O primeiro descreve a linguagem de um elemento $\mathcal{L}(a) = \{a\}$. Da mesma forma, o segundo descreve $\mathcal{L}(b) = \{b\}$. Assim, a string ab descreve a concatenação das duas, outra linguagem de um elemento.

$$\mathcal{L}(ab) = \mathcal{L}(a) \hat{} \mathcal{L}(b) = \{\sigma \in \Sigma^* \mid \sigma = \sigma_0 \hat{} \sigma_1 \text{ onde } \sigma_0 \in \mathcal{L}(a) \text{ e } \sigma_1 \in \mathcal{L}(b)\} = \{ab\}$$

A expressão regular a* descreve o fecho de Kleene da linguagem $\mathcal{L}(a)$, nomeadamente $\mathcal{L}(a^*) = \{a^n \mid n \in \mathbb{N}\}$. Concatená-la com $\mathcal{L}(ab)$ nos fornece:

$$\begin{aligned}\mathcal{L}(aba^*) &= \{\sigma \in \Sigma^* \mid \sigma = \sigma_0 \hat{} \sigma_1 \text{ onde } \sigma_0 \in \mathcal{L}(ab) \text{ e } \sigma_1 \in \mathcal{L}(a^*)\ \} \\ &= \{ab, aba, abaa, aba^3, \ldots\} = \{aba^n \mid n \in \mathbb{N}\}\end{aligned}$$

Terminamos esta subsecção com algumas construções que aparecem frequentemente. Estes exemplos usam $\Sigma = \{a, b, c\}$.

Exemplo 12.3.5

Descreva a linguagem que consiste em strings de a's cujo comprimento é um múltiplo de três, $\mathcal{L} = \{a^{3k} \mid k \in \mathbb{N}\} = \{\varepsilon, aaa, aaaaaa, \ldots\}$, com a expressão regular (aaa)*.

Note que a cadeia vazia é um membro dessa linguagem. Um erro comum é esquecer que a estrela significa para qualquer número de repetições, incluindo zero.

Exemplo 12.3.6

Para corresponder a qualquer caractere, podemos listá-los todos. A linguagem que consiste em palavras de três letras terminadas em bc é {abc , bbc, cbc}. A expressão regular (a|b|c)bc descreve-a. (Na prática, o alfabeto pode ser muito grande para que a listagem de todos os caracteres seja impraticável; veja a seção 12.3.1)

Exemplo 12.3.7

A expressão regular a*(ε|b) descreve a linguagem das cadeias que têm qualquer número de a's e opcionalmente terminam em um b, $\mathcal{L} = \{\varepsilon, b, a, ab, aa, aab , \ldots\}$. Da mesma forma, para descrever a linguagem que consiste em palavras tendo entre três e cinco a's, $\mathcal{L} = \{$ aaa, aaaa, aaaaa $\}$ podemos usar aaa(ε|a|aa).

Exemplo 12.3.8

A linguagem {b , bc, bcc, ab, abc, abcc, aab, ...} tem palavras que começam com qualquer número de a's (incluindo zero a's), seguidas de um único b, e depois terminam em menos de três c's. Para descrevê-la podemos usar a*b(ε|c|cc).

O resultado a seguir justifica o nosso estudo das expressões regulares porque mostra que elas descrevem as nossas linguagens de interesse.

Teorema (Teorema de Kleene)

Uma linguagem é reconhecida por uma máquina de estados finitos se e somente se essa linguagem for descrita por uma expressão regular.

Prová-lo-emos em metades separadas. As demonstrações utilizam máquinas não determinísticas, mas como podemos convertê-las em máquinas determinísticas, o resultado também se mantém ali.

Lema

Se uma linguagem é descrita por uma expressão regular, então existe uma máquina de estados finitos que reconhece essa linguagem.

Demonstração. Mostraremos que para qualquer expressão regular R existe uma máquina que aceita cadeias que correspondem a essa expressão. Utilizaremos indução na estrutura das expressões regulares.

Comecemos com expressões regulares que consistem num único caractere. Se $R = \emptyset$ então $\mathcal{L}(R) = \{\ \}$ e a máquina à esquerda abaixo reconhece $\mathcal{L}(R)$. Se $R = \varepsilon$ então $\mathcal{L}(R) = \{\varepsilon\}$ e a máquina do meio reconhece esta linguagem. Se a expressão regular for um caractere do alfabeto, tal como $R = \mathtt{a}$, então a máquina à direita funciona.

Terminamos com o tratamento das três operações. Sejam R_0 e R_1 expressões regulares; a hipótese indutiva nos dá uma máquina $\mathcal{M}_0$ cuja linguagem é descrita por R_0 e uma máquina $\mathcal{M}_1$ cuja linguagem é descrita por R_1.

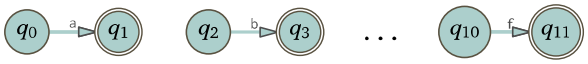
Primeiro considere a alternância, $R = R_0 \mid R_1$. Crie a máquina que reconhece a linguagem descrita por R unindo essas duas máquinas em paralelo: introduza um novo estado s e utilize transições ε para conectar s aos estados iniciais de $\mathcal{M}_0$ e $\mathcal{M}_1$. Reveja o Exemplo 12.2.12.

A seguir considere a concatenação, $R = R_0 \hat{} R_1$. Junte as duas máquinas em série: para cada estado de aceitação em $\mathcal{M}_0$, faça uma transição ε para o estado inicial de $\mathcal{M}_1$ e depois converta todos os estados de aceitação de $\mathcal{M}_0$ para estados de não aceitação. Reveja o Exemplo 12.2.13.

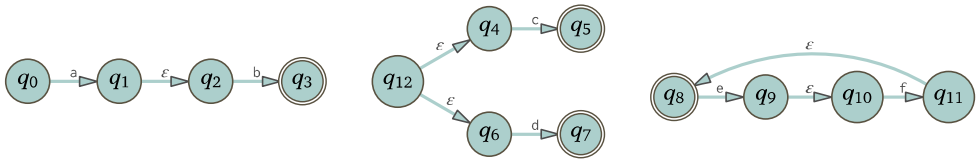
Finalmente considere o fecho de Kleene, $R = (R_0)^*$. Para cada estado de aceitação na máquina $\mathcal{M}_0$ que não é o estado inicial, faça uma transição ε para o estado inicial, e depois faça do estado inicial um estado de aceitação. Veja o Exemplo 12.2.14.

Exemplo 12.3.9

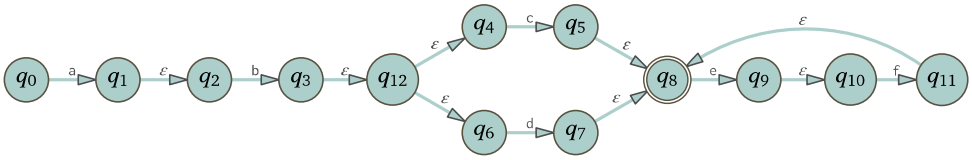
O processo de construção de uma máquina para a expressão regular `ab(c|d)(ef)*` começa com máquinas para os caracteres individuais.



Juntamos estes componentes atômicos



para obter a máquina completa.



Esta máquina é não-determinística. Para obter uma determinística, utilizamos o processo de conversão que vimos na seção anterior.

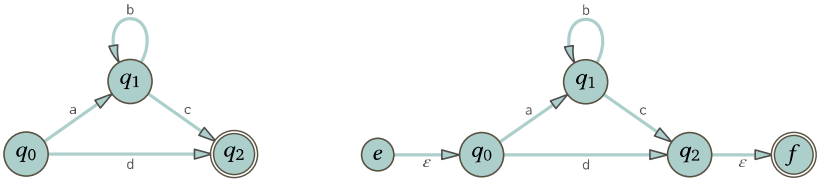
Lema

Qualquer linguagem reconhecida por uma máquina de estados finitos é descrita por uma expressão regular.

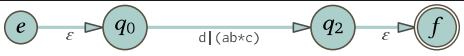
A nossa estratégia começa com uma máquina de estados finitos e elimina os seus estados um de cada vez. Abaixo está um exemplo, imagens antes e depois de parte de uma máquina maior, em que eliminamos o estado q.

Na imagem "depois" a aresta é rotulada por ab, com mais do que simplesmente um caractere. Para a demonstração, iremos generalizar os grafos de transição para permitir rótulos de arestas que são expressões regulares. Eliminaremos os estados, mantendo a mesma linguagem reconhecida. Terminaremos quando restarem apenas dois estados, com uma aresta entre eles. O rótulo da aresta será a expressão regular desejada.

Antes da demonstração, um exemplo. Considere a máquina à esquerda, abaixo.



A prova começa como acima à direita, introduzindo um novo estado inicial, e, com a garantia de não possuir arestas de entrada e um novo estado final, f, com a garantia de ser único. Depois a prova elimina q_1 como abaixo.



Claramente, esta máquina reconhece a mesma linguagem que a máquina inicial.

Demonstração. Chame a máquina de $\mathcal{M}$. Se não houver estados de aceitação, então a expressão regular é $\emptyset$ e estamos terminados. Caso contrário, transformaremos $\mathcal{M}$ numa nova máquina, $\hat{\mathcal{M}}$, com a mesma linguagem, na qual podemos executar a estratégia de eliminação de estados.

Primeiro providenciamos que $\hat{\mathcal{M}}$ tenha um único estado de aceitação. Crie um novo estado f e para cada um dos estados de aceitação de $\mathcal{M}$ faça uma transição ε para f (pelo parágrafo anterior há pelo menos um desses estados de aceitação). Mude todos os estados de aceitação para estados de não aceitação e depois faça f de aceitação.

Em seguida, introduza um novo estado de início, e. Faça uma transição ε entre ele e q_0 (assegurando que $\hat{\mathcal{M}}$ tem pelo menos dois estados permite-nos lidar uniformemente com máquinas de todos os tamanhos).

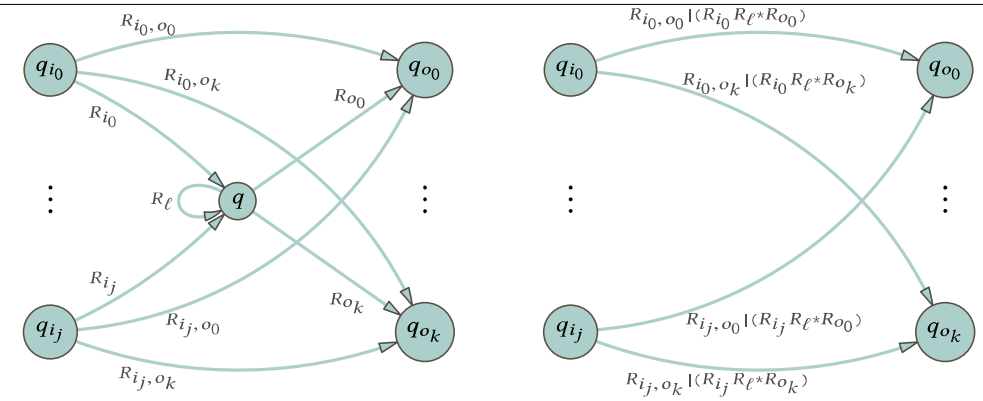
Como os rótulos das arestas são expressões regulares, podemos organizar para que de qualquer q_i para qualquer q_j haja no máximo uma aresta, porque se $\mathcal{M}$ tiver mais do que uma aresta, então em $\hat{\mathcal{M}}$ utilize a barra vertical, |, para combinar os rótulos, como aqui.

Faça o mesmo com loops, ou seja, casos em que $i = j$. Tal como as transformações anteriores, é evidente que isto não altera a linguagem das strings aceitas.

A última parte da transformação para $\hat{\mathcal{M}}$ é abandonar quaisquer estados inúteis. Se um nó de estado que não seja f não tem arestas de saída, então descarte-o juntamente com as arestas de entrada dele. A linguagem da máquina não mudará porque este estado não pode levar a um estado de aceitação, uma vez que não leva a lugar nenhum, e este estado não é por si só de aceitação, pois apenas f é de aceitação.

Na mesma linha, se um nó de estado q não for alcançável desde o início, então pode-se descartar esse nó juntamente com as suas arestas de entrada e de saída (consideramos clara a ideia de inalcançável, sem fornecermos uma definição formal).

Com isso, $\hat{\mathcal{M}}$ está pronta para a eliminação de estados. Abaixo encontram-se imagens antes e depois. A imagem antes mostra um estado q a ser eliminado. Há estados q_{i0}, … q_{ij} com uma aresta chegando em q, e estados q_{o0} , … q_{ok} que recebem uma aresta que sai de q. (Pelo trabalho de configuração acima, q tem pelo menos uma aresta de entrada e pelo menos uma de saída). Além disso, q pode ter um laço.



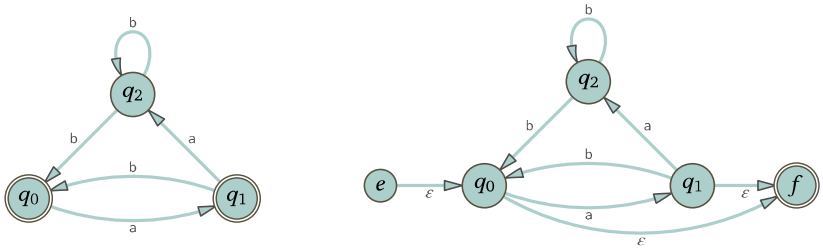
(Aqui está um ponto sutil: possivelmente alguns dos estados mostrados à esquerda de cada uma das duas imagens equivalem a alguns mostrados à direita. Por exemplo, possivelmente q_{i0} é igual a q_{o0}. Se assim for, então a aresta mostrada R_{i0},o_0 é um laço).

Elimine q e as arestas associadas, fazendo as substituições mostradas na imagem "depois". Observe que o conjunto de cadeias que levam a máquina de qualquer estado de entrada q_i para qualquer estado de saída q_o permanece inalterado. Portanto, a linguagem reconhecida pela máquina mantém-se inalterada.

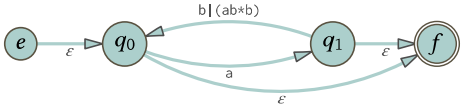
Repita esta eliminação até que tudo o que resta seja e e f, e a aresta entre eles. (A máquina tem um número finito de estados, portanto este procedimento deve eventualmente parar). A expressão regular desejada é o rótulo da aresta.

Exemplo 12.3.10

Considere $\mathcal{M}$ à esquerda. Introduza e e f para obter M̂ à direita.

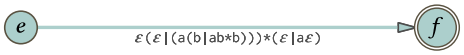


Comece eliminando q_2. Nos termos do passo chave da prova, q1 = qi0 e q0 = qo0. As expressões regulares são Ri0 = a, Ro0 = b, Ri0,o0 = b, e Rl = b. Isso nos dá esta máquina:



A seguir elimine q_1. Há um nó de entrada $q_0 = q_{i0}$ e dois nós de saída $q_0 = q_{o0}$ e $f = q_{o1}$ (Note que q_0 é tanto um nó de entrada como de saída; este é o ponto sutil mencionado na prova). As expressões regulares são $R_{i0} = a$, R_{o0} = b|(ab*b), e $R_{o1} = \varepsilon$.

Tudo o que resta é eliminar q_0. O único nó de entrada é $e = q_{i0}$ e o único nó de saída é $f = q_{o0}$, e assim $R_{i0} = \varepsilon$, $R_{o0} = \varepsilon|a\varepsilon$, e $R_1 = \varepsilon|a(b\ |ab*b)$.



Esta expressão regular pode ser simplificada. Por exemplo, $a\varepsilon = a$

12.3.1 Expressões Regulares na Prática

As expressões regulares são frequentemente utilizadas na prática. Por exemplo, imagine que você precisa procurar num log de servidor web os nomes de todos os PDF's baixados de um subdiretório. Um utilizador num sistema derivado do Unix poderia escrever algo assim no terminal:

```
grep "/linearalgebra/.*pdf" /var/log/apache2/access.log
```

O utilitário `grep` percorre o arquivo linha por linha, e se uma linha corresponder ao padrão, então o `grep` imprime essa linha. Esse padrão, começando com o subdiretório `/linearalgebra/`, é uma expressão regular estendida.

Ou seja, na prática precisamos frequentemente de operações em texto, e as expressões regulares são uma ferramenta importante. As linguagens modernas de programação como Python e Scheme incluem recursos para **expressões regulares estendidas**, por vezes chamadas **regexes**, que vão além dos exemplos da teoria em pequena escala que vimos anteriormente. Estas extensões enquadram-se em duas categorias. A primeira são construções convenientes que tornam mais fácil algo que de outra forma seria exequível, mas trabalhoso. A segunda é que algumas das extensões a expressões regulares em linguagens de programação modernas não se limitam a simples conveniências. Veremos um pouco mais sobre isto mais tarde.

Em primeiro lugar, as extensões de conveniência. Muitas delas são puramente relativas à escala: os nossos alfabetos anteriores tinham dois ou três caracteres mas na prática um alfabeto deve incluir pelo menos os caracteres ASCII imprimíveis: `a–z`, `A–Z`, `0–9`, espaço, tabulação, ponto, hífen, ponto de exclamação, sinal de percentagem, sinal de dólar, parênteses aberto e fechado, chaves abertas e fechadas, etc. Pode até mesmo conter todos os mais de cem mil caracteres Unicode. Precisamos de formas praticáveis para descrever conjuntos maiores de caracteres.

Considere a correspondência de um dígito. A expressão regular `(0|1|2|3|4|5|6|7|8|9)` é verbosa demais para uma lista frequentemente necessária. Uma abreviatura que as linguagens modernas permitem é `[0123456789]`, omitindo os caracteres de barra vertical e utilizando colchetes, que em expressões regulares estendidas são metacaracteres. Ou, como os caracteres dos dígitos são contíguos no conjunto de caracteres[11], podemos encurtá-la ainda mais para `[0-9]`. Na mesma linha, `[A-Za-z]` corresponde a uma única letra do alfabeto.

Para inverter o conjunto de caracteres correspondentes, coloque um circunflexo '^' como primeira coisa dentro do colchete (note que ele é um metacaractere). Assim, `[^0-9]` corresponde a um não-dígito e `[^A-Za-z]` corresponde a um caractere que não é uma letra ASCII.

11 Isto é verdade tanto em ASCII como em Unicode.

As listas mais comuns têm abreviaturas curtas. Outra abreviatura para os dígitos é `\d`. Use `\D` para os caracteres ASCII não-dígitos, `\s` para os caracteres de espaço em branco (espaço, tabulação, nova linha, formfeed, e retorno de linha) e `\S` para os caracteres ASCII que não são espaço em branco. Para abranger os caracteres alfanuméricos (letras maiúsculas e minúsculas ASCII, dígitos, e sublinhado) use `\w`, e para os caracteres não alfanuméricos ASCII use `\W`. E — o grande símbolo — o ponto '`.`' é um metacaractere que corresponde a qualquer membro do alfabeto.[12] Vimos o ponto sendo usado no exemplo de boas-vindas que iniciou esta discussão.

Exemplo 12.3.11

Os códigos postais canadenses têm sete caracteres: o quarto é um espaço, o primeiro, terceiro, e sexto são letras, e os outros são dígitos. A expressão regular

```
[a-zA-Z]\d[a-zA-Z] \d[a-zA-Z]\d
```

descreve-os.

Exemplo 12.3.12

As datas são frequentemente fornecidas no formato '`dd/mm/yy`'. Isto corresponde à regex[13]

```
\d\d/\d\d/\d\d
```

Exemplo 12.3.13

No formato de doze horas, algumas horas típicas são '`8:05 am`' ou '`10:15 pm`'. Esta regex poderia ser utilizada (note a cadeia vazia no início)

```
(|0|1)\d:\d\d\s(am|pm)
```

Recorde que na expressão regular `a(b|c)d` os parênteses e a barra vertical não estão lá para ser correspondidos. São metacaracteres, parte da sintaxe da expressão regular. Uma vez expandido o alfabeto Σ para incluir todos os caracteres, deparamo-nos com o problema de já estarmos utilizando alguns dos caracteres adicionais como metacaracteres.

Para fazer corresponder a um metacaractere, prefixamo-lo com uma contrabarra, '`\`'. Assim, para procurar pela cadeia '(Nota' coloque uma barra invertida antes do parêntese aberto `\(Nota`. Da mesma forma, `\|` corresponde a uma barra vertical e `\[` corresponde a um colchete aberto. Correspondente à própria contrabarra é `\\`. A isto chama-se **escapar** o metacaractere. O esquema descrito acima para representar listas por `\d`, `\D`, etc. é uma extensão do escape.

12 As linguagens de programação na prática, por padrão, fazem o ponto corresponder a qualquer caractere, exceto a nova linha. Além disso, têm uma forma de o fazer coincidir também com a nova linha.

13 Note todavia que isso considerará válida uma data como 98/99/10

A precedência dos operadores é: a repetição liga-se mais fortemente, depois a concatenação, e depois a alternância (force significados diferentes com parênteses). Assim, `ab*` é equivalente a `a(b*)`, e `ab|cd` é equivalente a `(ab)|(cd)`.

Quantificadores. Nos casos teóricos que vimos anteriormente, para corresponder a 'no máximo um a', utilizamos ε `|a`. Na prática, podemos escrever algo como `(|a)`, como fizemos acima para as horas no formato de doze horas. Mas representar a cadeia vazia, simplesmente não colocando nada lá, pode ser confuso. As linguagens modernas fazem do ponto de interrogação um metacaractere e permitem-lhe escrever `a?` para 'no máximo um `a`'.

Para 'pelo menos um a' linguagens modernas usam `a+`, por isso o sinal de mais é outro metacaractere. De uma forma mais geral, queremos frequentemente especificar quantidades. Por exemplo, para corresponder a cinco `a`'s, as expressões regulares estendidas usam as chaves como metacaracteres, com `a{5}`. Fazemos corresponder entre dois e cinco deles com `a{2,5}` e fazemos corresponder a pelo menos dois com `a{2,}`. Assim, `a+` é a abreviatura de `a{1,}`.

Como anteriormente, para corresponder a qualquer um destes metacaracteres, é necessário escapar-lhes. Por exemplo, `ser ou não ser\?` corresponde à famosa pergunta.

Receitas de bolo. Todas as extensões a expressões regulares que estamos vendo são impulsionadas pelos desejos dos programadores da vida real. Aqui está uma pilha de exemplos que as mostram realizando trabalhos práticos, correspondendo a coisas que gostaria de combinar.

Exemplo 12.3.14

Códigos postais do Brasil, chamados de CEP, são de cinco dígitos, seguidos de hífen e mais três dígitos. Podemos usar a regex: `\d{5}-\d{3}`.

Exemplo 12.3.15

Números de telefone norte-americanos coincidem com: `\d{3} \d{3}-\d{4}`.

Exemplo 12.3.16

A expressão regular `(-|\+)?\d+` corresponde a um número inteiro, positivo ou negativo. O ponto de interrogação torna o sinal opcional. O sinal de mais assegura a existência de pelo menos um dígito; escapa-se porque + é um metacaractere.

Exemplo 12.3.17

A expressão `[a-fA-F0-9]+` corresponde a uma representação numérica natural em hexadecimal. Os programadores prefixam frequentemente tal representação com `0x`, portanto a expressão se torna `(0x)?[a-fA-F0-9]+`.

Exemplo 12.3.18

Um identificador da linguagem C começa com uma letra ASCII ou sublinhado e depois pode ter arbitrariamente muitas mais letras, dígitos, ou sublinhados: `[a-zA-Z_]\w*`.

Exemplo 12.3.19

A correspondência com um nome de utilizador entre três e doze letras, dígitos, sublinhados, ou pontos pode ser feita com `[\w.]{3,12}`. Utilize `.{8,}` para corresponder a uma senha com pelo menos oito caracteres.

Exemplo 12.3.20

Para endereços de e-mail, `\S+@\S+` é uma expressão estendida comumente utilizada.[14]

Exemplo 12.3.21

Corresponde ao texto dentro de um único conjunto de parênteses a regex `\([^()]*\)`.

Exemplo 12.3.22

Isto corresponde a uma URL, um endereço web tal como `http://joshua.smcvt.edu/computing` . Esta regex é mais intrincada do que as anteriores, por isso merece alguma explicação. Baseia-se na quebra de URLs em três partes: um esquema como http seguido de dois pontos e duas barras para a frente, um servidor como joshua.smcvt.edu, e um caminho como /computing (o padrão também permite uma cadeia de consulta que segue uma interrogação mas este regex não lida com esses).

```
(https?|ftp)://([^\s/?\.#]+\.?){1,4}(/[^\s]*)?
```

Note o `https?`, para que o esquema possa ser http ou https, bem como ftp. Depois de dois pontos e duas barras para a frente vem a parte do servidor, constituída por alguns campos separados por pontos. Permitimos quase todos os caracteres nesses campos, exceto um espaço, uma interrogação, um ponto ou um jogo da velha. No final, vem um caminho. A especificação permite que os caminhos sejam sensíveis a maiúsculas e minúsculas, mas o regex aqui tem apenas minúsculas.

Mas espere! há mais! Você também pode fazer corresponder o início e o fim de uma linha com os metacaracteres '^' e o sinal de dólar '$'.

Exemplo 12.3.23

14 Isto é ingênuo na medida em que existem regras bem mais elaboradas para a sintaxe dos endereços de correio eletrônico. Além do fato de que um endereço sintaticamente correto pode simplesmente não existir. Mas é uma verificação de sanidade razoável.

Para correspondência com linhas começando por “Teorema” use `^Teorema`. Linhas terminando com “end{equation*}” poder ser correspondidas usando `end{equation\*}$`.

Os motores regex em linguagens modernas permitem-lhe especificar que a correspondência é insensível a maiúsculas e minúsculas (embora sejam diferentes quanto à sintaxe para tal).

Exemplo 12.3.24

Uma tag de documento HTML para uma imagem, tal como `<img src="logo.jpg">`, utiliza uma das chaves `src` ou `img` para fornecer o nome do arquivo que contém a imagem que será servida. Essas cadeias de caracteres podem ser em maiúsculas ou minúsculas, ou qualquer mistura. A linguagem de programação Racket utiliza uma sintaxe '?i:' para marcar parte do regex como insensível à capitalização: `\\s+(?i:(img|src))=` (notar também a dupla barra invertida, que é como Racket escapa o 's').

Para além da conveniência. Os motores de expressões regulares que vêm com linguagens de programação recentes têm capacidades para além de corresponderem apenas àquelas linguagens reconhecidas por máquinas de estados finitos.

Exemplo 12.3.25

A linguagem HTML de documentos da web utiliza tags tais como texto em negrito e texto italicizado. Fazer a correspondência para qualquer uma é simples, por exemplo `<b>[^<]*</b>`. Mas para uma única expressão que corresponda a todas elas parece que temos de fazer cada uma como um caso separado e depois combinar os casos com uma barra vertical. No entanto, em vez disso, podemos fazer com que o sistema se lembre do que encontra no início e procure isso novamente no final. Assim, o regex de Racket `<([^>]+)>[^<]*</\\1>` corresponde a tags HTML como as dadas. O seu segundo carácter é um abre parêntese, e o `\\1` refere-se a tudo o que se encontra entre esse parêntese aberto e o parêntese fechado correspondente. (Como se pode adivinhar pelo 1, também se pode ter uma segunda correspondência com `\\2`, etc.)

Isto é uma **referência para trás**. É muito conveniente. Contudo, essas extensões dão mais poder às expressões regulares estendidas do que às expressões regulares teóricas que estudamos anteriormente.

Exemplo 12.3.26

Esta é a linguagem dos **quadrados** sobre $\Sigma = \{$ `a`, `b` $\}$.

$$\mathcal{L} = \{\sigma \in \Sigma^* \mid \sigma = \tau \hat{} \tau \text{ para algum } \tau \in \Sigma^* \}$$

Alguns membros são `aabaab`, `baaabaaa`, e `aa`. A linguagem das quadrados não é regular[15]; portanto não pode ser descrita nem por um autômato finito, nem por uma expressão regular como vimos na teoria. No entanto, com expressões regulares estendidas, poderemos descrevê-la simplesmente como (.+)\1 , note a referência para trás.

15 Não demonstramos aqui.

Lado negativo. As expressões regulares são ferramentas poderosas, e ainda mais as regexes estendidas. Como ilustrado pelos exemplos acima, algumas das suas utilizações são: para validar nomes de usuário, pesquisar arquivos de texto, e filtrar resultados. Mas também elas podem trazer custos.

Por exemplo, a expressão regular para horas no formato de 12 horas `(ε|0|1)\d:\d\d\s(am|pm)` corresponde corretamente a '8:05 am' e '10:15 pm', mas fica aquém em alguns aspectos. Uma é que requer am ou pm no final, mas muitas vezes são dadas horas sem elas. Poderíamos mudar o final para `(ε |\s am|\s pm)`, o que é um pouco mais complexo mas resolve o problema.

Outra questão é que ela também corresponde a algumas cadeias que não desejamos, tais como as `13:00 am` ou `9:61 pm`. Podemos resolver isto como no parágrafo anterior, listando os casos.[16]

```
(01|02|...|11|12):(01|02|...|59|60)(\s am|\s pm)
```

Isto é como a solução do exemplo anterior, na medida em que resolve de fato a questão mas a um custo de complexidade, uma vez que equivale a uma lista das substrings permitidas.

Outro exemplo é que nem todas as strings que correspondem à expressão regular do CEP no Exemplo 12.3.14 têm um endereço correspondente. Atualmente existem menos de 1,5 milhão de CEPs válidos, sendo que há 10^8 strings que correspondem à regex. Portanto, a maioria das possibilidades de CEP não estão sendo utilizadas. Alterar a expressão regular para abranger apenas os códigos realmente em uso faria dela quase nada além de uma lista de cadeias (e que mudaria frequentemente).

Neste momento, as expressões regulares podem estar começando a parecer um pouco menos como um solucionador de problemas rápido e elegante e um pouco mais com um potencial problema de desenvolvimento e manutenção. A história completa é que por vezes uma expressão regular é exatamente o que se precisa para um trabalho rápido, e por vezes são boas também para tarefas mais complexas. Mas, em parte, o custo da complexidade supera o ganho em expressividade. Essa dicotomia entre poder e complexidade é frequentemente abordada online, citando este texto de Jamie Zawinski[17].

> A noção de que as regexps são a solução para todos os problemas é ... sem noção... . Algumas pessoas, quando confrontadas com um problema, pensam “Já sei, vou usar expressões regulares”. Agora elas têm dois problemas.

12.4. Créditos

Todas as seções, foram adaptadas (traduzidas e modificadas) de [1], que está disponível sob a licença Creative Commons Attribution-ShareAlike 4.0 (CC BY-SA 4.0).

16 As reticências correspondem a trechos da string que foram omitidos para deixar a string menor nesta apresentação.

17 Um dos programadores pioneiros do primeiro navegador internet de sucesso comercial, o Netscape Navigator.

12.5. Referências

1. Hefferon, Jim. *Theory of Computation: Making Connections*. Disponível em http://hefferon.net/computation/

12.6. Licença