Teoria da Computação

Exercícios Resolvidos

Daniel Castro Silva [1]

Setembro de 2016 (v2.0)

[1]Departamento de Engenharia Informática, Faculdade de Engenharia da Universidade do Porto / Laboratório de Inteligência Artificial e Ciência dos Computadores - dcs@fe.up.pt

Prefácio

Este livro tem como objetivo ser um auxiliar no estudo de Teoria da Computação[1], tal como lecionada na FEUP, apresentando exercícios resolvidos relativos às diversas temáticas abordadas. As soluções apresentadas não devem ser vistas como soluções únicas para os problemas, mas sim como possíveis resoluções dos mesmos.

Vários dos exercícios aqui resolvidos foram recolhidos de diversas fontes bibliográficas, sendo aqui dados os devidos créditos aos seus autores originais.

Bibliografia recomendada:

[Hopcroft et al., 2013] Hopcroft, J. H., Motwani, R., and Ullman, J. D. (2013). Introduction to Automata Theory, Languages, and Computation. Pearson, 3rd edition. ISBN: 978-1292039053.

[Linz, 2011] Linz, P. (2011). An Introduction to Formal Languages and Automata. Jones Bartlett Learning, 5th edition. ISBN: 978-1449615529.

[Sipser, 2013] Sipser, M. (2013). Introduction to the Theory of Computation. Cengage Learning, 3rd edition. ISBN: 978-1-133-18779-0.

[1] Os conteúdos aqui presentes não constituem de forma alguma fonte de estudo única ou suficiente para uma disciplina de Teoria de Computação.

Estrutura do Livro

Este livro está estruturado de forma idêntica à unidade curricular de Teoria da Computação do MIEIC da FEUP. Cada capítulo foca numa temática, apresentando um conjunto de exercícios resolvidos, e propondo também alguns exercícios para resolução.

No primeiro capítulo são abordados alguns métodos de prova, com foco na prova por indução, que será útil mais tarde. Nos capítulos 2 a 5 são abordadas as diferentes representações para Linguagens Regulares: autómatos finitos (deterministas e não deterministas), e expressões regulares, abordando-se no capítulo 6 as propriedades deste tipo de linguagens.

Nos capítulos 7 e 8 são abordadas duas representações para Linguagens Livres de Contexto - Gramáticas Livres de Contexto e Autómatos de Pilha - sendo no capítulo 9 abordadas as propriedades deste tipo de linguagens.

Finalmente, no capítulo 10 são abordadas Máquinas de Turing.

Capítulo 1

Introdução

Esta introdução aborda a temática de provas formais. Existem vários métodos de prova que podem ser usados, dependendo da hipótese a provar, ou refutar. Vamos aqui focar nas provas por indução e contradição, que são provavelmente as mais importantes em Teoria da Computação. Outros tipos de provas devem ser recordadas da unidade curricular Matemática Discreta, ou consultando fontes de informação auxiliares.

1.1 Prova por Contradição

Uma prova por contradição prova que o oposto daquilo que se quer provar é falso. Para isso, começamos por assumir o contrário do que queremos provar, e a partir daí tentamos encontrar uma contradição. Essa contradição implica que a assunção inicial (o contrário do que queríamos provar) estava errada, e portanto temos a nossa prova de que o que queríamos provar inicialmente é verdadeiro. De uma forma mais formal, supondo que queremos provar uma proposição P, começamos por assumir $\neg P$ como sendo verdadeiro. Usando premissas que sabemos serem verdadeiras é possível chegar a uma contradição $A \wedge \neg A$. Esta contradição implica que alguma das premissas usadas está errada, e sendo $\neg P$ a única premissa cuja veracidade era desconhecida, fica provado que $\neg P$ é falso, e consequentemente P é verdadeiro.

Exercício 1

Prove que um inteiro n é par se e apenas se n^2 for par (ou, de uma forma um pouco mais formal, $n = 2m \leftrightarrow n^2 = 2p$).

Este exercício introduz também o conceito de 'se e apenas se' (muitas vezes abreviado de iff, do inglês *if and only if*), que implica que na realidade é necessário realizar duas provas (ou seja, provar nos dois sentidos): provar que se n for par então n^2 também é par; e provar que se n^2 for par então n também é par.

Parte 1: $n = 2m \rightarrow n^2 = 2p$ (se um inteiro n for par, n^2 também é par)

Esta prova é simples, e requer apenas uma prova algébrica direta. Sabemos que um número par pode ser representado por $2m$.

Ora, $n^2 = (2m)^2 = 4m^2 = 2\,(2m^2)$. E está então provado que o quadrado de um número par é também um número par.

Parte 2: $n = 2m \leftarrow n^2 = 2p$ (se um quadrado perfeito n^2 for par, então n também é par)

Esta prova torna-se mais simples de realizar usando uma prova por contradição. Começamos então por assumir que n será ímpar, o que significa que pode ser escrito como $2m + 1$. Ora, se assim fosse, n^2 poderia ser escrito como $(2m + 1)^2$, o que resultaria em $4m^2 + 4m + 1 = 2\,(2m^2 + 2m) + 1$, que é também um número ímpar. Isto contradiz o pressuposto de que n^2 é par. Assim, fica provado que n não pode ser ímpar, e é portanto par[1].

Exercício 2

Na ilha de *Logos* existem dois tipos de habitantes: os *Aleteus*, que dizem sempre a verdade; e os *Pseudos*, que dizem sempre a mentira. Prove que a frase 'Eu sou mentiroso' não pode ser dita por um habitante de *Logos*.

Esta prova pode ser feita recorrendo a uma prova por contradição. Começamos então por assumir o contrário do que queremos provar: a frase foi dita por um habitante de *Logos*. Temos então duas hipóteses: ou se trata de um *Aleteu*, ou de um *Pseudo*. Vamos analisar cada um destes possíveis casos:

Caso 1: No caso de se tratar de um *Aleteu*, sabemos que ele diz sempre a verdade, pelo que a frase 'Eu sou mentiroso' é necessariamente uma falsidade, e portanto contraditória com a sua natureza, o que significa que

[1] Estamos a assumir alguns factos, como o facto de os números inteiros pares e ímpares seres mutuamente exclusivos, que deveriam também ser provados numa prova formal mais completa

chegámos a uma contradição.

Caso 2: No caso de se tratar de um *Pseudo*, sabemos que ele diz sempre a mentira, pelo que a frase 'Eu sou mentiroso' seria verdadeira, e portanto contraditória com a sua natureza, o que significa que chegámos também a uma contradição.

Como existem apenas dois tipos de habitantes em *Logos*, e em ambos os casos chegámos a uma contradição, então podemos concluir que a premissa inicial de que a frase 'Eu sou mentiroso' foi dita por um habitante de *Logos* é falsa, e portanto podemos concluir que a frase não foi dita por um habitante daquela ilha.

1.2 Prova Indutiva

A prova indutiva é das mais importantes em teoria da computação, uma vez que muitas das estruturas usadas possuem uma definição indutiva. Para realizar uma prova indutiva é necessário provar a hipótese a testar para um ou mais casos base e fazer depois a prova para o passo indutivo, isto é, provar que se a hipótese for assumida como verdadeira para um elemento, ela mantém-se verdadeira para o elemento seguinte da estrutura indutiva. Para sistematização, vamos apresentar sempre a prova indutiva em 4 pontos: estrutura de indução (qual a estrutura em que a prova será realizada; como se obtém o elemento seguinte a partir do atual); hipótese (o que queremos provar); caso base (prova para um ou mais casos de base); e passo indutivo (prova para o passo indutivo).

Exercício 1

Prove que $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a 0. O elemento base é o zero, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
- Caso Base. Para $n = 0$, podemos calcular $\sum_{i=0}^{0} 2^i = 2^0 = 1$. Do outro lado, temos $2^{0+1} - 1 = 2^1 - 1 = 1$. Obtemos uma igualdade, e a hipótese fica então provada para o caso base.

- Passo Indutivo. Vamos agora provar para qualquer inteiro positivo. Então, vamos assumir a hipótese como verdadeira e tentar fazer a prova para o passo indutivo. Mais formalmente, tentamos provar que

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1 \rightarrow \sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1$$

.

Desenvolvendo, temos que

$$\sum_{i=0}^{n+1} 2^i = 2^{n+2} - 1$$

$$\sum_{i=0}^{n} 2^i + 2^{n+1} = 2^{n+2} - 1$$

$$2^{n+1} - 1 + 2^{n+1} = 2^{n+2} - 1$$

$$2 \times 2^{n+1} - 1 = 2^{n+2} - 1$$

$$2^{n+2} - 1 = 2^{n+2} - 1$$

E obtemos então a igualdade que prova que a hipótese também se mantém para o passo indutivo, ficando assim provado que a hipótese é verdadeira.

Exercício 2 (TPC 2012/13)

Prove que para qualquer número inteiro maior ou igual a zero,

$$\sum_{i=0}^{n} i^2 = \frac{n\,(n+1)\,(2n+1)}{6}$$

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a zero. O elemento base é 0, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
- Caso Base. Para $n = 0$, podemos calcular $\sum_{i=0}^{0} i^2 = 0^2 = 0$. Do outro lado da igualdade, temos que $\frac{n(n+1)(2n+1)}{6} = \frac{0(0+1)(2\times 0+1)}{6} = 0$. Verifica-se então a hipótese para o caso base.

- Passo Indutivo. Está provado para n = 0. Agora, temos de provar para qualquer número maior do que zero. Então, temos a seguinte pergunta: Se assumirmos que a hipótese é válida para n, será válida também para $n + 1$? Dito de uma forma mais formal:
  $\sum_{i=0}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \rightarrow \sum_{i=0}^{n+1} i^2 = \frac{(n+1)(n+2)(2(n+1)+1)}{6}$
  Assumimos então a premissa como verdadeira e tentamos desenvolver a segunda parte de forma a usar a primeira equivalência e encontrar uma igualdade.

$$\sum_{i=0}^{n+1} i^2 = \frac{(n+1)(n+2)(2(n+1)+1)}{6}$$

$$\sum_{i=0}^{n} i^2 + (n+1)^2 = \frac{(n+1)(n+2)(2(n+1)+1)}{6}$$

$$\frac{n(n+1)(2n+1)}{6} + (n+1)^2 = \frac{(n+1)(n+2)(2(n+1)+1)}{6}$$

$$\frac{n(n+1)(2n+1)}{6} + \frac{6(n+1)^2}{6} = \frac{(n+1)(n+2)(2n+3)}{6}$$

$$\frac{n(n+1)(2n+1) + 6(n+1)(n+1)}{6} = \frac{(n+1)(2n^2+3n+4n+6)}{6}$$

$$\frac{(n+1)[n(2n+1) + 6(n+1)]}{6} = \frac{(n+1)(2n^2+7n+6)}{6}$$

$$(n+1)[n(2n+1) + 6(n+1)] = (n+1)\left(2n^2+7n+6\right)$$

$$(n+1)\left(2n^2+n+6n+6\right) = (n+1)\left(2n^2+7n+6\right)$$

$$2n^2+7n+6 = 2n^2+7n+6$$

Chegamos assim a uma equivalência, o que significa que a prova para o passo indutivo foi realizada com sucesso, e assim toda a prova indutiva foi concluída, podendo-se afirmar que a hipótese é verdadeira.

Exercício 3 (Desafio 2014/15)

Prove que $n\left(n^2+5\right)$ é divisível por 6, para qualquer número inteiro $n \geq 0$

- Estrutura de Indução. A estrutura é a dos números inteiros positivos mais o zero. O elemento base é 0, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $n\left(n^2+5\right)$ é divisível por 6, ou seja, $n\left(n^2+5\right) mod 6 = 0$, ou dito de outra forma, $n\left(n^2+5\right) = 6k$
- Caso Base. Para $n = 0$, podemos calcular $n\left(n^2+5\right) = 0\left(0^2+5\right) = 0$. Como zero é divisível por 6 $(0 = 6 \times 0)$, está provado para o caso base.
- Passo Indutivo. Para fazer a prova para o passo indutivo, vamos então assumir que a hipótese é válida para n, e tentar provar se será também válida para $n+1$. Dito de uma forma mais formal:

  $n\left(n^2+5\right) = 6k \rightarrow (n+1)\left((n+1)^2+5\right) = 6j$

  Então, tentamos desenvolver uma das partes, de forma a chegar a uma equivalência verdadeira. Começando por desenvolver a parte direita:

  $(n+1)\left(n^2+2n+1+5\right) = 6j$

  $n\left(n^2+2n+1+5\right) + \left(n^2+2n+1+5\right) = 6j$

  $n\left(n^2+5\right) + n\left(2n+1\right) + \left(n^2+2n+1+5\right) = 6j$

  $6k + n\left(2n+1\right) + \left(n^2+2n+1+5\right) = 6j$

  $6k + 2n^2 + n + n^2 + 2n + 1 + 5 = 6j$

  $6k + 3n^2 + 3n + 6 = 6j$

  $6\left(k+1\right) + n\left(3n+3\right) = 6j$

  $n\left(3n+3\right) = 6j - 6\left(k+1\right)$

  $3n\left(n+1\right) = 6\left(j-k+1\right)$

  Resta-nos agora provar que $3n\left(n+1\right)$ é também divisível por 6. Para isso, podemos optar por realizar nova prova indutiva para esta afirmação, ou então provar que $n\left(n+1\right)$ é um número par (e portanto possível de reescrever como $2m$, o que permitiria reescrever o termo como $3 \times 2m = 6m$, que é divisível por 6). Vamos fazer a prova para esta segunda opção.

  - Estrutura de Indução. Números inteiros positivos mais o zero. O elemento base é 0, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.

- Hipótese. $n\,(n+1) = 2a$
- Caso Base. Para $n = 0$, podemos calcular $n\,(n+1) = 0\,(0+1) = 0$. Como zero é divisível por 2 $(0 = 2 \times 0)$, está provado para o caso base.
- Passo Indutivo. Está provado para n = 0. Agora, temos de provar para qualquer número maior que zero. Novamente, se assumirmos que a hipótese é válida para n, será válida também para n+1? Dito de uma forma mais formal: $n\,(n+1) = 2a \rightarrow (n+1)\,(n+2) = 2b$

  Desenvolvendo então a parte da direita da implicação:

  $(n+1)\,n + (n+1)\,2 = 2b$

  $2a + (n+1)\,2 = 2b$

  $2\,(a+n+1) = 2b$

Podemos agora dizer que $n\,(n+1)$ é um número par, pelo que podemos reescrever o termo $3n\,(n+1)$ como $3 \times 2m$, ou $6m$, e ficamos então com:

$6m = 6\,(j-k+1)$

E provamos assim que $n\,(n^2+5)$ é divisível por 6.

Uma outra forma de resolver este exercício passa pelo uso do Paradoxo do Inventor, encontrando uma afirmação mais forte. Observando a sequência de valores gerados para valores crescentes de n, conseguimos encontrar uma nova expressão a provar:

$$n\left(n^2+5\right) = 6 \times \sum_{i=0}^{n-1} \left(\frac{i(i+1)}{2} + 1\right)$$

Vamos então realizar esta prova:

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a zero. O elemento base é 0, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $n\,(n^2+5) = 6 \times \sum_{i=0}^{n-1} \left(\frac{i(i+1)}{2} + 1\right)$
- Caso Base. Para $n = 0$, temos do lado esquerdo da igualdade que $0\,(0^2+5) = 0$. Do lado direito da igualdade, temos $6 \times \sum_{i=0}^{0-1} \left(\frac{i(i+1)}{2} + 1\right) = 6 \times 0 = 0$. Chegamos a uma igualdade, pelo que podemos afirmar ter provado o caso base.

- Passo Indutivo. Para realizar a prova para o passo indutivo, assumimos novamente que a hipótese é válida para n, tentando provar para $n+1$. Dito de uma forma mais formal: $n\left(n^2+5\right) = 6 \times \sum_{i=0}^{n-1}\left(\frac{i(i+1)}{2}+1\right) \rightarrow (n+1)\left((n+1)^2+5\right) = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$

$$(n+1)\left((n+1)^2+5\right) = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$(n+1)\left(n^2+2n+6\right) = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$n\left(n^2+2n+6\right) + n^2+2n+6 = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$n(n^2+5) + 2n^2+n+n^2+2n+6 = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$6 \times \sum_{i=0}^{n-1}\left(\frac{i(i+1)}{2}+1\right) + 3n^2+3n+6 = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$6 \times \sum_{i=0}^{n-1}\left(\frac{i(i+1)}{2}+1\right) + 6\left(\frac{n^2+n}{2}+1\right) = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$6 \times \sum_{i=0}^{n-1}\left(\frac{i(i+1)}{2}+1\right) + 6\left(\frac{n(n+1)}{2}+1\right) = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$6\left(\sum_{i=0}^{n-1}\left(\frac{i(i+1)}{2}+1\right) + \frac{n(n+1)}{2}+1\right) = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

$$6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right) = 6 \times \sum_{i=0}^{n}\left(\frac{i(i+1)}{2}+1\right)$$

E com esta equivalência fica assim provada a hipótese.

Exercício 4

Vamos agora ver um exemplo usando grafos como estrutura. Prove que num grafo completo $E = \frac{N(N-1)}{2}$, em que E é o número de arestas e N o número de nós do grafo.

- Estrutura de Indução. A estrutura é a dos grafos completos. O elemento base é grafo com apenas um nó, e nenhuma aresta. Os elementos seguintes são obtidos acrescentando um novo nó a um grafo existente, e as arestas necessárias para ligar a cada um dos nós desse grafo.

- Hipótese. $E = \frac{N(N-1)}{2}$

- Caso Base. Para o grafo simples, com apenas um nó (e nenhuma aresta), podemos calcular $\frac{N(N-1)}{2} = \frac{1(1-1)}{2} = 0$, ficando provada a hipótese para o caso base.

- Passo Indutivo. Vamos agora tentar fazer a prova para um grafo completo com um qualquer número de nós. Vamos assumir que para o grafo com N nós temos $E_N = \frac{N(N-1)}{2}$, e tentar verificar se acrescentando um nó ficamos com $E_{N+1} = \frac{(N+1)(N+1-1)}{2}$.

  Como sabemos, ao acrescentar um novo nó a um grafo com N nós, são necessárias mais N arestas, pelo que $E_{N+1} = E_N + N$.

  Assim, podemos substituir e ficamos com

$$E_N + N = \frac{(N+1)(N+1-1)}{2}$$

$$\frac{N(N-1)}{2} + N = \frac{(N+1)N}{2}$$

$$\frac{N(N-1)}{2} + \frac{2N}{2} = \frac{(N+1)N}{2}$$

$$\frac{N(N-1+2)}{2} = \frac{(N+1)N}{2}$$

$$\frac{N(N+1)}{2} = \frac{(N+1)N}{2}$$

E com esta equivalência fica assim provada a hipótese.

Exercício 5 (TPC 2010/11)

Prove por indução que, para todo o número natural n, $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a 1. O elemento base é 1, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $\sum_{i=0}^{n} i = \frac{n(n+1)}{2}$
- Caso Base. Para $n = 1$, podemos calcular $\sum_{i=0}^{n} i = 0 + 1 = 1$. Do outro lado da igualdade, temos que $\frac{1(1+1)}{2} = \frac{1\times 2}{2} = 1$. Verifica-se então a hipótese para o caso base.
- Passo Indutivo. Está provado para n = 1. Agora, temos de provar para qualquer número maior do que um. Se assumirmos que a hipótese é válida para n, será válida também para $n+1$? Dito de uma forma mais formal:
  $\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \rightarrow \sum_{i=0}^{n+1} i = \frac{(n+1)(n+2)}{2}$
  Assumimos então a premissa como verdadeira e tentamos desenvolver a segunda parte de forma a usar a primeira equivalência e encontrar uma igualdade.

$$\sum_{i=0}^{n+1} i = \frac{(n+1)(n+2)}{2}$$

$$\sum_{i=0}^{n} i + n + 1 = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1)}{2} + n + 1 = \frac{(n+1)(n+2)}{2}$$

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

$$\frac{(n+2)(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

Chegamos novamente a uma equivalência, o que significa que a prova para o passo indutivo foi realizada com sucesso, e podemos afirmar que a hipótese é verdadeira.

Exercício 6 (TPC 2011/12)

Prove por indução a seguinte igualdade, em que h é um número inteiro maior ou igual a 1: $\sum_{i=0}^{h-1} 2^i = 2^h - 1$

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a 1. O elemento base é 1, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $\sum_{i=0}^{h-1} 2^i = 2^h - 1$
- Caso Base. Para $h = 1$, podemos calcular $\sum_{i=0}^{1-1} 2^i = 2^0 = 1$. Do outro lado da igualdade, temos que $= 2^1 - 1 = 2 - 1 = 1$. Verifica-se então a hipótese para o caso base.
- Passo Indutivo. Está provado para h = 1. Agora, temos de provar para qualquer número maior do que um. Se assumirmos que a hipótese é válida para h, será válida também para $h + 1$? Mais formalmente:
  $\sum_{i=0}^{h-1} 2^i = 2^h - 1 \rightarrow \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$
  Assumimos então a premissa como verdadeira e tentamos desenvolver a segunda parte, usando a primeira equivalência até encontrar uma igualdade.

$$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$\sum_{i=0}^{h-1} 2^i + 2^h = 2^{h+1} - 1$$

$$2^h - 1 + 2^h = 2^{h+1} - 1$$

$$2 \times 2^h - 1 = 2^{h+1} - 1$$

$$2^{h+1} - 1 = 2^{h+1} - 1$$

Chegamos a uma equivalência, o que significa que a prova para o passo indutivo foi realizada com sucesso, e a hipótese é então verdadeira.

Exercício 7 (TPC 2013/14)

Prove por indução que a soma dos primeiros n cubos (ou seja, $1^3+2^3+...+n^3$) é dada por $\left(\frac{n(n+1)}{2}\right)^2$

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a 1. O elemento base é 1, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2$
- Caso Base. Para $n = 1$, podemos calcular $\sum_{i=1}^{1} i^3 = 1^3 = 1$. Do outro lado da igualdade, temos que $\left(\frac{1(1+1)}{2}\right)^2 = 1^2 = 1$. Verifica-se então a hipótese para o caso base.
- Passo Indutivo. Está provado para n = 1. Agora, temos de provar para qualquer número maior do que um. Se assumirmos que a hipótese é válida para n, será válida também para $n + 1$? Mais formalmente:

  $\sum_{i=1}^{n} i^3 = \left(\frac{n(n+1)}{2}\right)^2 \rightarrow \sum_{i=1}^{n+1} i^3 = \left(\frac{(n+1)(n+2)}{2}\right)^2$

  Assumimos então a premissa como verdadeira e tentamos desenvolver a segunda parte, usando a primeira equivalência até encontrar uma igualdade.

$$\sum_{i=1}^{n+1} i^3 = \left(\frac{(n+1)(n+2)}{2}\right)^2$$

$$\sum_{i=1}^{n} i^3 + (n+1)^3 = \frac{(n+1)^2(n+2)^2}{4}$$

$$\left(\frac{n(n+1)}{2}\right)^2 + (n+1)^3 = \frac{(n+1)^2(n+2)^2}{4}$$

$$\frac{n^2(n+1)^2}{4} + (n+1)^3 = \frac{(n+1)^2(n+2)^2}{4}$$

$$\frac{n^2(n+1)^2 + 4(n+1)^3}{4} = \frac{(n+1)^2(n+2)^2}{4}$$

$$\frac{n^2(n+1)^2 + 4(n+1)(n+1)^2}{4} = \frac{(n+1)^2\,(n+2)^2}{4}$$

$$\frac{(n^2 + 4(n+1))(n+1)^2}{4} = \frac{(n+1)^2\,(n+2)^2}{4}$$

$$\frac{(n^2 + 4n + 4)(n+1)^2}{4} = \frac{(n+1)^2\,(n+2)^2}{4}$$

$$\frac{(n+2)^2(n+1)^2}{4} = \frac{(n+1)^2\,(n+2)^2}{4}$$

Chegamos a uma equivalência, o que significa que a prova para o passo indutivo foi realizada com sucesso, e a hipótese é então verdadeira.

Exercício 8 (Exercício 2015/16)

Prove por indução a seguinte igualdade para $n \geq 1$: $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a 1. O elemento base é 1, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.
- Hipótese. $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1}$
- Caso Base. Para $n = 1$, podemos calcular $\sum_{i=1}^{1} \frac{1}{i(i+1)} = \frac{1}{1(1+1)} = \frac{1}{2}$. Do outro lado da igualdade, temos que $\frac{1}{1+1} = \frac{1}{2}$. Verifica-se então a hipótese para o caso base.
- Passo Indutivo. Está provado para n = 1. Agora, temos de provar para qualquer número maior do que um. Se assumirmos que a hipótese é válida para n, será válida também para $n + 1$? Mais formalmente:
  $\sum_{i=1}^{n} \frac{1}{i(i+1)} = \frac{n}{n+1} \rightarrow \sum_{i=1}^{n+1} \frac{1}{i(i+1)} = \frac{n+1}{n+2}$
  Novamente, assumimos a premissa como sendo verdadeira e tentamos desenvolver a segunda parte, usando a primeira equivalência na tentativa de encontrar uma igualdade.

$$\sum_{i=1}^{n+1} \frac{1}{i(i+1)} = \frac{n+1}{n+2}$$

$$\sum_{i=1}^{n} \frac{1}{i(i+1)} + \frac{1}{(n+1)(n+2)} = \frac{n+1}{n+2}$$

$$\frac{n}{n+1} + \frac{1}{(n+1)(n+2)} = \frac{n+1}{n+2}$$

$$\frac{n(n+2)+1}{(n+1)(n+2)} = \frac{(n+1)(n+1)}{(n+2)(n+1)}$$

$$\frac{n^2+2n+1}{(n+1)(n+2)} = \frac{(n+1)^2}{(n+2)(n+1)}$$

$$\frac{(n+1)^2}{(n+1)(n+2)} = \frac{(n+1)^2}{(n+2)(n+1)}$$

Chegamos a uma equivalência, o que significa que a prova para o passo indutivo foi realizada com sucesso, e a hipótese é então verdadeira.

Exercício 9 (Desafio 2015/16)

Prove que $n^3 - n$ é divisível por 3 para qualquer número inteiro positivo (ou seja, $n^3 - n = 3k$).

- Estrutura de Indução. A estrutura é a dos números inteiros maiores ou igual a 1. O elemento base é 1, e os elementos seguintes são obtidos somando 1 unidade ao elemento anterior.

- Hipótese. $n^3 - n = 3k$

- Caso Base. Para $n = 1$, podemos calcular $n^3 - n = 1 - 1 = 0$. Do outro lado da igualdade, para $k = 0$, temos que $3 \times 0 = 0$. Verifica-se então a hipótese para o caso base.

- Passo Indutivo. Está provado para n = 1. Agora, temos de provar para qualquer número maior do que um. Se assumirmos que a hipótese é válida para n, será válida também para $n + 1$? Mais formalmente:

  $n^3 - n = 3k \rightarrow (n+1)^3 - (n+1) = 3j$

  Novamente, assumimos a premissa como sendo verdadeira e tentamos desenvolver a segunda parte, usando a primeira equivalência na tentativa de encontrar uma igualdade.

$$(n+1)^3 - (n+1) = 3j$$

$$(n+1)(n^2 + 2n + 1) - n - 1 = 3j$$

$$n^3 + 2n^2 + n + n^2 + 2n + 1 - n - 1 = 3j$$

$$n^3 - n + 3n^2 + 3n = 3j$$

$$3k + 3n^2 + 3n = 3j$$

$$3(k + n^2 + n) = 3j$$

Como chegamos a um número que é múltiplo de 3 do lado esquerdo da equação, podemos afirmar que a hipótese é verdadeira. Poderíamos no entanto usar o paradoxo do inventor para tornar a prova mais forte, percebendo o fator de crescimento e provando que $n^3 - n = 3 \times \sum_{i=0}^{n-1}(i^2 + i)$. Tente fazer esta prova.

Exercício Proposto

a) Prove que $11^n - 6$ é divisível por 5 para qualquer número inteiro positivo (ou seja, $11^n - 6 = 5k$).

b) Consegue descobrir uma prova mais forte para usar o paradoxo do inventor?

Capítulo 2

Autómatos Finitos Deterministas (DFA)

Este capítulo aborda Autómatos Finitos Deterministas (ou DFAs, do inglês *Deterministic Finite Automata*). Os DFAs são uma das possibilidades de especificação de linguagens regulares (as outras possibilidades, Autómatos Finitos Não Deterministas e Expressões Regulares são abordadas nos próximos capítulos).

As linguagens regulares, e os autómatos, são muito úteis nas mais diversas áreas da informática, seja para ajudar na especificação de protocolos de comunicação (especificação do estados dos intervenientes e das possíveis mensagens que cada interveniente pode enviar a cada momento; isto é algo abordado nas disciplinas de Redes de Computadores ou Sistemas Distribuídos); em Diagramas de Estados, usados para especificação de sistemas (isto é algo abordado na disciplina de Engenharia de Software, entre outras); ...

São também muito usados (normalmente por geração automática a partir de expressões regulares, algo que vamos ver também mais à frente) para pesquisa de strings (por exemplo, com o comando *grep* em Linux); para validação de entrada em formulários (por exemplo para validar o formato de endereços de correio eletrónico, ou de um URL; isto é algo abordado nas disciplinas de Linguagens e Tecnologias Web e também em Laboratório de Bases de Dados e Aplicações Web, entre outras); ou na fase de análise lexical de um compilador (isto é abordado na disciplina de Compiladores), entre muitos outros usos.

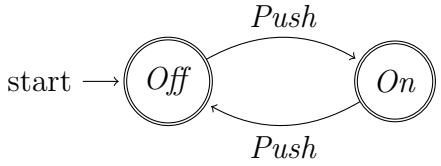
Definição

Um DFA representa um sistema que a qualquer momento pode estar num estado de vários possíveis. Vamos ver o exemplo muito simples (e clássico) de um interruptor, que pode estar ligado (On) ou desligado (Off), e que alterna consoante o interruptor é pressionado ($Push$).

Um DFA é definido formalmente como um tuplo $DFA = (Q, \Sigma, \delta, q_0, F)$, em que Q é o conjunto de estados do DFA, Σ é o conjunto de símbolos de entrada (alfabeto), δ representa a função de transição de estados e símbolos para estados ($\delta(q, a) = p$ denota uma transição do estado q para o estado p com o símbolo a), $q_0 \in Q$ representa o estado inicial, e $F \subseteq Q$ é o conjunto de estados finais.

$$Interruptor = (\{Off, On\}, \{Push\}, \{\delta(Off, Push) = On;$$
$$\delta(On, Push) = Off\}, Off, \{Off, On\})$$

Um DFA pode ser representado graficamente por um diagrama em que os nós representam os estados e as arestas as transições entre estados ($\delta(q, a) = p$ representado por um arco de q para p com etiqueta a); o estado inicial é denotado por uma seta de entrada sem origem etiquetada *Start*, e os nós reprentativos de estados finais possuem um duplo círculo.

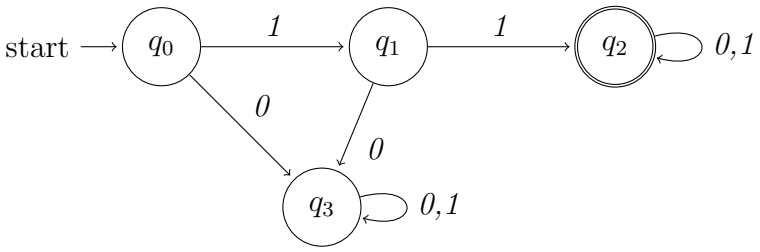


Um DFA pode ainda ser representado por uma tabela de transições, em que as linhas representam os estados, e as colunas os símbolos de entrada. Nas células de interseção entre estes encontra-se o estado de destino da transição respetiva. O estado inicial é indicado com uma seta e os estados finais com um asterisco.

	Push
→ * Off	On
* On	Off

Os DFAs são também usados no reconhecimento de linguagens, ou seja, permitem verificar se uma certa cadeia de caracteres pertence ou não a

uma determinada linguagem. Vejamos o exemplo da linguagem das cadeias binárias que começam por '11'. Ao construir o DFA é preciso garantir que conseguimos saber em que estado de reconhecimento este se encontra a cada momento: se ainda não reconheceu nenhum bit; se já reconheceu o primeiro 1, mas não o segundo; se já reconheceu os dois 1s (e portanto a cadeia é aceite); ou se reconheceu algo que não deveria estar ali, e portanto faz com que a cadeia não pertença à linguagem. Vamos ver a resolução em formato gráfico, para facilitar a interpretação.

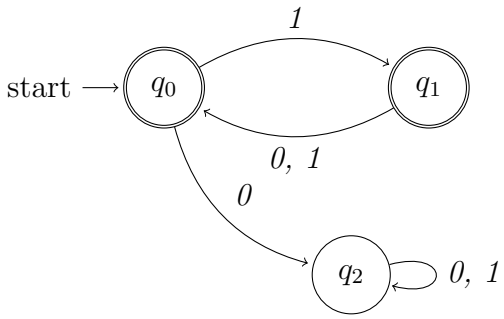


Temos então o DFA acima como solução para o problema de reconhecer cadeias binárias que começam por '11'. Se a cadeia de entrada for '1101', o DFA transita de q_0 para q_1 com o primeiro 1, e daí para q_2 com o segundo 1; daí para a frente, permanece em q_2 tanto com 0 como com 1, uma vez que a condição de aceitação (começar com 11) já foi cumprida. O estado q_3 serve para que o DFA fique completo (isto é, possuir transição para todos os pares *(estado, símbolo de entrada)*), sendo considerado um 'estado morto': há transições para ele de todos os estados com símbolos não usados até então, e do estado morto para ele próprio com todos os símbolos de entrada (isto garante que o DFA não morre até terminar de processar a cadeia de entrada).

Exercício 1

Apresente um DFA que permita reconhecer cadeias sobre o alfabeto {0,1} em que todas as posições ímpares têm um 1.

A solução para este problema passa por reconhecer que as transições representam posições ímpares ou pares, e garantir que nas transições ímpares temos um 1 (nas transições pares, tanto 0 como 1 são aceites).

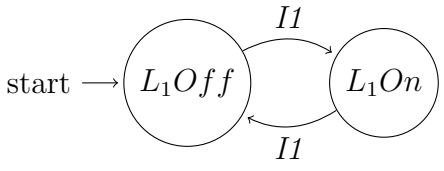


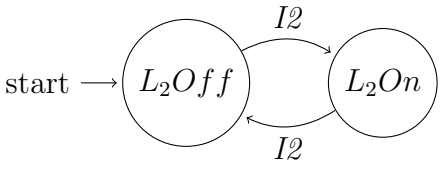
Se observarmos a solução acima, temos a transição de q_0 para q_1 a representar as posições ímpares e a de retorno as posições pares. Tanto q_0 como q_1 são estados finais, uma vez que em nenhum dos casos a regra (de posições ímpares terem 1) foi infrigida (note que a cadeia vazia, ε, faz também parte da linguagem). O estado q_2 é o estado morto que completa o DFA e trata dos dos casos em que uma posição ímpar é preenchida com 0.

Exercício 2

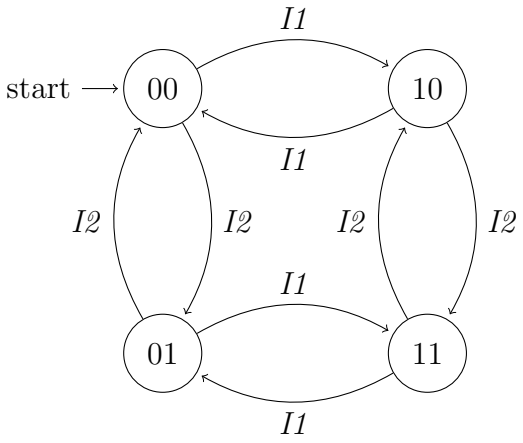
Apresente um DFA para modelar o estado das lâmpadas de uma sala com duas lâmpadas e respetivos interruptores.

Vamos considerar que cada lâmpada (L1 e L2) tem dois possíveis estados: ligada e desligada (*On* e *Off*); e que os dois interruptores (I1 e I2) afetam o estado das lâmpadas respetivas. Podemos modelar cada uma das lâmpadas de forma independente e depois fazer o produto dos dois DFAs daí resultantes, obtendo assim o DFA completo para as duas lâmpadas. Ficam assim os dois DFAs:





Para obter um único DFA é então necessário fazer o produto dos dois DFAs acima. Para isso, o novo DFA terá um estado para cada tuplo de estados dos DFAs anteriores (S_{DFA_1}, S_{DFA_2}) . Vamos representar cada novo estado como um número binário em que o primeiro bit representa o estado de L_1 (0 para Off e 1 para On), e o segundo bit representa o estado de L_2.

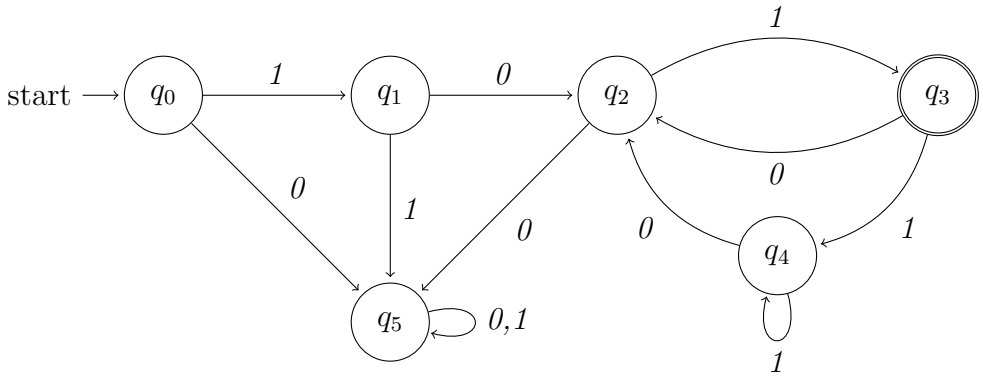


As transições devem ter em atenção quais as alterações que provocam nos estados dos dois DFAs. Neste caso, transições com $I1$ afetam apenas L_1, pelo que transições com $I1$ no autómato produto levam a estados sem alteração no 'sub-estado' de L_2.

Exercício 3 (Desafio 2014/15)

Modele um DFA que permita reconhecer cadeias binárias que iniciem em 10, terminem em 01, e não possuam nenhuma sequência com dois 0's. Ex.: 1001 não pertence à linguagem; 10101 pertence à linguagem.

Para resolver este exercício, podemos começar por tentar dividir o problema em 3 sub-problemas: início das cadeias, não possuir sequência com dois 0's consecutivos, e término das cadeias. Temos depois, para obter a solução final, de ver como integrar cada uma dessas partes, de forma a evitar estados desnecessários, e garantir que todas as cadeias pertencentes à linguagem (mesmo as mais curtas, como 101) são aceites.

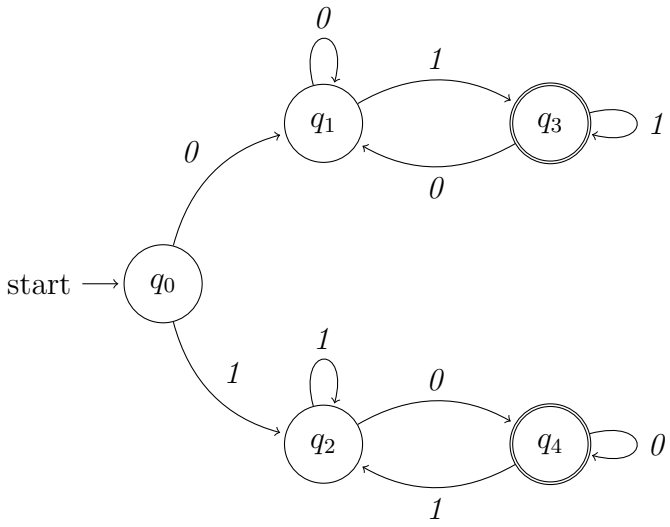


Analisando o autómato acima, podemos ver a parte inicial nos estados q_0, q_1 e q_2, em que temos a sequência '10'. Os estados q_1, q_2 e q_3 representam a parte final, com a sequência '01' para término, sendo o estado q_3 o estado de aceitação; Os estados q_1 e q_2 são partilhados entre as partes de início e de término de forma a garantir que a cadeia '101', que faz parte da linguagem, é também aceite pelo DFA. O estado q_4 e as transições com '0' de q_4 e de q_3 para q_2 garantem que não existem dois zeros consecutivos e também que podem existir vários 1's consecutivos. O estado q_5 é o estado morto, e existe apenas para tornar o DFA completo, garantindo que todos os estados possuem transições para todos os símbolos de entrada.

Exercício 4

Modele um DFA que permita reconhecer cadeias no alfabeto {0,1} com símbolos inicial e final diferentes.

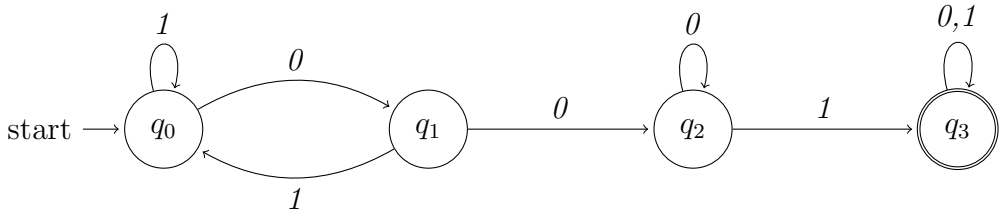
Precisamos aqui de distinguir os dois casos iniciais, e assim dividir o DFA em dois 'ramos', um para cada um dos possíveis símbolos iniciais, e que terá de garantir símbolo final diferente.



Os estados q_3 e q_4 são os estados de aceitação, uma vez que é para eles que transitam os símbolos contrários dos símbolos iniciais de cada um dos respetivos ramos. Quando temos uma transição com o mesmo símbolo que o símbolo inicial, os estados de destino (q_1 e q_2) deixam de ser estados finais.

Exercício 5

Considere o seguinte DFA. Apresente a definição formal e a representação em tabela para o DFA, e indique por palavras qual a linguagem representada. Indique ainda quais das cadeias '010101', '0100101' e '1010010' são aceites pelo DFA.



Para obter a definição formal e a representação tabular do DFA é apenas necessário interpretar o DFA e mudar o formato da representação.

DFA A = ($\{q_0, q_1, q_2, q_3\}$, $\{0, 1\}$, δ, q_0, $\{q_3\}$), em que δ representa as seguintes transições: $\delta(q_0, 0) = q_1$; $\delta(q_0, 1) = q_0$; $\delta(q_1, 0) = q_2$; $\delta(q_1, 1) = q_0$; $\delta(q_2, 0) = q_2$; $\delta(q_2, 1) = q_3$; $\delta(q_3, 0) = q_3$; $\delta(q_3, 1) = q_3$.

E a representação em tabela:

	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_2	q_0
q_2	q_2	q_3
$^* q_3$	q_3	q_3

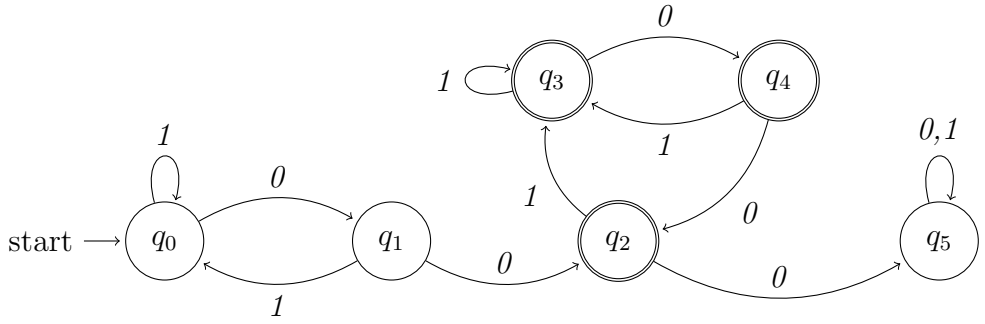
Em relação à linguagem aceite pelo autómato, temos de tentar ver as transições que levam ao estado final. Podemos ver que uma vez atingido o estado final não voltamos a um estado não-final, o que significa que queremos reconhecer a sequência de símbolos que levam até ele. Ao olhar para as transições, vemos que a sequência é 001: q_0 é o estado em que ainda não se reconheceu qualquer símbolo desta sequência; q_1 é o estado em que já se reconheceu o primeiro 0; q_2 é o estado em que já se reconheceram os dois 0s e finalmente o estado final quando toda a sequência foi reconhecida.

Em relação às cadeias aceites, podemos calcular a função de transição estendida $\delta\hat{}$ para cada uma das cadeias e verificar se é atingido o estado final. $\delta\hat{}(q_0, 010101) = q_0$; $\delta\hat{}(q_0, 0100101) = q_3$; $\delta\hat{}(q_0, 010010) = q_3$. Assim, apenas as duas últimas cadeias são aceites pelo DFA.

Exercício 6

Modele um DFA que permita reconhecer cadeias binárias que contenham a sequência 00, mas não a sequência 000.

Uma solução para este exercício passa por reconhecer que quando temos a sequência 000, há uma transição para o estado morto, e portanto a cadeia não será aceite. Por outro lado, um estado de aceitação só poderá ser atingido a partir do momento em que apareçam dois 0s consecutivos. Assim o DFA terá de manter nos seus estados informação de quantos 0s consecutivos foram lidos, e se já atingiu a condição de aceitação ou não. Podemos pensar nesta solução como uma espécie de contador de zeros, em que há um *reset* quando é lido um 1.



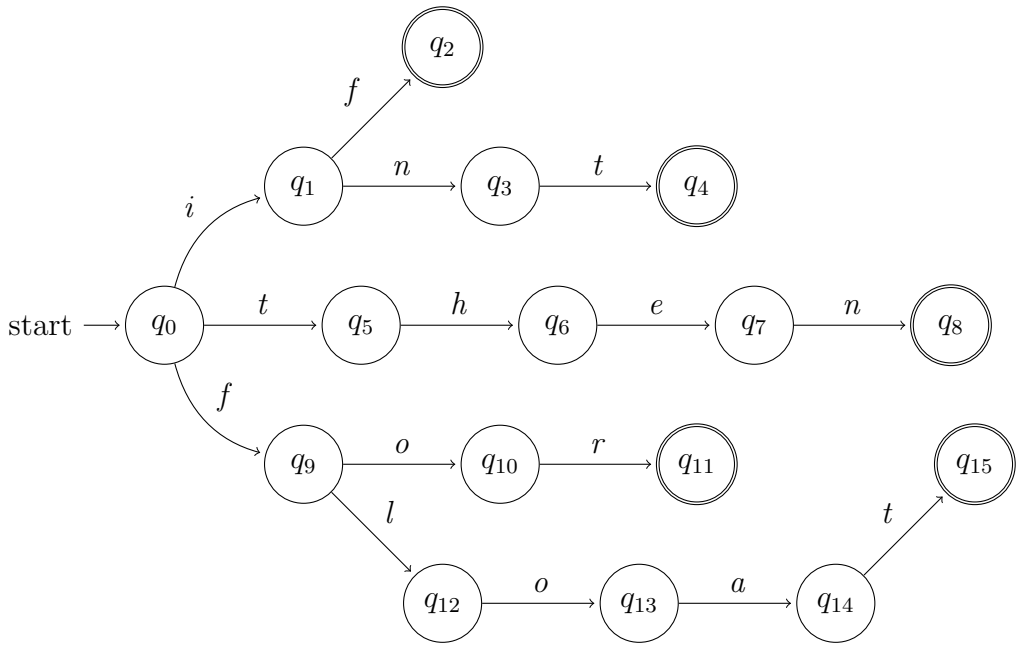
Olhando para o DFA acima, podemos ver que só atingimos um estado de aceitação quando há dois 0s consecutivos (passando de q_0 para q_1 e daí para q_2). Depois de atingir q_2, a condição de aceitação foi atingida, mas é necessário ainda garantir que a restrição dos três zeros consecutivos não é quebrada, e para isso necessitamos dos estados q_3 e q_4, que são uma 'cópia' dos estados q_0 e q_1, respetivamente, mas neste caso sendo estados de aceitação. O estado q_5 e a transição para ele a partir de q_2 (o estado que representa dois zeros consecutivos) garante que ao terceiro 0, o DFA não atinge nunca mais um estado de aceitação.

Exercício 7 (TPC 2010/11)

Desenhe um DFA que reconheça as cadeias de caracteres: *if*, *then*, *int*, *for*, e *float*. De seguida, represente o DFA usando a notação formal.

Partindo das palavras que se pretender ver reconhecidas, podemos definir o nosso alfabeto como contendo apenas esses caracteres. Assim, o alfabeto a

usar será $\Sigma = \{a, e, f, h, i, l, n, o, r, t\}$. Como temos palavras começadas pelo mesmo símbolo (por exemplo *if* e *int* começam ambos por i), temos de ter cuidado para que exista apenas uma transição possível em cada ponto. Uma possível solução será a seguinte (note que se apresenta o DFA incompleto, por questões de simplificação):



Temos então o reconhecimento de *if* em q_2, *int* em q_4, *then* em q_8, *for* em q_{11} e *float* em q_{15}. Em cada estado há apenas uma hipótese de transição para cada entrada, faltando apenas o estado morto e respetivas transições para que o DFA esteja completo.

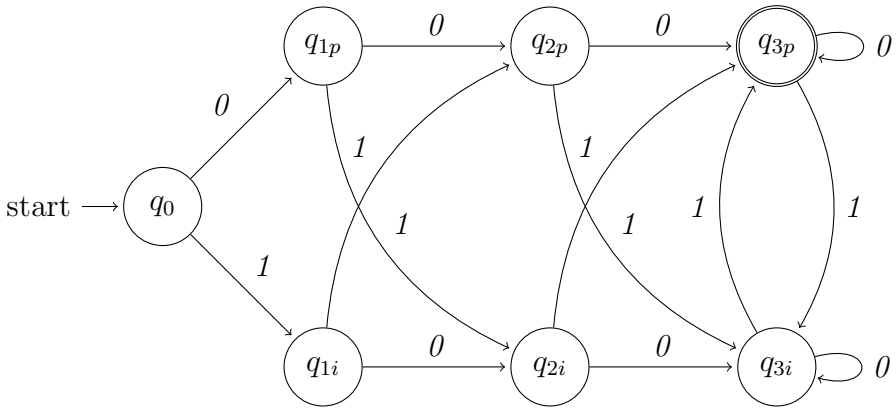
Para a representação formal do DFA, temos apenas de contruir o tuplo que o representa, ficando com

DFA A = ($\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7, q_8, q_9, q_{10}, q_{11}, q_{12}, q_{13}, q_{14}, q_{15}\}$, $\{$a,e,f,h,i,l,n,o,r,t$\}$, δ, q_0, $\{q_2, q_4, q_8, q_{11}, q_{15}\}$), em que δ representa as seguintes transições: $\delta(q_0, i) = q_1$; $\delta(q_1, f) = q_2$; $\delta(q_1, n) = q_3$; $\delta(q_3, t) = q_4$; $\delta(q_0, t) = q_5$; $\delta(q_5, h) = q_6$; $\delta(q_6, e) = q_7$; $\delta(q_7, n) = q_8$; $\delta(q_0, f) = q_9$; $\delta(q_9, o) = q_{10}$; $\delta(q_{10}, r) = q_{11}$; $\delta(q_9, l) = q_{12}$; $\delta(q_{12}, o) = q_{13}$; $\delta(q_{13}, a) = q_{14}$; $\delta(q_{14}, t) = q_{15}$.

Exercício 8 (TPC 2012/13)

Desenhe um DFA que reconheça cadeias sobre o alfabeto $\Sigma = \{0, 1\}$, de tamanho maior ou igual a 3, e com um número par de 1's.

Para manter informação de tamanho da cadeia e de paridade de 1's, precisamos de 6 estados (mais o estado inicial, para tamanho 0). Uma possível solução será então:

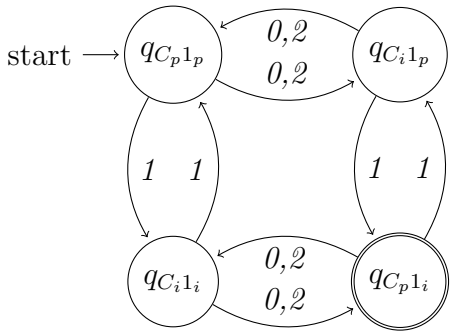


Os nomes dos estados indicam número de símbolos já lidos, e a paridade dos 1's. Como podemos ver, a parte de cima contém os estados com um número par de 1's, e os estados de baixo com um número ímpar de 1's, sendo que as transições com 0 mantém-se na metade respetiva do autómato, e as transições com 1 trocam entre a parte de cima e a de baixo do autómato. Olhando da esquerda para a direita, podemos ver o aumento do número de símbolos já lidos, sendo que o estado de aceitação será apenas quando já temos 3 ou mais símbolos lidos, e o número de 1's é par.

Exercício 9 (TPC 2013/14)

Desenhe um DFA completo que permita reconhecer cadeias no alfabeto $\Sigma = \{0, 1, 2\}$ de comprimento par e com um número ímpar de 1's.

Para este exercício, necessitamos novamente de verificar duas condições: paridade do comprimento da cadeia e paridade dos 1's. A introdução de um 0 ou um 2 altera apenas a paridade do comprimento da cadeia, mas mantém a paridade dos 1's. Já a introdução de um 1 altera não só a paridade dos 1's na cadeia como também a paridade do comprimento da cadeia. Uma solução possível será então:

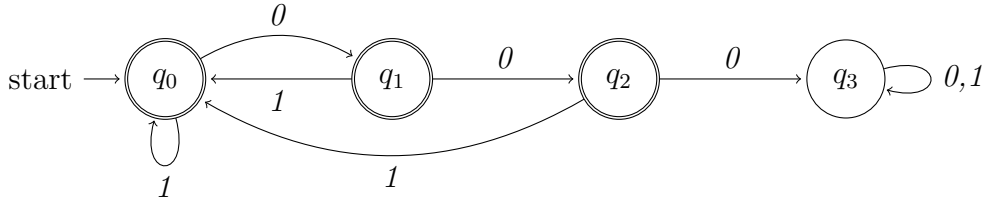


Os estado indicam a paridade do comprimento (C_p para comprimento par e C_i para comprimento ímpar) e do número de 1's (1_p para número par de 1's e 1_i para número ímpar de 1's). Como podemos ver, os estados de cima têm número par de 1's, e os estados de baixo têm um número ímpar de 1's. Assim, as transições com 0 e 2 mantém-se na sua parte (cima/baixo), alterando apenas a paridade do comprimento da cadeia. As transições com 1 alteram a paridade tanto do comprimento da cadeia como dos 1's.

Exercício 10 (Desafio 2015/16)

Modele um DFA completo que permita reconhecer cadeias binárias em que cada sub-cadeia de comprimento 3 não tenha mais de dois zeros.
Ex.: 1001 pertence à linguagem ; 10101 pertence à linguagem ; 10001 não pertence à linguagem

Uma forma simples de resolver este exercício é reformular o enunciado: dizer que para uma cadeia pertencer à linguagem, cada sub-cadeia de comprimento 3 não pode ter mais do que dois zeros, será equivalente a dizer que não podem existir sub-cadeias de comprimento 3 com 3 zeros, ou seja, não podem existir 3 zeros consecutivos. Apesar de ser a mesma linguagem, esta formulação aponta-nos logo para uma possível solução: um DFA que reconheça a sequência '000', e que aceite as cadeias até esse ponto, deixando de as aceitar caso a sequência seja reconhecida. Olhando então para uma possível resolução:

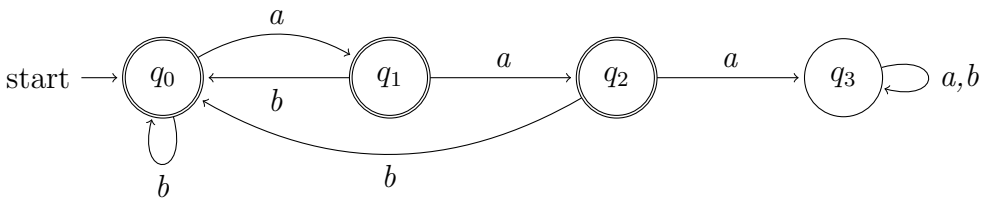


Como podemos ver, só não são aceites as cadeias que tenham chegado ao estado q_3, e para isso é necessário que existam pelo menos 3 zeros consecutivos. Assim, consegue-se rejeitar essas cadeias, e aceitar as restantes, ou seja, aquelas em que em cada sub-cadeia de 3 símbolos, não existem mais de dois zeros.

Exercício 11 (Exercício 2015/16)

Desenhe um DFA completo que reconheça cadeias no alfabeto a, b em que cada 'aa' é seguido imediatamente de um 'b'.

Para este exercício, uma forma mais simples de resolver será novamente reformular o enunciado, e perceber que a linguagem não aceita cadeias com 3 a's consecutivos, aceitando todas as outras cadeias. Uma possível solução será então:



Se repararmos, este autómato é estruturalmente igual ao DFA obtido no exercício anterior, sendo que apenas as transições apresentam símbolos diferentes. Conseguimos então verificar que os dois exercícios são formulações diferentes (e com alfabeto diferente) para o mesmo problema.

Exercício Proposto 1 Desenhe um DFA que aceite a linguagem das cadeias binárias com pelo menos dois 0s consecutivos e que não contenha dois 1s consecutivos.

Exercício Proposto 2 Desenhe um DFA que reconheça cadeias no alfabeto Σ={0, 1, 2, 3, 4, 5, 6, 7, 8, 9} em que a soma dos seus dígitos seja um múltiplo de 3. Exemplo de cadeias aceites: ε, 30, 45, 81.

Capítulo 3

Autómatos Finitos Não Deterministas (NFA)

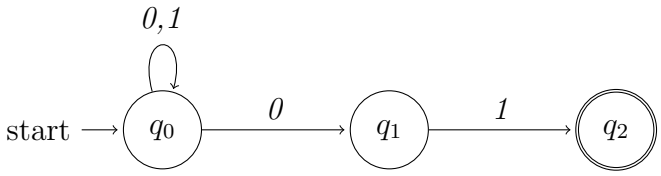
Os autómatos finitos não deterministas (ou NFAs, do inglês *Non-Deterministic Finite Automata*) são uma extensão dos DFAs, permitindo que um determinado símbolo de entrada possa conduzir a mais do que um estado. Este não-determinismo não acrescenta novas capacidades a estes autómatos, em comparação com os DFAs, mas pode permitir simplificar muito a criação de autómatos para resolução de alguns problemas (no entanto, esta 'simplificação' pode acarretar uma maior complexidade na interpretação e no desenho dos NFAs). Vamos também ver como os NFAs podem ser transformados em DFAs equivalentes.

Definição

A definição de um NFA continua a ser o tuplo $NFA = (Q, \Sigma, \delta, q_0, F)$, mas em que agora δ representa a função de transição de estados e símbolos para um conjunto de estados ($\delta(q, a) = M$, em que $M \subseteq Q$ representa o conjunto dos estados possíveis de destino.

Este não-determinismo leva a que seja necessário manter em aberto todas as possibilidades de transição do NFA. Para que uma cadeia seja aceite pelo NFA é necessário que pelo menos um dos 'caminhos' leve a um estado final. Isto pode ser visto como um processamento em paralelo de todas as possibilidades.

Vamos ver o exemplo de um NFA que permita reconhecer cadeias no alfabeto $\{0, 1\}$ terminadas em 01. Vejamos a solução em forma de diagrama:



Com esta solução, e para processar uma cadeia, '*escolhemos*' permanecer em q_0 até ao penúltimo símbolo de entrada. Se este for 0, leva-nos a q_1, e se o último símbolo for 1, o NFA fica assim em q_2, aceitando a cadeia.

Por exemplo, para a cadeia '1001', ao processar o primeiro 1, o NFA permanece em q_0. Ao ler o segundo símbolo (0), o NFA poderia nessa altura transitar para q_1, ou permanecer em q_0, pelo que temos agora de manter as duas hipóteses em aberto. Ao receber o terceiro símbolo (0), caso o NFA esteja em q_1 irá morrer, pois não há nenhuma transição definida; caso esteja em q_0, temos novamente duas transições possíveis: q_0 e q_1. Ao receber o último símbolo (1), temos novamente duas hipóteses: se o NFA estiver em q_0, transita novamente para q_0; se estiver em q_1, transita para q_2. Assim, no final da cadeia '1001', o NFA está ou em q_0 ou em q_2; como pelo menos um destes estados é um estado final, podemos dizer que a cadeia 1001 é aceite por este NFA.

Por outro lado, para a cadeia '0110', ao processar o primeiro 0, o NFA pode permanecer em q_0 ou ir para q_1. Ao receber o segundo símbolo (1), se estiver em q_0 mantém-se em q_0, e se estiver em q_1 vai para q_2. Ao receber o terceiro símbolo (1), se estiver em q_0, mantém-se em q_0, e se estiver em q_2 morre, pois não há nenhuma transição possível. Ao receber o último símbolo (0), e como apenas q_0 se mantém em aberto, podemos transitar para q_0 ou q_1, o que significa que a cadeia não é aceite, pois nenhum destes estados é final.

As tabelas abaixo mostram estas possibilidades de transição de uma forma mais compacta.

1	0	0	1
q_0	q_0	q_0	q_0
		q_1	q_2
	q_1	-	

0	1	1	0
q_0	q_0	q_0	q_0
			q_1
q_1	q_2	-	

As outras representações possíveis para o NFA (formal e tabular) mantêm-se também, com a diferença de as transições serem feitas para um conjunto de estados. Vamos ver então as representações formal e tabular para o NFA acima:

$$NFA = (\{q_0, q_1, q_2\}, \{0, 1\}, \{\delta(q_0, 0) = \{q_0, q_1\}; \delta(q_0, 1) = \{q_0\};$$
$$\delta(q_1, 0) = \emptyset; \delta(q_1, 1) = \{q_2\}; \delta(q_2, 0) = \emptyset; \delta(q_2, 1) = \emptyset\}, q_0, \{q_2\})$$

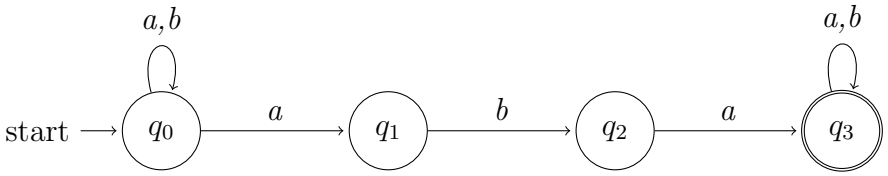
	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$* q_2$	$\emptyset$	$\emptyset$

Repare-se que quando não existem transições, é usado o símbolo de conjunto vazio, $\emptyset$, para expressar esse facto.

Exercício 1

Obtenha um NFA que permita reconhecer cadeias no alfabeto {a,b} que contenham a sub-cadeia *aba*.

Para resolver este exercício, podemos ver que a única coisa que interessa é saber se a sequência 'aba' ocorre na cadeia de entrada. Então, podemos modelar o seguinte NFA:



Se olharmos para a solução, aquilo que temos nas transições entre q_0 e q_3 é o reconhecimento da cadeia 'aba'. Em q_0 a transição para o próprio estado permite que o NFA permaneça nesse estado até 'à altura certa', isto é, até que a sub-cadeia *aba* apareça na entrada. Após reconhecimento da cadeia *aba*, chegando ao estado q_3, ficamos em q_3, que é um estado de aceitação.

Olhando para o exemplo da cadeia abbabab, temos as seguintes possibilidades de transição, representadas na tabela abaixo.

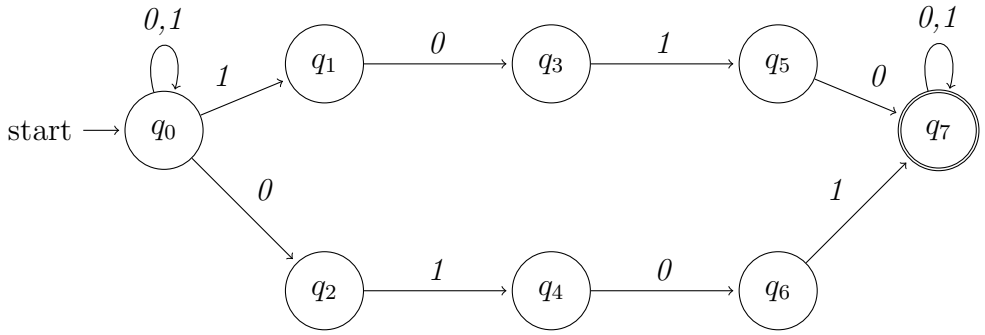
Como podemos ver, existe um caminho que leva a q_3 quando a sequência *aba* aparece na entrada, e permanece em q_3 até ao final do processamento da cadeia, o que permite aceitar a cadeia abbabab.

<table>
<tr><td>a</td><td>b</td><td>b</td><td>a</td><td>b</td><td>a</td><td>b</td></tr>
<tr><td rowspan="3">q_0</td><td rowspan="3">q_0</td><td rowspan="3">q_0</td><td rowspan="2">q_0</td><td rowspan="2">q_0</td><td>q_0</td><td>q_0</td></tr>
<tr><td>q_1</td><td>q_2</td></tr>
<tr><td>q_1</td><td>q_2</td><td>q_3</td><td>q_3</td></tr>
<tr><td>q_1</td><td>q_2</td><td>-</td><td></td><td></td><td></td><td></td></tr>
</table>

Exercício 2

Modele um NFA (tirando o máximo partido do não determinismo) que reconheça cadeias do alfabeto binário que contenham a sequência 1010 ou a sequência 0101.

Neste exercício temos algo semelhante ao anterior, mas em que queremos reconhecer uma coisa ou outra. Podemos modelar isto com uma divisão do caminho possível em dois 'ramos':



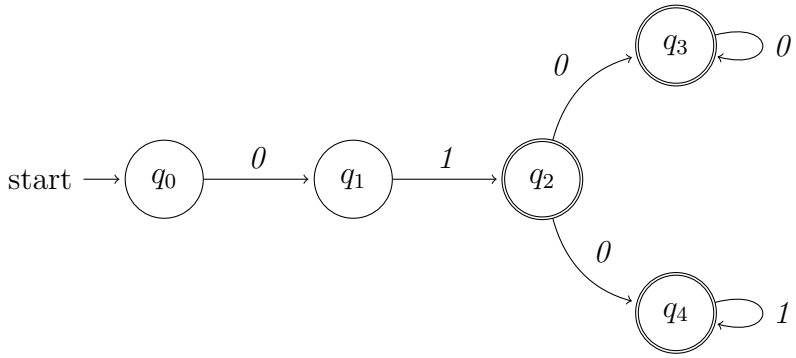
Podemos ver que esta solução é muito semelhante à solução para o problema anterior, mas desta vez com dois caminhos alternativos entre o estado inicial e o estado final.

Exercício 3

Obtenha um NFA com no máximo 5 estados para a linguagem das cadeias binárias no formato 0101^n ou 010^n ($n \geq 0$).

Neste exercício temos não só o desafio de modelar a linguagem como um NFA mas também o limite de 5 estados.

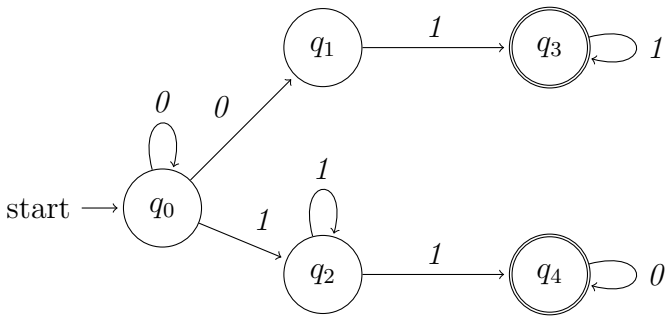
Se não tivessemos esse limite, poderíamos separar logo no início o NFA em dois ramos e cada um trataria da sua sub-linguagem. Assim, com esta limitação, podemos tentar aproveitar o facto de ambas as possibilidades de cadeias da linguagem começarem por 01 e colocar essa parte do caminho conjuntamente, e separar apenas quando houver necessidade disso.



Como podemos ver na solução, temos a parte inicial do NFA em comum para o reconhecimento do início das cadeias (01). Depois, temos a divisão em dois ramos, sendo que o ramo de cima permite reconhecer as cadeias com pelo menos um zero, o que juntando ao 01 já reconhecidos permite reconhecer as cadeias do formato 010^n com $n \geq 1$. Para permitir reconhecer com $n = 0$, o estado q_2 também necessita de ser final. Depois, no ramo de baixo, temos a possibilidade de reconhecer cadeias no formato 01^n, com $n \geq 0$, o que juntando ao 01 iniciais permite as cadeias no formato 0101^n com $n \geq 0$. Não poderíamos continuar a ter os estados q_3 e q_4 juntos, como até então, porque de um lado temos obrigatoriamente um zero, e do outro lado temos zero ou mais zeros, e não conseguiríamos distinguir os dois casos se a transição com 0 a partir de q_2 fosse para o mesmo estado.

Exercício 4

Considere o NFA abaixo.



Indique $\delta\hat{}\,(q_0, S)$ para cada uma das cadeias C: 00, 11, 1100, 1101, 011, 0011. Indique quais destas cadeias são aceites pelo NFA. Descreva a linguagem aceite pelo NFA.

Para determinar $\delta\hat{}\,(q_0, S)$ temos de verificar a cada passo quais os possíveis estados atingíveis, para no final termos o conjunto total de estados atingíveis. Se virmos no formato tabela como acima, equivale a indicar todos os estados presentes na última coluna da tabela. Assim, temos:
$\delta\hat{}\,(q_0, 00) = \{q_0, q_1\}$
$\delta\hat{}\,(q_0, 11) = \{q_2, q_4\}$
$\delta\hat{}\,(q_0, 1100) = \{q_4\}$
$\delta\hat{}\,(q_0, 1101) = \emptyset$
$\delta\hat{}\,(q_0, 011) = \{q_3, q_2, q_4\}$
$\delta\hat{}\,(q_0, 0011) = \{q_3, q_2, q_4\}$

As cadeias aceites são as que incluem pelo menos um estado de aceitação do NFA, ou seja, 11, 1100, 011 e 0011.

Em relação à linguagem aceite pelo NFA, temos de pensar nos caminhos que levam a estados de aceitação, neste caso os ramos de cima e de baixo. Olhando para o ramo de cima, temos um qualquer número de 0s aceite no estado inicial, seguido de um 0 na transição de q_0 para q_1 seguido de um 1 na transição para q_3, podendo aqui aceitar um qualquer número de 1s. Assim, temos um conjunto de cadeias no formato 0^i1^j com $i \geq 1, j \geq 1$.

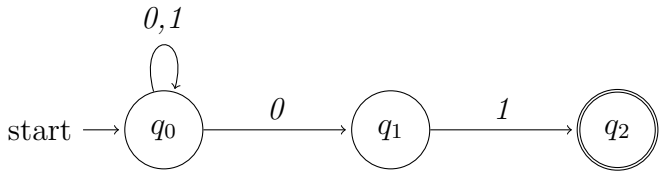
Olhando para o ramo de baixo, temos novamente um qualquer número de 0s no estado inicial, seguido de um 1 na transição para q_2, o qual por sua vez aceita um qualquer número de 1s. Temos novamente um 1 obrigatório na transição para q_4, o qual permite depois um qualquer número de 0s. Assim, temos um conjunto de cadeias no formato $0^i1^j0^k$ com $i \geq 0, j \geq 2, k \geq 0$.

Juntando os dois, obtemos então uma definição da linguagem do DFA: $\{0^i1^j : i \geq 1, j \geq 1\} \cup \{0^i1^j0^k : i \geq 0, j \geq 2, k \geq 0\}$.

Conversão de NFA para DFA

A conversão de NFA para DFA pode-se fazer facilmente através da técnica de construção de sub-conjuntos. Esta técnica simula o paralelismo dos caminhos possíveis do NFA, ao construir um DFA cujos estados representam as combinações de estados do NFA original a cada momento.

Voltando ao exemplo do NFA para reconhecer cadeias no alfabeto $\{0, 1\}$ terminadas em 01, recordemos o NFA:



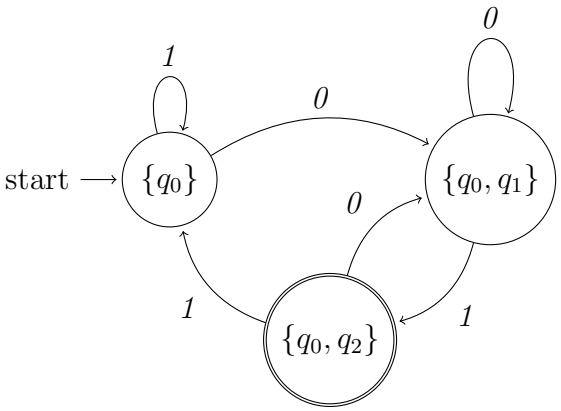
Para passar para DFA, podemos construir a tabela do DFA, começando com a primeira linha, que será o estado inicial do NFA:

	0	1
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$

Continuamos agora a preencher a tabela, adicionando na coluna de estados os que ainda não se encontram presentes, e nas colunas das transições, colocam-se todos os estados de destino possíveis. Como podemos ver na tabela acima, o estado constutuído por q_0 já se encontra na tabela, e portanto precisamos apenas de acrescentar o estado constituído pelos estados q_0 e q_1. Como podemos ver na tabela abaixo, já completa, a partir deste estado que contém q_0 e q_1 podemos transitar com 0 para esse mesmo estado, e com 1 para um novo estado que contém q_0 e q_2, o qual necessita de ser acrescentado à tabela.

	0	1
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$^* \{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

No final, quando já todos os estados de destino de transições estiverem presentes na tabela, chegamos ao final da conversão. Neste caso, obtivemos na mesma 3 estados (embora em teoria o número de estados do DFA possa atingir 2^N, em que N é o número de estados do NFA, na prática é raro ultrapassar em muito o número de estados do NFA). Vejamos o DFA resultante graficamente:

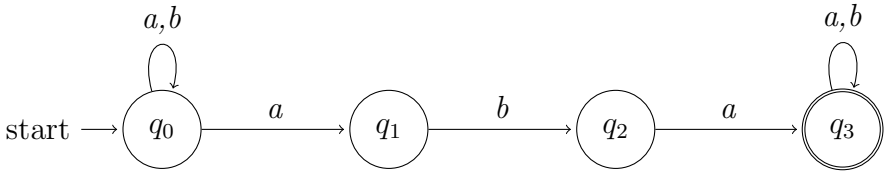


Apesar de manter o mesmo número de estados (3), o DFA apresenta mais transições do que o NFA equivalente.

Exercício 5

Converta o NFA obtido no Exercício 1 para o DFA equivalente.

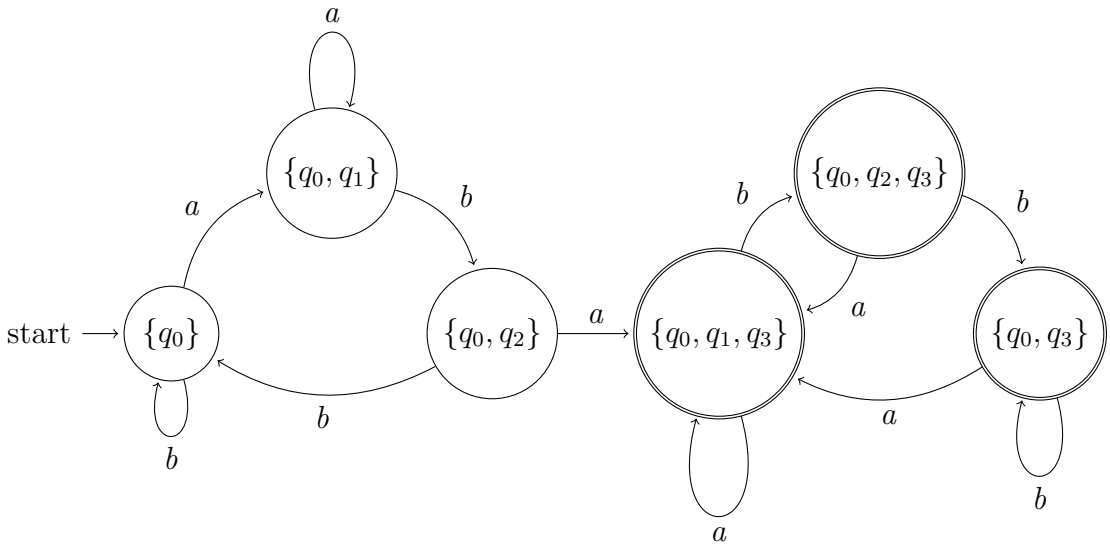
Recordando o DFA do exercício 1:



Convertendo para DFA, temos então a seguinte tabela:

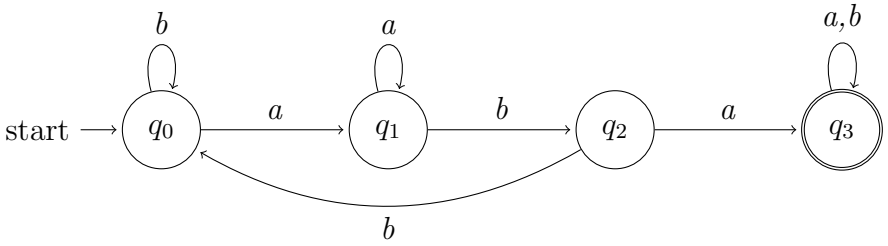
	a	b
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\{q_0, q_2\}$	$\{q_0, q_1, q_3\}$	$\{q_0\}$
* $\{q_0, q_1, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_2, q_3\}$
* $\{q_0, q_2, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$
* $\{q_0, q_3\}$	$\{q_0, q_1, q_3\}$	$\{q_0, q_3\}$

Representando o DFA graficamente:



Podemos ver que neste caso obtemos mais estados do que no NFA original.

Podemos também tentar comparar com uma solução obtida diretamente ao pensar num DFA para resolver o problema. Ao tentar reconhecer a sequência aba, necessitavamos apenas de estados que mantivessem informação sobre que parte da sequência foi já reconhecida. Uma solução típica seria:

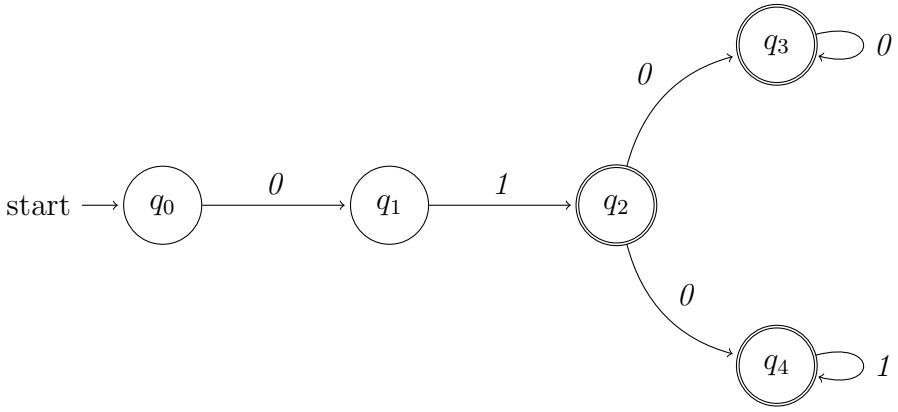


Mais à frente vamos falar sobre como minimizar o número de estados de um DFA, para eliminar estados desnecessários. Nessa altura, podemos voltar a este exemplo, para mostrar que os 3 estados finais obtidos acima pela conversão do NFA são equivalentes entre si, e portanto conseguimos minimizar esse DFA para um igual ao que construímos diretamente.

Exercício 6

Converta o NFA obtido no Exercício 3 para o DFA equivalente.

Relembrando o NFA do exercício 3:



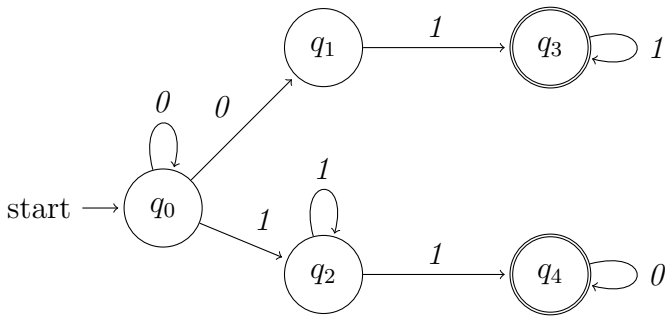
Ora, contruíndo então a tabela do DFA equivalente:

	0	1
$\rightarrow \{q_0\}$	$\{q_1\}$	$\emptyset$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$* \{q_2\}$	$\{q_3, q_4\}$	$\emptyset$
$* \{q_3, q_4\}$	$\{q_3\}$	$\{q_4\}$
$* \{q_3\}$	$\{q_3\}$	$\emptyset$
$* \{q_4\}$	$\emptyset$	$\{q_4\}$

Exercício 7

Converta o NFA do Exercício 4 para o DFA equivalente.

Relembrando o NFA do exercício 4:



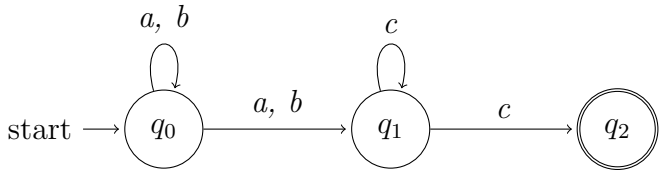
	0	1
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_2\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_2, q_3\}$
$\{q_2\}$	$\emptyset$	$\{q_2, q_4\}$
$^* \{q_2, q_3\}$	$\emptyset$	$\{q_2, q_3, q_4\}$
$^* \{q_2, q_4\}$	$\{q_4\}$	$\{q_2, q_4\}$
$^* \{q_2, q_3, q_4\}$	$\{q_4\}$	$\{q_2, q_3, q_4\}$
$^* \{q_4\}$	$\{q_4\}$	$\emptyset$

Construíndo a tabela do DFA equivalente:

Como podemos ver, o DFA tem 7 estados, em comparação com os 5 estados do NFA, o que fica ainda bastante aquém dos 32 (2^5) estados possíveis. Este é então mais um exemplo em que o número de estados do DFA não é muito superior ao número de estados do NFA que lhe deu origem.

Exercício 8 (TPC 2010/11)

Considere o autómato finito abaixo sobe o alfabeto $\Sigma = \{a, b, c\}$. Represente o autómato usando a notação formal e converta-o para um DFA que aceite a mesma linguagem, desenhando o DFA completo resultante.



Olhando rapidamente para o autómato, podemos ver que ele reconhece a linguagem das cadeias com pelo menos um a ou um b, e com pelo menos um c, e em que todos os c's aparecem no final da cadeia.

Para a representação na notação formal, temos apenas de lembrar a representação formal como um tuplo com 5 elementos e representar a informação nesse formato, ficando então com:

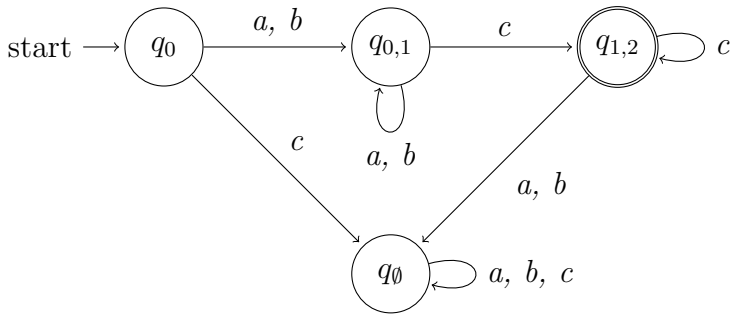
$(Q, \Sigma, q_0, \delta, F)$

em que $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{a, b, c\}$, $F = \{q_2\}$ e δ é definido por $\delta(q_0, a) = \{q_0, q_1\}$; $\delta(q_0, b) = \{q_0, q_1\}$; $\delta(q_0, c) = \emptyset$; $\delta(q_1, a) = \emptyset$; $\delta(q_1, b) = \emptyset$; $\delta(q_1, c) = \{q_1, q_2\}$; $\delta(q_2, a) = \emptyset$; $\delta(q_2, b) = \emptyset$; $\delta(q_2, c) = \emptyset$

Transformando este NFA num DFA, começamos novamente por q_0 e construímos a tabela de transições

	a	b	c
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_1, q_2\}$
$* \{q_1, q_2\}$	$\emptyset$	$\emptyset$	$\{q_1, q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

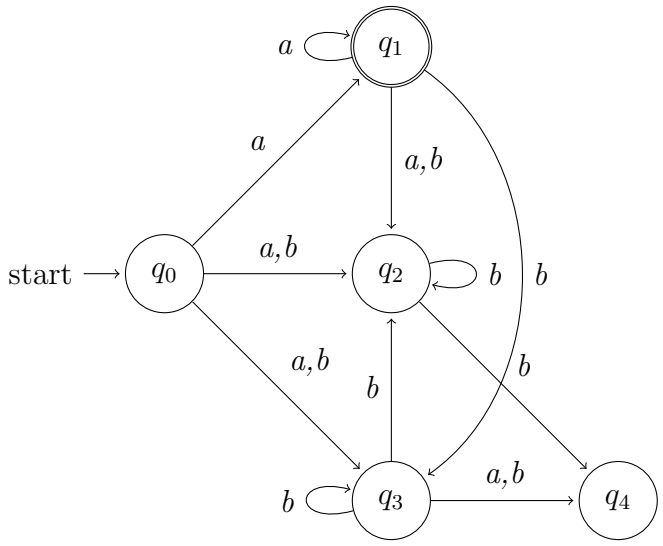
A partir da tabela podemos então desenhar também o DFA completo.



Como podemos ver, o DFA não tem muitos mais estados do que o NFA equivalente, sendo apenas acrescentado o estado morto para completar o DFA.

Exercício 9 (TPC 2011/12)

Represente o DFA abaixo usando a notação formal e converta-o para um DFA equivalente, apresentando o diagrama resultante.



Para representar o DFA na notação formal, temos apenas de representar os vários componentes:

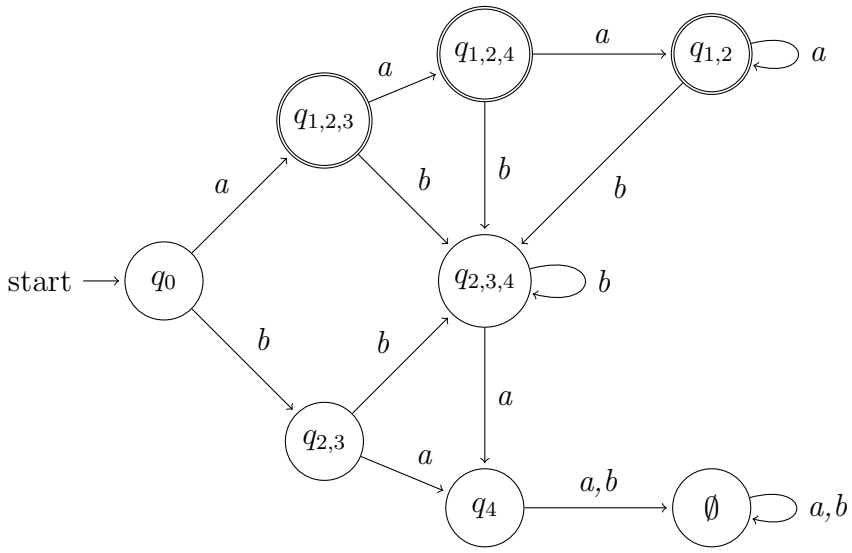
(Q, Σ, δ, q_0, F) = ($\{q_0, q_1, q_2, q_3, q_4\}$, {a, b}, δ, q_0, $\{q_1\}$)

em que δ é definido por: $\delta(q_0, a) = \{q_1, q_2, q_3\}$; $\delta(q_0, b) = \{q_2, q_3\}$; $\delta(q_1, a) = \{q_1, q_2\}$; $\delta(q_1, b) = \{q_2, q_3\}$; $\delta(q_2, b) = \{q_2, q_4\}$; $\delta(q_3, a) = \{q_4\}$; $\delta(q_3, b) = \{q_2, q_3, q_4\}$.

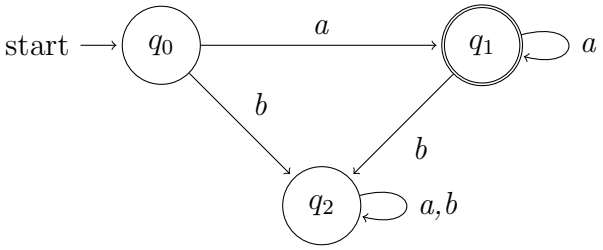
Para obter o DFA equivalente, podemos então usar o método de construção de sub-conjuntos, obtendo a seguinte tabela:

	a	b
$\rightarrow \{q_0\}$	$\{q_1, q_2, q_3\}$	$\{q_2, q_3\}$
* $\{q_1, q_2, q_3\}$	$\{q_1, q_2, q_4\}$	$\{q_2, q_3, q_4\}$
$\{q_2, q_3\}$	$\{q_4\}$	$\{q_2, q_3, q_4\}$
* $\{q_1, q_2, q_4\}$	$\{q_1, q_2\}$	$\{q_2, q_3, q_4\}$
$\{q_2, q_3, q_4\}$	$\{q_4\}$	$\{q_2, q_3, q_4\}$
$\{q_4\}$	$\emptyset$	$\emptyset$
* $\{q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2, q_3, q_4\}$
$\emptyset$	$\emptyset$	$\emptyset$

Podemos agora desenhar o DFA obtido:



Repare-se que o DFA não é muito maior do que o NFA original (8 estados vs 5 estados no NFA original). Note-se ainda que este DFA não é o DFA ideal para o reconhecimento desta linguagem. Na verdade, olhando para o NFA original, podemos ver que a linguagem reconhecida pelo autómato é a das cadeias constituídas apenas por a's e de comprimento maior ou igual a 1 (os estados q_2, q_3 e q_4 não permitem conduzir a um estado de aceitação). Para reconhecer esta mesma linguagem, bastava-nos ter um simples DFA com dois estados (mais o estado morto):



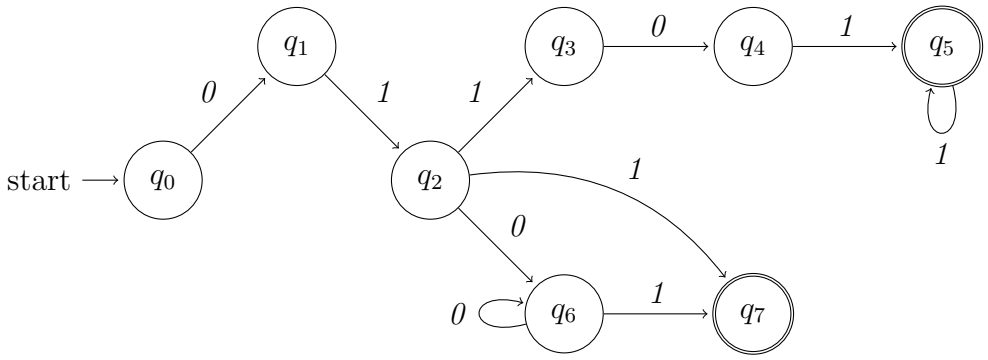
Mais tarde vamos então abordar a minimização de DFAs, e o método que nos permite chegar do primeiro DFA ao segundo.

Exercício 10 (Desafio 2014/15)

Modele um NFA que permita reconhecer cadeias do alfabeto {0,1} no formato $01101^i, i \geq 1$ ou no formato $010^j1, j \geq 0$.

Ex: 01101 pertence à linguagem; 0110 não pertence à linguagem; 011 pertence à linguagem; 0101 pertence à linguagem.
Converta o NFA obtido para o DFA equivalente.

Olhando para a linguagem, podemos simplesmente separar o autómato em dois ramos, e tratar cada um deles independentemente. No entanto, como ambos os ramos começam com os mesmos símbolos, podemos juntar a parte inicial, e assim reduzir o número de estados do NFA. Uma possível solução seria então:

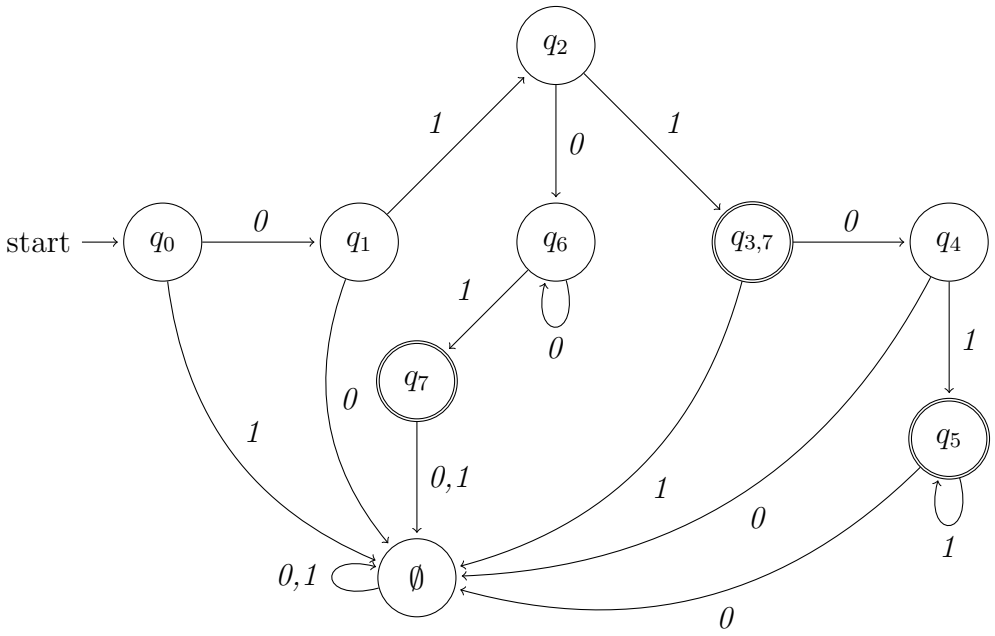


Olhando para a solução, vemos então a parte inicial e coincidente dos dois ramos nas primeiras transições, e a partir de q_2 os dois ramos para as duas partes da linguagem. A parte de cima reflete a primeira parte $(01101^i, i \geq 1)$, e a parte de baixo para a segunda parte $(010^j1, j \geq 0)$.

Convertendo para DFA, ficamos com a seguinte tabela:

	0	1
$\rightarrow \{q_0\}$	$\{q_1\}$	$\emptyset$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$\{q_2\}$	$\{q_6\}$	$\{q_3, q_7\}$
$\{q_6\}$	$\{q_6\}$	$\{q_7\}$
* $\{q_3, q_7\}$	$\{q_4\}$	$\emptyset$
* $\{q_7\}$	$\emptyset$	$\emptyset$
$\{q_4\}$	$\emptyset$	$\{q_5\}$
* $\{q_5\}$	$\emptyset$	$\{q_5\}$
$\emptyset$	$\emptyset$	$\emptyset$

Desenhando o DFA ficamos então com:



Novamente, podemos constatar que o número de estados do DFA resultante não é muito maior do que o número de estados do NFA original.

Exercício Proposto 1 Desenhe um NFA que aceite a linguagem das cadeias binárias em que o último dígito é diferente do primeiro. Converta o NFA obtido para um DFA e compare-o com a solução do Exercício 4 do capítulo anterior.

Exercício Proposto 2 Desenhe um NFA que aceite a linguagem das cadeias binárias com um número ímpar de zeros e número par de uns. Converta o NFA obtido para um DFA e compare os dois.

Capítulo 4

NFAs com Transições Épsilon (ε-NFA)

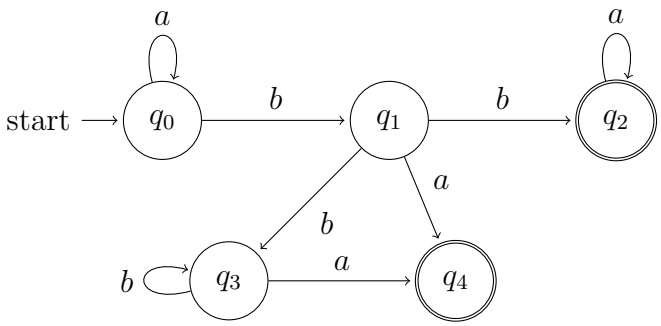
Os autómatos finitos não deterministas com transições épsilon (e-NFAs ou ε-NFAs, do inglês *Non-Deterministic Finite Automata with ε-Moves*) são uma extensão dos NFAs, permitindo transições espontâneas entre estados (isto é, sem consumir nenhum símbolo de entrada). Estas transições introduzem não-determinismo aos autómatos, uma vez que num determinado momento podem estar no estado de origem ou de destino de uma transição ε. Novamente, este não-determinismo não acrescenta novas capacidades a estes autómatos, mas pode permitir simplificar a criação de autómatos em alguns problemas. Vamos também ver como os ε-NFAs podem ser convertidos em DFAs.

Definição

A definição de um ε-NFA continua a ser o tuplo $\varepsilon\text{-NFA} = (Q, \Sigma, \delta, q_0, F)$, sendo que as transições podem ser efetuadas com um símbolo do alfabeto ou com a cadeia vazia (representada pelo símbolo ε).

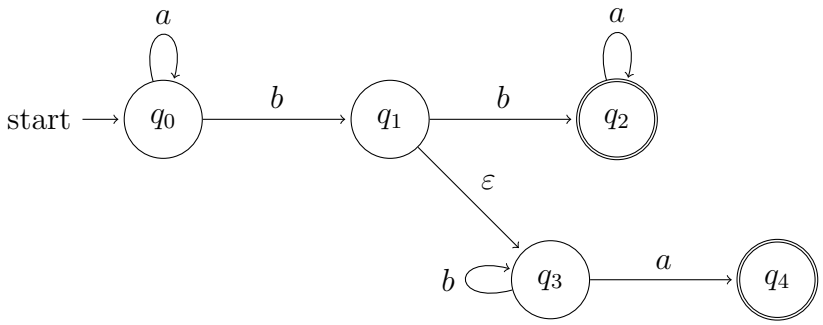
Vamos ver o exemplo de um ε-NFA para a linguagem das cadeias no alfabeto {a, b} constituídas por cadeias no formato $a^i bb^j a$ com $i \geq 0, j \geq 0$, ou no formato $a^k bba^m$ com $k \geq 0, m \geq 0$.

Este problema pode ser resolvido com um NFA sem transições-ε, mas um ε-NFA pode facilitar a criação e interpretação do autómato. Vamos olhar para a solução usando um NFA sem transições-ε:



Com o NFA, temos a parte comum entre os dois formatos de cadeias representada entre q_0 e q_1; no 'ramo de cima' temos a representação das cadeias no formato $a^k bba^m$; já nos ramos de baixo, temos as possibilidades para representação das cadeias no formato $a^i bb^j a$, podendo transitar diretamente de q_1 para q_4 no caso de não existirem b's, ou transitar de q_1 para q_3 caso exista pelo menos um b.

Um ε-NFA permite uma solução com menos transições, e com um aspeto que transmite mais facilmente o formato das cadeias da linguagem:



Neste ε-NFA, e em comparação com o NFA acima, a parte de cima do autómato continua igual, permitindo o reconhecimento das cadeias no formato $a^k bba^m$. Já a parte de baixo torna-se mais simples de interpretar, uma vez que se identifica mais facilmente o formato das cadeias reconhecidas ($a^i bb^j a$). A transição-ε permite então que no momento em que o autómato transita para q_1 com o primeiro b possa imediatamente transitar para q_3 sem consumir qualquer símbolo de entrada. Assim, se o próximo símbolo da cadeia for um a, o autómato transita para q_4; se o símbolo for um b, pode transitar para q_2 ou para q_3, permitindo assim um funcionamento semelhante ao do NFA acima mas permitindo uma leitura mais simplificada.

Vamos ver a tabela de transições para este autómato.

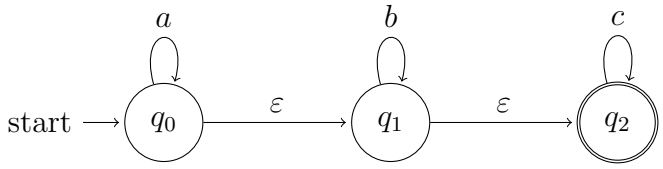
	ε	a	b
$\rightarrow q_0$	$\emptyset$	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_3\}$	$\emptyset$	$\{q_2\}$
$^* q_2$	$\emptyset$	$\{q_2\}$	$\emptyset$
q_3	$\emptyset$	$\{q_4\}$	$\{q_3\}$
$^* q_4$	$\emptyset$	$\emptyset$	$\emptyset$

Repare-se que na tabela de transição do ε-NFA é acrescentada uma coluna para as transições-ε, como se se tratasse de mais um símbolo do alfabeto.

Exercício 1

Construa um ε-NFA para a linguagem das cadeias do alfabeto {a,b,c} no formato $a^x b^y c^z$ em que $x \geq 0, y \geq 0, z \geq 0$. Apresente também a tabela de transições.

Uma solução para este problema tem de contemplar a cadeia vazia (que faz também parte da linguagem), e não deve permitir símbolos distintos intercalados. Vamos ver uma solução possível:



Esta solução permite transitar diretamente de q_0 para q_1 e daí para q_2 sem consumir qualquer símbolo de entrada, aceitando assim a cadeia vazia. Também permite um qualquer número de a's, b's e c's nos estados q_0, q_1 e q_2 respetivamente, garantindo assim a ordem e o formato das cadeias da linguagem.

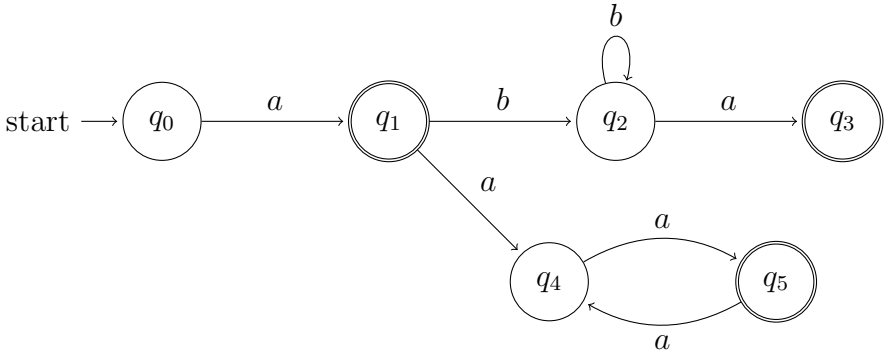
A tabela de transições é agora fácil de obter.

	ε	a	b	c
$\rightarrow q_0$	$\{q_1\}$	$\{q_0\}$	$\emptyset$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$	$\{q_1\}$	$\emptyset$
$* q_2$	$\emptyset$	$\emptyset$	$\emptyset$	$\{q_2\}$

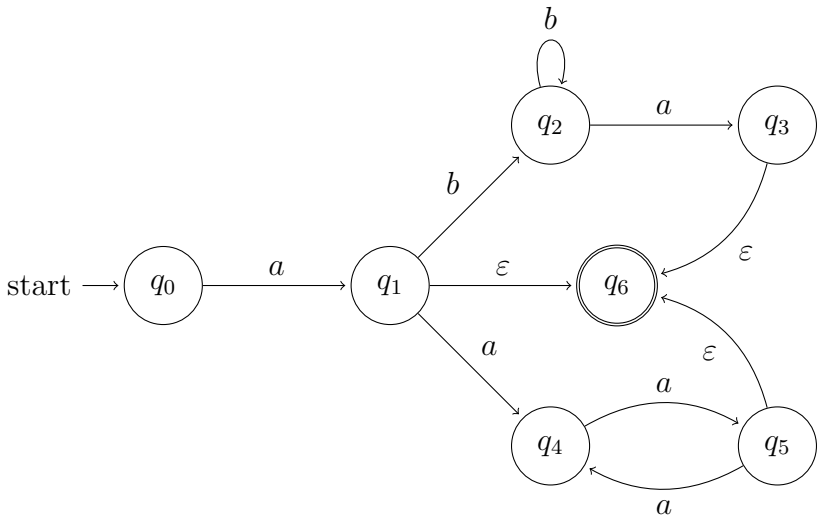
Exercício 2

Construa um ε-NFA para a linguagem das cadeias do alfabeto {a,b} no formato a^{2n+1} com $n \geq 0$ e no formato ab^ia com $i \geq 1$ com apenas um estado de aceitação.

Para que o autómato tenha apenas um estado de aceitação, podemos construir o autómato como se fosse um NFA e acrescentar depois um novo estado (o de aceitação) para o qual acrescentamos transições-ε a partir de todos os estados de aceitação do NFA (os quais deixam de o ser). Assim, podemos olhar para um possível NFA para resolver o problema:



Podemos ver nesta solução que o 'ramo' de cima permite reconhecer as cadeias no formato ab^ia, enquanto que o 'ramo' de baixo permite reconhecer as cadeias com um número ímpar de a's. Note-se que q_1 é também um estado de aceitação para garantir que a cadeia só com 1 a é aceite. Fazendo então a substituição de estados finais, ficamos com o seguinte ε-NFA:

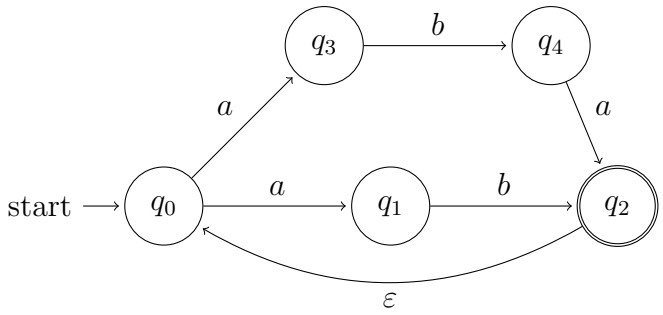


Note-se que a solução permanece igual, exceto que agora apenas existe um estado de aceitação, o qual não tem qualquer transição de saída. Uma outra solução poderia também ter sido atingida aproveitando o estado q_3 como estado final único, e adicionando apenas transições-ε de q_1 e de q_5 para q_3, permitindo assim poupar o estado extra adicionado na solução acima.

Exercício 3

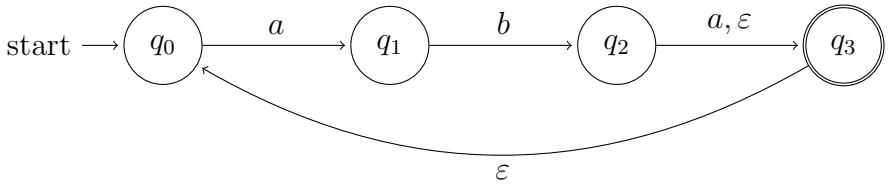
Apresente um ε-NFA que permita reconhecer cadeias do alfabeto {a, b} constituída por uma ou mais repetições de *aba* ou *ab*.

Uma possibilidade de resolução envolve criar um NFA para reconhecer as duas possibilidades, e acrescentar depois uma transição-ε do estado final para o estado inicial, de forma a permitir repetições dessas mesmas cadeias.



Podemos ver as duas possíveis sub-cadeias constituíntes das cadeias da linguagem nos 'ramos' de cima e do meio do autómato. No estado final, a transição-ε permite então começar de novo uma outra sequência *aba* ou *ab*.

Podemos ver abaixo uma outra solução possível, tirando um pouco mais de partido do não-determinismo:

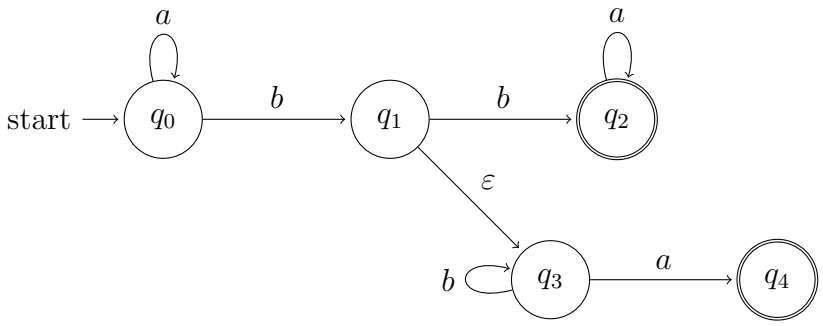


Nesta solução, as partes comuns são colocadas conjuntamente, e é usada uma transição-ε para permitir cadeias *ab* ou *aba*.

Conversão de ε-NFA para DFA

Para converver um ε-NFA para DFA socorremo-nos do fecho-ε dos estados do ε-NFA como auxiliar para o método da construção de sub-conjuntos já visto no capítulo anterior. O fecho-ε (ou *E-Close* ou ε-*Close*) de um estado inclui o próprio estado e todos aqueles que se possam alcançar com transições ε. Incluem-se aqui não só os estados atingíveis diretamente através de uma transição-ε mas também todos aqueles atingíveis a partir desses, e assim sucessivamente.

Voltemos ao exemplo acima, e recordemos o ε-NFA obtido:



Para transformar em DFA, começamos primeiro por identificar o fecho-ε de cada estado:

Podemos agora criar a tabela do DFA pelo método de construção de sub-conjuntos. O estado inicial do DFA resultante (primeira linha na tabela abaixo) será o fecho-ε do estado inicial do ε-NFA original. Para cada

Estado	Fecho-ε
q_0	$\{q_0\}$
q_1	$\{q_1, q_3\}$
q_2	$\{q_2\}$
q_3	$\{q_3\}$
q_4	$\{q_4\}$

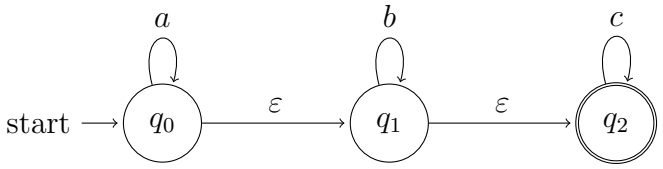
transição (células na tabela), deve ser sempre usado o fecho-ε dos estados de destino. Por exemplo, podemos ver na primeira linha que a partir de q_0 e com um b é possível ir para q_1; devemos na célula colocar não apenas q_1 mas sim o fecho-ε de q_1, ou seja $\{q_1, q_3\}$. Continuamos depois tal como anteriormente, acrescentando linhas na tabela que ainda não estejam presentes.

	a	b
$\rightarrow \{q_0\}$	$\{q_0\}$	$\{q_1, q_3\}$
$\{q_1, q_3\}$	$\{q_4\}$	$\{q_2, q_3\}$
$* \{q_4\}$	$\emptyset$	$\emptyset$
$* \{q_2, q_3\}$	$\{q_2, q_4\}$	$\{q_3\}$
$* \{q_2, q_4\}$	$\{q_2\}$	$\emptyset$
$\{q_3\}$	$\{q_4\}$	$\{q_3\}$
$* \{q_2\}$	$\{q_2\}$	$\emptyset$
$\emptyset$	$\emptyset$	$\emptyset$

Exercício 4

Calcule o fecho-ε de cada estado do ε-NFA obtido no exercício 1 e converta-o para um DFA.

Recordando o ε-NFA do exercício 1:



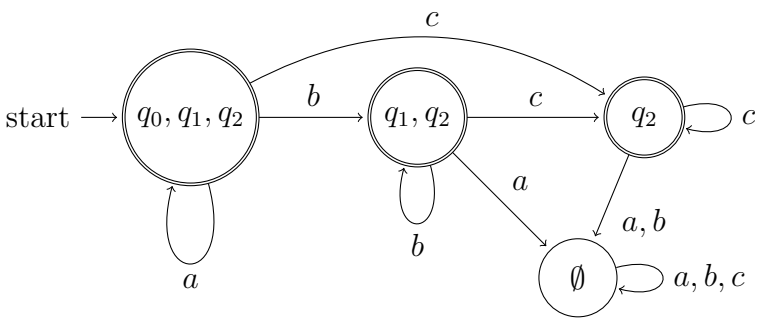
Vamos então calcular o fecho-ε de cada estado. Não esqueçamos que o fecho tem uma definição recursiva, e por isso o fecho-ε de q_0 vai inclui não só q_1 como também q_2.

Estado	Fecho-ε
q_0	$\{q_0, q_1, q_2\}$
q_1	$\{q_1, q_2\}$
q_2	$\{q_2\}$

A partir daqui, podemos construir a tabela para conversão para DFA, começando a preencher a tabela pelo estado inicial, ou seja, usando o fecho-ε do estado inicial do ε-NFA.

	a	b	c
$\rightarrow$ * $\{q_0, q_1, q_2\}$	$\{q_0, q_1, q_2\}$	$\{q_1, q_2\}$	$\{q_2\}$
* $\{q_1, q_2\}$	$\emptyset$	$\{q_1, q_2\}$	$\{q_2\}$
* $\{q_2\}$	$\emptyset$	$\emptyset$	$\{q_2\}$
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

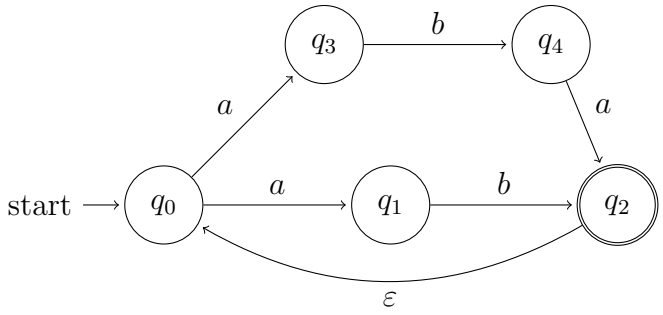
Temos aqui um DFA resultante com o mesmo número de estados do que o ε-NFA que lhe deu origem (mais o estado morto), mas com mais algumas transições, e em que todos os estados são de aceitação. Vamos ver o diagrama equivalente.



Exercício 5

Calcule o fecho-ε de cada estado de cada um dos ε-NFAs obtidos no exercício 3 e converta-os para DFAs.

Recordando o primeiro ε-NFA do exercício 3:



Fazendo o cálculo do fecho-ε dos estados do autómato:

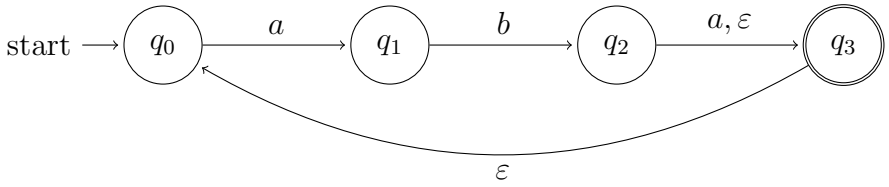
Estado	Fecho-ε
q_0	$\{q_0\}$
q_1	$\{q_1\}$
q_2	$\{q_2, q_0\}$
q_3	$\{q_3\}$
q_4	$\{q_4\}$

A partir daqui, usamos então a técnica de construção de sub-conjuntos para obter o DFA equivalente (por questões de simplificação de representação, vamos omitir o estado morto).

Como podemos ver, o DFA resultante tem apenas 4 estados, sendo dois deles estados de aceitação.

	a	b
$\rightarrow \{q_0\}$	$\{q_1, q_3\}$	$\emptyset$
$\{q_1, q_3\}$	$\emptyset$	$\{q_0, q_2, q_4\}$
$* \{q_0, q_2, q_4\}$	$\{q_0, q_1, q_2, q_3\}$	$\emptyset$
$* \{q_0, q_1, q_2, q_3\}$	$\{q_1, q_3\}$	$\{q_0, q_2, q_4\}$

Olhando agora para a segunda solução do mesmo exercício:



Fazendo o cálculo do fecho-ε dos estados do autómato:

Estado	Fecho-ε
q_0	$\{q_0\}$
q_1	$\{q_1\}$
q_2	$\{q_2, q_3, q_0\}$
q_3	$\{q_3, q_0\}$

A partir daqui, usamos então a técnica de construção de sub-conjuntos para obter o DFA equivalente:

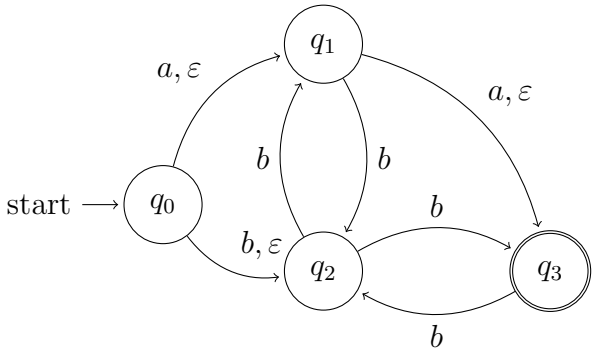
	a	b
$\rightarrow \{q_0\}$	$\{q_1\}$	$\emptyset$
$\{q_1\}$	$\emptyset$	$\{q_2, q_3, q_0\}$
$* \{q_2, q_3, q_0\}$	$\{q_3, q_0, q_1\}$	$\emptyset$
$* \{q_3, q_0, q_1\}$	$\{q_1\}$	$\{q_2, q_3, q_0\}$

O DFA resultante tem também 4 estados, sendo dois deles estados de aceitação. Podemos também notar que o DFA obtido a partir deste segundo

ε-NFA é equivalente estruturalmente ao DFA obtido a partir do primeiro ε-NFA.

Exercício 6

Considere o ε-NFA abaixo. Calcule o fecho-ε de cada estado e converta o autómato para um DFA.



Fazendo o cálculo do fecho-ε dos estados do autómato:

Estado	Fecho-ε
q_0	$\{q_0, q_1, q_2, q_3\}$
q_1	$\{q_1, q_3\}$
q_2	$\{q_2\}$
q_3	$\{q_3\}$

Note-se que o fecho-ε de q_0 inclui não só q_1 e q_2, mas também q_3, dada a definição recursiva de fecho. A partir daqui usamos então a técnica de construção de sub-conjuntos para obter o DFA equivalente.

O DFA resultante tem apenas então cinco estados, sendo quatro deles estados de aceitação.

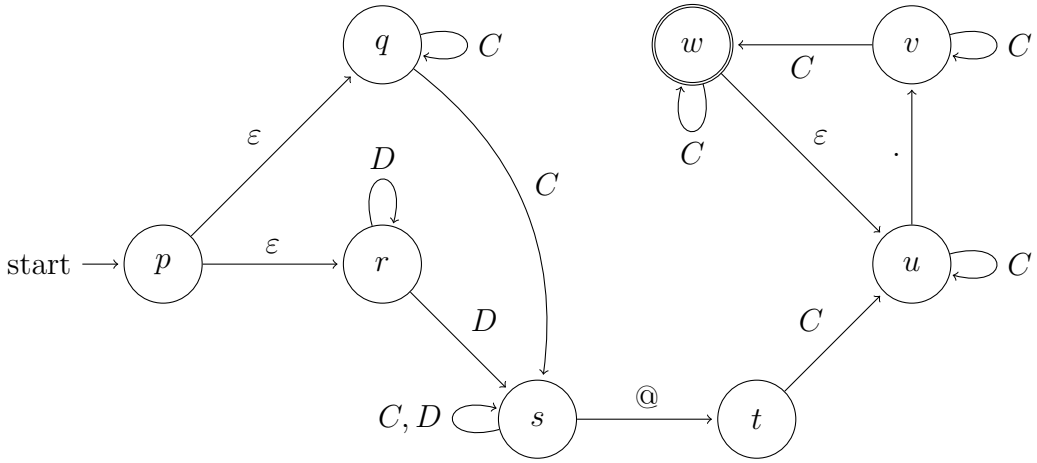
	a	b
$\rightarrow$ * $\{q_0, q_1, q_2, q_3\}$	$\{q_1,\ q_3\}$	$\{q_1, q_2, q_3\}$
* $\{q_1,\ q_3\}$	$\{q_3\}$	$\{q_2\}$
* $\{q_1, q_2, q_3\}$	$\{q_3\}$	$\{q_1, q_2, q_3\}$
* $\{q_3\}$	$\emptyset$	$\{q_2\}$
$\{q_2\}$	$\emptyset$	$\{q_1, q_3\}$

Exercício 7 (Desafio 2014/15)

Considere o ε-NFA apresentado de seguida, onde C representa um qualquer caráter do alfabeto (a-z) e D um dígito (0-9). Para que poderá servir este autómato? Converta-o no DFA equivalente.

	ε	C	D	@	.
$\rightarrow$ p	{q,r}	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
q	$\emptyset$	{q,s}	$\emptyset$	$\emptyset$	$\emptyset$
r	$\emptyset$	$\emptyset$	{r,s}	$\emptyset$	$\emptyset$
s	$\emptyset$	{s}	{s}	{t}	$\emptyset$
t	$\emptyset$	{u}	$\emptyset$	$\emptyset$	$\emptyset$
u	$\emptyset$	{u}	$\emptyset$	$\emptyset$	{v}
v	$\emptyset$	{v,w}	$\emptyset$	$\emptyset$	$\emptyset$
* w	{u}	{w}	$\emptyset$	$\emptyset$	$\emptyset$

Olhando apenas para a tabela de transições, torna-se complicado perceber qual a funcionalidade do autómato, muito embora a presença do símbolo '@' no alfabeto nos possa já dar uma dica. Vamos representar o autómato graficamente, para facilitar a interpretação do mesmo:



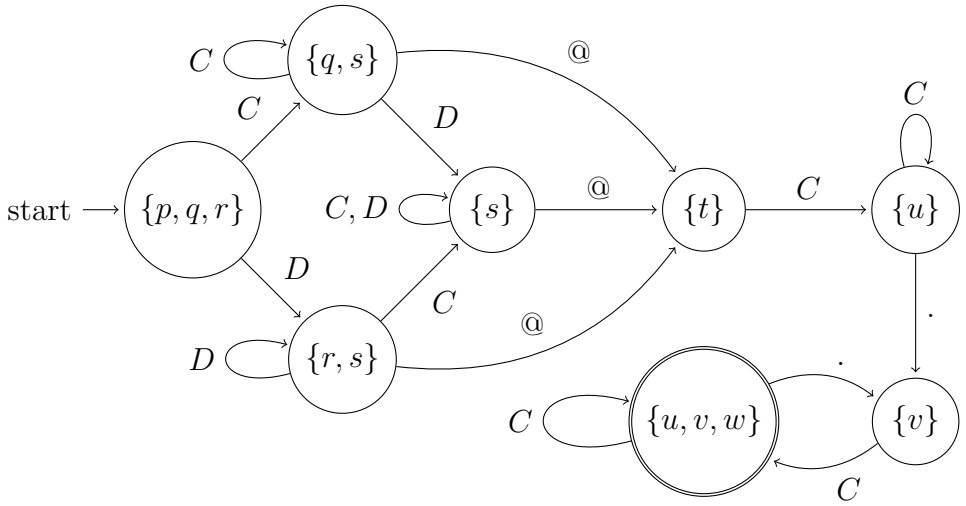
Com o diagrama já se torna mais simples perceber que este autómato reconhece cadeias no formato de um endereço de correio eletrónico, podendo ter um conjunto de dígitos ou letras de comprimento maior ou igual a 1, seguido de um '@', seguido de uma ou mais letras, seguido de um ponto e novamente uma ou mais letras, sendo que a parte final (ponto e letras) pode ser repetida várias vezes.

Para converter para DFA, convém primeiro determinar o fecho-ε de cada estado. Temos então que:

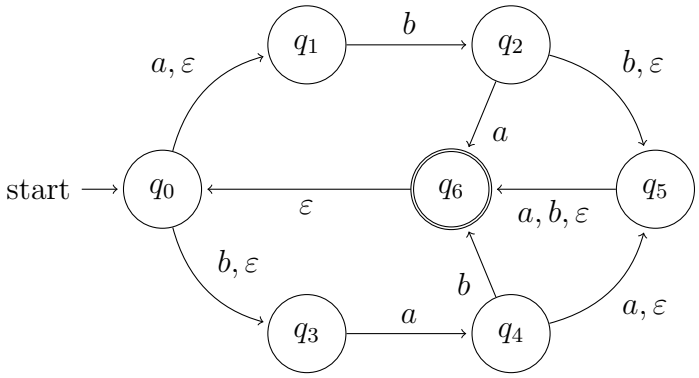
Estado	Fecho-ε
p	$\{p, q, r\}$
q	$\{q\}$
r	$\{r\}$
s	$\{s\}$
t	$\{t\}$
u	$\{u\}$
v	$\{v\}$
w	$\{w, u\}$

Podemos agora fazer a transformação do ε-NFA num DFA, tendo tanto a tabela como o diagrama respetivos:

	C	D	@	.
$\rightarrow \{p,q,r\}$	$\{q,s\}$	$\{r,s\}$	$\emptyset$	$\emptyset$
$\{q,s\}$	$\{q,s\}$	$\{s\}$	$\{t\}$	$\emptyset$
$\{r,s\}$	$\{s\}$	$\{r,s\}$	$\{t\}$	$\emptyset$
$\{s\}$	$\{s\}$	$\{s\}$	$\{t\}$	$\emptyset$
$\{t\}$	$\{u\}$	$\emptyset$	$\emptyset$	$\emptyset$
$\{u\}$	$\{u\}$	$\emptyset$	$\emptyset$	$\{v\}$
$\{v\}$	$\{u,v,w\}$	$\emptyset$	$\emptyset$	$\emptyset$
$*\ \{u,v,w\}$	$\{u,v,w\}$	$\emptyset$	$\emptyset$	$\{v\}$



Exercício Proposto 1 Considere o ε-NFA abaixo. Indique todas as cadeias de comprimento menor do que 3 aceites pelo autómato, e converta-o para um DFA.



Capítulo 5

Expressões Regulares

As Expressões Regulares são uma outra forma de expressar linguagens regulares, sendo uma alternativa de representação aos autómatos vistos até aqui, e constituindo uma forma mais compacta de representação.

Definição

Uma expressão regular simples pode ser obtida com um qualquer símbolo do alfabeto ou com o símbolo de cadeia vazia (ε). Adicionalmente, pode ser usado o símbolo $\emptyset$ para descrever a linguagem vazia. Agora, e de forma indutiva, uma nova expressão regular pode ser obtida por concatenação (representada por um '.', normalmente omitido), união (representada por um '+') ou fecho (representado por um *) de expressões existentes.

Vamos olhar para alguns exemplos no alfabeto binário (símbolos 0 e 1).

No caso de querermos reconhecer a cadeia 10, a expressão regular que o permite fazer será apenas 10, sendo aqui omitido o símbolo de concatenação.

Se quisermos reconhecer as cadeias 10 ou 01, podemos fazer a união das expressões para cada uma das opções, e ficamos com a expressão 10 + 01.

Se quisermos reconhecer cadeias iniciadas por 1, necessitamos de poder dizer que depois do 1 podem existir zero ou mais ocorrências dos símbolos 0 ou 1. Assim, podemos usar o fecho para permitir essa repetição, usando parêntesis para indicar a associação dos operadores, ficando com a expressão 1(0+1)*.

Exercício 1

Obtenha uma expressão regular para a linguagem das cadeias do alfabeto

{0,1} que quando interpretadas como um número em binário são múltiplos de 4.

Dizer que uma cadeia em binário representa números múltiplos de 4 é equivalente a dizer que as cadeias devem terminar em 00. Assim, a expressão regular torna-se simples de escrever:

(0+1)*00

Permite-se assim reconhecer qualquer sequência de zeros e uns inicialmente, mas obriga-se a que a cadeia termine em dois zeros consecutivos.

Exercício 2

Obtenha uma expressão regular para a linguagem das cadeias sobre o alfabeto {a,b} cujo comprimento seja múltiplo de 3.

Este problema pode resolver-se de forma simples, permitindo repetições de 3 símbolos do alfabeto:

((a+b)(a+b)(a+b))*

Com esta expressão, garante-se que cada bloco existente na cadeia final tem exatamente 3 símbolos, o que permite formar cadeias de comprimento múltiplo de 3 (incluindo a cadeia vazia).

Exercício 3

Obtenha uma expressão regular para a linguagem das cadeias do alfabeto {a,b} em que cada sequência de dois *a*'s é obrigatoriamente seguida de uma sequência de 3 *b*'s.

Para resolver o problema, necessitamos de restringir a quantidade de *a*'s consecutivos e no caso de existirem dois *a*'s, deve-se forçar a existência de 3 *b*'s. Isto pode ser conseguido com a seguinte expressão:

(ab + aabbb + b)* (a + ε)

Olhando para esta solução, podemos verificar que as possibilidades de existência de *a*'s estão sempre associadas a ocorrências de *b*'s: se tivermos dois *a*'s, obriga-se a ter três *b*'s; se existir apenas um *a*, obriga-se a ter um *b*, de forma a evitar *a*'s consecutivos. Ao adicionar a possibilidade de ter um b em qualquer ponto, e usando o fecho para a união destas três hipóteses,

obtém-se uma solução parcial para o problema. Falta apenas permitir que as cadeias terminem em a, o que é conseguido com a parte $(a + \varepsilon)$ no final (repare-se que não há possibilidade de ter dois a's consecutivos).

Uma outra possível solução seria:
b* (abb* + aabbbb*)* (a + ε)
Nesta solução, permite-se o início da cadeia com a's ou b's, e um qualquer número de b's após o número obrigatório de b's. No final, a componente (a + ε) permite novamente que a cadeia termine possivelmente em a.

Exercício 4

Obtenha uma expressão regular para a linguagem das cadeias sobre o alfabeto {a,b} em que o número de a's é par.

Para resolver este problema, é necessário que cada a seja acompanhado obrigatoriamente de um outro a, garantindo assim a condição de paridade dos a's. Vamos ver algumas possibilidade de resposta.

b* (ab*ab*)*
Esta solução garante que a existência de a's é sempre em número par, permitindo um qualquer número de b's entre os a's. O b* inicial assegura que a cadeia possa iniciar em b, sendo que a parte da direita da expressão já permite cadeias terminadas em a ou em b.

(b + ab*a)*
Esta solução garante que os a's aparecem sempre em número par, e por outro lado permite que existam b's em qualquer ponto da cadeia (o fecho da expressão permite a existência de qualquer número de b's antes ou depois do par de a's, e o b* entre os a's permite também ter b's entre os a's de cada par de a's).

Exercício 5 (Desafio 2014/15)

Encontre uma expressão regular sobre o alfabeto {a, b, c} que permita reconhecer cadeias em que existem pelo menos dois caracteres idênticos consecutivos. Por exemplo, as cadeias 'aba' e 'acaba' não pertencem à linguagem,

enquanto que as cadeias 'abba' e 'baaca' pertencem à linguagem.

Tendo o alfabeto 3 símbolos, existem três hipóteses de ter caracteres idênticos consecutivos: aa, bb ou cc. Então, essa parte será descrita pela sub-expressão aa+bb+cc. Temos ainda de permitir que o resto da cadeia tenha qualquer sequência de símbolos do alfabeto, o que pode ser conseguido com a sub-expressão (a+b+c)*. Juntando tudo, obtemos a expressão

(a+b+c)* (aa+bb+cc) (a+b+c)*

Exercício 6 (Exame 2014/15)

Apresente uma expressão regular que permita reconhecer códigos postais nacionais com 7 dígitos. Considere o símbolo 'D' como representando qualquer dígito (0 a 9), 'M' como representando uma letra maiúscula, 'm' como representando uma letra minúscula (incluindo letras acentuadas), 'E' como representando um espaço e 'T' como representando um traço. A expressão deve permitir reconhecer códigos postais como '4200-465 Porto', '4400-017 Vila Nova de Gaia' ou '4490-001 A Ver-o-Mar'.

A parte inicial da expressão é relativamente simples de obter (assumindo que não são colocadas restrições adicionais, como por exemplo o código postal não poder começar com 0): DDDDTDDDE. Falta agora a parte da expressão que permita reconhecer nomes de localidades. Estes nomes devem ter pelo menos uma palavra (embora possam ter mais), sendo que a primeira palavra deve começar por maiúscula, e as palavras devem ser separadas por um espaço ou um traço. Assim, uma possibilidade será Mm*((E+T)(M+m)m*)*. Assim, a expressão final será:

DDDDTDDDEMm*((E+T)(M+m)m*)*

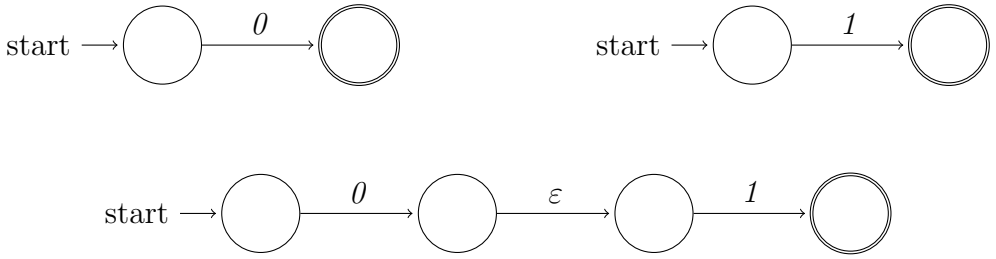
Conversão de Expressões Regulares em Autómatos

As expressões regulares podem ser convertidas em autómatos (ε-NFAs) através do uso de *templates* para cada uma das possibilidades da definição indutiva das expressões regulares. Recorde-se que, a partir daí, é possível converter para um DFA.

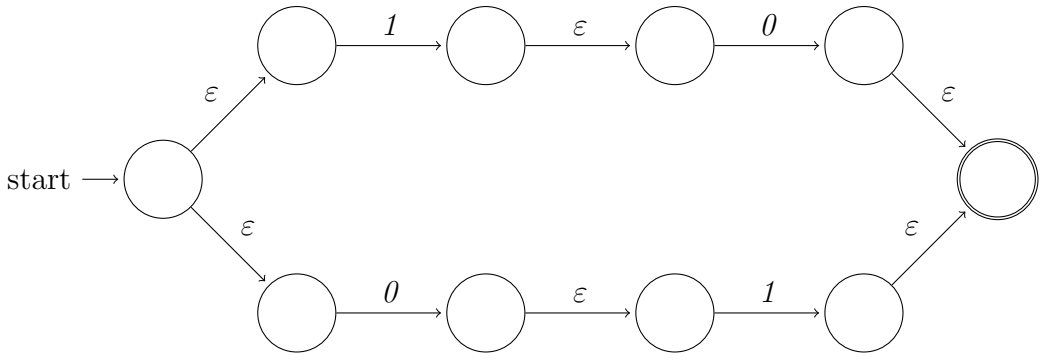
A conversão faz-se então substituíndo os elementos básicos (cada símbolo do alfabeto, cadeia vazia, ou símbolo de linguagem vazia) pelos templates

equivalentes, e usando depois os templates das operações de concatenação, união e fecho para ir construíndo um ε-NFA de sub-expressões cada vez mais complexas, até atingir a expressão completa.

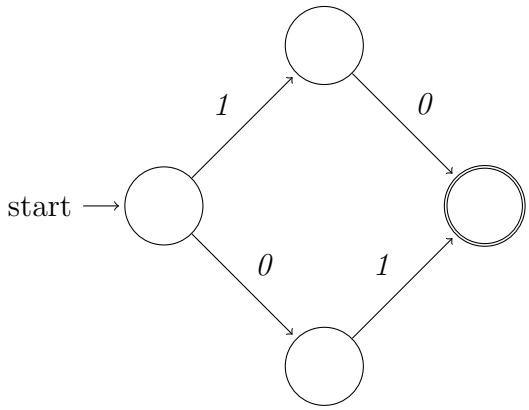
Vamos ver o exemplo da expressão 01 + 10. As imagens abaixo mostram os vários passos para a criação do autómato par a expressão. Na primeira imagem, vemos as representações dos símbolos 0 e 1 (cada um dos símbolos representado por um autóato). Na imagem seguinte, pode ver-se a construção do autómato para a expressão 01 (a construção do autómato para a expressão 10 seria equivalente), podendo ver-se como é feita a concatenação dos autómatos de duas expressões previamente convertidas.



Na figura seguinte mostra-se o resultado final da união das duas opções (10 + 01), sendo possível ver como é feita a união de dois autómatos previamente obtidos.

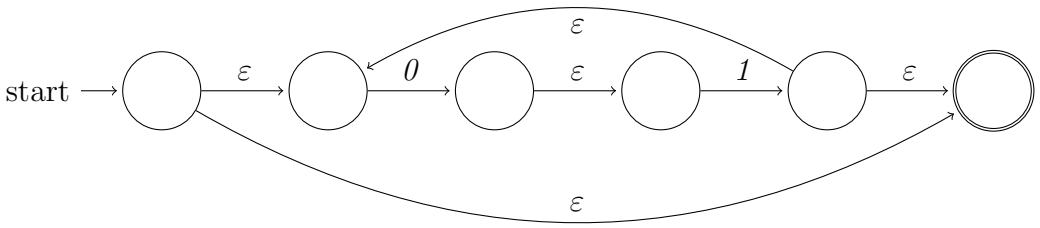


Podemos ver que esta solução tem um conjunto de transições ε excessivo e desnecessário. Podemos simplificar o autómato acima para o seguinte:

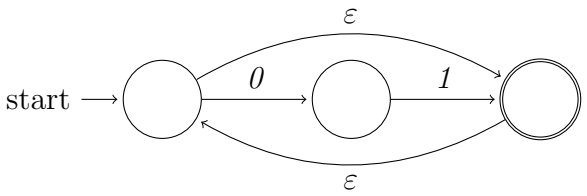


Podemos ver neste autómato que continua a aceitar as sequências 10 e 01

Vamos ver ainda um exemplo com o fecho, para a expressão (01)*. O autómato da expressão 01 é idêntico ao obtido anteriormente e mostrado acima. Ao realizar o fecho, o autómato ficará assim:



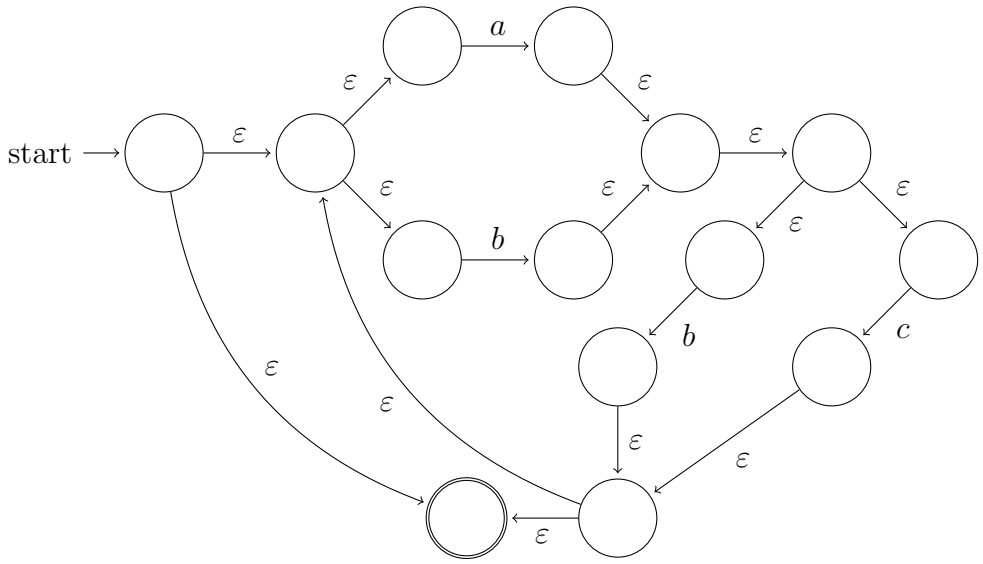
Novamente, este autómato tem ainda um conjunto de transições ε que aparentam ser desnecessárias. Podemos então simplificar o autómato, ficando com a seguinte versão:



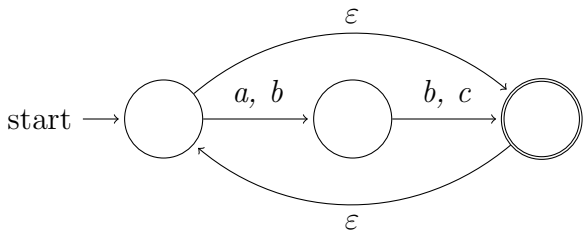
Exercício 7

Converta a expressão regular ((a+b)(b+c))* num autómato.

Para realizar a conversão, podemos optar pela utilização dos *templates*, obtendo o seguinte ε-NFA para a expressão:



Novamente, podemos ver que o autómato obtido por este método tem várias transições desnecessárias. Podemos agora tentar simplificar este autómato (ou alternativamente, tentar produzir diretamente um autómato a partir da expressão regular). Uma possível solução (mais compacta) seria:

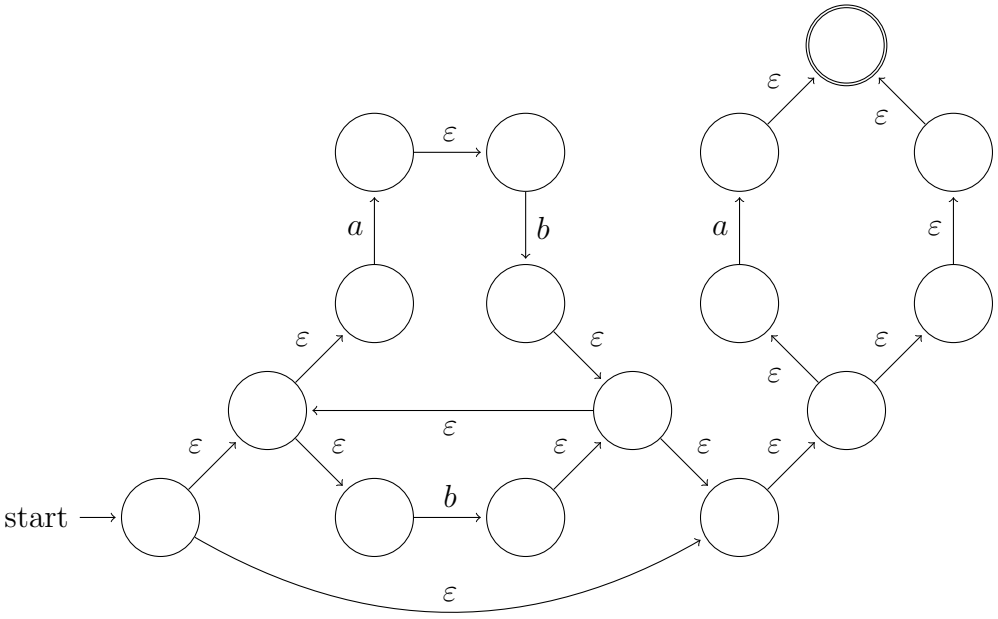


Repare-se que este autómato é estruturalmente equivalente ao visto anteriormente para a expressão (10)* dado que as duas expressões são também semelhantes.

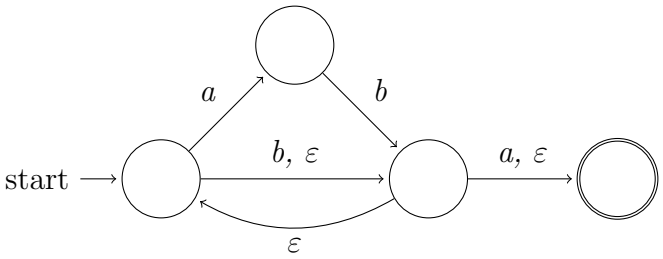
Exercício 8

Obtenha um autómato para a expressão regular (ab+b)*(a+ε).

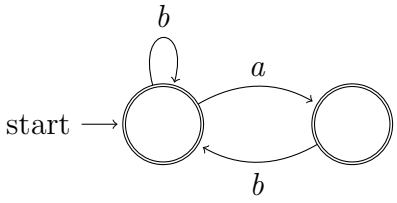
Usando novamente os *templates*, chegamos ao seguinte autómato:



Novamente, este autómato tem várias transições ε desnecessárias. Podemos tentar simplificar o autómato, produzindo uma representação mais compacta, como por exemplo:



Podíamos ainda tentar condensar mais o autómato, e representá-lo apenas com dois estados:



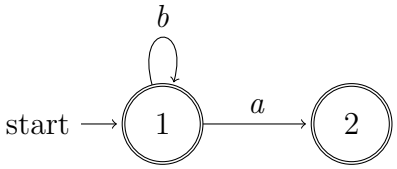
Conversão de Autómatos em Expressões Regulares

Vamos ver dois métodos para realizar a conversão em sentido contrário, denominados de construção de caminhos e eliminação de estados.

Construção de Caminhos

Para obter a expressão regular a partir de um autómato pelo método de construção de caminhos, começamos por numerar os nós do autómato de 1 até N, em que N é o número de estados do autómato. Depois, vão sendo obtidos caminhos sucessivamente mais complexos, no formato $R^k_{i,j}$, representando os caminhos possíveis do nó i até ao nó j passando no máximo pelo nó numerado como k. No final, a expressão regular para a linguagem é obtida pela união de todos os caminhos desde o estado inicial até cada um dos estados finais com k igual a N.

Vamos ver um exemplo para o seguinte autómato:



Os caminhos para $k = 0$ são obtidos vendo as ligações entre i e j que não passam por mais nenhum nó. Assim, temos os seguintes caminhos:

$R^0_{1,1} = b + \varepsilon \quad ; \quad R^0_{1,2} = a \quad ; \quad R^0_{2,1} = \emptyset \quad ; \quad R^0_{2,2} = \varepsilon$

Note-se que existe sempre a possibilidade de transitar para o próprio nó com ε, e que quando não existe caminho direto entre nós distintos é usado o símbolo $\emptyset$ para o indicar.

Os caminhos de índice k mais elevado são obtidos usando a fórmula:

$$R^k_{i,j} = R^{k-1}_{i,j} + R^{k-1}_{i,k}(R^{k-1}_{k,k})^* R^{k-1}_{k,j}$$

Assim, podemos calcular os caminhos para $k = 1$:

Podemos agora calcular os caminhos para $k = 2$. Como apenas interessam os caminhos que levam do estado inicial a cada um dos estados finais, basta calcular $R^2_{1,1}$ e $R^2_{1,2}$, pelo que podemos evitar alguns cálculos desnecessários.

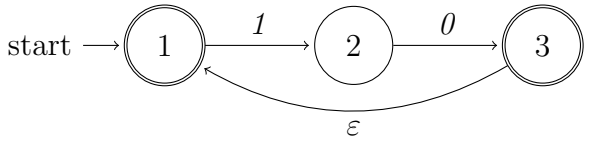
$R^1_{1,1}$	$R^0_{1,1} + R^0_{1,1}(R^0_{1,1})^*R^0_{1,1} = (b+\varepsilon) + (b+\varepsilon)(b+\varepsilon)^*(b+\varepsilon) = b^*$
$R^1_{1,2}$	$R^0_{1,2} + R^0_{1,1}(R^0_{1,1})^*R^0_{1,2} = a + (b+\varepsilon)(b+\varepsilon)^*a = b^*a$
$R^1_{2,1}$	$R^0_{2,1} + R^0_{2,1}(R^0_{1,1})^*R^0_{1,1} = \emptyset + \emptyset(b+\varepsilon)^*(b+\varepsilon) = \emptyset$
$R^1_{2,2}$	$R^0_{2,2} + R^0_{2,1}(R^0_{1,1})^*R^0_{1,2} = \varepsilon + \emptyset(b+\varepsilon)^*a = \varepsilon$

$R^2_{1,1}$	$R^1_{1,1} + R^1_{1,2}(R^1_{2,2})^*R^1_{2,1} = b^* + b^*a\varepsilon^*\emptyset = b^*$
$R^2_{1,2}$	$R^1_{1,2} + R^1_{1,2}(R^1_{2,2})^*R^1_{2,2} = b^*a + b^*a\varepsilon^*\varepsilon = b^*a$

A expressão final será a união destes dois termos, ficando b* + b*a, ou, escrito de outra forma, b*(a+ε).

Exercício 9

Use o método de construção de caminhos para encontrar uma expressão regular para a linguagem definida pelo seguinte autómato.



A tabela abaixo contém as expressões que representam os caminhos parciais para $0 \leq k \leq 2$.

A expressão final será $R^3_{1,1} + R^3_{1,3}$, pelo que, usando a fórmula, podemos desdobrar na seguintes expressão:

$R^2_{1,1} + R^2_{1,3}(R^2_{3,3})^*R^2_{3,1} \ + \ R^2_{1,3} + R^2_{1,3}(R^2_{3,3})^*R^2_{3,3}$

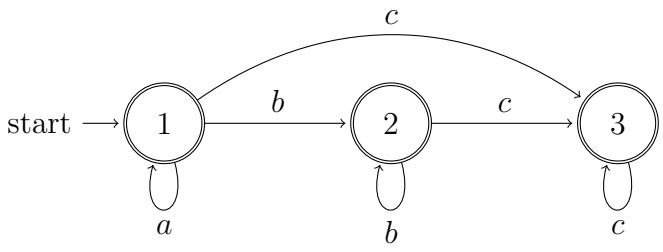
A expressão final será então $\varepsilon + 10(\varepsilon + 10)^*\varepsilon + 10 + 10(\varepsilon + 10)^*(\varepsilon + 10)$ podendo ser simplificada para $(10)^*$.

Podemos, de forma a poupar esforço, começar pelo final, isto é, olhar para os termos de $k = 3$ que constituem a expressão regular da linguagem, e verificar queis os termos de $k = 2$ necessários, fazendo depois o mesmo para $k = 1$.

$k=0$	$k=1$	$k=2$
$R^0_{1,1} = \varepsilon$	$R^1_{1,1} = \varepsilon + \varepsilon\varepsilon^*\varepsilon = \varepsilon$	$R^2_{1,1} = \varepsilon + 1\varepsilon^*\emptyset = \varepsilon$
$R^0_{1,2} = 1$	$R^1_{1,2} = 1 + \varepsilon\varepsilon^*1 = 1$	$R^2_{1,2} = 1 + 1\varepsilon^*\varepsilon = 1$
$R^0_{1,3} = \emptyset$	$R^1_{1,3} = \emptyset + \varepsilon\varepsilon^*\emptyset = \emptyset$	$R^2_{1,3} = \emptyset + 1\varepsilon^*0 = 10$
$R^0_{2,1} = \emptyset$	$R^1_{2,1} = \emptyset + \emptyset\varepsilon^*\varepsilon = \emptyset$	$R^2_{2,1} = \emptyset + \varepsilon\varepsilon^*\emptyset = \emptyset$
$R^0_{2,2} = \varepsilon$	$R^1_{2,2} = \varepsilon + \emptyset\varepsilon^*1 = \varepsilon$	$R^2_{2,2} = \varepsilon + \varepsilon\varepsilon^*\varepsilon = \varepsilon$
$R^0_{2,3} = 0$	$R^1_{2,3} = 0 + \emptyset\varepsilon^*\emptyset = 0$	$R^2_{2,3} = 0 + \varepsilon\varepsilon^*0 = 0$
$R^0_{3,1} = \varepsilon$	$R^1_{3,1} = \varepsilon + \varepsilon\varepsilon^*\varepsilon = \varepsilon$	$R^2_{3,1} = \varepsilon + 1\varepsilon^*\emptyset = \varepsilon$
$R^0_{3,2} = \emptyset$	$R^1_{3,2} = \emptyset + \varepsilon\varepsilon^*1 = 1$	$R^2_{3,2} = 1 + 1\varepsilon^*\varepsilon = 1$
$R^0_{3,3} = \varepsilon$	$R^1_{3,3} = \varepsilon + \varepsilon\varepsilon^*\emptyset = \varepsilon$	$R^2_{3,3} = \varepsilon + 1\varepsilon^*0 = \varepsilon + 10$

Exercício 10

Considere o autómato abaixo e calcule $R^3_{1,3}$



Começando pelo final, e usando a fórmula para determinar quais os termos para $k = 2$ necessários, temos então:
$R^3_{1,3} = R^2_{1,3} + R^2_{1,3}R^2_{3,3}{}^*R^2_{3,3}$

Aplicando a fórmula a cada um dos termos:
$R^2_{1,3} = R^1_{1,3} + R^1_{1,2}R^1_{2,2}{}^*R^1_{2,3}$
$R^2_{3,3} = R^1_{3,3} + R^1_{3,2}R^1_{2,2}{}^*R^1_{2,3}$

Aplicando novamente a fórmula aos termos distintos:
$R^1_{1,3} = R^0_{1,3} + R^0_{1,1}R^0_{1,1}{}^*R^0_{1,3} = c + (a+\varepsilon)(a+\varepsilon)^*c = a^*c$
$R^1_{1,2} = R^0_{1,2} + R^0_{1,1}R^0_{1,1}{}^*R^0_{1,2} = b + (a+\varepsilon)(a+\varepsilon)^*b = a^*b$

$R^1_{2,2} = R^0_{2,2} + R^0_{2,1}R^0_{1,1}{}^*R^0_{1,2} = (b+\varepsilon) + \emptyset(a+\varepsilon)^*b = b+\varepsilon$
$R^1_{2,3} = R^0_{2,3} + R^0_{2,1}R^0_{1,1}{}^*R^0_{1,3} = c + \emptyset(a+\varepsilon)^*c = c$
$R^1_{3,3} = R^0_{3,3} + R^0_{3,1}R^0_{1,1}{}^*R^0_{1,3} = (c+\varepsilon) + \emptyset(a+\varepsilon)^*c = c+\varepsilon$
$R^1_{3,2} = R^0_{3,2} + R^0_{3,1}R^0_{1,1}{}^*R^0_{1,2} = \emptyset + \emptyset(a+\varepsilon)^*b = \emptyset$

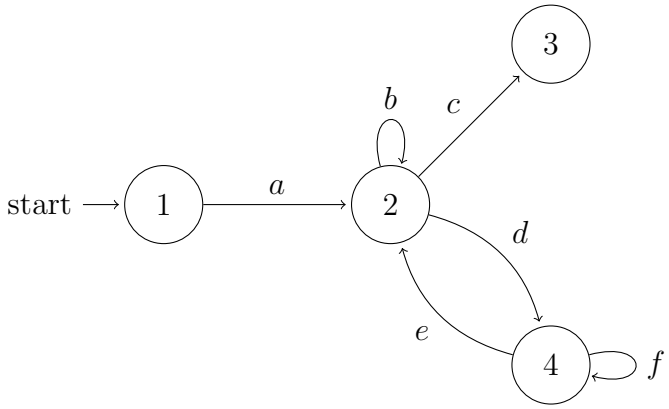
Podemos agora calcular os termos para $k = 2$:
$R^2_{1,3} = a^*c + a^*b(b+\varepsilon)^*c =$ a*b*c
$R^2_{3,3} = c + \varepsilon + \emptyset(b+\varepsilon)^*c = c+\varepsilon$

E agora podemos calcular o termo pretendido:
$R^3_{1,3} = a^*b^*c + a^*b^*c(c+\varepsilon)^*(c+\varepsilon) =$a*b*cc*

Eliminação de Estados

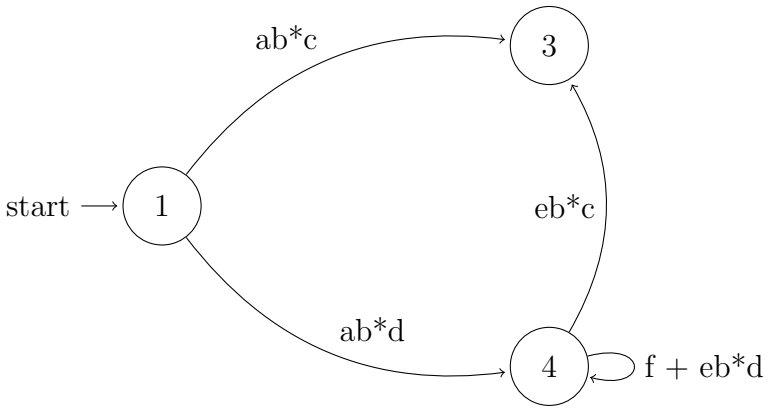
O segundo método consiste em eliminar sucessivamente estados não finais do autómato, substituindo-os pelas ligações equivalentes, até ficar apenas com o estado inicial e estados finais. A expressão regular da linguagem é obtida pela união das expressões que levam do estado inicial a cada um dos estados finais.

Vamos ver um exemplo de eliminação de um estado, no seguinte autómato:



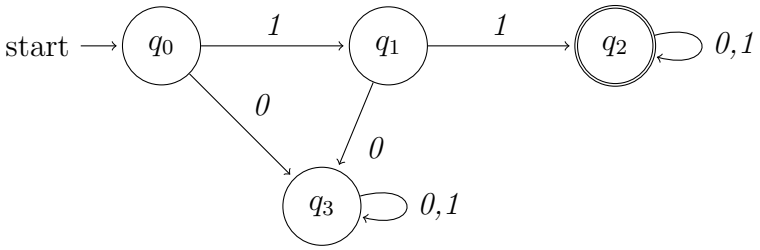
Se escolhermos eliminar o estado 2, será necessário inserir novas ligações que substituam todas aquelas em que o estado 2 participa. Então, vamos ver todas as ligações de entrada no estado 2, e todas as ligações de saída do estado 2, acrescentando depois novas ligações entre cada par de estados (entrada, saída). Temos entradas a partir dos estados 1 e 4, e saídas para os estados 3 e 4, pelo que vamos ter ligações entre os pares de estados (1,3), (1,4), (4,3) e (4,4). A expressão em cada nova ligação é obtida concatenando a expressão do arco que ligava ao estado a ser eliminado com o fecho do

loop do estado a ser eliminado com a expressão do arco que liga ao arco de destino. Eliminando o estado 2, ficamos então com o seguinte autómato:

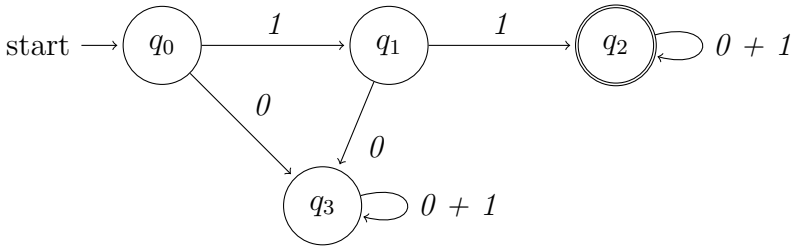


Exercício 11

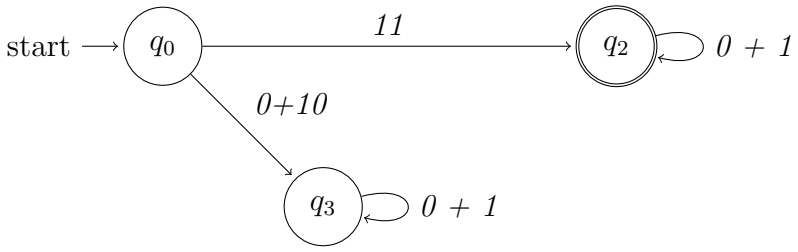
Use o método de eliminação de estados para obter uma expressão regular para a linguagem representada pelo seguinte autómato:



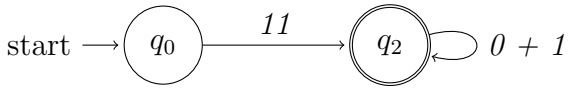
Temos de começar por substituir as transições nos arcos por expressões regulares equivalentes:



Vamos começar por eliminar q_1. Temos uma transição de entrada, vinda de q_0, e duas de saída, dirigidas a q_2 e q_3. Ao eliminar q_1, ficamos então com o seguinte autómato:



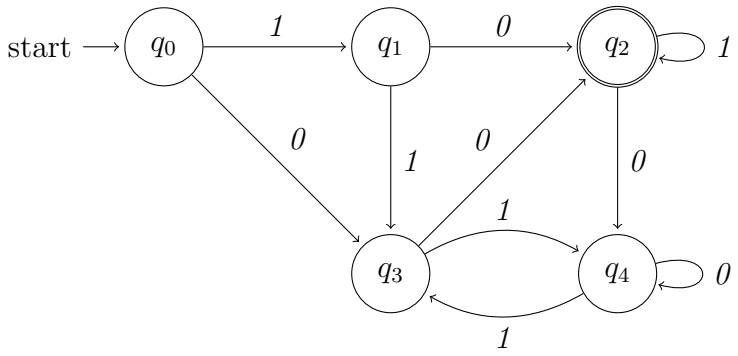
Podemos eliminar q_3 sem necessitar de adicionar qualquer nova transição, uma vez que não existem arcos de saída de q_3.



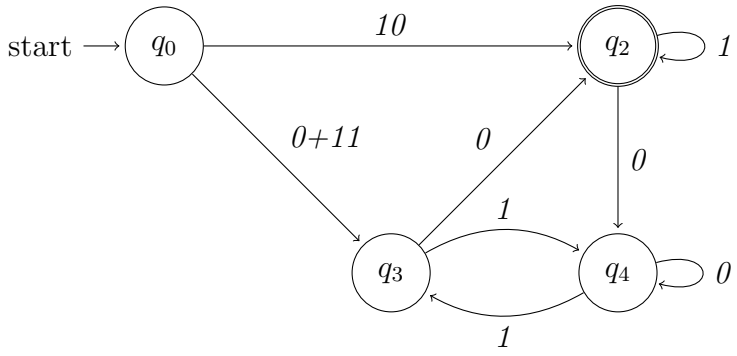
A expressão regular da linguagem será então 11(0+1)*.

Exercício 12

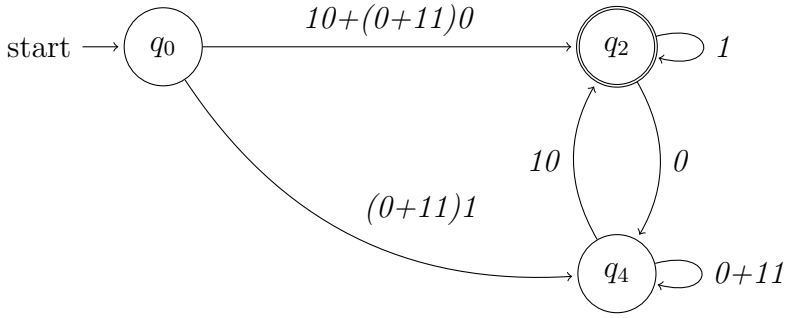
Use o método de eliminação de estados para obter uma expressão regular para a linguagem representada pelo seguinte autómato:



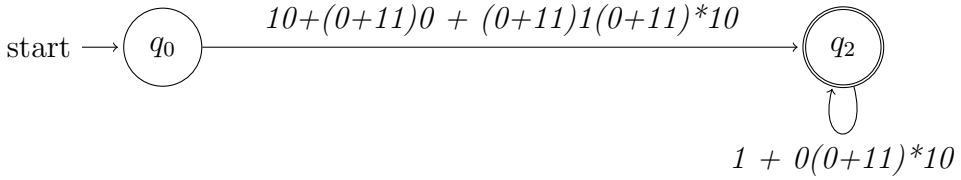
Olhando para este autómato, podemos ver que temos 3 estados a eliminar: q_1, q_3 e q_4. Para eliminar q_1, vamos precisar de duas novas ligações (temos uma de entrada, vinda de q_0, e duas de saída, para q_2 e q_3), ficando assim com o seguinte autómato:



Para eliminar q_3, olhamos para as transições de entrada (q_1 e q_4) e para as transições de saída (q_2 e q_4), ficando assim com 4 novas transições.



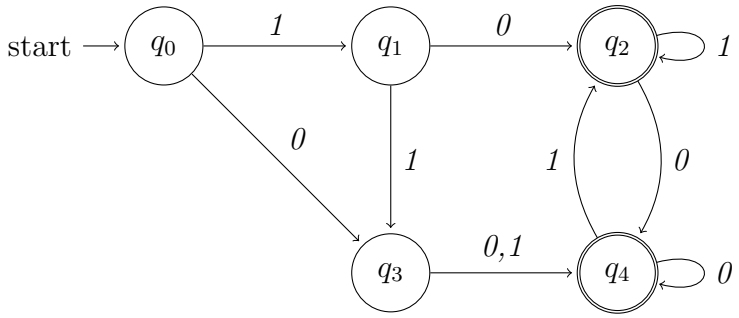
Finalmente, para eliminar q_4, é necessário ver as transições de entrada (q_0 e q_2) e as transições de saída (q_2), ficando assim com duas novas transições.



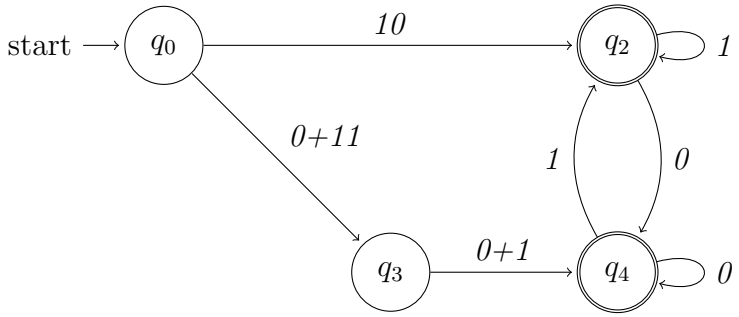
A expressão final será então (10+(0+11)0+(0+11)1(0+11)*10) (1+0(0+11)*10)*.

Exercício 13

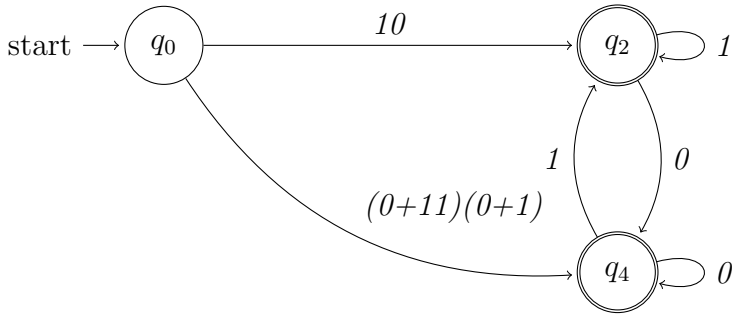
Use o método de eliminação de estados para obter uma expressão regular para a linguagem representada pelo seguinte autómato:



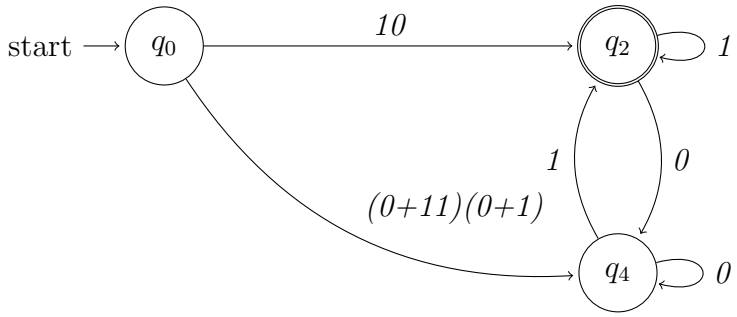
Olhando para o autómato, temos então dois estados a eliminar, q_1 e q_3. Começando por eliminar q_1, ficamos com o seguinte autómato:



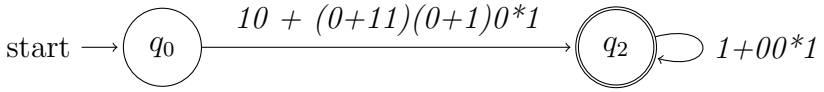
Eliminando agora q_3, ficamos com o seguinte autómato:



Neste momento, estão eliminados todos os estados exceto o estado inicial e estados finais. Para obter a expressão final será necessário reunir as expressões que levam do estado inicial a cada um dos estados finais, pelo que se torna necessário dividir o autómato em dois 'ramos', um para processar cada um dos estados finais. Ficamos assim por um lado com q_2 como estado final:

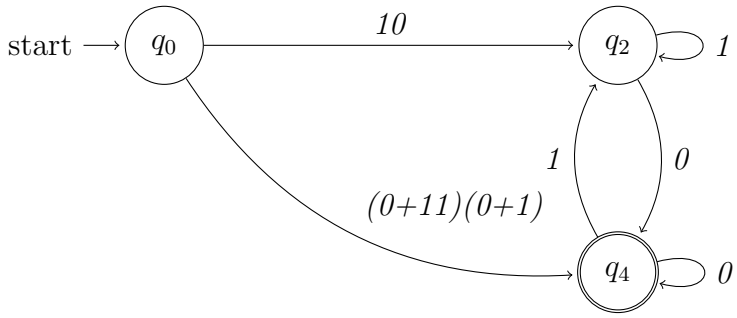


Neste caso, temos de eliminar q_4, ficando assim com o seguinte autómato:

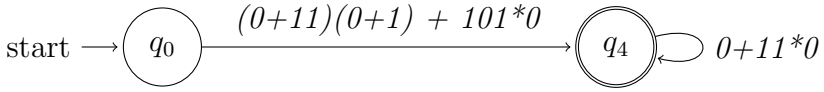


A expressão regular será assim (10 + (0+11)(0+1)0*1) (1+00*1)*.

Por outro lado, ficamos com q_4 como estado final:



Eliminando q_2 ficamos com o seguinte autómato:



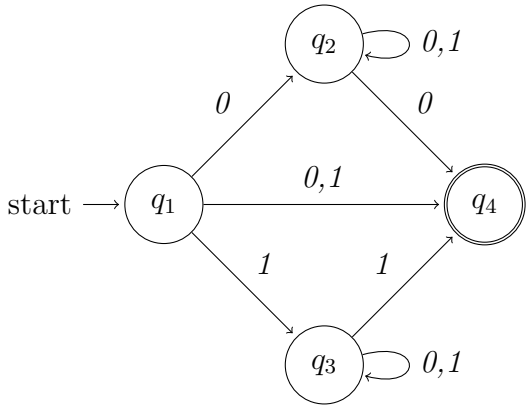
A expressão regular será assim ((0+11)(0+1) + 101*0) (0+11*0)*.

Quando se combinam as duas expressões parciais , obtém-se a expressão para a linguagem:

(10 + (0+11)(0+1)0*1) (1+00*1)* + ((0+11)(0+1) + 101*0) (0+11*0)*

Exercício 14 (TPC 2013/14)

Obtenha expressões regulares para a linguagem definida pelo seguinte autómato, usando o método de construção de caminhos, e o método de eliminação de estados.



Começando pelo método de construção de caminhos, aproveitando que os estados estão já numerados a partir de 1, e dado que temos apenas um estado final, a expressão final desejada será $R^4_{1,4}$, que podemos ir desenvolvendo para ver quais os termos necessários:

$R^4_{1,4} = R^3_{1,4} + R^3_{1,4}(R^3_{4,4})^*R^3_{4,4}$

Desenvolvendo cada um dos termos de ordem 3:

$R^3_{1,4} = R^2_{1,4} + R^2_{1,3}(R^2_{3,3})^*R^2_{3,4}$

$R^3_{4,4} = R^2_{4,4} + R^2_{4,3}(R^2_{3,3})^*R^2_{3,4}$

Desenvolvendo agora cada um dos termos de ordem 2:

$R^2_{1,3} = R^1_{1,3} + R^1_{1,2}(R^1_{2,2})^*R^1_{2,3}$

$R^2_{1,4} = R^1_{1,4} + R^1_{1,2}(R^1_{2,2})^*R^1_{2,4}$

$R^2_{3,3} = R^1_{3,3} + R^1_{3,2}(R^1_{2,2})^*R^1_{2,3}$

$R^2_{3,4} = R^1_{3,4} + R^1_{3,2}(R^1_{2,2})^*R^1_{2,4}$

$R^2_{4,3} = R^1_{4,3} + R^1_{4,2}(R^1_{2,2})^*R^1_{2,3}$

$R^2_{4,4} = R^1_{4,4} + R^1_{4,2}(R^1_{2,2})^*R^1_{2,4}$

Vendo agora os termos de ordem 1:

$R^1_{1,2} = R^0_{1,2} + R^0_{1,1}(R^0_{1,1})^*R^0_{1,2} = 0 + \varepsilon(\varepsilon)^*0 = 0$

$R^1_{1,3} = R^0_{1,3} + R^0_{1,1}(R^0_{1,1})^*R^0_{1,3} = 1 + \varepsilon(\varepsilon)^*1 = 1$

$R^1_{1,4} = R^0_{1,4} + R^0_{1,1}(R^0_{1,1})^*R^0_{1,4} = 0 + 1 + \varepsilon(\varepsilon)^*(0+1) = 0 + 1$

$R^1_{2,2} = R^0_{2,2} + R^0_{2,1}(R^0_{1,1})^*R^0_{1,2} = \varepsilon + 0 + 1 + \emptyset(\varepsilon)^*0 = \varepsilon + 0 + 1$

$R^1_{2,3} = R^0_{2,3} + R^0_{2,1}(R^0_{1,1})^*R^0_{1,3} = \emptyset + \emptyset(\varepsilon)^*1 = \emptyset$

$R^1_{2,4} = R^0_{2,4} + R^0_{2,1}(R^0_{1,1})^*R^0_{1,4} = 0 + \emptyset(\varepsilon)^*(0+1) = 0$
$R^1_{3,2} = R^0_{3,2} + R^0_{3,1}(R^0_{1,1})^*R^0_{1,2} = \emptyset + \emptyset(\varepsilon)^*0 = \emptyset$
$R^1_{3,3} = R^0_{3,3} + R^0_{3,1}(R^0_{1,1})^*R^0_{1,3} = \varepsilon + 0 + 1 + \emptyset(\varepsilon)^*1 = 0 + 1$
$R^1_{3,4} = R^0_{3,4} + R^0_{3,1}(R^0_{1,1})^*R^0_{1,4} = 1 + \emptyset(\varepsilon)^*(0+1) = 1$
$R^1_{4,2} = R^0_{4,2} + R^0_{4,1}(R^0_{1,1})^*R^0_{1,2} = \emptyset + \emptyset(\varepsilon)^*0 = \emptyset$
$R^1_{4,3} = R^0_{4,3} + R^0_{4,1}(R^0_{1,1})^*R^0_{1,3} = \emptyset + \emptyset(\varepsilon)^*1 = \emptyset$
$R^1_{4,4} = R^0_{4,4} + R^0_{4,1}(R^0_{1,1})^*R^0_{1,4} = \varepsilon + \emptyset(\varepsilon)^*(0+1) = \varepsilon$

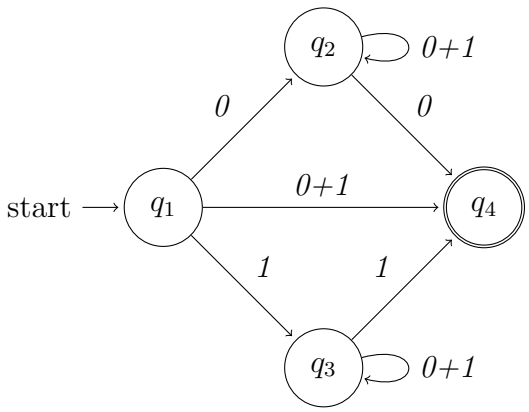
Podemos agora calcular as expressões para os termos de ordem 2:
$R^2_{1,3} = R^1_{1,3} + R^1_{1,2}(R^1_{2,2})^*R^1_{2,3} = 1 + 0(\varepsilon + 0 + 1)^*\emptyset = 1$
$R^2_{1,4} = R^1_{1,4} + R^1_{1,2}(R^1_{2,2})^*R^1_{2,4} = 0 + 1 + 0(\varepsilon + 0 + 1)^*0 = 0 + 1 + 0(0+1)^*0$
$R^2_{3,3} = R^1_{3,3} + R^1_{3,2}(R^1_{2,2})^*R^1_{2,3} = 0 + 1 + \emptyset(\varepsilon + 0 + 1)^*\emptyset = 0 + 1$
$R^2_{3,4} = R^1_{3,4} + R^1_{3,2}(R^1_{2,2})^*R^1_{2,4} = 1 + \emptyset(\varepsilon + 0 + 1)^*0 = 1$
$R^2_{4,3} = R^1_{4,3} + R^1_{4,2}(R^1_{2,2})^*R^1_{2,3} = \emptyset + \emptyset(\varepsilon + 0 + 1)^*\emptyset = \emptyset$
$R^2_{4,4} = R^1_{4,4} + R^1_{4,2}(R^1_{2,2})^*R^1_{2,4} = \varepsilon + \emptyset(\varepsilon + 0 + 1)^*0 = \varepsilon$

Calculamos agora as expressões para os termos de ordem 3:
$R^3_{1,4} = R^2_{1,4} + R^2_{1,3}(R^2_{3,3})^*R^2_{3,4} = 0 + 1 + 0(0+1)^*0 + 1(0+1)^*1$
$R^3_{4,4} = R^2_{4,4} + R^2_{4,3}(R^2_{3,3})^*R^2_{3,4} = \varepsilon + \emptyset(0+1)^*1 = \varepsilon$

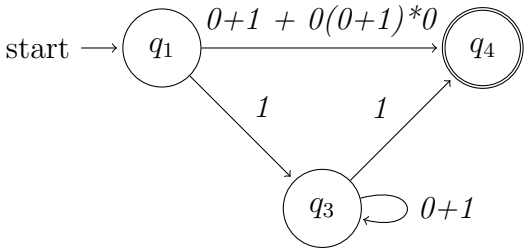
E finalmente podemos calcular a expressão final:
$R^4_{1,4} = R^3_{1,4} + R^3_{1,4}(R^3_{4,4})^*R^3_{4,4} = 0 + 1 + 0(0+1)^*0 + 1(0+1)^*1 + (0 + 1 + 0(0+1)^*0 + 1(0+1)^*1)\,(\varepsilon)^*\varepsilon = 0 + 1 + 0(0+1)^*0 + 1(0+1)^*1$

E temos então a expressão final para a linguagem definida pelo autómato.

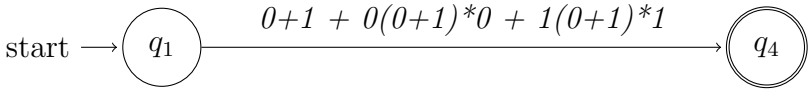
Usando agora o método de eliminação de estados, começamos por transformar as etiquetas das transições em expressões regulares:



De seguida, escolhemos um dos estados q_2 ou q_3 para eliminar primeiro. Eliminando q_2:



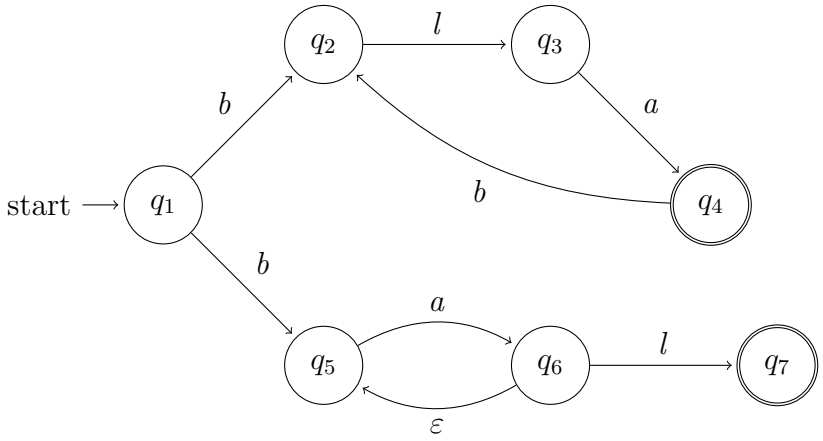
Eliminando agora o estado q_3:



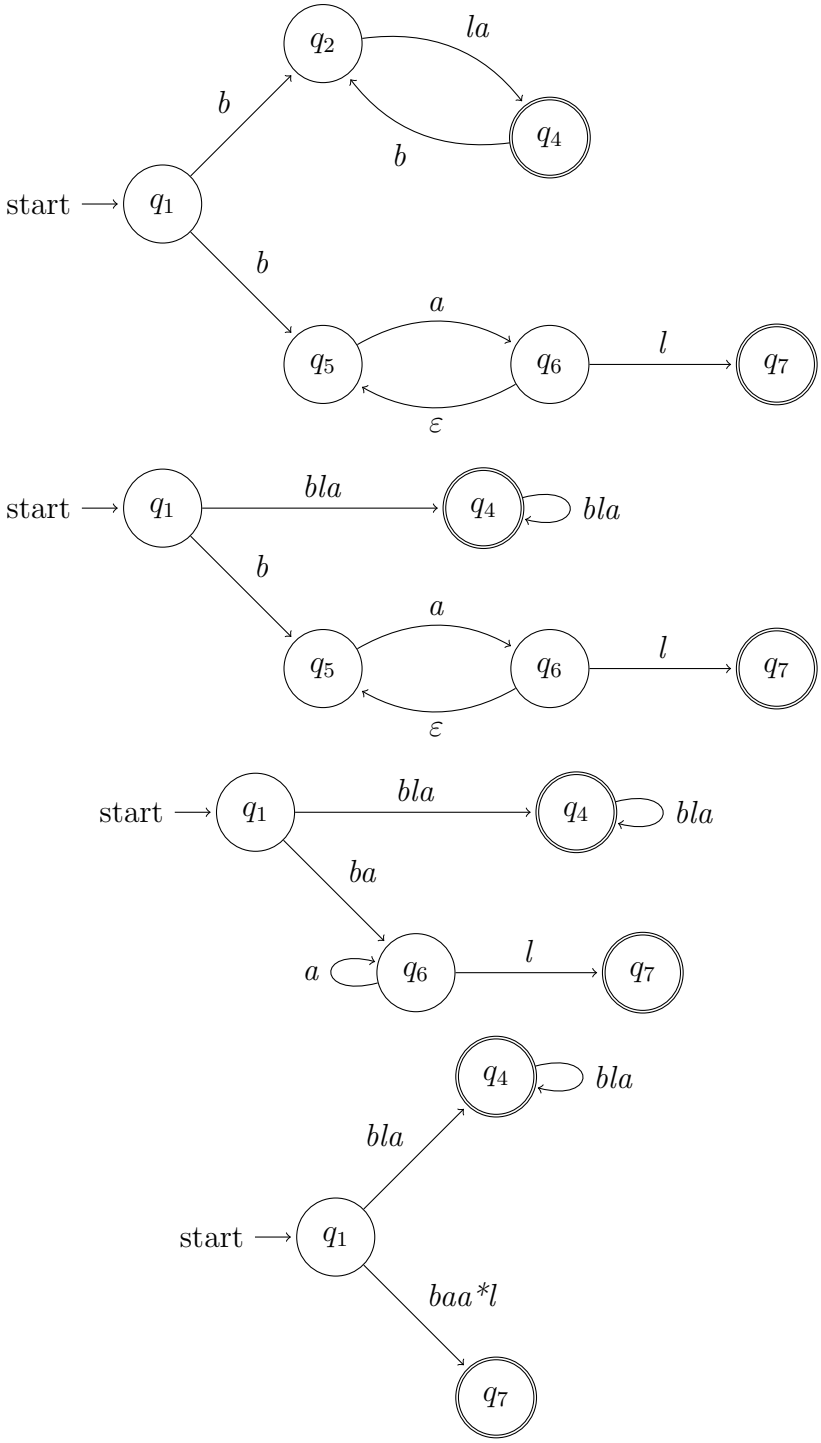
E chegamos então à expressão que nos define a linguagem:
0 + 1 + 0(0+1)*0 + 1(0+1)*1
Como podemos ver, ambos os métodos resultam na mesma expressão, embora o método de eliminação de estados o permita fazer com menos trabalho.

Exercício 15 (Desafio 2014/15)

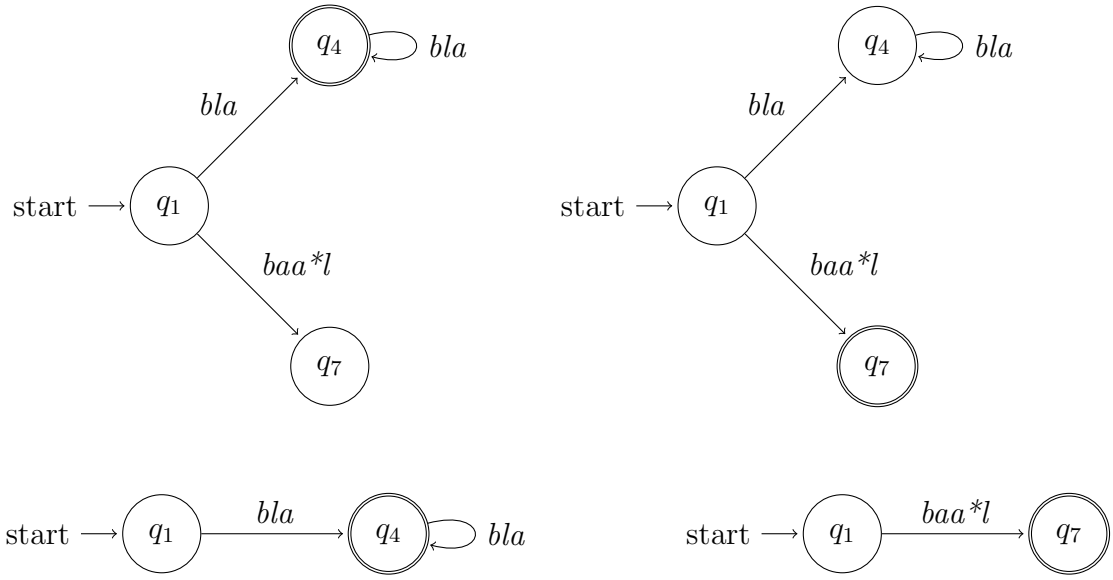
Converta o seguinte autómato (sobre o alfabeto {a, b, l}) na expressão regular equivalente, usando o método de eliminação de estados. Mostre todos os passos intermédios na sua solução.



Como todas as transições têm apenas um símbolo, podemos começar já com a eliminação dos estados. Eliminando os estados q_3, q_2, q_5 e q_6 (por esta ordem), temos então a seguinte sequência de passos:



Dividindo agora o autómato nos dois estados de aceitação, temos:



Ficamos assim com a expressão bla(bla)* para o primeiro caso e baa*l para o segundo caso. Unindo as duas expressões, ficamos então com a seguinte expressão para a linguagem: bla(bla)* + baa*l

Leis Algébricas

As expressões regulares, enquanto representação algébrica de linguagens regulares, permitem ser-lhes aplicado um conjunto de leis algébricas, as quais podem ser úteis na simplificação de expressões, ou para verificar se duas expressões regulares representam a mesma linguagem.

Exercício 16

Indique se as expressões (0+11*0)* e (1*0)* representam a mesma linguagem.

Para verificar se as expressões representam a mesma linguagem, podemos tentar transformar uma na outra. Assim, começando pela primeira expressão, e aplicando sucessivamente leis algébricas das expressões regulares:

$$
\begin{array}{rl}
(0+11^*0)^* & \\
(\varepsilon 0+11^*0)^* & \text{(identidade da concatenação)} \\
((\varepsilon+11^*)0)^* & \text{(distributividade da concatenação)} \\
(1^*0)^* & \varepsilon + 11^* = 1^*
\end{array}
$$

Assim, podemos afirmar que as duas expressões representam a mesma linguagem.

Exercício 17

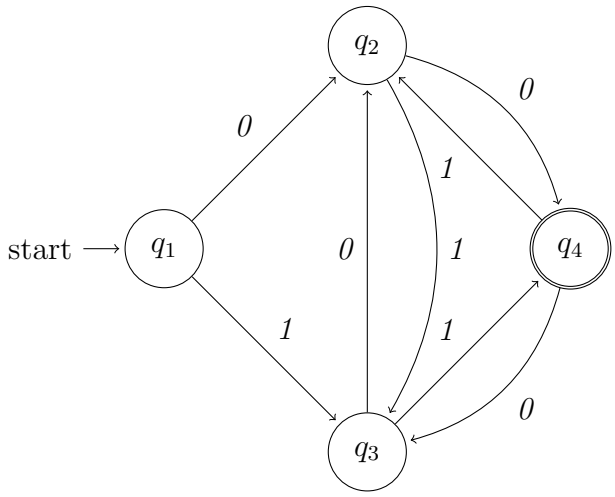
Simplifique a seguinte expressão regular:
(a+b+ε)* + ((a*)* + (b*)*)*

$$
\begin{array}{rll}
(a+b+\varepsilon)^* + (\ (a^*)^* + (b^*)^*\ )^* & & \\
(a+b)^* + (\ (a^*)^* + (b^*)^*\ )^* & & (L+\varepsilon)^* = L^* \\
(a+b)^* + (\ a^* + b^*\ )^* & & (L^*)^* = L^*
\end{array}
$$

Temos agora uma expressão que pode ainda ser simplificada. Olhando para a parte da expressão (a* + b*)*, podemos tentar reescrever de uma forma simplificada: (a+b)* Vamos então tentar ver que cadeias fazem parte desta linguagem: da parte esquerda, são cadeias constituídas por 0 ou mais repetições de sequências de a's ou sequências de b's de comprimento maior ou igual a zero, ou seja, todas as cadeias sobre o alfabeto {a,b}. Do lado direito, são cadeias constituídas por 0 ou mais repetições de a ou de b, ou seja, novamente todas as cadeias sobre o alfabeto. Então, temos uma equivalência, pelo que podemos reescrever a expressão como (a+b)* + (a+b)* , a qual pode ser reescrita como (a+b)* (idempotência da união).

Exercício Proposto 1 Escreva uma expressão regular para reconhecer cadeias sobre o alfabeto {a,b} em que não existem subcadeias de 2 a's seguidas de 2 b's. Por exemplo, a cadeia abba pertence à linguagem, mas a cadeia baaabba já não pertence.

Exercício Proposto 2 Obtenha uma expressão regular para a linguagem representada pelo seguinte autómato.


q_2
0
0
1
start
q_1
0
1
q_4
1
1
0
q_3

Capítulo 6

Propriedades das Linguagens Regulares

Neste capítulo exploram-se algumas propriedades das lingugens regulares.

6.1 Propriedades de Fecho

Referem-se aqui um conjunto de operações que quando aplicadas sobre linguagens regulares preservam a regularidade da linguagem resultado.

Alguns exemplos dessas operações são a Concatenação, Fecho, União, Intersecção, Diferença, Complemento, Reverso, Homomorfismo e Homomorfismo Inverso.

Para os operadores de Complemento e Reverso, as operações podem ser feitas diretamente no DFA da linguagem:

- Para o complemento, basta trocar todos os estados de aceitação para não-aceitação e vice-versa, passando assim o autómato a aceitar todas as cadeias que antes rejeitava. Note que esta operação tem de ser realizada no DFA completo (de forma a que o estado morto, se existir, passe a ser um estado de aceitação do novo autómato).

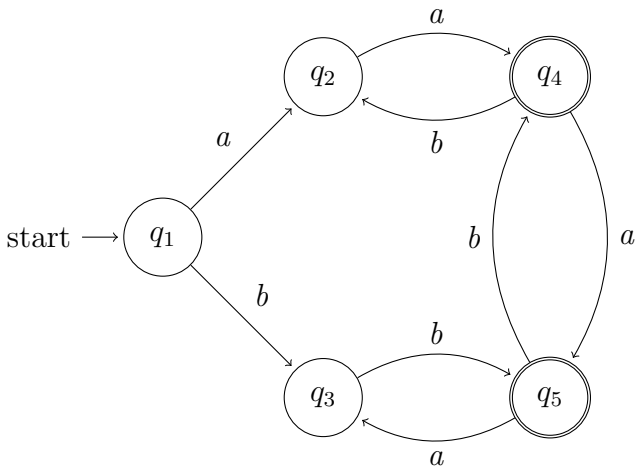
- Para o reverso, deve-se inverter a direção de todas as setas de transições, acrescentar um novo estado, que será o estado inicial do novo auómato, com transições ε para os estados de aceitação do autómato original (os quais devem deixar de ser estados de aceitação), e tornando o estado inicial do autómato original como único estado de aceitação do novo autómato.

Para os operadores de União, Interseção e Diferença, as operações podem ser feitas a partir dos DFAs das linguagens originais, fazendo o produto

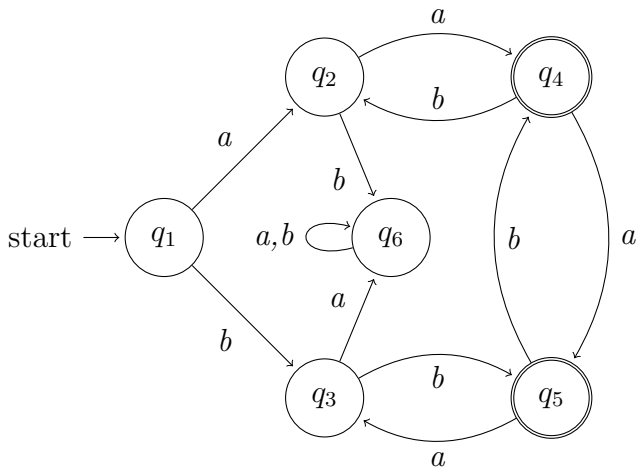
dos dois autómatos, e verificando quais os estados de aceitação para o novo autómato. No caso da união, os estados de aceitação devem incluir os estados de aceitação tanto de um autómato como de outro; para a interseção, apenas aqueles que incluam estados de aceitação de ambos os autómatos originais; para a diferença, apenas os estados que sejam de aceitação para o autómato da primeira linguagem mas não para o da segunda linguagem.

Exercício 1

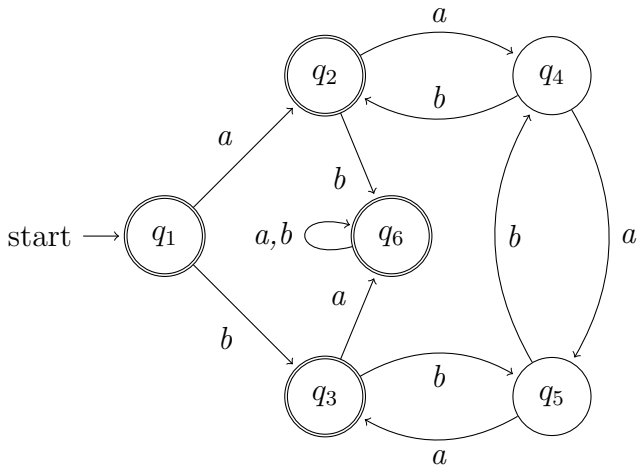
Obtenha o autómato para o complemento da linguagem representada pelo seguinte autómato.



Para obter o complemento da linguagem, é necessário estar a trabalhar sobre o DFA completo, para garantir que qualquer 'estado morto', caso exista, está presente no autómato (uma vez que este será um estado de aceitação no autómato para o complemento da linguagem). Assim, acrescentamos o estado morto, para ficarmos com o DFA completo.



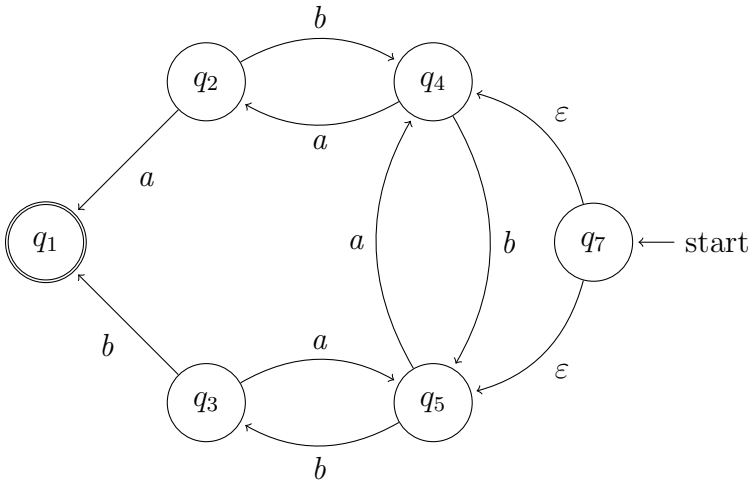
Agora, necessitamos apenas de trocar a aceitação dos estados, fazendo com que as cadeias aceites pela linguagem original deixem de o ser, e todas as cadeias não aceites pela linguagem original passem a ser aceites. Ficamos então com o seguinte DFA:



Exercício 2

Obtenha o autómato para o reverso da linguagem representada pelo autómato do exercício anterior.

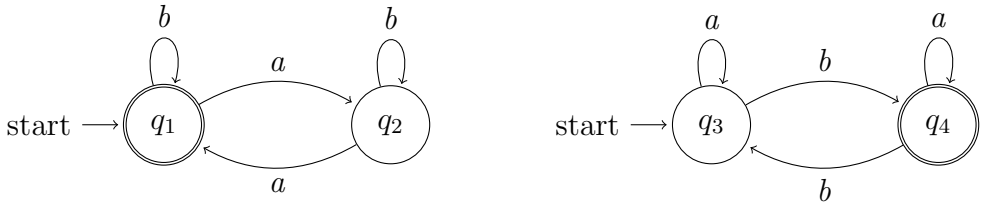
Para obter uma solução, necessitamos então de realizar 3 alterações ao autómato original: acrescentar um novo estado inicial com transições ε para cada um dos estados finais originais; fazer com que o estado inicial do autómato original passe a ser o único estado de aceitação do novo autómato; e inverter o sentido de todas as transições. Note-se que para obter o reverso não é necessário trabalhar com o DFA completo – uma vez que o estado morto do DFA original só tem transições de entrada e nenhuma de saída, invertendo o sentido das transições ficaríamos com um estado só com transições de saída mas nenhuma de entrada, ou seja, um estado não atingível. Fazendo as alterações necessárias, ficamos então com o seguinte autómato como resultado:



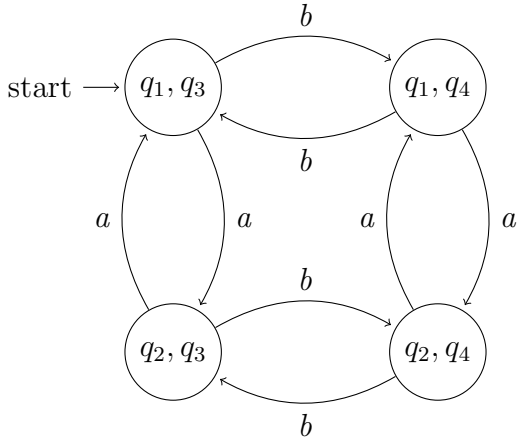
Exercício 3

Considere as linguagens A e B sobre o alfabeto {a,b}, em que A representa cadeias com número par de a's, e B representa cadeias com número ímpar e b's. Obtenha autómatos para as linguagens $C = A \cup B$, $D = A \cap B$, $E = A - B$ e $F = B - A$.

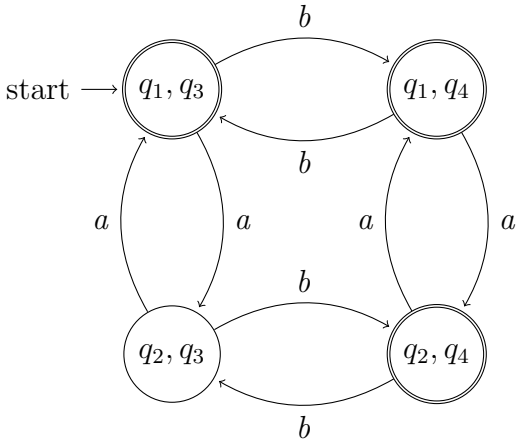
Vamos começar por representar cada um dos DFAs para as linguagens A e B. Ficamos então com os seguintes dois autómatos:



Ao fazer o produto destes dois autómatos, ficamos com um autómato com os estados compostos por um estado de cada um dos autómatos originais, sendo as transições calculadas com base nos estados de destino de cada um dos autómatos. O estado inicial será o estado composto pelos estados iniciais de cada um dos autómatos. Assim, ficamos com o seguinte autómato produto:

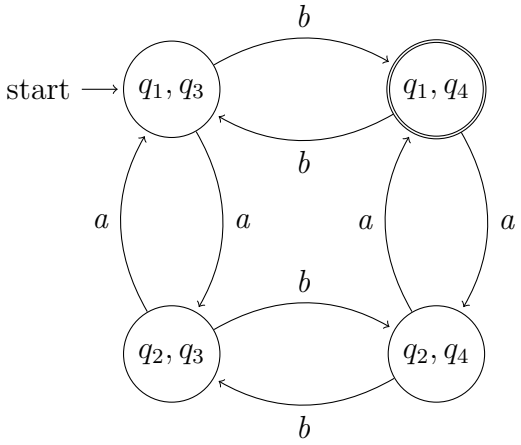


Para obter cada uma das linguagens pretendidas, será apenas necessário indicar quais os estados finais em cada caso. No caso de $C = A \cup B$, os estados finais serão todos aqueles que incluam estados que eram finais em A ou em B, ou seja, todos os estados que contenham q_1 (número par de a's) ou q_4 (número ímpar de b's). Assim, o autómato para C será:

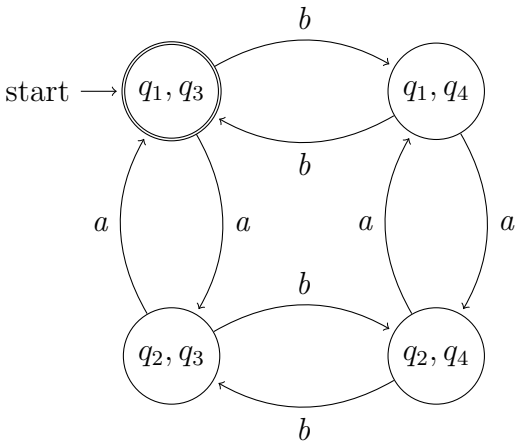


Repare-se que o único estado não final é aquele composto por q_2 e q_3, ou seja, o estado que representa cadeias em que o número de a's é ímpar e o número de b's é par. Os outros estados, representando número par de a's e/ou número ímpar de b's são todos estados de aceitação.

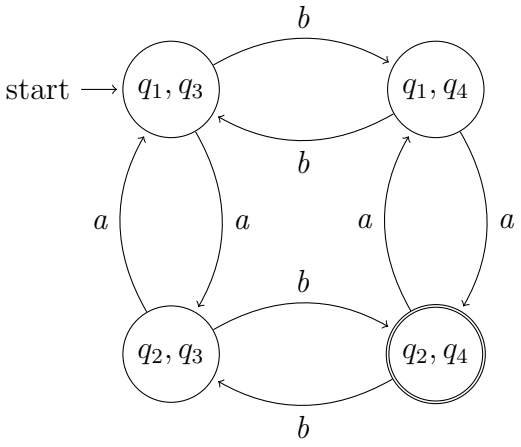
Para $D = A \cap B$, será necessário ter como estados finais apenas aqueles compostos por estados finais tanto em A como em B. Neste caso, será o estado composto por q_1 e q_4, resultando no seguinte autómato:



Para a linguagem $E = A - B$, será necessário que os estados finais sejam aqueles que são finais em A mas não o sejam em B. Neste caso, serão finais os estados que contenham q_1, mas não q_4 (ou seja, estados em que o número de a's seja par, mas garantindo não aceitar cadeias quando o número de b's é ímpar), resultando no seguinte autómato:



Finalmente, para a linguagem $F = B - A$, será necessário fazer algo semelhante ao que foi feito para E, mas em que neste caso os estados finais serão aqueles que contêm q_4 mas não q_1:

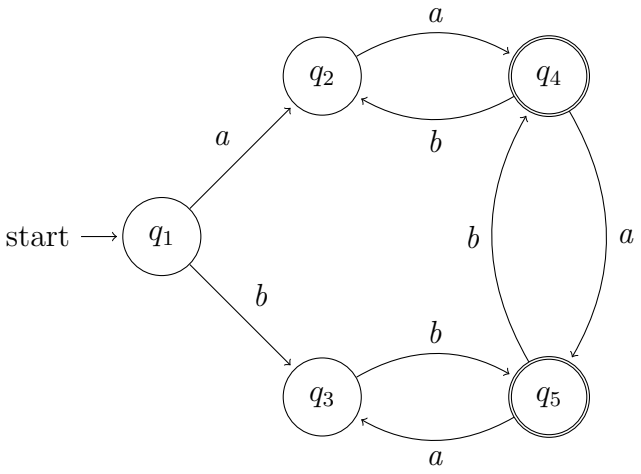


6.2 Propriedades de Decisão

Referem-se aqui algumas questões que podem ser colocadas relativamente a linguagens regulares, nomeadamente determinar se uma dada linguagem é vazia, determinar se uma cadeia pertence a uma linguagem, e determinar se duas representações de linguagens correspondem à mesma linguagem (para esta última, vamos olhar para a possibilidade de minimização de autómatos).

Exercício 1

Indique, das seguintes cadeias, quais pertencem à linguagem representada pelo autómato abaixo: abba, aaba, baba, abab, bbab, baab, baaa, aaab, bbba. Indique ainda se a linguagem representada é vazia.



Para verificar quais as cadeias que pertencem à linguagem, basta verificar quais são aceites pelo autómato, simulando o comportamento deste à medida que processa a cadeia de entrada. Neste caso, são aceites as cadeias aaba, bbab, aaab e bbba. As cadeias abba, baba, abab, baab e baab não são aceites pelo autómato, e portanto não pertencem à linguagem. Podemos ver, por exemplo, que o *percurso* do autómato para a cadeia aaba será $q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_4 \xrightarrow{b} q_2 \xrightarrow{a} q_4$; sendo q_4 um estado final, a cadeia é aceite. Já para a cadeia abba, depois de transitar de q_1 para q_2 com o primeiro a, o autómato 'morre' pois não existe transição possível de q_2 com b.

Para a linguagem ser vazia, não poderiam existir no autómato estados finais alcançáveis a partir do estado inicial. Neste caso, existem estados finais alcançáveis a partir do estado inicial, pelo que a linguagem não é vazia (como, aliás, se verificou pelo facto de existirem cadeias pertencentes à linguagem).

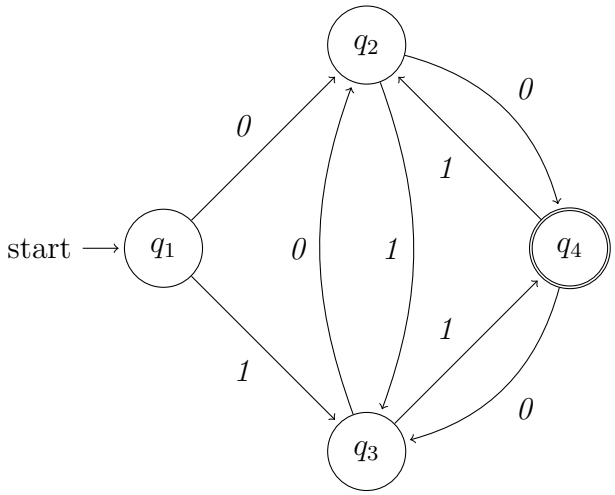
6.2.1 Minimização de Autómatos

A minimização de autómatos serve não só para encontrar uma representação mais compacta para determinada linguagem / autómato, mas pode também

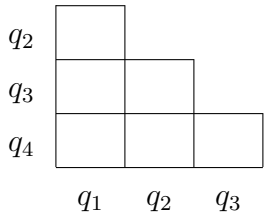
ser usada para determinar se duas representações correspondem à mesma linguagem, verificando se os dois autómatos minimizados são estruturalmente equivalentes (caso o sejam, correspondem à mesma linguagem).

A minimização de autómatos é feita recorrendo a uma tabela triangular de equivalências de estados, a qual é preenchida iterativamente com condições de equivalência, e impossibilidades de equivalência de estados.

Vamos ver um exemplo de construção desta tabela para o seguinte autómato:



Começa-se por desenhar uma tabela triangular como apresentada abaixo, em que quer num eixo quer no outro se representam os estados. As células representam a equivalência (ou não) entre os estados cuja interseção resulta nessa célula. Por exemplo, a célula que se encontra na interseção entre q_2 e q_3 vai conter informação sobre equivalência entre esses dois estados. Repare-se que não é representada a linha diagonal da tabela, e portanto no eixo vertical começamos pelo segundo estado, e no eixo horizontal não representamos o último estado.



O próximo passo será assinalar na tabela quais os estados que à partida não são equivalentes, que são as intereseções entre um estado que seja de aceitação e um estado que não seja de aceitação. Neste caso, temos apenas um estado de aceitação, q_4, e marcamos então com um X todas as células

que resultam da interseção de q_4 com qualquer um dos outros 3 estados do autómato:

q_2			
q_3			
q_4	X	X	X
	q_1	q_2	q_3

De sequida, preenchem-se as células ainda vazias com as condições necessárias para que os estados sejam equivalentes. Estas condições são a equivalência dos estados de destino de cada um dos estados para cada um dos símbolos da linguagem. Neste caso, temos apenas os símbolos 0 e 1, pelo que existem apenas duas condições em cada célula. Por exemplo, para a célula que resulta da interseção de q_1 com q_2, temos a condição de $q_2 = q_4$ uma vez que q_1 transita para q_2 com 0, e q_2 transita para q_4 com 0. Depois de preencher as células vazias com estas condições, ficamos então com a seguinte tabela:

q_2	$q_2 = q_4$ $q_3 = q_3$		
q_3	$q_2 = q_2$ $q_3 = q_4$	$q_4 = q_2$ $q_3 = q_4$	
q_4	X	X	X
	q_1	q_2	q_3

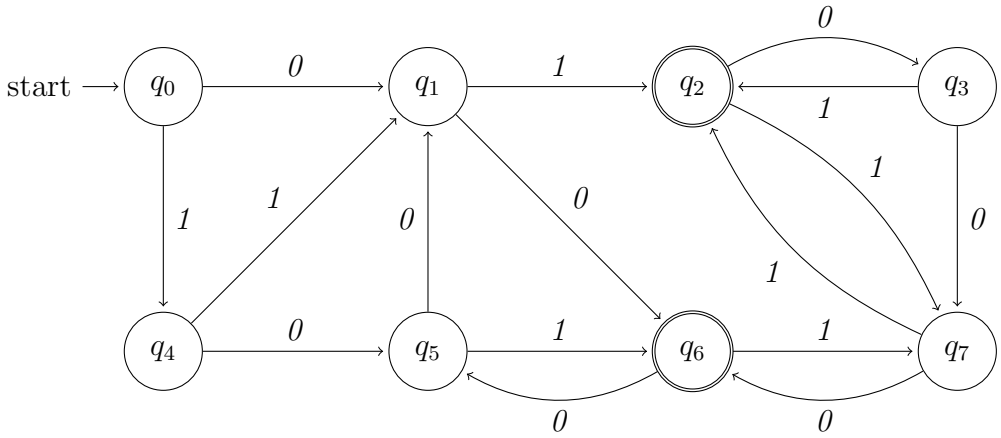
O próximo passo é ir verificando, de uma forma sistemática (por exemplo, linha por linha ou coluna por coluna, repetindo o processo até não haver mais alterações), quais os estados com pelo menos uma condição de equivalência falsa, o que leva a que os estados não possam ser considerados equivalentes, marcando então a célula com um X. Neste caso, logo na primeira iteração, temos condições em todas as 3 células que envolvem q_4, pelo que nenhum par de estados é equivalente, ficando então com a seguinte tabela:

Podemos então concluir que o autómato apresentado é já o autómato mínimo para a linguagem que representa, não podendo ser simplificado.

q_2	X		
q_3	X	X	
q_4	X	X	X
	q_1	q_2	q_3

Exercício 2 (Desafio 2014/15)

Obtenha o DFA mínimo para a linguagem representada pelo seguinte autómato.



Para obter o DFA mínimo, usamos a tabela de equivalência de estados. Na tabela abaixo, temos já marcados com um 'X' as células onde se intersetam um estado final com um estado não-final. Neste caso, tendo q_2 e q_6 como estados finais, serão marcadas todas as células de interseção entre um destes estados e um dos estados não finais.

O próximo passo será então a colocação das condições de equivalência necessárias para que cada par de estados possa ser considerado equivalente. Tendo dois símbolos na linguagem (0 e 1), cada célula terá duas condições (correspondentes à equivalência dos estados de destino das transições com cada símbolo da linguagem). Por questões de organização da tabela, e de forma a facilitar a leitura da mesma, representa-se sempre na linha de cima de cada célula a condição referente ao símbolo 0 e na linha de baixo a condição referente ao símbolo 1. Adicionalmente, e em cada condição, o estado à esquerda é o estado de destino considerando o estado na vertical, e o estado à direita é o estado de destino considerando o estado na horizontal. Assim, a título de exemplo, na célula de interseção entre q_3 e q_5, na linha de cima

	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_1							
* q_2	X	X					
q_3			X				
q_4			X				
q_5			X				
* q_6	X	X		X	X	X	
q_7			X				X

temos $q_7 = q_1$, sendo que q_7 é o estado de destino da transição com o símbolo 0 a partir de q_3, e q_1 é o estado de destino da transição com o símbolo 1 a partir de q_5. A tabela abaixo mostra então este preenchimento.

	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_1	$q_1 = q_6$ $q_4 = q_2$						
* q_2	X	X					
q_3	$q_1 = q_7$ $q_4 = q_2$	$q_6 = q_7$ $q_2 = q_2$	X				
q_4	$q_1 = q_5$ $q_4 = q_1$	$q_6 = q_5$ $q_2 = q_1$	X	$q_7 = q_5$ $q_2 = q_1$			
q_5	$q_1 = q_1$ $q_4 = q_6$	$q_6 = q_1$ $q_2 = q_6$	X	$q_7 = q_1$ $q_2 = q_6$	$q_5 = q_1$ $q_1 = q_6$		
* q_6	X	X	$q_3 = q_5$ $q_7 = q_7$	X	X	X	
q_7	$q_1 = q_6$ $q_4 = q_2$	$q_6 = q_6$ $q_2 = q_2$	X	$q_7 = q_6$ $q_2 = q_2$	$q_5 = q_6$ $q_1 = q_2$	$q_1 = q_6$ $q_6 = q_2$	X

O próximo passo será o de marcar com um X as células com condições de equivalência já marcadas com um X. Para isso, vamos percorrer a tabela (neste caso, foi percorrida coluna a coluna, da esquerda para a direita, e em cada coluna de cima para baixo), e verificar o estado das células indicadas

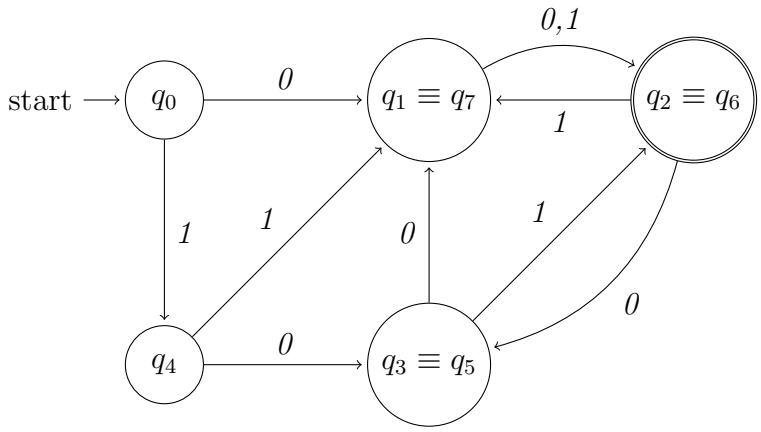
pelas condições de equivalência; se pelo menos uma dessas células estiver marcada com um X, então marcamos também a célula atual com um X, significando que também os estados que se intersetam nesta célula não serão equivalentes.

q_1	X						
* q_2	X	X					
q_3	X	X	X				
q_4	$q_1 = q_5$ $q_4 = q_1$	X	X	X			
q_5	X	X	X	$q_7 = q_1$ $q_2 = q_6$	X		
* q_6	X	X	$q_3 = q_5$ $q_7 = q_7$	X	X	X	
q_7	X	$q_6 = q_6$ $q_2 = q_2$	X	X	X	X	X
	q_0	q_1	q_2	q_3	q_4	q_5	q_6

Como podemos ver, a grande maioria das células foi já marcada com um X. Na próxima iteração deste processo obtemos a tabela abaixo.

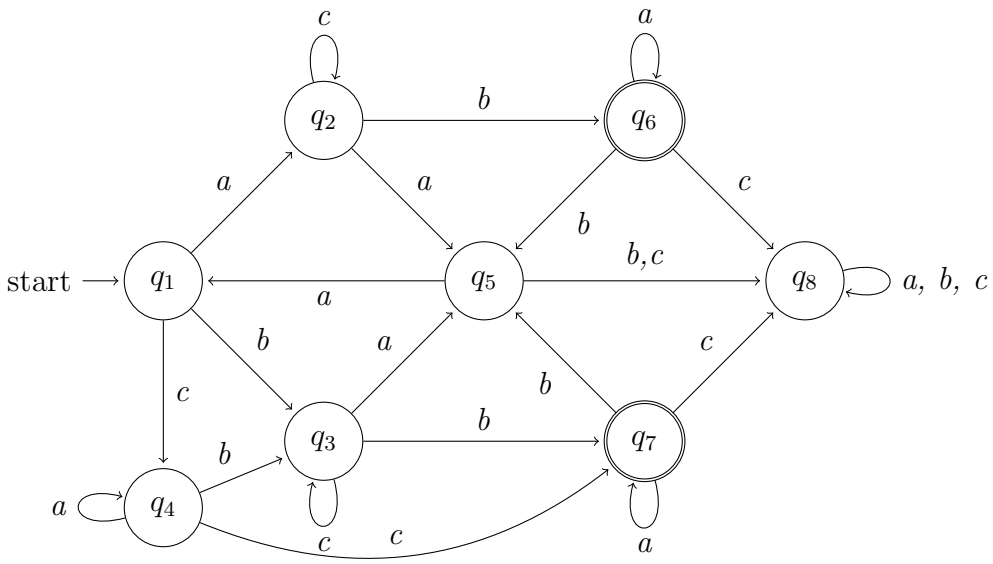
Agora já não existem mais alterações a realizar à tabela (novas passagens deixam a tabela inalterada), pelo que podemos verificar que o autómato original pode ser simplificado. As células que não estão marcadas com um X representam a equivalência dos estados respetivos. Por exemplo, a célula na interseção entre q_1 e q_7 não está marcada com um X, o que significa que q_1 e q_7 são dois estados equivalentes. Temos então as seguintes equivalências:
$q_1 \equiv q_7$
$q_2 \equiv q_6$
$q_3 \equiv q_5$
Podemos agora desenhar o novo DFA para a mesma linguagem, mas já com menos estados que o DFA original:

q_1	X						
* q_2	X	X					
q_3	X	X	X				
q_4	X	X	X	X			
q_5	X	X	X	$q_7 = q_1$ $q_2 = q_6$	X		
* q_6	X	X	$q_3 = q_5$ $q_7 = q_7$	X	X	X	
q_7	X	$q_6 = q_6$ $q_2 = q_2$	X	X	X	X	X
	q_0	q_1	q_2	q_3	q_4	q_5	q_6



Exercício 3

Simplifique o automato seguinte.



Para simplificar o autómato, vamos então usar a tabela de equivalência de estados. Na tabela abaixo, temos já marcados com um 'X' as células onde se intersetam um estado final com um estado não-final.

	q_1	q_2	q_3	q_4	q_5	q_6	q_7
q_2							
q_3							
q_4							
q_5							
* q_6	X	X	X	X	X		
* q_7	X	X	X	X	X		
q_8						X	X

Neste caso, existem dois estados finais (q_6 e q_7), sendo que nas células de interseção entre um destes dois estados e dos um estado não finais é então marcada com um X.

O próximo passo será então o de colocar em cada célula as condições de equivalência necessárias para que os estados dessa célula sejam equivalentes. Como temos 3 símbolos na linguagem (a, b e c), cada célula terá 3 condições, como mostra a próxima iteração da tabela. De forma a simplificar

a interpretação, as linhas correspondem aos três símbolos, a, b e c, respetivamente, encontrando-se na esquerda das igualdades o estado de destino com a transição a partir do estado indicado na coluna respetiva, e no lado direito da igualdade o estado de destino com a transição a partir do estado indicado na linha respetiva.

	q_1	q_2	q_3	q_4	q_5	q_6	q_7
q_2	$q_2 = q_5$ $q_3 = q_6$ $q_4 = q_2$						
q_3	$q_2 = q_5$ $q_3 = q_7$ $q_4 = q_3$	$q_5 = q_5$ $q_6 = q_7$ $q_2 = q_3$					
q_4	$q_2 = q_4$ $q_3 = q_3$ $q_4 = q_7$	$q_5 = q_4$ $q_6 = q_3$ $q_2 = q_7$	$q_5 = q_4$ $q_7 = q_3$ $q_3 = q_7$				
q_5	$q_2 = q_1$ $q_3 = q_8$ $q_4 = q_8$	$q_5 = q_1$ $q_6 = q_8$ $q_2 = q_8$	$q_5 = q_1$ $q_7 = q_8$ $q_3 = q_8$	$q_4 = q_1$ $q_3 = q_8$ $q_7 = q_8$			
* q_6	X	X	X	X	X		
* q_7	X	X	X	X	X	$q_6 = q_7$ $q_5 = q_5$ $q_8 = q_8$	
q_8	$q_2 = q_8$ $q_3 = q_8$ $q_4 = q_8$	$q_5 = q_8$ $q_6 = q_8$ $q_2 = q_8$	$q_5 = q_8$ $q_7 = q_8$ $q_3 = q_8$	$q_4 = q_8$ $q_3 = q_8$ $q_7 = q_8$	$q_1 = q_8$ $q_8 = q_8$ $q_8 = q_8$	X	X

Após uma primeira iteração, são marcadas com um X todas as células em que pelo menos uma das condições de equivalência está também marcada com um X. Neste caso, foi optou-se por percorrer a tabela coluna a coluna, de cima para baixo, da esquerda para a direita. Ficamos assim com a seguinte tabela após esta primeira iteração:

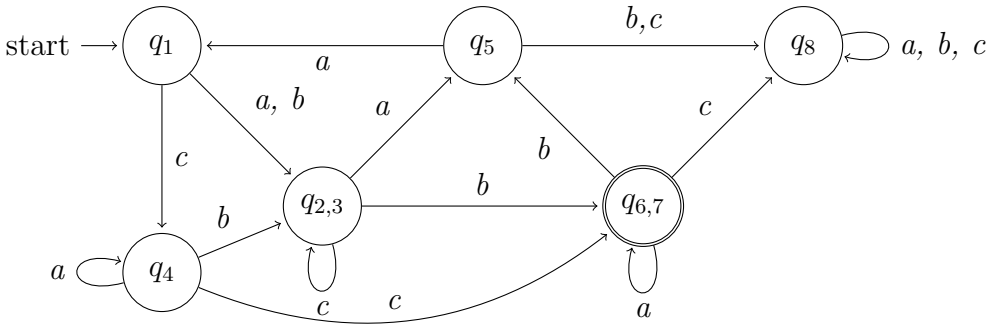
Após mais uma iteração, obtemos a seguinte tabela, que será já a tabela final (iterações seguintes não produzem mais alterações na tabela). Aqui podemos ver que apenas duas células não estão preenchidas com X, indicando

	q_1	q_2	q_3	q_4	q_5	q_6	q_7
q_2	X						
q_3	X	$q_5 = q_5$ $q_6 = q_7$ $q_2 = q_3$					
q_4	X	X	X				
q_5	X	X	X	X			
* q_6	X	X	X	X	X		
* q_7	X	X	X	X	X	$q_6 = q_7$ $q_5 = q_5$ $q_8 = q_8$	
q_8	$q_2 = q_8$ $q_3 = q_8$ $q_4 = q_8$	X	X	X	$q_1 = q_8$ $q_8 = q_8$ $q_8 = q_8$	X	X

que os estados que deram origem a essas células são equivalentes. Neste caso, podemos constatar que q_2 e q_3 são equivalentes entre si, e que q_6 e q_7 são também equivalentes entre si.

	q_1	q_2	q_3	q_4	q_5	q_6	q_7
q_2	X						
q_3	X	$q_5 = q_5$ $q_6 = q_7$ $q_2 = q_3$					
q_4	X	X	X				
q_5	X	X	X	X			
* q_6	X	X	X	X	X		
* q_7	X	X	X	X	X	$q_6 = q_7$ $q_5 = q_5$ $q_8 = q_8$	
q_8	X	X	X	X	X	X	X

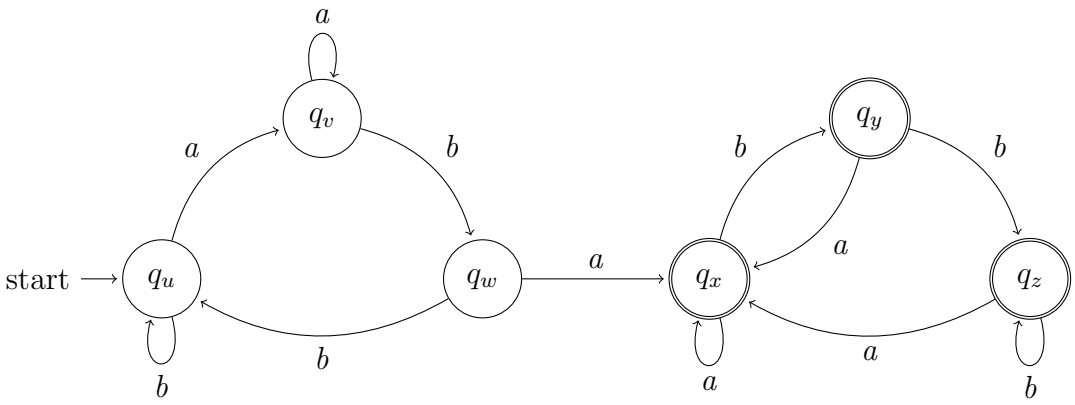
Então, o DFA mínimo para a linguagem terá apenas 6 estados, em comparação com os 8 do autómato inicial.



Exercício 4

Recorde o autómato obtido no **Exercício 5** do Capítulo 3, e minimize-o.

Recordando o DFA obtido no exercício, com os estados renomeados para simplificação:



Começamos então por construir a tabela de equivalência de estados, marcando já os estados não equivalentes (células na interseção de um estado final com um estado não final):

	q_u	q_v	q_w	q_x	q_y
q_v					
q_w					
* q_x	X	X	X		
* q_y	X	X	X		
* q_z	X	X	X		

Colocamos agora as condições de equivalência em cada célula não preenchida com um X:

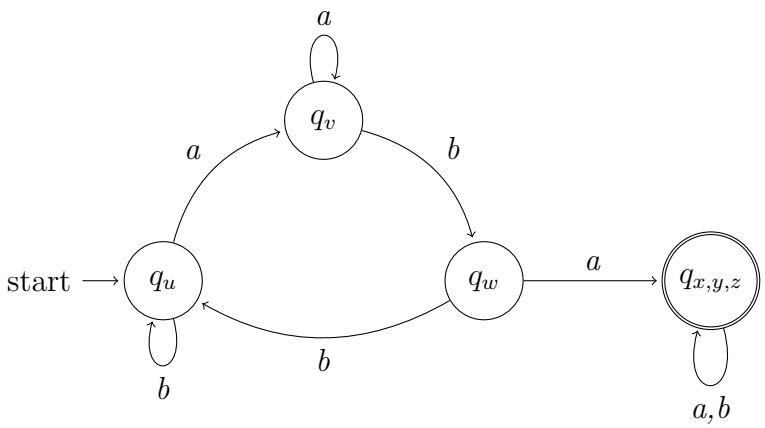
	q_u	q_v	q_w	q_x	q_y
q_v	$q_v = q_v$ $q_u = q_w$				
q_w	$q_v = q_x$ $q_u = q_u$	$q_v = q_x$ $q_w = q_u$			
* q_x	X	X	X		
* q_y	X	X	X	$q_x = q_x$ $q_y = q_z$	
* q_z	X	X	X	$q_x = q_x$ $q_y = q_z$	$q_x = q_x$ $q_z = q_z$

Após duas iterações do processo (marcar células com X quando pelo menos uma das células referenciadas nas condições de equivalência está já marcada com X), chegámos à seguinte tabela:

Temos então as seguintes equivalências: $q_x \equiv q_y \equiv q_z$.

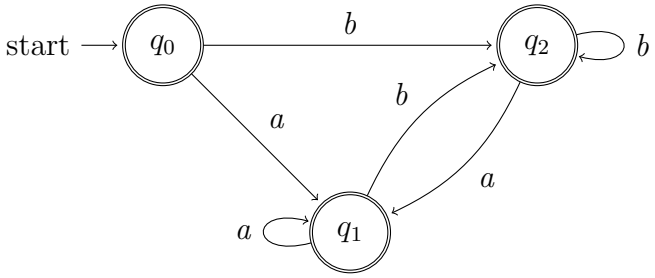
Podemos agora desenhar o novo DFA minimizado:

	q_u	q_v	q_w	q_x	q_y
q_v	X				
q_w	X	X			
* q_x	X	X	X		
* q_y	X	X	X	$q_x = q_x$ $q_y = q_z$	
* q_z	X	X	X	$q_x = q_x$ $q_y = q_z$	$q_x = q_x$ $q_z = q_z$



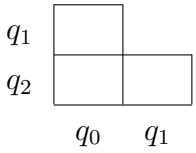
Exercício 5 (Exercício 2015/16)

Minimize o seguinte DFA, desenhando o DFA resultante, e indique qual a expressão regular que representa a linguagem por ele aceite.



Começamos então por construir a tabela de equivalências, cortando as células que representam interseção entre um estado final e um estado não final. Note-se que, dado que os três estados são estados finais, não há nenhuma célula cortada inicialmente (tabela da esquerda).

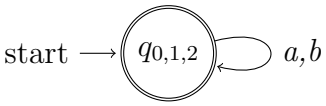
Colocamos agora as condições de equivalências necessárias para cada uma das células da tabela (tabela da direita).



q_1	$q_1 = q_1$ $q_2 = q_2$	
q_2	$q_1 = q_1$ $q_2 = q_2$	$q_1 = q_1$ $q_2 = q_2$
	q_0	q_1

Como podemos verificar, todas as condições de equivalência são verdadeiras, pelo que todas as células ficam por cruzar, indicando que todos os pares de estados são equivalentes (ou seja, $q_0 \equiv q_1 \equiv q_2$).

Podemos então desenhar o seguinte autómato, tendo em conta as equivalências encontradas:



Em relação à expressão regular para a linguagem, dado que temos apenas um estado, o qual é final, a expressão será simplesmente (a+b)*.

Exercício 6

Verifique se os seguintes dois autómatos representam a mesma linguagem.

	a	b
$\rightarrow q_w$	q_x	q_y
q_x	q_y	q_z
q_y	q_z	q_w
$*\ q_z$	q_w	q_x

	a	b
$\rightarrow q_0$	q_2	q_6
$*\ q_1$	q_0	q_4
q_2	q_3	q_7
q_3	q_1	q_5
q_4	q_6	q_1
q_5	q_4	q_3
q_6	q_7	q_0
$*\ q_7$	q_5	q_2

Para resolver o problema podemos optar por duas estratégias:

- Minimizar cada um dos autómatos e comparar depois os resultados obtidos, verificando se os dois autómatos minimizados são iguais (a menos de nomes dos estados).

- Juntar os estados dos dois autómatos na mesma tabela de minimização, e verificar se os estados iniciais dos dois autómatos são equivalentes.

Vamos usar este segundo método, começando então por apresentar a tabela de equivalência de estados contendo os estados dos dois autómatos (dado o tamanho da tabela, apresenta-se já preenchida com os estados não equivalentes marcados, e com as condições de equivalência para as restantes células).

	q_w	q_x	q_y	q_z	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_x	$q_x = q_y$ $q_y = q_z$										
q_y	$q_x = q_z$ $q_y = q_w$	$q_y = q_z$ $q_z = q_w$									
* q_z	X	X	X								
q_0	$q_x = q_2$ $q_y = q_6$	$q_y = q_2$ $q_z = q_6$	$q_z = q_2$ $q_w = q_6$	X							
* q_1	X	X	X	$q_w = q_0$ $q_x = q_4$	X						
q_2	$q_x = q_3$ $q_y = q_7$	$q_y = q_3$ $q_z = q_7$	$q_z = q_3$ $q_w = q_7$	X	$q_2 = q_3$ $q_6 = q_7$	X					
q_3	$q_x = q_1$ $q_y = q_5$	$q_y = q_1$ $q_z = q_5$	$q_z = q_1$ $q_w = q_5$	X	$q_2 = q_1$ $q_6 = q_5$	X	$q_3 = q_1$ $q_7 = q_5$				
q_4	$q_x = q_6$ $q_y = q_1$	$q_y = q_6$ $q_z = q_1$	$q_z = q_6$ $q_w = q_1$	X	$q_2 = q_6$ $q_6 = q_1$	X	$q_3 = q_6$ $q_7 = q_1$	$q_1 = q_6$ $q_5 = q_1$			
q_5	$q_x = q_4$ $q_y = q_3$	$q_y = q_4$ $q_z = q_3$	$q_z = q_4$ $q_w = q_3$	X	$q_2 = q_4$ $q_6 = q_3$	X	$q_3 = q_4$ $q_7 = q_3$	$q_1 = q_4$ $q_5 = q_3$	$q_6 = q_4$ $q_1 = q_3$		
q_6	$q_x = q_7$ $q_y = q_0$	$q_y = q_7$ $q_z = q_0$	$q_z = q_7$ $q_w = q_0$	X	$q_2 = q_7$ $q_6 = q_0$	X	$q_3 = q_7$ $q_7 = q_0$	$q_1 = q_7$ $q_5 = q_0$	$q_6 = q_7$ $q_1 = q_0$	$q_4 = q_7$ $q_3 = q_0$	
* q_7	X	X	X	$q_w = q_5$ $q_x = q_2$	X	$q_0 = q_5$ $q_4 = q_2$	X	X	X	X	X

A próxima etapa passa então por percorrer a tabela iterativamente, marcando com um X as células em que pelo menos uma das condições de equi-

valência seja falsa (isto é, esteja já marcada). A tabela abaixo apresenta já o resultado final deste processo, não havendo já mais alterações que possam ser realizadas à tabela.

	q_w	q_x	q_y	q_z	q_0	q_1	q_2	q_3	q_4	q_5	q_6
q_x	X										
q_y	X	X									
* q_z	X	X	X								
q_0	$q_x = q_2$ $q_y = q_6$	X	X	X							
* q_1	X	X	X	$q_w = q_0$ $q_x = q_4$	X						
q_2	X	$q_y = q_3$ $q_z = q_7$	X	X	X	X					
q_3	X	X	$q_z = q_1$ $q_w = q_5$	X	X	X	X				
q_4	X	$q_y = q_6$ $q_z = q_1$	X	X	X	X	$q_3 = q_6$ $q_7 = q_1$	X			
q_5	$q_x = q_4$ $q_y = q_3$	X	X	X	$q_2 = q_4$ $q_6 = q_3$	X	X	X	X		
q_6	X	X	$q_z = q_7$ $q_w = q_0$	X	X	X	X	$q_1 = q_7$ $q_5 = q_0$	X	X	
* q_7	X	X	X	$q_w = q_5$ $q_x = q_2$	X	$q_0 = q_5$ $q_4 = q_2$	X	X	X	X	X

Podemos então ver que existem vários pares de estados equivalentes, incluindo os estados iniciais de cada um dos autómatos, q_w e q_0, o que significa que os dois autómatos são equivalentes.

Lema da Bombagem

O Lema da Bombagem diz-nos que se uma linguagem é regular, então segue o lema, ou seja, que para cadeias '*grandes*' existe uma parte da cadeia que se pode repetir, dando origem a mais cadeias da linguagem. Isto permite-nos usar autómatos finitos para representar linguagens infinitas, usando *loops* no autómato para repetir parte da cadeia.

Como o que nos interessa é a prova da regularidade ou não de uma linguagem, temos de usar a prova por contra-positiva, ou seja, se uma linguagem não segue o Lema da Bombagem, então não é regular (repare-se que uma linguagem pode seguir o lema, mas mesmo assim não ser regular, como vamos ver mais à frente). Para provar que uma linguagem é regular, podemos apresentar um autómato ou uma expressão regular que a represente (se formos capazes de o fazer, é porque a linguagem é regular).

Então, e de uma forma mais formal, o Lema diz-nos que se L é uma linguagem regular, existe uma constante m (um comprimento mínimo, que será dependente da linguagem) tal que para todas as cadeias w pertencentes a L com comprimento maior ou igual a m existe uma decomposição de w em três partes ($w = xyz$, com $y \neq \varepsilon$ e $|xy| \leq m$), tal que para qualquer $n \geq 0$, as cadeias geradas por n repetições de y também pertencem à linguagem (note-se que não estamos a dizer que são geradas todas as cadeias da linguagem, mas sim que todas as cadeias geradas pertencem à linguagem). De uma forma simplificada, será algo como:

$$\exists m : \forall w \in L, |w| \geq m : \exists w = xyz, y \neq \varepsilon, |xy| \leq m : \forall n : xy^n z \in L$$

.

Para fazer a prova, temos de negar o lema, o que significa que será necessário inverter os quantificadores. Então, temos que para uma linguagem L não seguir o lema, qualquer que seja o valor de m (por muito grande que seja), existe pelo menos uma cadeia w pertencente a L com comprimento maior ou igual a m, tal que qualquer que seja a divisão dessa cadeia em três partes, existem pelo menos um n tal que $xy^n z$ não pertence a L. Novamente, de uma forma simplificada, será algo como:

$$\forall m : \exists w \in L, |w| \geq m : \forall w = xyz, y \neq \varepsilon, |xy| \leq m : \exists n : xy^n z \notin L$$

Vamos ver um exemplo, para L = $\{vv^R : v \in \{0,1\}^* \}$, ou seja, cadeias binárias (de comprimento par) em que a segunda metade da cadeia é igual ao reverso da primeira metade da cadeia.

Podemos ver desde já que a linguagem não é regular, dado que um DFA para reconhecer estas cadeias teria de ser infinito, de forma a memorizar toda a primeira metade da cadeia, para garantir que a segunda metade da cadeia seria o reverso da primeira metade. Vamos fazer a prova usando o Lema da Bombagem. Começamos por assumir que existe um m, provando então que o Lema falha qualquer que seja o valor escolhido para m.

Agora, temos de escolher uma cadeia na qual não seja possível realizar a divisão e bombear a parte do meio. Vamos escolher uma cadeia 'grande', por exemplo a cadeia no formato $(01)^m(10)^m$, com comprimento $4m$. Numa cadeia com comprimento $4m$, e como sabemos que $y \neq \varepsilon$ e $|xy| \leq m$, então tanto x como y estão necessariamente na primeira metade da cadeia, e z abarca ainda parte da primeira metade, para além da segunda metade da cadeia. Agora, temos de provar que para cada possibilidade de divisão da cadeia, existe um n tal que $xy^nz \notin L$. Com este formato de cadeia, temos sempre 0's e 1's intercalados, com exceção do ponto central da cadeia, em que existem dois 1's consecutivos. Ora, como y não pode ser vazio, y vai conter pelo menos um símbolo, o que significa que para $n = 0$, a cadeia $xy^0z = xz$ vai ficar mais curta à esquerda do par de uns do que à direita, o que significa que este deixa de ser o ponto central da cadeia, o que significa que a cadeia deixa de estar no formato vv^R, o que significa que não pertence à linguagem.

Note-se que existem cadeias para as quais pode existir uma divisão que permite bombear a parte central, como por exemplo as cadeias apenas com 0's ou apenas com 1's (ao bombear por exemplo os dois primeiros símbolos da cadeia é possível gerar cadeias pertencentes à linguagem), e por isso é essencial que a escolha da cadeia seja feita com cuidado, para garantir que não escapa nenhuma cadeia que permita fazer a prova.

Podemos concluir que, uma vez que provamos que a linguagem não segue o Lema da Bombagem, então a linguagem não é regular.

Exercício 7

Determine se a linguagem $L = \{a^nb^p : n \leq p \leq 2n\}$ é ou não regular usando o Lema da Bombagem.

Vamos então começar por assumir a existência de um m, escolhendo depois uma cadeia de comprimento maior ou igual a m, como por exemplo $w = a^mb^m$, que pertence a L (repare-se que esta escolha vai diminuir o número de divisões possíves mais à frente).

No próximo passo, ao fazer a divisão de w em três partes, e como $|xy| \leq m$, sabemos que y estará na primeira parte da cadeia, e vai conter apenas a's (pelo menos 1, dado que $y \neq \varepsilon$). Ao bombear y com n maior ou igual a 2, a cadeia xy^nz deixa de pertencer a L, dado que vamos ficar com mais a's do que b's.

De uma forma mais formal, as divisões possíveis são com $x = a^p$, $p \geq 0$; $y = a^q$, $q \geq 1$; e $z = a^{m-p-q}b^m$. Para $n = 2$, a cadeia resultante será $xy^2z = xyyz = a^pa^qa^qa^{m-p-q}b^m = a^{m+q}b^m$. Como sabemos que $q > 0$, então vamos ter mais a's do que b's, o que significa que a cadeia não pertence à linguagem.

Como o valor de m não foi instanciado, significa que por muito grande que este seja, a cadeia w com o dobro desse comprimento no formato a^mb^m (que pertence a L) não pode ser dividida em três partes e ter a parte do meio bombeada, pois isso originaria cadeias com mais a's do que b's.

Fica então provado que a linguagem não é regular, uma vez que não segue o Lema da Bombagem.

Exercício 8

Determine se a linguagem $L = \{0^a1^b0^c : a > 4, b > 2, c \leq b\}$ é ou não regular usando o Lema da Bombagem.

Começamos então por assumir a existência de uma constante m, e escolhemos uma cadeia pertencente a L. Vamos escolher a cadeia no formato 000001^m0^m, que pertence à linguagem, desde que m seja maior do que 2 (o que se pode assumir, dado que o comprimento mínimo da cadeia é maior do que 2, e não faria sentido ter um valor de m menor do que o comprimento mínimo das cadeias da linguagem). Esta escolha da cadeia é pensada de forma a conseguirmos ao mesmo tempo diminuir o número de casos de partições a analisar e também encontrar um caso (um valor de n) em cada divisão tal que $xy^nz \notin L$.

Então, com a cadeia 0000001^m0^m, e considerando que $y \neq \varepsilon$ e $|xy| \leq m$, podemos ter três possíveis divisões da cadeia:

– y contém apenas 0's. Neste caso, como a cadeia escolhida tem apenas 5 zeros inicialmente, quando $n = 0$, a cadeia gerada (xz) não vai pertencer à linguagem, dado que irá ter no máximo quatro 0's (e a linguagem especifica $a > 4$).

– y contém 0's e 1's. Neste caso, para $n = 0$, a cadeia irá também ter no máximo quatro zeros, pelo que também não irá pertencer à linguagem.

– y contém apenas 1's. Neste caso, para $n = 0$, a cadeia irá ter um número de 1's inferior ao número de 0's do final da cadeia. Dado que uma das condições da linguagem é $c \leq b$, esta cadeia deixa de pertencer a L.

Prova-se então que não existe nenhuma divisão da cadeia que permita bombear a parte do meio da mesma, pelo que podemos afirmar que, não seguindo o Lema da Bombagem, a linguagem é não regular.

Exercício 9

Determine se a linguagem $L = \{pp^Rq : p, q \in \{0,1\}^+\}$ é ou não regular usando o Lema da Bombagem.

Esta linguagem já permite verificar o Lema da Bombagem. Escolhendo $m = 4$, por exemplo, conseguimos realizar a prova para qualquer cadeia da linguagem.

Para cadeias com $|p| = 1$, podemos fazer a divisão de forma a que $x = pp^R$ e y seja o primeiro símbolo de q, com z representando a restante cadeia. Seja qual for o valor de n, a cadeia xy^nz pertence a L, pois a primeira parte da cadeia (x) permanece inalterada, mantendo-se as cadeias resultantes no formato pp^Rq.

Para cadeias com $|p| > 1$, podemos fazer a divisão de forma a que $x = \varepsilon$, y seja o primeiro símbolo de p, e z represente a restante cadeia. Seja qual for o valor de n, a cadeia irá pertencer à linguagem: para $n = 0$, estamos basicamente a cortar o primeiro símbolo de p; basta assumir que o último símbolo de p^R passa agora a pertencer a q para que a cadeia continue no formato pp^Rq. Para $n > 1$, vamos ter repetições do primeiro símbolo; as duas primeiras repetições desse símbolo podem ser vistas como pp^R, em que p é agora constituído apenas por um símbolo, e toda a restante cadeia pertencerá a q, ficando assim a cadeia também no formato pp^Rq.

Então, acabamos de provar que o Lema da Bombagem se verifica para esta linguagem L: existe realmente um valor m (neste caso até relativamente pequeno, usamos 4), para o qual todas as cadeias $w \in L$ com $|w| \geq m$ podem ser divididas em três partes ($w = xyz$), tal que $xy^nz \in L$ para $n \geq 0$.

Na realidade, se tentarmos fazer o DFA para a linguagem, podemos constatar que teríamos de ter um DFA infinito para conseguir fazer o reconhecimento de pp^R, e sabemos que a linguagem não é regular.

No entanto, não conseguindo provar que a linguagem não segue o Lema da Bombagem, não podemos retirar qualquer conclusão sobre a regularidade da linguagem.

Capítulo 7

Gramáticas Livres de Contexto (CFG)

As Gramáticas Livres de Contexto (ou CFGs, do inglês *Context-Free Grammars*) são uma das formas de representação de Linguagens Livres de Contexto (ou CFLs, do inglês *Context-Free Languages*), sendo a outra forma autómatos de pilha, abordados no próximo capítulo. As CFGs permitem especificar linguagens que as expressões regulares e autómatos vistos até agora não permitem, como por exemplo as que foram vistas no final do último capítulo, nas provas de não regularidade.

As gramáticas livres de contexto podem ser vistas como um conjunto de regras, ou produções, no formato H $\rightarrow$ B, em que H (a cabeça da regra) representa uma variável, e B (o corpo da regra) representa um conjunto de terminais (símbolos da linguagem) e não-terminais (variáveis) pelos quais a cabeça pode ser substituída.

Começando pela variável de arranque da gramática, podem fazer-se substituições de cada variável por uma das suas produções; as cadeias obtidas constituídas apenas por terminais são aquelas que constituem a linguagem.

Vamos ver um exemplo da gramática para a linguagem L = $\{vv^R : v \in \{0,1\}^* \}$, que vimos já no capítulo anterior não ser uma linguagem regular.

A gramática pode ser constituída por apenas uma variável, com três produções, duas capazes de, recursivamente, produzir as cadeias desejadas, e uma que funciona como base de recursão:

$S \rightarrow 0S0 \mid 1S1 \mid \varepsilon$

A definição formal de uma gramática é feita com um tuplo de 4 elementos G = (V, T, P, S), em que V representa o conjunto de símbolos não-terminais (variáveis), T o conjunto de símbolos terminais (os símbolos do alfabeto), P o conjunto de produções e S $\in$ V a variável de arranque da gramática. Para a gramática apresentada, a sua definição formal será G = ({S}, {0, 1}, P, S),

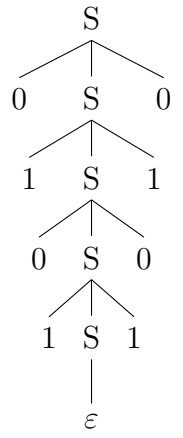
em que P representa as produções acima.

Se começarmos com o símbolo S, podemos produzir a cadeia vazia (ε), ou cadeias que iniciam e terminam no mesmo símbolo, e que no meio podem voltar a gerar novas cadeias nesse formato.

Para obter, por exemplo, a cadeia 01011010, começamos pela variável de arranque da gramática, S, substituindo a cada passo de derivação uma variável pelo corpo de uma das suas produções. Assim, para obter a cadeia desejada, temos:

$S \Rightarrow 0S0 \Rightarrow 01S10 \Rightarrow 010S010 \Rightarrow 0101S1010 \Rightarrow 0101\varepsilon1010$

Esta derivação pode também ser representada pela seguinte árvore de análise, em que os nós internos representam as variáveis da linguagem, e as folhas os terminais (cada expansão de um nó interno nos seus filhos corresponde a um passo da derivação, com os filhos sendo os símbolos presentes na produção, por ordem, da esquerda para a direita):



A colheita da árvore (leitura das folhas da esquerda para a direita) produz a cadeia que a árvore representa.

As derivações podem ainda ser classificadas como derivação mais à esquerda (*leftmost*) ou mais à direita (*rightmost*). Numa derivação mais à esquerda, a cada passo de derivação, é sempre escolhida para substituição a variável mais à esquerda na cadeia intermédia. Por outro lado, numa derivação mais à direita, em cada passo de derivação é escolhida sempre a variável mais à direita na cadeia intermédia. O resultado final é o mesmo (a cadeia pretendida), sendo uma distinção apenas no método de a obter. No exemplo apresentado, ambas as derivação mais à esquerda e mais à direita teriam o mesmo aspeto, já que existe sempre apenas uma variável a substituir.

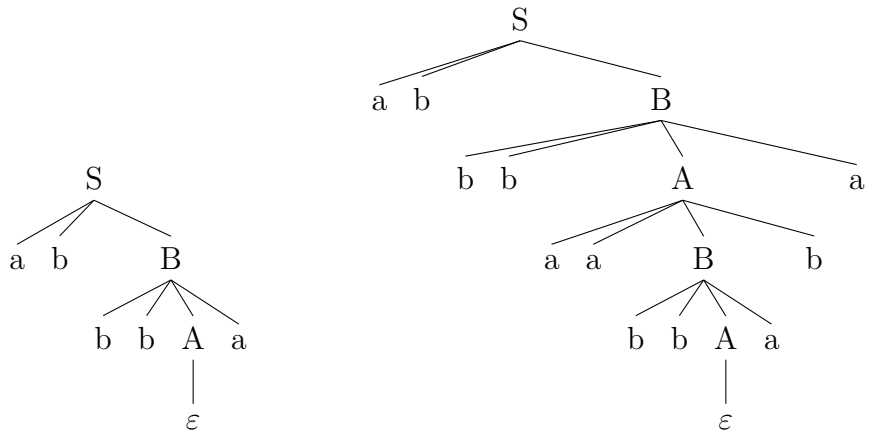
Exercício 1

Considere a seguinte gramática, em que S é a variável de arranque, e apresente uma derivação mais à esquerda para as cadeias *abbba* e *abbbaabbaba*, assim como as respetivas árvores de análise.

$S \to abB$
$A \to aaBb \mid \varepsilon$
$B \to bbAa$

Iniciando por S, podemos fazer a seguinte derivação para a cadeia *abbba*:
$S \underset{lm}{\Longrightarrow} abB \underset{lm}{\Longrightarrow} abbbAa \underset{lm}{\Longrightarrow} abbb\varepsilon a$

Para a cadeia *abbbaabbaba*, temos a seguinte derivação:
$S \underset{lm}{\Longrightarrow} abB \underset{lm}{\Longrightarrow} abbbAa \underset{lm}{\Longrightarrow} abbbaaBba \underset{lm}{\Longrightarrow} abbbaabbAaba \underset{lm}{\Longrightarrow} abbbaabb\varepsilon aba$
Podemos agora também desenhar as respetivas árvores de análise:



Exercício 2 (Desafio 2014/15)

Obtenha uma CFG para a linguagem L = $\{a^n b^m c^k : n, m > 0, k = n + m, n$ par, m ímpar$\}$. Obtenha ainda a derivação mais à esquerda e uma árvore de análise para a cadeia *aaaabbbccccccc*.

Para obter cadeias neste formato, torna-se necessário garantir que existe pelo menos um a e um b, e que o número de c's é igual ao número de a's mais o número de b's. De forma a garantir a paridade desejada, acrescentam-se

sempre símbolos aos pares, com exceção da produção base que contém b, de forma a obter um número ímpar. Uma possível solução é a seguinte:

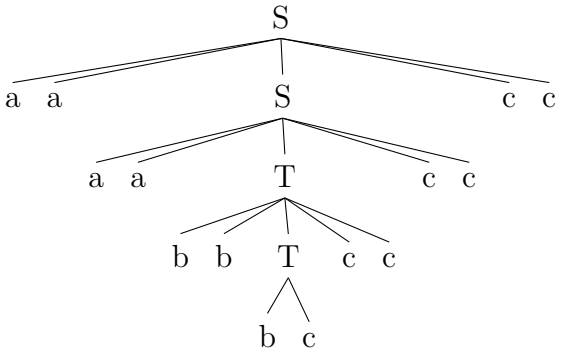
$S \rightarrow aaScc \mid aaTcc$
$T \rightarrow bbTcc \mid bc$

Com a variável S, variável de arranque desta gramática, garante-se que o número de a's é igual ao número de c's e que existem pelo menos dois a's, podendo existir mais, sempre em número par. A 'base de recursão' de S é quando é aplicada a produção $aaTcc$, que nos leva à variável T, a qual assegura um número igual de b's e de c's, assim como garante, com a sua 'base de recursão', a existência de pelo menos um b (assim como também assegura que o número total de b's é ímpar). No total, o número de c's é igual à soma do número de a's e de b's.

Vamos então fazer a derivação mais à esquerda para a cadeia pretendida, começando pela variável de arranque, S, e fazendo a cada passo de derivação a substituição de uma variável pelo corpo de uma das suas produções:

$$S \underset{lm}{\Longrightarrow} aaScc \underset{lm}{\Longrightarrow} aaaaTcccc \underset{lm}{\Longrightarrow} aaaabbTcccccc \underset{lm}{\Longrightarrow} aaaabbbccccccc$$

A partir da derivação podemos então criar a árvore de análise da cadeia, começando pela raíz (variável de arranque da gramática) e fazendo as substituições conforme a derivação.



Como podemos ver, a colheira da árvore produz a cadeia pretendida.

Exercício 3 (Desafio 2014/15)

Considere a seguinte CFG, com T = {t, s, c, d}, V = {S, W, C, D}, e variável inicial S.

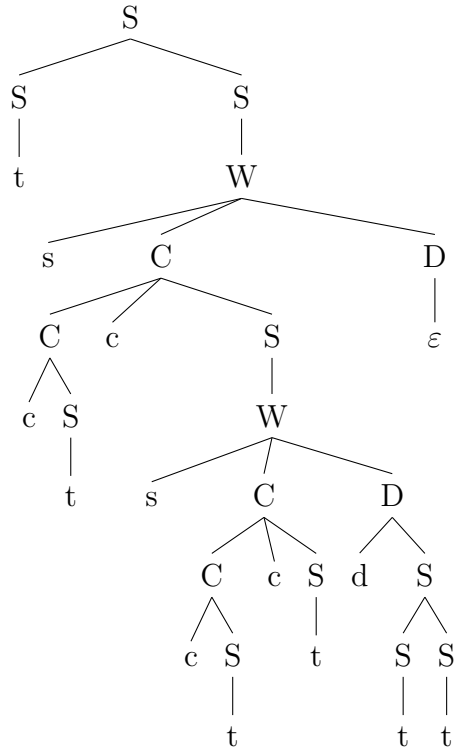
$S \to W \mid t \mid SS$
$W \to sCD$
$C \to cS \mid CcS$
$D \to dS \mid \varepsilon$

Obtenha uma derivação mais à esquerda para a cadeia *tsctcsctctdtt*, assim como a respetiva árvore de análise. Esta gramática é ambígua? Prove-o.

Podemos então começar por obter uma derivação para a cadeia, começando pela variável S, e verificando qual das possibilidades de derivação permite obter a cadeia pretendida.

$$S \underset{lm}{\Longrightarrow} SS \underset{lm}{\Longrightarrow} tS \underset{lm}{\Longrightarrow} tW \underset{lm}{\Longrightarrow} tsCD \underset{lm}{\Longrightarrow} tsCcSD \underset{lm}{\Longrightarrow} tscScSD \underset{lm}{\Longrightarrow} tsctcSD \underset{lm}{\Longrightarrow} tsctcWD \underset{lm}{\Longrightarrow} tsctcsCDD \underset{lm}{\Longrightarrow} tsctcsCcSDD \underset{lm}{\Longrightarrow} tsctcscScSDD \underset{lm}{\Longrightarrow} tsctcsctcSDD \underset{lm}{\Longrightarrow} tsctcsctctDD \underset{lm}{\Longrightarrow} tsctcsctctdSD \underset{lm}{\Longrightarrow} tsctcsctctdSSD \underset{lm}{\Longrightarrow} tsctcsctctdtSD \underset{lm}{\Longrightarrow} tsctcsctctdttD \underset{lm}{\Longrightarrow} tsctcsctctdtt$$

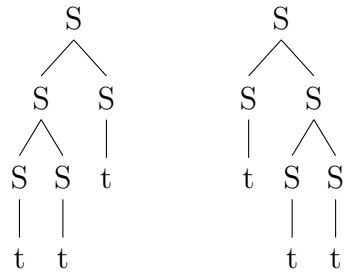
Agora, podemos também desenhar a árvore:



Esta derivação e árvore são um pouco mais complexas de obter do que as anteriores, dado o comprimento da cadeia e número de produções, sendo

necessário ter um pouco mais de atenção a quais as produções que podem produzir cada terminal e a forma como elas se relacionam entre si, de forma a encontrar o 'caminho certo'.

Em relação à ambiguidade, sim, podemos afirmar que esta gramática é ambígua, uma vez que é possível pensar numa cadeia da linguagem para a qual existem pelo menos duas árvores de análise diferentes. A título de exemplo, a cadeia *ttt* possui as seguintes duas árvores de análise:



Exercício 4

Obtenha uma CFG para a linguagem L = $\{w_1 c w_2 : w_1, w_2 \in \{a,b\}^+, |w_1| = |w_2|, w_1 \neq w_2^R\}$

Para obter uma solução para este problema, temos de garantir ao mesmo tempo que as 'partes exteriores' das cadeias têm o mesmo comprimento, mas não são o reverso uma da outra, o que significa que é necessário detetar essa diferença. Uma possível solução seria:
$S \rightarrow aSa \mid bSb \mid aTb \mid bTa$
$T \rightarrow aTa \mid aTb \mid bTa \mid bTb \mid c$

Com esta solução, e sendo S a variável de arranque da gramática, ficamos em S enquanto o início e o final da cadeia forem o reverso um do outro; quando se detetam símbolos distintos, que quebram esse padrão, passamos para T, onde se pode continuar a inserir qualquer par de símbolos, um de cada lado do *c* final, de forma a garantir comprimento igual de cada lado; no final, T é substituído por um c, terminando a cadeia.

Exercício 5

Obtenha uma CFG para a linguagem L=$\{a^p b^q c^r : p, q, r \geq 0, p = q \vee q \neq r\}$. A gramática obtida é ambígua? Justifique. Obtenha ainda uma derivação mais à direita para a cadeia *aabbccc*.

Como temos um 'ou' na definição da linguagem L, podemos separar a construção em duas sub-linguagens diferentes (L1 e L2), e no final a linguagem L será a união dessas linguagens. Vamos então ver cada caso.

Para L1=$\{a^p b^q c^r : p, q, r \geq 0, p = q\}$, temos apenas de garantir que o número de a's e de b's é igual. Para isso, podemos usar a seguinte gramática:

$S1 \rightarrow XC$

$X \rightarrow aXb \mid \varepsilon$

$C \rightarrow cC \mid \varepsilon$

Para L2=$\{a^p b^q c^r : p, q, r \geq 0, q \neq r\}$, temos de assegurar que o número de b's e de c's é diferente, o que pode ser conseguido assegurando que o número de b's é menor do que o número de c's, ou vice-versa. Assim, uma gramática possível será:

$S2 \rightarrow AY \mid AZ$

$A \rightarrow aA \mid \varepsilon$

$Y \rightarrow bYc \mid bY \mid b$

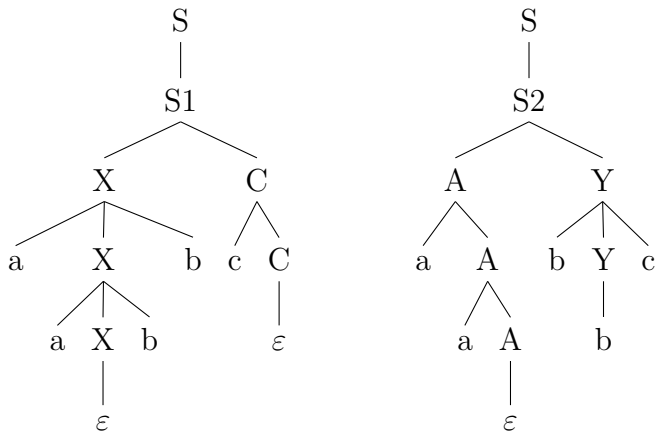
$Z \rightarrow bZc \mid Zc \mid c$

A variável A permite a existência de um qualquer número de a's. A variável Y permite assegurar a existência de pelo menos mais um b do que c's na cadeia, enquanto que a variável Z assegura o oposto, isto é, que o número de c's na cadeia é superior ao número de b's em pelo menos uma unidade.

Para obter a gramática final, serão necessárias as produções acima, mais a seguinte, para juntar as duas opções:

$S \rightarrow S1 \mid S2$

A gramática obtida é ambígua. Para o provar, podemos dar o exemplo de uma cadeia que tenha mais do que uma árvore de análise diferente. Por exemplo, a cadeia *aabbc* tem as seguintes duas possibilidades de interpretação:



Em relação à derivação mais à direita para a cadeia *aabbccc*, e para a gramática apresentada acima, teremos a seguinte derivação:
$S \underset{rm}{\Longrightarrow} S1 \underset{rm}{\Longrightarrow} XC \underset{rm}{\Longrightarrow} XcC \underset{rm}{\Longrightarrow} XccC \underset{rm}{\Longrightarrow} XcccC \underset{rm}{\Longrightarrow} Xccc\varepsilon \underset{rm}{\Longrightarrow} aXbccc \underset{rm}{\Longrightarrow} aaXbbccc \underset{rm}{\Longrightarrow} aa\varepsilon bbccc$

Exercício 6 (Exercício 2015/16)

Obtenha uma gramática para a linguagem $L1 = \{a^i b^j c^k : i, j, k \geq 0\}$, e para a linguagem $L2 = \{a^i b^j c^k : i, j, k \geq 0, j \geq i + k\}$. Consegue obter uma expressão regular para a linguagem L1? E para a linguagem L2?

Para a linguagem L1, podemos facilmente dividir a gramática em 3 componentes, cada uma para gerar cada um dos símbolos do alfabeto, ficando assim com a seguinte gramática:
$S \rightarrow ABC$
$A \rightarrow aA \mid \varepsilon$
$B \rightarrow bB \mid \varepsilon$
$C \rightarrow cC \mid \varepsilon$

Para a linguagem L2, podemos constatar que esta é um sub-conjunto de L1, mas que para gerar estas cadeias temos agora de relacionar o número de símbolos gerados. Uma possível solução será:
$S \rightarrow ABC$

$A \to aAb \mid \varepsilon$
$B \to bB \mid \varepsilon$
$C \to bCc \mid \varepsilon$

Com esta gramática, a partir da variável A consegue-se gerar um número igual de a's e de b's; com a variável C, consegue-se gerar um número igual de b's e de c's; com estas duas variáveis consegue-se garantir que o número de b's é igual à soma de a's e de c's, sendo que com a variável B se conseguem gerar b's adicionais, garantindo assim a condição de que $j \geq i + k$.

Para a linguagem L1, consegue-se facilmente pensar numa expressão regular, uma vez que se pretende gerar um qualquer número de cada um dos símbolos, ficando com L1 = a*b*c*.

Para L2, já não é possível encontrar uma expressão regular, uma vez que o número de símbolos se relacionam entre si. Podemos utilizar o lema da bombagem para provar que a linguagem não é regular, sendo portanto impossível de criar uma expressão regular para a mesma.

Exercício Proposto 1 Obtenha uma CFG para a linguagem L = $\{a^p b^q c^r : r = |p - q|\}$.

Exercício Proposto 2 Obtenha uma CFG para a linguagem L = $\{abcb^R : a, b, c \in \{0, 1\}^+\}$.

Gramáticas Regulares

Uma vez que as Linguagens Livres de Contexto englobam as Linguagens Regulares, podemos usar CFGs para representar linguagens regulares.

Surge então aqui o conceito de gramática regular: uma gramática diz-se ser regular se for consistentemente linear à esquerda ou consistentemente linear à direita. Uma gramática é linear quando todas as suas produções têm apenas uma variável no seu corpo. Para que a gramática seja consistentemente linear à direita, todas as produções são da forma $A \to xB$ ou $A \to x$, em que x representa um conjunto de símbolos terminais e B uma variável (não-terminal). Por outro lado, uma gramática diz-se consistentemente li-

near à esquerda se todas as suas produções são da forma $A \to Bx$ ou $A \to x$.

Para fazer a passagem de uma linguagem regular (representada por um DFA) para uma CFG, podemos aplicar os seguintes passos de conversão:

- Os estados do DFA tornam-se nas variáveis da gramática (estado inicial será o símbolo inicial da gramática)
- Os símbolos do alfabeto tornam-se nos terminais da gramática
- Para cada transição δ é criada uma produção: $\delta(q_1, a) = q_2$ dá origem a uma produção $Q_1 \to aQ_2$; Para cada estado final do DFA, acrescenta-se uma nova produção epsilon (por exemplo, se q_2 é um estado final, acrescenta-se uma produção $Q_2 \to \varepsilon$).

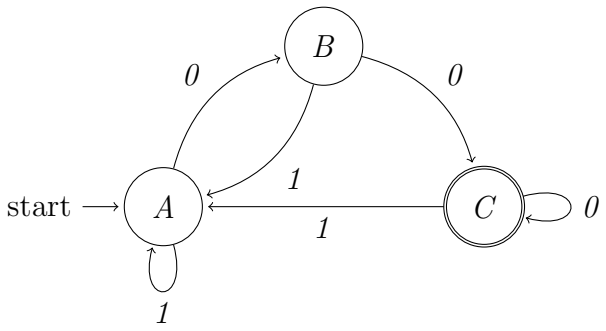
Podemos também aplicar esta tranformação a partir de um ε-NFA, adicionando ainda uma produção unitária para cada transição vazia. Por exemplo, para uma transição $\delta(q_1, \varepsilon) = q_2$ adiciona-se uma produção $Q_1 \to Q_2$.

Conversamente, podemos também fazer a passagem de uma gramática regular para um autómato que represente essa mesma linguagem, aplicando as regras de conversão em sentido contrário.

Exercício 7

Considere a linguagem das cadeias binárias que quando interpretadas como números em decimal são divisíveis por quatro. Desenhe um autómato para esta linguagem e faça a conversão para uma gramática equivalente. Indique ainda uma derivação e árvore de análise para a cadeia *10100*.

Uma possibilidade de reconhecer números divisíveis por quatro é reconhecer cadeias que terminam com dois zeros. Podemos fazê-lo facilmente com o seguinte autómato:



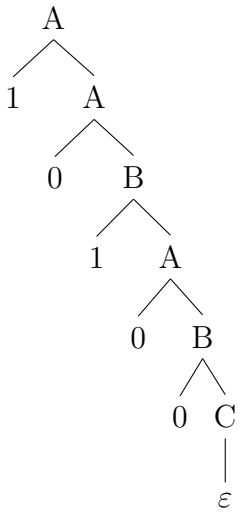
Para fazer a passagem para uma CFG, temos apenas de aplicar as regras. Identificamos então os estados como sendo as variáveis da gramática, os símbolos do alfabeto como sendo os símbolos terminais da gramática e as transições como sendo as produções da gramática. Ficamos então com as seguintes produções:
$A \to 1A \mid 0B$
$B \to 0C \mid 1A$
$C \to 0C \mid 1A \mid \varepsilon$

Como a conversão foi feita a partir de um DFA, não existe ambiguidade na gramática, havendo sempre apenas uma derivação possível a cada passo.

Para a cadeia 10100, temos a seguinte derivação:

$A \Rightarrow 1A \Rightarrow 10B \Rightarrow 101A \Rightarrow 1010B \Rightarrow 10100C \Rightarrow 10100\varepsilon$

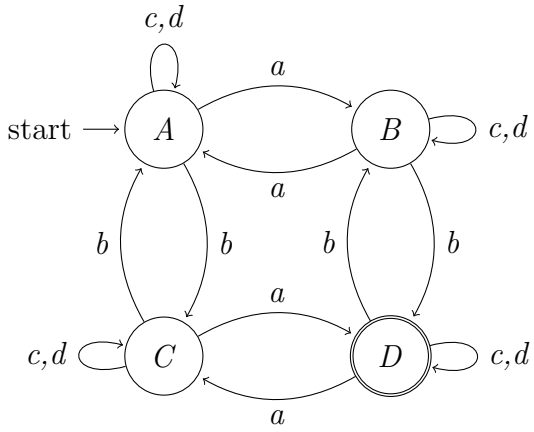
A árvore de análise correspondente seria a seguinte:



Exercício 8

Partindo da seguinte gramática, em que A é a variável de arranque, obtenha o DFA para a linguagem por ela definida e descreva a linguagem aceite.
$A \to aB \mid bC \mid cA \mid dA$
$B \to aA \mid bD \mid cB \mid dB$
$C \to aD \mid bA \mid cC \mid dC$
$D \to aC \mid bB \mid cD \mid dD \mid \varepsilon$

Sendo a gramática linear à direita, podemos então aplicar as regras de conversão de autómato para gramática em sentido inverso, e obter o autómato para esta linguagem. Iremos ter então quatro estados, conseguindo também identificar 4 símbolos no alfabeto, e um estado final.



Olhando para o autómato, torna-se mais simples de perceber que a linguagem definida tanto pela gramática como pelo autómato é a linguagem das cadeias no alfabeto a,b,c,d em que tanto o número de a's como o número de b's são ímpares.

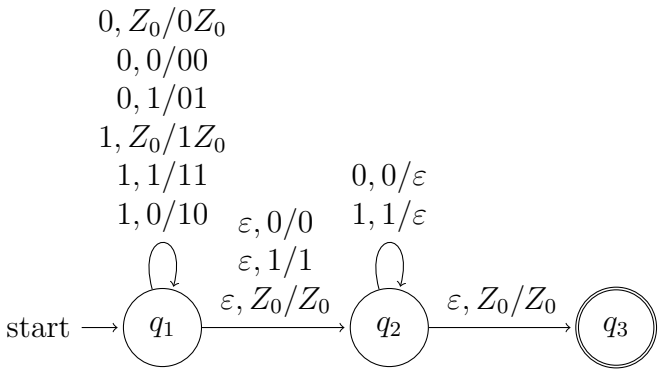
Exercício Proposto 3 É também possível fazer a conversão de uma linguagem regular para uma gramática livre de contexto partindo de uma expressão regular. Consegue determinar um algoritmo que o permita fazer?

Capítulo 8

Autómatos de Pilha (PDA)

Os autómatos de pilha, ou PDAs (do inglês *Push-Down Automata*) são extensões dos ε-NFAs com uma pilha infinita que permite armazenar informação. Uma transição tem agora que ter em atenção não só o estado atual e o símbolo na cadeia de entrada, mas também o símbolo que se encontra no topo da pilha, substituíndo ao mesmo tempo o topo da pilha (fazendo *push* de um novo símbolo; fazendo *pop* do símbolo que lá estivesse; substituindo o símbolo existente por um novo; ou mantendo a pilha inalterada). De uma forma mais formal, um PDA é visto como um tuplo $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, em que Q é o conjunto de estados do PDA, Σ o alfabeto (símbolos de entrada), Γ o alfabeto da pilha, δ o conjunto de transições, $q_0 \in Q$ o estado inicial, $Z_0 \in \Gamma$ o símbolo inicial da pilha, e $F \subseteq Q$ o conjunto de estados finais. As transições são no formato $\delta(q, a, X) = \{(p_1, \gamma_1), (p_2, \gamma_2), ...\}$, em que $q, p_1, p_2, ... \in Q$ são estados do PDA, $a \in \Sigma$ é um símbolo de entrada, $X \in \Gamma$ é um símbolo da pilha e $\gamma_1, \gamma_2, ...$ são conjuntos de símbolos da pilha, pelos quais X é substituído (pode ser usado ε para fazer *pop* da pilha).

Vamos olhar novamente para o exemplo da linguagem L = $\{vv^R : v \in \{0,1\}^*\}$. Uma possível implementação desta linguagem com um PDA será usando um estado para ler os símbolos de v, fazendo *push* desses símbolos na pilha; outro estado para ler os símbolos de v^R, fazendo *pop* dos símbolos da pilha ao confirmar que v^R é efetivamente o reverso de v; e um estado final, para aceitar a cadeia caso esta cumpra efetivamente com o formato das cadeias da linguagem. Teríamos então um $PDA = (\{q_1, q_2, q_3\}, \{0,1\}, \{Z_0, 0, 1\}, \delta, q_1, Z_0, \{q_3\})$, cuja representação gráfica seria:



Repare-se que em q_1 existem transições que fazem *push* do símbolo lido (0 ou 1) para todos os possíveis símbolos da pilha (Z_0, 0 e 1); no estado q_2 é feito *pop* de um símbolo da pilha quando o símbolo correspondente aparece na entrada; finalmente, a transição para q_3 acontece apenas quando a pilha já está vazia (isto é, todos os símbolos que foram colocados na pilha foram já retirados). A aceitação num PDA como este, de aceitação por estado final, acontece quando o autómato está num estado final e já não existem mais símbolos na cadeia de entrada (independentemente do conteúdo da pilha).

Note-se que com este autómato, é necessário 'adivinhar' quando se atinge o meio da cadeia, de forma a fazer a passagem de q_1 para q_2 (o que, como vamos ver mais à frente, faz com que este seja um PDA não determinista).

Para determinar se uma cadeia é aceite pelo PDA, pode ver-se a sequência de descrições instantâneas do PDA, verificando se o estado final é atingível. A descrição instantânea do estado do PDA inclui o estado em que este se encontra, a cadeia na entrada que ainda não foi processada, e o conteúdo da pilha. Por exemplo, para a cadeia 10100101, a sequência de descrições que conduz à aceitação seria:

$(q_1, 10100101, Z_0) \vdash (q_1, 0100101, 1Z_0) \vdash (q_1, 100101, 01Z_0) \vdash (q_1, 00101, 101Z_0) \vdash (q_1, 0101, 0101Z_0) \vdash (q_2, 0101, 0101Z_0) \vdash (q_2, 101, 101Z_0) \vdash (q_2, 01, 01Z_0) \vdash (q_2, 1, 1Z_0) \vdash (q_2, \varepsilon, Z_0) \vdash (q_3, \varepsilon, Z_0)$

Note-se ainda que existe, em cada momento, uma outra alternativa, que seria passar para q_2 ainda nos primeiros símbolos, ou manter em q_1 mesmo após ter percorrido metade da cadeia. No entanto, essas transições levariam a estados de não aceitação, pelo que seriam 'abandonados' por não se revelarem úteis.

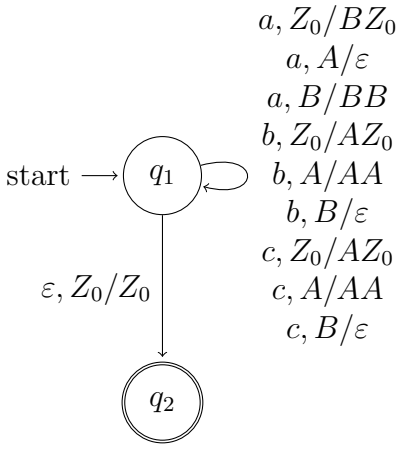
A aceitação das cadeias processadas por um PDAs pode, em alternativa ao uso de estados finais, ser determinado pelo facto de esvaziar por completo a pilha. Na representação formal, estes PDAs de aceitação por pilha vazia têm apenas os seis primeiros elementos, não incluíndo o conjunto de estados finais, pois estes deixam de existir. Estes PDAs podem ser facilmente obti-

dos a partir de um PDA de aceitação por estado final se, para cada estado final, for realizada uma transição para um novo estado, que faz *pop* de todos os símbolos da pilha, efetivamente esvaziando a pilha. Por outro lado, um PDA de aceitação por pilha vazia pode também ser convertido num PDA de aceitação por estado final se for adicionado um novo símbolo inicial na pilha antes do processamento, e em cada estado do PDA original for acrescentada uma transição para um estado final fazendo *pop* desse novo símbolo inicial.

Exercício 1

Obtenha um PDA para a linguagem das cadeias no alfabeto {a, b, c} em que o número de a's é igual à soma do número de b's e de c's.

Uma possível solução para este problema consiste em manter um contador que permita saber a relação entre o número de a's que já foi processado e o número de b's e c's. Como apenas possuímos uma pilha, podemos implementar este contador usando dois símbolos, um para representar valores positivos, e outro para representar valores negativos. Vamos ver essa solução:

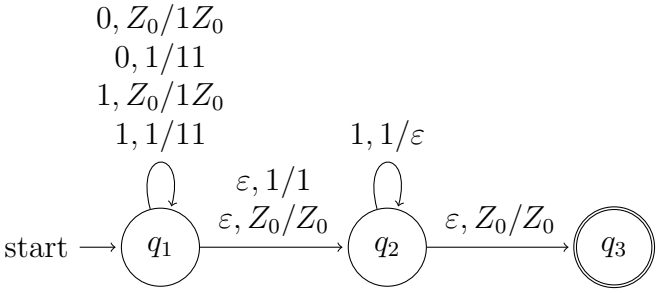


Podemos ver que no estado q_1, as transições associadas ao símbolo b são idênticas às associadas ao símbolo c, fazendo *pop* da pilha se nela existir um B, ou fazendo *push* de um A caso contrário. As transições associadas ao símbolo a fazem exatamente o contrário, isto é, é feito um *pop* da pilha, no caso de esta conter um A, ou *push* de um B caso contrário. A transição para q_2 permite que, quando a pilha esteja vazia (isto é, não há a's a mais nem a menos), o PDA transite para o estado de aceitação.

Exercício 2 (Desafio 2014/15)

Obtenha um PDA que reconheça a linguagem das cadeias binárias no formato $(0+1)^n 1^n : n \geq 0$. Apresente a sequência de descrições instantâneas para a cadeia 101111 e indique se esta é aceite pelo PDA.

Para obter um PDA para esta linguagem, necessitamos de um estado para reconhecer a primeira parte da cadeia, fazendo *push* de um símbolo para a pilha, e outro para reconhecer os 1's da segunda parte da cadeia, fazendo *pop* dos símbolos da pilha. Será ainda necessário um estado final para indicar aceitação.



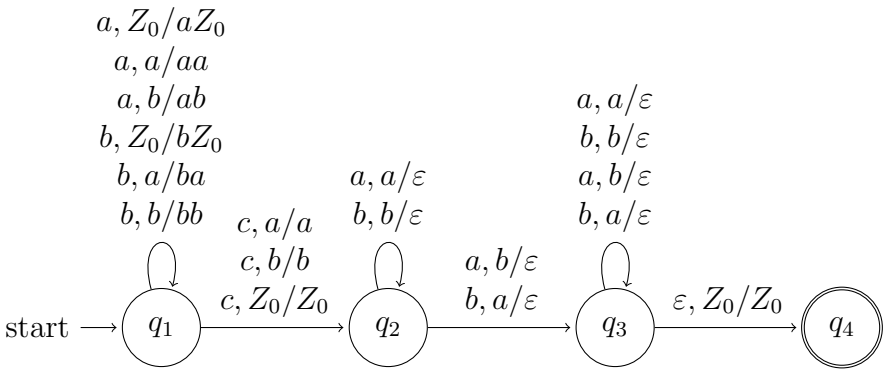
Podemos ver que esta solução é muito semelhante à do exemplo apresentado acima (as linguagens são também muito semelhantes). Para a cadeia pretendida, temos então a seguinte sequência de descrições intantâneas que conduz à aceitação da cadeia:

$(q_1, 101111, Z_0) \vdash (q_1, 01111, 1Z_0) \vdash (q_1, 1111, 11Z_0) \vdash (q_1, 111, 111Z_0)$
$\vdash (q_2, 111, 111Z_0) \vdash (q_2, 11, 11Z_0) \vdash (q_2, 1, 1Z_0) \vdash (q_2, \varepsilon, Z_0) \vdash (q_3, \varepsilon, Z_0)$

Exercício 3

Obtenha um PDA para a linguagem $\mathrm{L} = \{w_1 c w_2 : w_1, w_2 \in \{a,b\}^+, |w_1| = |w_2|, w_1 \neq w_2^R\}$. Apresente ainda a sequência de descrições instantâneas para as cadeias *abacaba* e *abacaab*, indicando se são ou não aceites pelo PDA.

Para obter uma solução para este problema, será necessário usar a pilha para guardar a primeira sequência de símbolos (w_1), e usar dois estados para validar que a segunda sequência (w_2) não é o reverso de w_1. Uma possível solução seria então:



No estado q_1 estamos então a preencher a pilha com os símbolos da primeira sequência (w_1), fazendo *push* do símbolo lido da entrada qualquer que seja o símbolo já na pilha. A passagem de q_1 para q_2 dá-se quando é lido o c que marca o meio da cadeia (podendo ter qualquer símbolo na pilha). Em q_2 e q_3 é feito o pop do símbolo no topo da pilha por cada símbolo na cadeia de entrada; em q_2 estamos ainda numa situação em que a segunda sequência está a corresponder ao reverso da primeira, pois cada símbolo lido é igual ao símbolo retirado da pilha; a passagem para q_3 acontece quando é lido um símbolo diferente do que está no topo da pilha; a partir deste momento, e como já temos a garantia de que $w_1 \neq w_2^R$, interessa-nos apenas contar o número de símbolos lidos, de forma a garanir que $|w_1| = |w_2|$. Quando a pilha é esvaziada, é realizada a transição para q_4.

Vejamos então as sequências para as cadeias indicadas:

$(q_1,\ abacaba,\ Z_0) \vdash (q_1,\ bacaba,\ aZ_0) \vdash (q_1,\ acaba,\ baZ_0) \vdash (q_1,\ caba,\ abaZ_0) \vdash (q_2,\ aba,\ abaZ_0) \vdash (q_2,\ ba,\ baZ_0) \vdash (q_2,\ a,\ aZ_0) \vdash (q_2,\ \varepsilon,\ Z_0)$

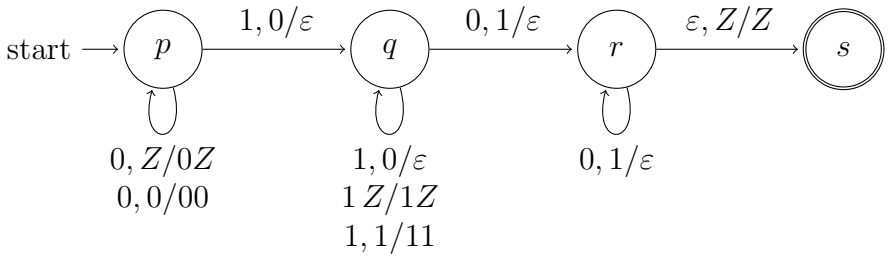
$(q_1,\ abacaab,\ Z_0) \vdash (q_1,\ bacaab,\ aZ_0) \vdash (q_1,\ acaab,\ baZ_0) \vdash (q_1,\ caab,\ abaZ_0) \vdash (q_2,\ aab,\ abaZ_0) \vdash (q_2,\ ab,\ baZ_0) \vdash (q_3,\ b,\ aZ_0) \vdash (q_3,\ \varepsilon,\ Z_0) \vdash (q_4,\ \varepsilon,\ Z_0)$

A primeira cadeia não é aceite pelo PDA (permanece no estado q_2, um estão de não aceitação), enquanto que a segunda cadeia já é aceite pelo PDA (termina em q_4, um estado de aceitação).

Exercício 4 (Exercício 2015/16)

Apresente um PDA para a linguagem $L = \{0^p 1^{p+q} 0^q : p, q > 0\}$, e a sequência de descrições instantâneas para a cadeia 011100.

Para garantir tanto a ordem dos símbolos na cadeia como a relação entre o número de zeros e de uns, torna-se necessário guardar a quantidade de 0's lidos inicialmente, assim como a quantidade de 1's que ultrapassa os zeros iniciais (e que terá de corresponder aos zeros finais). Uma possível solução, considerando um PDA com aceitação por estado final, e com símbolo inicial da pilha Z, será então:



Olhando para esta solução, podemos ver que no estado p estão a ser lidos os zeros iniciais da cadeia, colocando esses zeros na pilha, de forma a manter a sua contagem. Na transição para o estado q e no estado q, são lidos os uns, retirando os zeros da pilha enquanto ainda existem zeros, e colocando uns na pilha quando é ultrapassado o número de zeros inicial. Na passagem para o estado r e no estado r são lidos os zeros finais, sendo retirados da pilha os uns correspondentes. Finalmente, a passagem ao estado s garante a aceitação quando termina a leitura.

Para a cadeia 011100, temos então a seguinte sequência de descrições instantâneas:

$(p,\ 011100,\ Z) \vdash (p,\ 11100,\ 0Z) \vdash (q,\ 1100,\ Z) \vdash (q,\ 100,\ 1Z) \vdash (q,\ 00,\ 11Z) \vdash (r,\ 0,\ 1Z) \vdash (r,\ \varepsilon,\ Z) \vdash (s,\ \varepsilon,\ Z)$

Como no final da computação a cadeia de entrada foi completamente processada e o autómato está no estado s, significa que a aceia é aceite.

Equivalência entre CFGs e PDAs

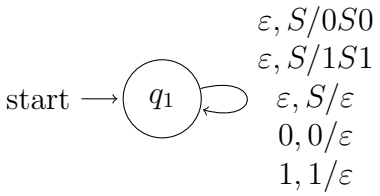
Sendo os PDAs e as CFGs duas representações de CFLs, podem ser convertidos entre si. A conversão de uma CFG num PDA de aceitação por pilha vazia é simples, bastando a existência de um estado, com várias transições para si próprio: para cada variável, uma transição que substitui essa variável pelo corpo de cada uma das suas produções na pilha, sem consumir qualquer símbolo na entrada; para cada símbolo, uma transição que consome o símbolo

e faz *pop* desse mesmo símbolo do topo da pilha. Inicialmente, a pilha contém apenas o símbolo correspondente à variável de arranque da gramática.

Vamos ver um exemplo tendo por base a gramática para a linguagem L $= \{vv^R : v \in \{0,1\}^* \}$, vista anteriormente:

$S \to 0S0 \mid 1S1 \mid \varepsilon$

O PDA equivalente terá então apenas um estado, com duas transições para os símbolos da linguagem (0 e 1) e três transições que representam as três produções da gramática. Temos assim o seguinte PDA:



Vejamos o exemplo de processamento da cadeia 101101:

(q_1, 101101, S) $\vdash$ (q_1, 101101, $1S1$) $\vdash$ (q_1, 01101, $S1$) $\vdash$ (q_1, 01101, $0S01$) $\vdash$ (q_1, 1101, $S01$) $\vdash$ (q_1, 1101, $1S101$) $\vdash$ (q_1, 101, $S101$) $\vdash$ (q_1, 101, 101) $\vdash$ (q_1, 01, 01) $\vdash$ (q_1, 1, 1) $\vdash$ (q_1, ε, ε)

Estando a pilha completamente vazia no final do processamento da cadeia de entrada, podemos afirmar que a cadeia é aceite pelo PDA.

Exercício 5

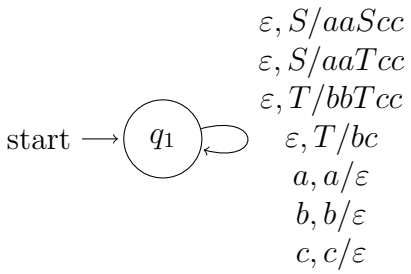
Considere as gramáticas do Exercícios 2, 3 e 4 do capítulo anterior. Converta cada uma delas para um PDA de aceitação por pilha vazia.

Começando pelo **Exercício 2**, e recordando a gramática obtida:

$S \to aaScc \mid aaTcc$

$T \to bbTcc \mid bc$

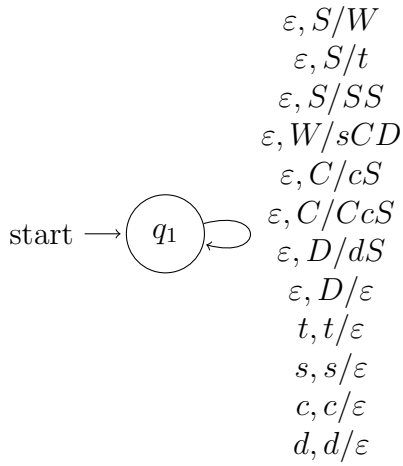
Podemos então converter esta gramática num PDA com apenas um estado usando o método visto acima:



Temos novamente apenas um estado, as quatro transições correspondentes às quatro produções da gramática, e as três transições correspondentes aos três símbolos do alfabeto da linguagem.

No caso do **Exercício 3**, temos a seguinte gramática:
$S \to W|t|SS$
$W \to sCD$
$C \to cS|CcS$
$D \to dS|\varepsilon$

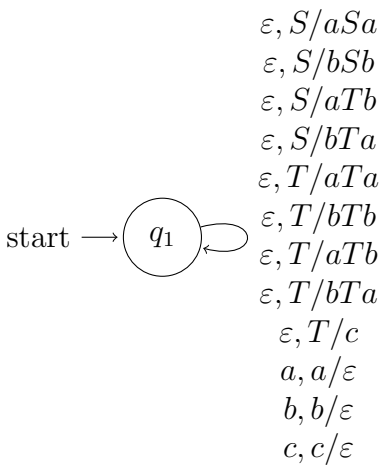
Fazendo igualmente a conversão num PDA:



Neste PDA existem bastante mais transições, dado que a gramática tem também mais produções.

Finalmente, recordado a gramática do **Exercício 4**:
$S \to aSa \mid bSb \mid aTb \mid bTa$
$T \to aTa \mid aTb \mid bTa \mid bTb \mid c$

Podemos ter então o seguinte PDA:

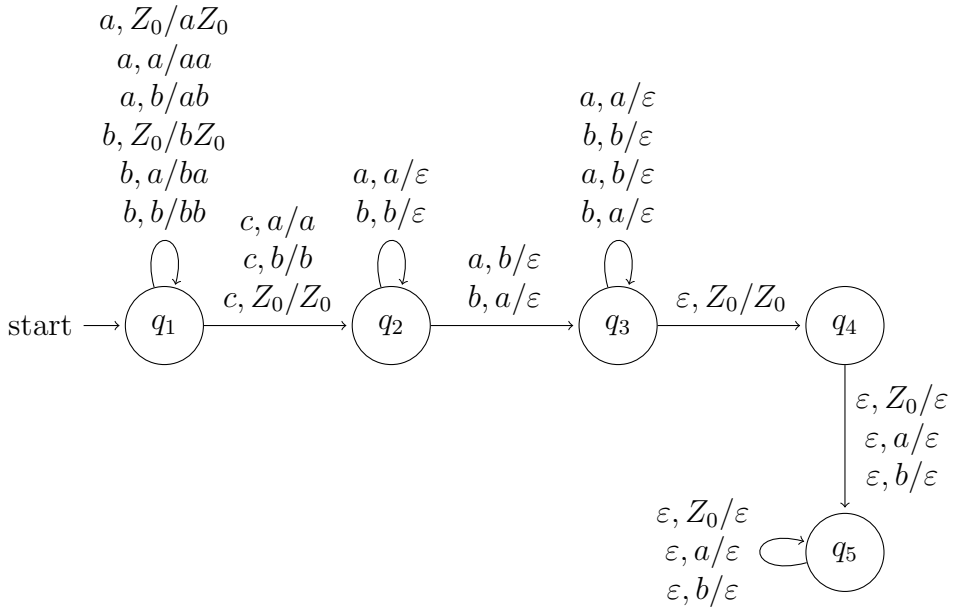


Repare-se neste último exemplo, que a linguagem pedida é a mesma que foi pedida acima, no **Exercício 3**, e para o qual foi obtido um PDA com quatro estados (sendo um deles usado apenas para garantir a aceitação das cadeias).

Exercício 6

Converta o PDA obtido no **Exercício 3** para um PDA de aceitação por pilha vazia, e o PDA obtido no **Exercício 5** por conversão da Gramática obtida no **Exercício 4** do capitulo anterior para um PDA de aceitação por estado final. Apresente as definições formais de cada um dos PDAs resultantes.

Este exercício envolve novamente os dois PDAs para a mesma linguagem. Começando pelo PDA obtido no **Exercício 3** acima, a conversão para um PDA de aceitação por pilha vazia é feita inserindo um novo estado no PDA, e acrescentando transições de cada um dos estados finais do PDA original para este novo estado em que é feito um *pop* de qualquer símbolo do topo da pilha sem consumir nada na entrada. No novo estado, deve também existir uma transição para o próprio estado, sendo também feito *pop* de qualquer símbolo da pilha. Vamos ver como fica então o PDA modificado (todas as modificações foram inseridas na metade de baixo do PDA, para ser mais simples de as identificar):



O novo estado q_5 será então o estado que tratará de esvaziar a pilha, garantindo que a pilha fica vazia no final do processamento. Repare-se que o estado q_4 deixou de ser estado final, uma vez que num PDA de aceitação por pilha vazia estes deixam de ser úteis.

Note-se ainda que, neste caso, o estado q_5 seria desnecessário, e que teria bastado modificar a transição de q_3 para q_4 de $\varepsilon, Z_0/Z_0$ para $\varepsilon, Z_0/\varepsilon$, esvaziando assim a pilha no único ponto onde esta poderia ser esvaziada.

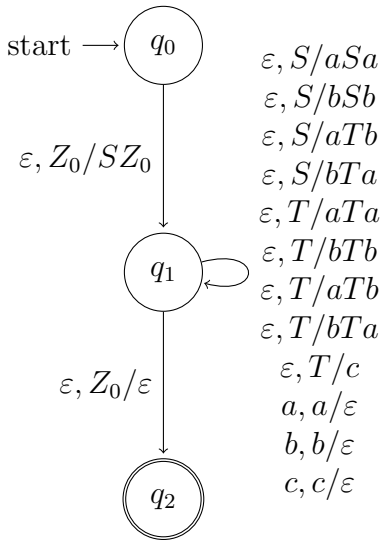
Em relação à definição formal, temos então a seguinte definição:

PDA V = $(\{q_1, q_2, q_3, q_4, q_5\}, \{a, b, c\}, \{a, b, Z_0\}, \delta, q_1, Z_0)$

Sendo δ constituído pelas seguintes transições: $\delta(q_1, a, Z_0) = \{(q_1, aZ_0)\}$, $\delta(q_1, a, a) = \{(q_1, aa)\}$, $\delta(q_1, a, b) = \{(q_1, ab)\}$, $\delta(q_1, b, Z_0) = \{(q_1, bZ_0)\}$, $\delta(q_1, b, a) = \{(q_1, ba)\}$, $\delta(q_1, b, b) = \{(q_1, bb)\}$, $\delta(q_1, c, Z_0) = \{(q_2, Z_0)\}$, $\delta(q_1, c, a) = \{(q_2, a)\}$, $\delta(q_1, c, b) = \{(q_2, b)\}$, $\delta(q_2, a, a) = \{(q_2, \varepsilon)\}$, $\delta(q_2, b, b) = \{(q_2, \varepsilon)\}$, $\delta(q_2, a, b) = \{(q_3, \varepsilon)\}$, $\delta(q_2, b, a) = \{(q_3, \varepsilon)\}$, $\delta(q_3, a, a) = \{(q_3, \varepsilon)\}$, $\delta(q_3, a, b) = \{(q_3, \varepsilon)\}$, $\delta(q_3, b, a) = \{(q_3, \varepsilon)\}$, $\delta(q_3, b, b) = \{(q_3, \varepsilon)\}$, $\delta(q_3, \varepsilon, Z_0) = \{(q_4, Z_0)\}$, $\delta(q_4, \varepsilon, Z_0) = \{(q_5, \varepsilon)\}$, $\delta(q_4, \varepsilon, a) = \{(q_5, \varepsilon)\}$, $\delta(q_4, \varepsilon, b) = \{(q_5, \varepsilon)\}$, $\delta(q_5, \varepsilon, Z_0) = \{(q_5, \varepsilon)\}$, $\delta(q_5, \varepsilon, a) = \{(q_5, \varepsilon)\}$, $\delta(q_5, \varepsilon, b) = \{(q_5, \varepsilon)\}$

Vendo agora o processo inverso, para o PDA obtido no **Exercício 5** acima, será necessário introduzir dois novos estados: um novo estado inicial (e um novo símbolo inicial para a pilha) que introduza o símbolo inicial original (variável de arranque da gramática); e um novo estado final, para o qual devem ser inseridas transições a partir de cada um dos estados do PDA

sem consumir nenhum símbolo da cadeia de entrada e tendo na pilha o novo símbolo. Vejamos então como fica o PDA modificado:



O novo estado inicial tem então uma transição para o estado inicial do PDA original, em que é feito *push* do símbolo inicial da pilha original (S), sendo agora usado Z_0 como símbolo inicial da pilha. Existe também uma transição de cada um dos estados do PDA original (neste caso só havia um estado, q_1) para o novo estado final (q_2) quando é detetado o novo símbolo inicial da pilha (o que siginifica que a pilha teria ficado completamente vazia no PDA original).

Em relação à definição formal, temos então a seguinte definição:

PDA A = $(\{q_0, q_1, q_2\}, \{a, b, c\}, \{a, b, c, S, T, Z_0\}, \delta, q_0, Z_0, \{q_2\})$

Sendo δ constituído pelas seguintes transições: $\delta(q_0, \varepsilon, Z_0) = \{(q_1, SZ_0)\}$,
$\delta(q_1, \varepsilon, S) = \{(q_1, aSa), (q_1, bSb), (q_1, aTb), (q_1, bTa)\}$,
$\delta(q_1, \varepsilon, T) = \{(q_1, aTa), (q_1, bTb), (q_1, aTb), (q_1, bTa), (q_1, c)\}$,
$\delta(q_1, a, a) = \{(q_1, \varepsilon)\},\ \delta(q_1, b, b) = \{(q_1, \varepsilon)\},\ \delta(q_1, c, c) = \{(q_1, \varepsilon)\}$,
$\delta(q_1, \varepsilon, Z_0) = \{(q_2, \varepsilon)\}$

Determinismo dos PDAs

Os PDAs vistos até aqui são maioritariamente não determinísticos, isto é, em cada momento, poderia ser realizada mais do que uma transição, dando efetivamente origem não a uma única sequência de descrições instantâneas,

mas sim a uma *árvore*[1]. No entanto, os PDAs podem ser deterministas (chamando-se normalmente de DPDA, do inglês *Deterministic PDA*), o que pode facilitar a sua execução, e análise. Um exemplo de DPDA foi obtido no **Exercício 3**, em que a cada momento apenas uma transição pode ser executada.

Para ser determinista, um PDA só pode ter uma escolha em cada momento. Para isso, e em cada estado, só pode existir no máximo uma transição possível para cada par (símbolo de entrada, símbolo na pilha). Isto implica também especial cuidado com transições espontâneas, em que não é consumido nenhum símbolo da entrada: no caso de existir uma transição destas para um determinado símbolo na pilha, não poderá existir mais nenhuma transição com esse símbolo na pilha (uma vez que essa transição-ε significa que pode ser feita com qualquer símbolo na entrada).

Exercício 7

Classifique os PDAs obtidos nos exercícios anteriores quanto ao seu determinismo.

Começando pelo **Exercício 1**, o PDA obtido não é determinista, dado que em q_1 existe uma transição para q_2 com Z_0 no topo da pilha e qualquer símbolo presente na entrada, e existem ainda transições de q_1 para si próprio também com Z_0 no topo da pilha.

No caso do **Exercício 2**, o PDA volta a ser não determinista, novamente devido a transições ε: em q_1 existem transições para si próprio com Z_0 e com 1 no topo da pilha (com 0 e 1 como símbolos de entrada), mas existem também transições-ε para q_2 com Z_0 e 1 no topo da pilha, o que faz com que o PDA seja não determinista.

No **Exercício 3**, já temos um DPDA: em cada estado existe apenas uma possibilidade para cada par (símbolo de entrada, símbolo no topo da pilha), e a transição-ε de q_3 para q_4 é feita com Z_0 no topo da pilha, e como não existe nenhuma transição em q_3 com Z_0 no topo da pilha, o determinismo não é invalidado.

No **Exercício 4**, temos também um PDA determinista: em cada estado temos apenas uma única possibilidade de transição para cada par (símbolo de entrada, símbolo no topo da pilha). Como a transição ε do estado r para

[1]Por questões de simplificação, temos estado a apresentar apenas uma sequência de descrições instantâneas que conduza à aceitação da cadeia, podendo, como já foi referido, existir possíveis caminhos alternativos (que poderiam ou não conduzir a aceitação).

s é feita com Z no topo da pilha e não há mais nenhuma transição em r considerando Z no topo da pilha, mantém-se o determinismo.

No **Exercício 5** temos vários PDAs, sendo que nenhum deles é determinista, uma vez que em cada um deles existe mais do que uma transição-ε para um determinado símbolo no topo da pilha. Por exemplo, e em qualquer um dos três PDAs apresentados, para o símbolo S existem sempre pelo menos duas possibilidades de substituição (as quais advêm da existência de mais do que uma produção para aquele símbolo (variável) na gramática que deu origem aos PDAs), o que faz com que estes PDAs não sejam deterministas. Aliás, como se pode ver, para quem um PDA obtido por este método de conversão de CFG para PDA seja determinista, a gramática original não pode ter mais do que uma produção por cada variável.

Finalmente, no **Exercício 6**, temos novamente dois PDAs. O primeiro pode ser considerado determinístico, uma vez que em cada estado existe apenas uma possibilidade de transição para cada par (símbolo de entrada, símbolo no topo da pilha). Se repararmos, todas as transições-ε são feitas em estados para os quais não existem mais nenhuma transição para cada um dos símbolos da pilha usados nessas transições. Por exemplo, em q_5 existem quatro transições-ε para o próprio estado, mas todas elas são feitas com símbolos diferentes no topo da pilha, e não existe mais nenhuma transição possível para nenhum desse símbolos. Já o segundo PDA não pode ser considerado determinístico, uma vez que apresenta várias possibilidades de transição em q_1 (sendo novamente um PDA obtido por transformação de uma CFG com várias produções para cada variável, são apresentadas várias opções de transição para os símbolos S e T no topo da pilha).

Exercício Proposto 1 Obtenha um PDA (aceitação por estado final) para a linguagem L das cadeias binárias em que o reverso dos primeiros símbolos da cadeia pode ser encontrado no resto da cadeia ($L = \{w_1 w_2 : w_1^R \subseteq w_2\}$), e converta-o para um PDA de aceitação por pilha vazia. Obtenha também uma CFG para esta mesma linguagem, converta-a para um PDA (aceitação por pilha vazia), e finalmente converta esse PDA para um PDA de aceitação por estado final. Classifique cada um dos PDAs obtidos em relação ao seu determinismo. Apresente ainda uma sequência de descrições instantâneas para a cadeia 011010. Exemplos: 01010 pertence a L (por exemplo, se $w_1 = 01$, $w_2 = 010$, que contém w_1^R (10))

Capítulo 9

Propriedades das Linguagens Livres de Contexto

As Linguagens Livres de Contexto (CFL, do inglês *Context-Free Language*) gozam de um conjunto de propriedades que são aqui exploradas.

9.1 Simplificação de CFGs e Forma Normal de Chomsky

Existem formas de transformar as CFGs de forma a encontrar representações alternativas para a mesma linguagem que permitam facilitar o processamento ou a realização de provas ou operações sobre as gramáticas.

Uma das formas mais usadas é a Forma Normal de Chomsky (CNF, ou *Chomsky Normal Form*), em que todas as produções da gramática dão origem a duas variáveis ou a um terminal, ou seja, são produções na forma $A \to BC$ ou $A \to a$, em que A, B e C são variáveis e a um terminal. Note-se que esta forma normal não permite obter a cadeia vazia, pelo que a gramática G de uma linguagem L que inclua a cadeia vazia irá ter uma gramática correspondente em CNF para $L \backslash \{\varepsilon\}$.

Para chegar a esta forma, podem seguir-se um conjunto de passos, eliminando produções-ε, produções unitárias e símbolos inúteis.

Para eliminar produções-ε, é necessário primeiro identificar quais as variáveis anuláveis, que são aquelas que podem produzir a cadeia vazia. Se houver uma produção $A \to \varepsilon$, então a variável A é anulável. Se houver uma produção $B \to CD$, em que tanto C como D são variáveis anuláveis, então B será também anulável. Depois de identificar todas as variáveis anuláveis, é necessário identificar todas as produções em que essas variáveis aparecem no corpo, e criar novas produções em que se fornece a alternativa em que a

variável anulável não esteja presente. Por exemplo, se A for anulável, e houver uma produção $B \to AC$ então esta será transformada em $B \to AC \mid C$. Todas as produções $A \to \varepsilon$ são também obviamente eliminadas.

Uma forma simples de eliminar produções unitárias (aquelas no formato $A \to B$) seria iterativamente ir substituindo o corpo da produção unitária pelas produções da variável (ou seja, para o caso de $A \to B$, substituir por $A \to \alpha$, em que α são as produções de B). No entanto, se existirem ciclos, este método torna-se falível. Então, torna-se necessário usar um método mais robusto, identificando os pares unitários, e depois, para cada par unitário (A, B), incluir as produções não unitárias de B em A. Os pares (X, X) são considerados pares unitários. Um par (X, Y) é considerado um par unitário se houver uma sequência de produções unitárias que permita gerar Y a partir de X. Por exemplo, considerando as produções $A \to B$ e $B \to C$, temos (A, A), (B, B) e (C, C) como pares unitários; olhando para as produções existentes, temos que (A, B) e (B, C) são também pares unitários, assim como (A, C).

Para eliminar símbolos inúteis, é preciso identificar e eliminar símbolos não geradores, e símbolos não atingíveis. Símbolos não geradores são aqueles que não dão origem a terminais. Identificam-se os símbolos geradores tendo em conta que os terminais são geradores e se houver uma produção $A \to \alpha$ em que α é composto apenas por geradores (terminais e/ou variáveis), então A também é gerador. Em relação aos símbolos atingíveis, o símbolo de arranque é atingível por definição, e todos símbolos presentes no corpo das produções de um símbolo atingível são também atingíveis.

Vamos ver um exemplo, usando a seguinte gramática:
$S \to traX \mid X$
$X \to laX \mid \varepsilon$
$Y \to alY \mid \varepsilon$

Para obter a gramática equivalente na Forma Normal de Chomsky precisamos então de realizar os três passos de simplificação.

Para eliminar as produções-ε, começamos por identificar quais as produções que podem dar origem à cadeia vazia. Neste caso, originalmente apenas X e Y têm essa possibilidade (pelas produções $X \to \varepsilon$ e $Y \to \varepsilon$, respetivamente). Olhando depois para os corpos das restantes produções, podemos ver que S também é anulável (pela produção $S \to X$). Então, para obter a gramática resultante da aplicação deste primeiro passo, devemos retirar as produções-ε da gramática, e acrescentar novas produções para os casos em que X, Y ou S possam não existir. Ficamos então com a seguinte gramática:
$S \to traX \mid X \mid tra$
$X \to laX \mid la$

$Y \to alY \mid al$

No segundo passo, para eliminar as produções unitárias, podemos identificar apenas uma produção unitária: $S \to X$, pelo que temos apenas de a substituir pelas produções de X, ficando com:
$S \to traX \mid laX \mid la \mid tra$
$X \to laX \mid la$
$Y \to alY \mid al$

Finalmente, no terceiro passo eliminam-se símbolos inúteis. Apesar de todos os símbolos serem geradores, nem todos são atingíveis: Y não é atingível a partir de S. Então, no final, ficamos com apenas duas produções:
$S \to traX \mid laX \mid la \mid tra$
$X \to laX \mid la$

Com exceção da cadeia vazia (que não é contemplada por gramáticas na CNF), a linguagem aceite por esta gramática é a mesma que a aceite pela grmática original (isto é, cadeias iniciadas por 'tra', seguidas de um qualquer número de 'la's).

Falta agora apenas colocar a gramática no formato final, em que todas as produções dão origem a duas variáveis ou a um terminal. Podemos começar por definir novas variáveis para produção dos terminais:
$T \to t$
$R \to r$
$A \to a$
$L \to l$

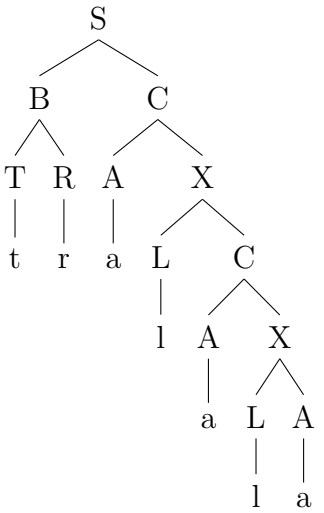
Fazendo a substituição, ficamos com as seguintes produções:
$S \to TRAX \mid LAX \mid LA \mid TRA$
$X \to LAX \mid LA$

Acrescentando as produções
$B \to TR$
$C \to AX$

Podemos substituir e ficar com a gramática final:
$S \to BC \mid LC \mid LA \mid BA$
$X \to LC \mid LA$
$B \to TR$
$C \to AX$
$T \to t$

$R \to r$
$A \to a$
$L \to l$

Podemos ver que a gramática não é tão simples de interpretar, mas tem a vantagem de produzir apenas árvores binárias: todos os nós derivam dois novos nós, ou uma folha. Por exemplo, para a cadeia *tralala* teremos a seguinte árvore:



Exercício 1 (Desafio 2014/15)

Converta a seguinte gramática para a Forma Normal de Chomsky e apresente árvores de análise para a cadeia 1001001 na gramática original e na CNF.
$S \to 1AB0 \mid 0BA1 \mid AA \mid BB \mid SS$
$A \to 0A \mid \varepsilon$
$B \to 1B \mid \varepsilon$

Para converter a gramática para CNF, necessitamos então de seguir os três passos de simplificação. Para eliminar as produções-ε, começamos por identificar que tanto A como B podem produzir ε, e por isso são anuláveis. S, por sua vez, tem pelo menos uma produção composta apenas por símbolos anuláveis ($S \to AA$), e será por isso também anulável. Então, é necessário eliminar as produções-ε e acrescentar novas produções, ficando a gramática nesta forma:

$S \to 1AB0 \mid 0BA1 \mid AA \mid BB \mid SS \mid 1B0 \mid 1A0 \mid 10 \mid 0B1 \mid 0A1 \mid 01 \mid A \mid B \mid S$
$A \to 0A \mid 0$
$B \to 1B \mid 1$

No próximo passo, é necessário identificar as produções unitárias, e vemos que existem em S três produções unitárias ($S \to A \mid B \mid S$). Como não há problemas de ciclos, podemos fazer diretamente a substituição pelas produções das variáveis respetivas, ficando assim com a seguinte gramática:
$S \to 1AB0 \mid 0BA1 \mid AA \mid BB \mid SS \mid 1B0 \mid 1A0 \mid 10 \mid 0B1 \mid 0A1 \mid 01 \mid 0A \mid 0 \mid 1B \mid 1$
$A \to 0A \mid 0$
$B \to 1B \mid 1$

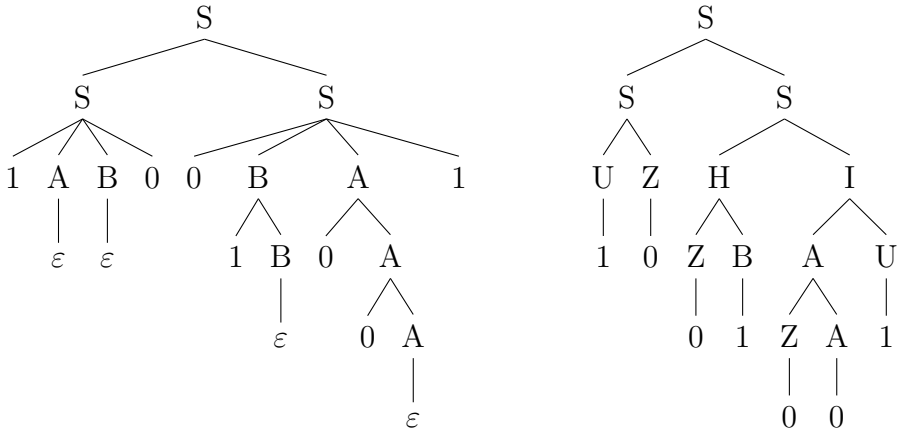
De seguida, temos de identificar símbolos não geradores, ou não atingíveis. Neste caso, todos os símbolos geram terminais, pelo que todos são geradores. Começando em S, é também possível atingir todos os restantes símbolos, pelo que não existem também símbolos não atingíveis.

Então, falta apenas o passo final de colocar todas as produções no formato da CNF. Começamos por criar novas variáveis para os terminais da linguagem:
$Z \to 0$
$U \to 1$

Substituímos agora na gramática as ocorrências de terminais não isolados pelas novas variáveis:
$S \to UABZ \mid ZBAU \mid AA \mid BB \mid SS \mid UBZ \mid UAZ \mid UZ \mid ZBU \mid ZAU \mid ZU \mid ZA \mid 0 \mid UB \mid 1$
$A \to ZA \mid 0$
$B \to UB \mid 1$
$Z \to 0$
$U \to 1$

Agora, falta apenas adicionar novas variáveis para reduzir produções de tamanho maior do que 2. Uma solução possível seria:
$S \to FG \mid HI \mid AA \mid BB \mid SS \mid UG \mid FZ \mid UZ \mid HU \mid ZI \mid ZU \mid ZA \mid 0 \mid UB \mid 1$
$A \to ZA \mid 0$
$B \to UB \mid 1$
$F \to UA$
$G \to BZ$
$H \to ZB$
$I \to AU$
$Z \to 0$
$U \to 1$

Em relação à cadeia 1001001, temos então as seguintes árvores:



A árvore da esquerda é obtida usando a gramática original, sendo uma possível árvore para a cadeia. A árvore da direita é obtida usando a gramática na CNF, sendo portanto uma árvore binária (note-se que cada nó interno ou tem dois filhos que são também nós, ou um filho que é folha).

Exercício 2

Converta a seguinte gramática para a CNF.
$S \to aXYb \mid bYXa$
$X \to XaaX \mid \varepsilon$
$Y \to YbY \mid YcT \mid \varepsilon$
$T \to XY \mid Z$
$Z \to cT$

Novamente, vamos seguir os vários passos de simplificação. Começamos por eliminar das produções-ε, identificando que tanto X como Y podem produzir ε, e por isso são anuláveis. Por sua vez, T também é anulável uma vez que tem uma produção $T \to XY$, que é composta apenas por símbolos anuláveis. Então, é necessário eliminar as produções-ε e acrescentar novas produções, ficando a gramática nesta forma:
$S \to aXYb \mid bYXa \mid aYb \mid aXb \mid ab \mid bYa \mid bXa \mid ba$
$X \to XaaX \mid aaX \mid Xaa \mid aa$
$Y \to YbY \mid YcT \mid bY \mid Yb \mid b \mid cT \mid Yc \mid c$
$T \to XY \mid Z \mid X \mid Y$

$Z \to cT \mid c$

Agora, identificamos as produções unitárias, e vemos que existem em T três produções unitárias ($T \to A|B|S$), sendo as únicas existentes. Como não há problemas de ciclos, fazemos diretamente a substituição pelas produções das variáveis respetivas, ficando assim com a seguinte gramática:
$S \to aXYb \mid bYXa \mid aYb \mid aXb \mid ab \mid bYa \mid bXa \mid ba$
$X \to XaaX \mid aaX \mid Xaa \mid aa$
$Y \to YbY \mid YcT \mid bY \mid Yb \mid b \mid cT \mid Yc \mid c$
$T \to XY \mid cT \mid c \mid XaaX \mid aaX \mid Xaa \mid aa \mid YbY \mid YcT \mid bY \mid Yb \mid b \mid Yc$
$Z \to cT \mid c$

Agora identificamos símbolos não geradores ou não atingíveis. Neste caso, todos os símbolos geram terminais, pelo que todos são geradores. Começando em S, consegue-se atingir X e Y, e a partir de Y atinge-se T, mas nenhuma destas variáveis permite atingir Z, pelo que Z pode ser eliminada por ser uma produção não-atingível e portanto inútil.
$S \to aXYb \mid bYXa \mid aYb \mid aXb \mid ab \mid bYa \mid bXa \mid ba$
$X \to XaaX \mid aaX \mid Xaa \mid aa$
$Y \to YbY \mid YcT \mid bY \mid Yb \mid b \mid cT \mid Yc \mid c$
$T \to XY \mid cT \mid c \mid XaaX \mid aaX \mid Xaa \mid aa \mid YbY \mid YcT \mid bY \mid Yb \mid b \mid Yc$

Falta agora apenas o passo final de colocar todas as produções no formato da CNF. Começamos por criar novas variáveis para os terminais da linguagem:
$A \to a$
$B \to b$
$C \to c$

Substituindo na gramática as ocorrências de terminais não isolados pelas novas variáveis:
$S \to AXYB \mid BYXA \mid AYB \mid AXB \mid AB \mid BYA \mid BXA \mid BA$
$X \to XAAX \mid AAX \mid XAA \mid AA$
$Y \to YBY \mid YCT \mid BY \mid YB \mid b \mid CT \mid YC \mid c$
$T \to XY \mid CT \mid c \mid XAAX \mid AAX \mid XAA \mid AA \mid YBY \mid YCT \mid BY \mid YB \mid b \mid YC$

Agora, falta apenas adicionar novas variáveis para reduzir produções de tamanho maior do que 2. Uma solução possível seria:
$S \to JK \mid LM \mid AK \mid JB \mid AB \mid LA \mid BM \mid BA$
$X \to MJ \mid AJ \mid MA \mid AA$
$Y \to KY \mid YN \mid BY \mid YB \mid b \mid CT \mid YC \mid c$

$T \to XY \mid CT \mid c \mid MJ \mid AJ \mid MA \mid AA \mid KY \mid YN \mid BY \mid YB \mid b \mid YC$
$J \to AX$
$K \to YB$
$L \to BY$
$M \to XA$
$N \to CT$
$A \to a$
$B \to b$
$C \to c$

9.2 Propriedades de Fecho das CFLs

Tal como as linguagens regulares, existe um conjunto de propriedades que se aplicam às linguagens livres de contexto, incluindo um conjunto de operações que produzem como resultado também uma linguagem livre de contexto.

Exemplos de operações fechadas para as linguagens livres de contexto são a união, a concatenação, o fecho, o reverso, homomorfismo e homomorfismo inverso. A interseção com uma linguagem regular é uma operação fechada, muito embora a interseção com uma linguagem livre de contexto não resulte garantidamente numa linguagem livre de contexto.

Um exemplo poderá ser visto olhando para as linguagens $A = \{a^i b^j c^k, i = j\}$ e $B = \{a^i b^j c^k, j = k\}$, ambas linguagens livres de contexto (embora nenhuma das duas seja uma linguagem regular). A interseção destas duas linguagens será a linguagem $C = A \cap B = \{a^i b^j c^k, i = j = k\}$, a qual não é uma linguagem livre de contexto, uma vez que não é possível encontrar uma gramática ou um autómato de pilha para esta linguagem (mais à frente poderemos ver uma prova forma em como esta linguagem não é uma CFL).

Exercício 1

Considere as linguagens L1 e L2, sendo L1 definida pela gramática
$A \to 0A1 \mid \varepsilon$
e L2 por
$B \to aBb \mid \varepsilon$.
Apresente gramáticas para as linguagens $L3 = L1 \cup L2$, $L4 = L2.L1$, $L5 = L1^*$ e $L6 = L2^R$.

A operação de união entre duas CFLs é relativamente simples de fazer: a linguagem resultado vai ter as mesmas produções que as duas CFLs a unir

(com o devido cuidado no caso de haver nomes de variáveis repetidas entre as duas), mais uma nova variável, que será a variável de arranque da linguagem resultado, que vai ter como produções as antigas variáveis de arranque das linguagens originais. Então, L3 poderá ser definida pela seguinte gramática (com C como variável de arranque):
$C \to A \mid B$
$A \to 0A1 \mid \varepsilon$
$B \to aBb \mid \varepsilon$

A concatenação de duas CFLs é também simples de realizar, bastando acrescentar uma nova variável que contenha uma produçao que concatene as variáveis de arranque das duas gramáticas originais. Assim, L4 poderá ser definida pela seguinte gramática (com D como variável de arranque):
$D \to AB$
$A \to 0A1 \mid \varepsilon$
$B \to aBb \mid \varepsilon$

O fecho de uma CFL é também simples de fazer, bastando ter uma nova variável de arranque que permita ter 0 ou mais repetições da antiga variável de arranque da linguagem original. Então, L5 poderá ser definida pela seguinte gramática (com E como variável de arranque):
$E \to AE \mid \varepsilon$
$A \to 0A1 \mid \varepsilon$

O reverso de uma gramática é um pouco mais trabalhoso, sendo necessário reverter a ordem dos símbolos em todas as produções da gramática. Então, L6 pode ser definida pela seguinte gramática:
$B \to bBa \mid \varepsilon$

Exercício 2

Partindo do conhecimento que L1 é uma CFL e L2 não é uma CFL, e que $L1 \cap L3 = L2$ o que pode dizer acerca de L3?

Ora, sabemos que a interseção não é uma operação fechada para as CFLs, mas também sabemos que a interseção entre uma CFL e uma linguagem regular resulta numa CFL. Então, se sabemos que L2 não é uma CFL, e que L1 é uma CFL, sabemos que L3 não pode ser uma linguagem regular (se o fosse, então o resultado da interseção com L1 seria também uma CFL). Por

outro lado, não podemos afirmar mais nada sobre L3 (isto é, sobre se esta é ou não uma CFL).

Exercício 3 (Desafio 2015/16)

Partindo do conhecimento de que a linguagem $L1 = \{a^p b^p c^p b^q a^q : p, q \geq 0\}$ não é uma linguagem livre de contexto (CFL) e que a linguagem $L2 = 0^n 1^n : n \geq 0$ é uma CFL, prove, usando as propriedades de fecho das CFLs, que a linguagem $L3 = a^n b^n c^n : n \geq 0$ não é uma CFL.

Para fazer a prova, é necessário usar um conjunto de operações fechadas para as linguagens livres de contexto. Para podemos fazer a prova, temos de começar por ter todas as linguagens no mesmo alfabeto. Usando o homomorfismo, podemos, em L2, substituir os 0's por b's e os 1's por a's, obtendo assim a linguagem $L2' = \{b^n a^n : n \geq 0\}$. Observando as linguagens $L1$, $L2'$ e $L3$, podemos constatar que existe uma relação entre elas: $L1 = L3.L2'$. Como a concatenação é uma operação fechada para as CFLs, sabemos que se ambas $L3$ e $L2'$ fossem CFLs então $L1$ seria também uma CFL. No entanto, como sabemos que $L1$ não é uma CFL, então pelo menos um dos operandos também não será uma CFL. Como sabemos que $L2'$ é uma CFL (uma vez que foi obtida a partir de $L2$ aplicando uma operação fechada para as CFLs), então $L3$ não poderá ser uma CFL.

9.3 Propriedades de Decisão das CFLs

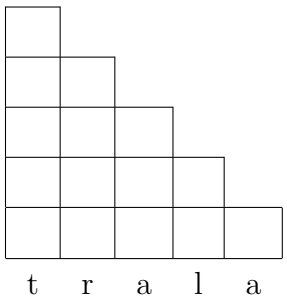
Entende-se por propriedades de decisão a capacidade de responder a um conjunto de questões relativamente a CFLs. Curiosamente, existem várias questões para as quais não são atualmente conhecidos algoritmos capazes de dar uma resposta. São exemplos disso questões relativas à ambiguidade de uma gramática ou de uma linguagem: não existe uma forma universal de provar que uma dada gramática (ou linguagem) é ambígua ou não. Da mesma forma, não existe também forma de provar se duas CFGs (ou PDAs) definem a mesma linguagem, ou se a interseção de duas CFGs (ou PDAs) resulta na linguagem vazia (ou seja, se não têm qualquer cadeia em comum).

Uma das questões respondidas aqui é relativamente à pertença de uma cadeia à linguagem definida por uma gramática. Um dos algoritmos usados para verificar essa pertença é o algoritmos de Cocke-Younger-Kasami (CYK),

o qual se baseia numa construção da árvore de análise da cadeia de baixo para cima. Este método necessita que a gramática da linguagem esteja na Forma Normal de Chomsky, algo que como foi visto anteriormente, é sempre possível de obter para qualquer CFG.

Começa-se por desenhar uma tabela triangular, tendo na base da tabela a cadeia a testar. Depois, essa tabela é preenchida de baixo para cima, verificando para cada posição da tabela qual ou quais as variáveis que podem dar origem às subcadeias respetivas. No final, caso a variável de arranque esteja presente no topo da tabela, significa que é possível obter a cadeia com a gramática. Vamos ver um exemplo, usando a gramática que foi já vista acima em CNF e testando a pertença da cadeia *trala*. Recordando a gramática já em CNF:
$S \to BC \mid LC \mid LA \mid BA$
$X \to LC \mid LA$
$B \to TR$
$C \to AX$
$T \to t$
$R \to r$
$A \to a$
$L \to l$

Começamos então por desenhar uma tabela triangular que na base tem 5 casas, tantas quantas o número de símbolos na cadeia.



Podemos interpretar o conteúdo de cada célula como expresso na seguinte tabela, em que a célula X_{ij} vai conter as variáveis que permitem gerar a subcadeia i-j. Por exemplo, a célula X_{24} será preenchida com as variáveis que permitam gerar a subcadeia 2-4, ou seja, *ral*. Então, a célula X_{15} irá conter as variáveis capazes de gerar a cadeia completa, daí que se esta célula contiver a variável de arranque da gramática, significa que a cadeia pertence à gramática.

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}
t	r	a	l	a

Começamos então por preencher a linha de baixo da tabela com a variável ou variáveis que permitem gerar a sub-cadeia respetiva, ou seja, o símbolo que está debaixo dessa célula. Neste exemplo, olhando para a gramática, temos apenas uma variável capaz de gerar cada um dos símbolos, pelo que preenchemos as células com esses símbolos respetivos (note-se que como podemos ter mais do que uma variável, devemos sempre usar conjuntos):

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
X_{12}	X_{23}	X_{34}	X_{45}	
{T}	{R}	{A}	{L}	{A}
t	r	a	l	a

Passando agora para a segunda linha da tabela, vamos ver quais as variáveis que permitem gerar sub-sequências de comprimento dois. Para isso, vamos olhar para a célula imediatamente abaixo e para a célula em baixo à direita. Então, para X_{12} temos de olhar para as células X_{11} e X_{22}, e ver todas as variáveis que permitam gerar uma qualquer das possíveis combinações (sequências) de variáveis. Neste caso, temos apenas uma combinação, TR, e olhando para a gramática, apenas uma variável que permite produzir TR: B. X_{12} fica então preenchido com {B}. Da mesma forma, X_{23} irá ter as variáveis que permitem obter RA (não havendo nenhuma, a célula fica vazia). X_{34} irá conter as variáveis capazes de gerar AL (novamente, nenhuma), e finalmente X_{45} irá conter as variáveis capazes de gerar LA (existem duas, S e X). Ficamos então com a seguinte tabela:

X_{15}				
X_{14}	X_{25}			
X_{13}	X_{24}	X_{35}		
$\{B\}$			$\{S, X\}$	
{T}	{R}	{A}	{L}	{A}
t	r	a	l	a

Nas linhas seguintes, já fica um pouco mais complexo: como as sub-cadeias já terão comprimento maior do que dois, temos de ver todas as hipóteses de gerar essa sub-cadeia. Isso implica que não iremos ver apenas duas células, mas vários pares de células capazes de gerar cada sub-cadeia. As células a consultar ficam sempre na vertical e na diagonal da célula a analisar, e uma está tanto mais próxima como a outra estará mais distante.

No caso da célula X_{13}, vamos olhar para o par constituído pela célula imediatamente abaixo (X_{12}) e pela mais distante na diagonal (X_{33}), e depois para o par constituído pela célula mais abaixo verticalmente (X_{11}), e mais próxima diagonalmente (X_{23}). No primeiro caso, temos apenas uma combinação possível, que resulta na sequência BA, a qual pode ser produzida pela variável S. No segundo caso, não temos nenhuma sequência possível, dado que a célula X_{23} está vazia.

No caso da célula X_{24}, vamos olhar para o par X_{23} e X_{44}, e depois para o par X_{22} e X_{34}. Tanto no primeiro como no segundo casos não existem quaisquer sequências possíveis, dado que as célula X_{23} e X_{34} estão vazias.

Finalmente, para a célula X_{35}, vamos olhar para o par X_{34} e X_{55} e depois para o par X_{33} e X_{45}. No primeiro caso temos uma célula vazia, pelo que não temos nenhuma possibilidade. No segundo caso, temos duas combinações possíveis, as sequências AS e AX, sendo que apenas a variável C é capaz de gerar a sequência AX. Ficamos então com a seguinte tabela:

X_{15}				
X_{14}	X_{25}			
$\{S\}$		$\{C\}$		
$\{B\}$			$\{S, X\}$	
{T}	{R}	{A}	{L}	{A}
t	r	a	l	a

Passando para a linha seguinte, vamos então olhar para as duas células.

No caso da célula X_{14}, temos de analisar o par X_{13} e X_{44}, seguido do par X_{12} e X_{34}, seguido do par X_{11} e X_{24}. No primeiro par, temos apenas uma combinação que gera a sequência SL. No entanto, não há nenhuma variável capaz de gerar esta sequência, pelo que não teremos aqui nenhuma variável. No segundo par, temos uma célula vazia, pelo que não teremos também nenhuma variável a acrescentar. Finalmente, no terceiro par, temos também uma célula vazia. Como nenhum dos pares tem nenhuma possibilidade de geração, esta célula irá ficar vazia.

No caso da célula X_{25}, temos de analisar o par X_{24} e X_{55}, seguido do par X_{23} e X_{45}, seguido do par X_{22} e X_{35}. Tanto no primeiro par como no segundo deparamo-nos com uma célula vazia, pelo que estes pares não irão contribuir com nenhuma variável. No terceiro par, temos uma combinação possível, que resulta na sequência RC, a qual, no entanto, não tem qualquer variável capaz de a gerar, ficando esta célula também vazia.

Não havendo alterações na tabela, passamos finalmente para a última linha, olhando para a célula X_{15}. Temos aqui de analisar o par X_{14} e X_{55}, seguido do par X_{13} e X_{45}, seguido do par X_{12} e X_{35}, seguido do par X_{11} e X_{25}. Para o primeiro caso, temos uma célula vazia (X_{14}), pelo que não resulta em nenhuma variável. No segundo caso, temos duas combinações possíveis, as sequências SS e SX, sendo, no entanto, que nenhuma delas é possível de gerar com a gramática. No terceiro caso, temos uma combinação, a sequência BC, a qual é possível de gerar com a variável S. Finalmente, no quarto caso, temos também uma célula vazia, pelo que não teremos mais nenhuma variável a acrescentar. Ficamos então com a seguinte tabela no final:

$\{S\}$				
$\{S\}$		$\{C\}$		
$\{B\}$			$\{S, X\}$	
{T}	{R}	{A}	{L}	{A}
t	r	a	l	a

Como a célula do topo da tabela contém a variável de arranque da gramática (S), podemos concluir que a cadeia efetivamente pertence à linguagem definida por esta gramática.

9.4 Lema da Bombagem para CFLs

Tal como acontece com o Lema da Bombagem para as linguagens regulares, também para as CFLs existe uma forma de provar que uma linguagem não é uma linguagem livre de contexto. O lema da bombagem para CFLs funciona de forma semelhante ao lema da bombagem para linguagens regulares, com as necessárias alterações.

À semelhança do que acontecia nas linguagens regulares, também aqui caso L seja uma linguagem livre de contexto, então L segue o Lema da Bombagem para CFLs. Novamente, aquilo que nos interessa é provar que uma linguagem não é uma CFL, o que acontece quando a linguagem não segue o Lema da Bombagem para CFLs.

O lema aqui baseia-se no facto de, numa gramática finita, cadeias muito longas necessitarem de provocar repetição de produções. Pensando em termos da árvore de análise de uma cadeia longa, haverá necessariamente partes internas da árvore com estrutura repetida, e é essa repetição que é explorada no lema.

Então, aquilo que o lema nos diz é que se L for uma CFL, então existe uma constante m (um tamanho mínimo, dependente da linguagem), tal que todas as cadeias z pertencentes a L e com um tamanho maior ou igual a m podem ser decompostas em cinco partes ($z = uvwxy$), em que $vx \neq \varepsilon$ (ou seja, pelo menos um dos elementos v e x será não vazio), e $|vwx| \leq m$ (ou seja, a parte do meio da cadeia não é demasiado comprida), tal que para $k \geq 0$, as cadeias geradas por repetição de v e x, no formato uv^kwx^ky, também pertencem a L. De uma forma simplificada, será algo como:

$$\exists m : \forall z \in L, |z| \geq m : \exists z = uvwxy, vx \neq \varepsilon, |vwx| \leq m : \forall k : uv^kwx^ky \in L$$

Para fazer a prova, temos de negar o lema, o que implica inverter todos os quantificadores. Então, temos que para uma linguagem L não seguir o lema, qualquer que seja o valor de m (por muito grande que seja), existe pelo menos uma cadeia z pertencente a L com comprimento maior ou igual a m, tal que qualquer que seja a divisão dessa cadeia em cinco partes, existem pelo menos um k tal que uv^kwx^ky não pertence a L. Novamente, de uma forma simplificada, será algo como:

$$\forall m : \exists z \in L, |z| \geq m : \forall z = uvwxy, vx \neq \varepsilon, |vwx| \leq m : \exists k : uv^kwx^ky \notin L$$

Vamos ver um exemplo para a linguagem L = $\{\ vv : v \in \{0,1\}^*\ \}$, ou seja cadeias binárias de comprimento par que podem ser divididas em duas metades iguais.

Se tentássemos construir um PDA ou uma CFG para a linguagem iríamos ver que não seria possível fazê-lo. Vamos então ver uma prova mais formal em como L não é uma CFG.

Começamos por escolher uma cadeia que pertença a L e cujo comprimento seja maior do que m, como por exemplo a cadeia $z = 0^m1^m0^m1^m$, com comprimento $4m > m$. Temos agora de ver as divisões possíveis da cadeia tal que $z = uvwxy$ em que $vx \neq \varepsilon$ e $|vwx| \leq m$:

- No caso de vwx abranger apenas a primeira ou apenas a segunda metade da cadeia, então bombeamentos com $k \neq 1$ irão originar cadeias não pertencentes a L (a outra metade da cadeia irá permanecer inalterada, e não acompanha a parte bombeada).

- No caso de vwx abranger parte da primeira metade da cadeia e parte da segunda metade, então um bombeamento também causará cadeias não pertencentes à linguagem (poderão ser bombeados 1's da primeira metade da cadeia e/ou 0's da segunda metade da cadeia, não sendo acompanhados pelas partes que se mantém inalteradas).

Na realidade a única hipótese de continuar a gerar cadeias pertencentes à linguagem seria se v contivesse um ou mais zeros (ou uns) da primeira metade da cadeia e x o mesmo número de zeros (ou uns) da segunda metade da cadeia. No entanto, para isso acontecer, vwx seria necessariamente maior do que m (comprimento mínimo de $m + 2$ para abranger um mínimo de um dígito de cada lado), pelo que tal divisão não é possível.

Então, como vimos que para qualquer divisão possível da cadeia, existe sempre pelo menos um valor de k que gera cadeias não pertencentes à linguagem, provamos que a linguagem L não é uma CFL.

Exercício 1 (TPC 2011/12)

Considere a linguagem $L1 = \{0^n10^n1, n \geq 0\}$. Mostre que L1 obedece ao Lema da Bombagem para CFLs. L1 é uma CFL?

Para provar que a linguagem segue o Lema, devemos encontrar um valor de m, a partir do qual todas as cadeias da linguagem tenham uma divisão que respeite as condições impostas pelo Lema, e que resulte em cadeias aceites ao bombear as partes repetíveis da cadeia. Neste caso concreto, podemos fazer a prova para m = 4. Cadeias com tamanho superior ou igual a 4 terão o formato $0^a10^a1, a \geq 1$, as quais podem ser divididas da seguinte forma: $u = 0^{a-1}$; $v = 0$; $w = 1$; $x = 0$; $y = 0^{a-1}1$. Com esta divisão, válida para qualquer

cadeia com tamanho igual ou superior a 4, qualquer que seja o valor de k, a cadeia uv^kwx^ky continua a pertencer à linguagem L1. Para $k = 0$, teremos a cadeia $uwy = 0^{a-1}10^{a-1}1$; como $a \geq 1$, resulta numa cadeia pertencente a L1. Para $k > 1$, teremos cadeias no formato $0^{a-1}0^k10^k0^{a-1}1 = 0^{a+k-1}10^{a+k-1}1$, as quais pertencem também a L1.

Embora esta prova em como a linguagem segue o Lema da Bombagem não seja prova de que a linguagem é uma CFL, podemos tentar criar uma CFG ou um PDA para a linguagem, provando assim que a linguagem é uma CFL. Podemos ter então a seguinte gramática:

$S \to A1$

$A \to 0A0 \mid 1$

Exercício 2 (TPC 2011/12)

Prove, usando o Lema da Bombagem para CFLs, que a linguagem $L = \{0^n1^n0^n1^n, n \geq 0\}$ não é uma CFL.

Ora para provar que L não é uma CFL usando o Lema da Bombagem começamos por escolher uma cadeia da linguagem. Vamos escolher $z = 0^m1^m0^m1^m$, a qual tem comprimento $4m$, e por isso respeita as condições do Lema. Temos agora de verificar todas as divisões possíveis da cadeia em cinco partes. Dadas as limitações da divisão, nomeadamente a condição de que $|vwx| \leq m$, a parte 'do meio' da cadeia (vwx) não pode conter mais do que duas 'partes' consecutivas da cadeia (considerando cada dígito como uma parte, a cadeia terá então 4 partes), podendo conter apenas uma ou duas delas.

- No caso de vwx incluir apenas um dos dígitos, repetições de v e x irão gerar cadeias em que esse dígito se repete mais vezes do que os restantes, gerando assim cadeias que não pertencem a L.

- No caso de vwx incluir dois dígitos, repetições de v e x irão gerar cadeias em que essas duas partes irão ter tamanho superior às restantes duas partes.

Estes dois casos abrangem então todas as divisões possíveis da cadeia, pelo que conseguimos provar que L não segue eo Lema da Bombagem, e não é portanto uma CFL.

Exercício 3

Tente provar, usando o Lema da Bombagem para CFLs, que a linguagem

L=$\{a^n b^n : n \geq 0\}$ não é uma CFL.

Começamos novamente por escolher uma cadeia pertencente à linguagem, como por exemplo $z = a^m b^m$, que tem comprimento $2m$. De seguida, temos de ver todas as divisões possíveis da cadeia em cinco partes.

Uma das divisões possíveis é $z = uvwxy$ em que $u = a^{n-1}$, $v = a$, $w = \varepsilon$, $x = b$ e $y = b^{n-1}$. Para $k = 0$, a nova cadeia será $z' = uwy = a^{n-1}b^{n-1}$, que pertence à linguagem; para $k \geq 1$, teremos as novas cadeias como $z' = uv^k wx^k y$, o que resulta em $a^{n-1+k}b^{n-1+k}$, que também pertencem à linguagem. Então, havendo uma divisão para a qual todos os valores de k resultam em cadeias pertencentes à linguagem, significa que o Lema é cumprido no caso da cadeia escolhida.

Poder-se-ia tentar escolher outra cadeia com a qual fazer a prova, mas qualquer todas as cadeias terão o mesmo formato que a escolhida, pelo que será sempre possível realizar a divisão da cadeia como feito acima, e mostrar que o Lema é seguido. Não conseguimos então provar que a linguagem não é uma CFL (efetivamente, se tentarmos obter um PDA ou uma CFG para a linguagem, facilmente podemos ver que é possível fazê-lo e que a linguagem é, de facto, uma CFL).

Exercício Proposto 1 Encontre árvores de análise alternativas (gramática original e na CNF) às obtidas no **Exercício 1** da secção Simplificação de CFGs e Forma Normal de Chomsky para a mesma cadeia *1001001*.

Exercício Proposto 2 Considerando a gramática do mesmo **Exercício 1** da secção Simplificação de CFGs e Forma Normal de Chomsky, verifique, usando o algoritmo CYK, se as cadeias 10110 e 0011 pertencem à linguagem definida pela gramática.

Capítulo 10

Máquinas de Turing

As Máquinas de Turing (TM, do inglês *Turing Machine*) são autómatos capazes de reconhecer ou processar qualquer tipo de linguagem. Podem ser vistas como uma máquina com uma cabeça de processamento que a cada momento se encontra numa posição de uma fita unidimensional infinita, e que se pode deslocar para a esquerda ou para a direita, mantendo ou alterando o símbolo presente na fita.

Formalmente uma TM é constituída por um conjunto Q de estados, um conjunto Σ de símbolos do alfabeto, um conjunto Γ de símbolos da fita, as transições δ associadas, o estado inicial $q_0 \in Q$, o símbolo (pertencente a Γ) que representa a inexistência de informação na fita (Branco) e o conjunto de estados finais $F \subseteq Q$. Uma transição é dependente do estado em que a máquina se encontra e qual o símbolo que está presente na posição atual da fita, indicando qual o estado de destino, o símbolo que passará a substituir o símbolo atual na fita, e a direção de movimentação da cabeça (esquerda ou direita). Formalmente, TM $= (Q, \Sigma, \Gamma, \delta, q_0, B, F)$, sendo as transições definidas como $\delta(q, X) = (p, Y, D)$, em que $q, p \in Q$, $X, Y \in \Gamma$, e D indica a direção de movimento (esquerda ou direita).

Uma TM pode ser representada por um autómato em que em cada transição se indica o símbolo atual na fita, o símbolo pelo qual é substituído e a direção de movimento da cabeça. De forma a conseguir acompanhar a evolução da computação numa TM, pode-se também apresentar a sequência de descrições instantâneas, sendo que em cada descrição instantânea se apresenta o conteúdo da fita e o estado atual da cabeça, sendo este representado imediatamente à esquerda do símbolo atual, de forma a transmitir também a sua posição.

Vejamos um exemplo de uma TM para reconhecimento de cadeias no formato $a^n b^n c^n, n \geq 1$, que sabemos não ser uma CFL e por isso impossível

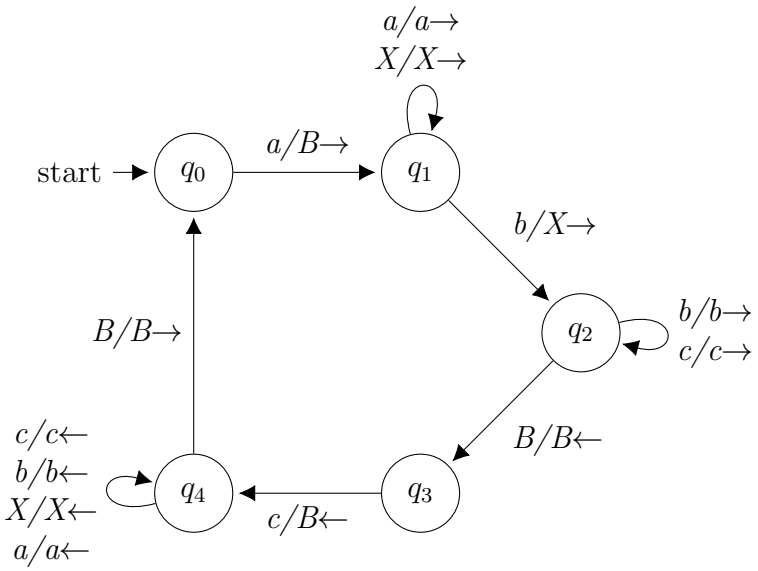
de modelar usando autómatos de pilha ou gramáticas livres de contexto.

O primeiro passo para construir uma TM passa por delinear um algoritmo que permita resolver o problema tendo em conta o funcionamento de uma TM. Uma possível solução consiste em fazer várias passagens pela cadeia de forma a garantir que o número de a's, b's e c's é igual, marcando as partes da cadeia já processadas de forma a evitar contar duas vezes a mesma entrada. Como não existe preocupação em garantir que no final do processamento a cadeia se mantenha inalterada, podemos optar por apagar 'as pontas' da cadeia (os a's e os c's) e marcar apenas os b's já processados. Olhando para a cadeia $aaabbbccc$ como exemplo, a ideia será obter uma TM que permita os seguintes passos, assumindo o símbolo X como marca para os b's já processados:

$$BaaabbbcccB$$
$$BBaaXbbccBB$$
$$BBBaXXbcBBB$$
$$BBBBXXXBBBB$$

Neste ponto final, não havendo mais a's ou c's e se não houver mais b's (isto é, todos estarão transformados em X's), então a cadeia será aceite.

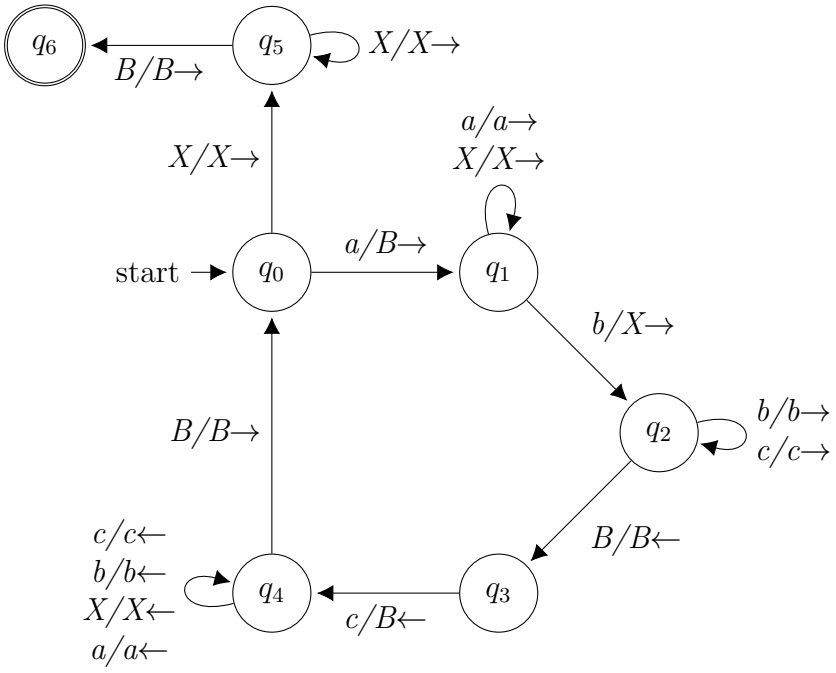
Tendo um algoritmo que funcione para a resolução do problema, podemos passar para o segundo passo, que consiste na implementação do algoritmo usando uma TM. Uma das 'partes' da máquina terá de lidar com as passagens iniciais pela cadeia, apagando o primeiro a, marcando o primeiro b que encontrar, e apagando o último c. Isso pode ser realizado pela seguinte TM parcial:



Analisando esta parte da TM, podemos ver que, partindo de q_0, o primeiro a é reconhecido e apagado (substituído por B). Depois, em q_1, a máquina anda para a direita 'vendo' a's ou X's até encontrar o primeiro b, o qual substitui por X, passando para q_2. Uma vez em q_2, a máquina continua para a direita através de b's e c's até ao final da cadeia, altura em que, passando para q_3, apaga o último c, passando para q_4. Em q_4 voltamos atrás, até atingir o início da cadeia, altura em que volta a q_0.

Falta-nos agora garantir que caso não seja encontrado mais nenhum a, todos os b's foram convertidos em X's, e todos os c's foram apagados.

Para isso, temos de acrescentar essa possibilidade, acrescentando uma nova possibilidade em q_0 de encontrar logo um X e não um a. Uma possibilidade de solução seria então:



Embora na primeira fase não seja validado que o formato é o correto (por exemplo, a cadeia *aaabbbcbccc* poderia ser processada), este último passo completa as garantias relativas ao formato e tamanho da cadeia, passando ao estado de aceitação (q_6) apenas nos casos devidos. Quando a cadeia não está no formato requerido, haverá um momento em que não há mais transições possíveis. Por exemplo, se a cadeia não terminar em c, em q_3 não haverá transição possível, e a TM morre nesse ponto; não se tratando de um estado de aceitação, a cadeia não é aceite.

A Máquina de Turing pode ainda ser representada de forma tabular, colocando num eixo os estados e no outro os símbolos do alfabeto da fita,

sendo as células preenchidas com as transições respetivas. Neste exemplo, teríamos a seguinte tabela:

	a	b	c	X	B
$\rightarrow q_0$	(q_1, B, R)			(q_5, X, R)	
q_1	(q_1, a, R)	(q_2, X, R)		(q_1, X, R)	
q_2		(q_2, b, R)	(q_2, c, R)		(q_3, B, L)
q_3			(q_4, B, L)		
q_4	(q_4, a, L)	(q_4, b, L)	(q_4, c, L)	(q_4, X, L)	(q_0, B, R)
q_5				(q_5, X, R)	(q_6, B, R)
$*q_6$					

Os estados estão indicados nas linhas, enquanto os símbolos da fita estão presentes nas colunas. Podemos ver que cada célula contém o destino da transição correspondente. Foi utilizado L e R para indicar movimento da cabeça para a esquerda e direita, respetivamente. Uma célula em branco significa que não existe transição definida, sendo que a máquina termina a sua execução se se encontrar numa configuração que não permita qualquer transição.

Vamos ainda ver a sequência de descrições instantâneas para o processamento das cadeias *abc* e *aabcbc*:

$Bq_0abcB \vdash BBq_1bcB \vdash BXq_2cB \vdash BXcq_2B \vdash BXq_3cB \vdash Bq_4XBB \vdash Bq_4BXB \vdash Bq_0XB \vdash BXq_5B \vdash BXBq_6B$

Neste momento, não havendo mais nenhuma transição possível, o processamento termina. Dado que q_6 é um estado de aceitação, a cadeia *abc* é aceite pela TM.

Note-se que a indicação do símbolo B é opcional, podendo ser omitida exceto quando se encontra imediatamente à direita da cabeça da máquina.

$q_0aabcbc \vdash q_1abcbc \vdash aq_1bcbc \vdash aXq_2cbc \vdash aXcq_2bc \vdash aXcbq_2c \vdash aXcbcq_2B \vdash aXcbq_3c \vdash aXcq_4b \vdash aXq_4cb \vdash aq_4Xcb \vdash q_4aXcb \vdash q_4BaXcb \vdash q_0aXcb \vdash q_1Xcb \vdash Xq_1cb$

Neste momento, não existe transição definida, pelo que o processamento termina. Como q_1 não é um estado de aceitação, esta cadeia não é aceite.

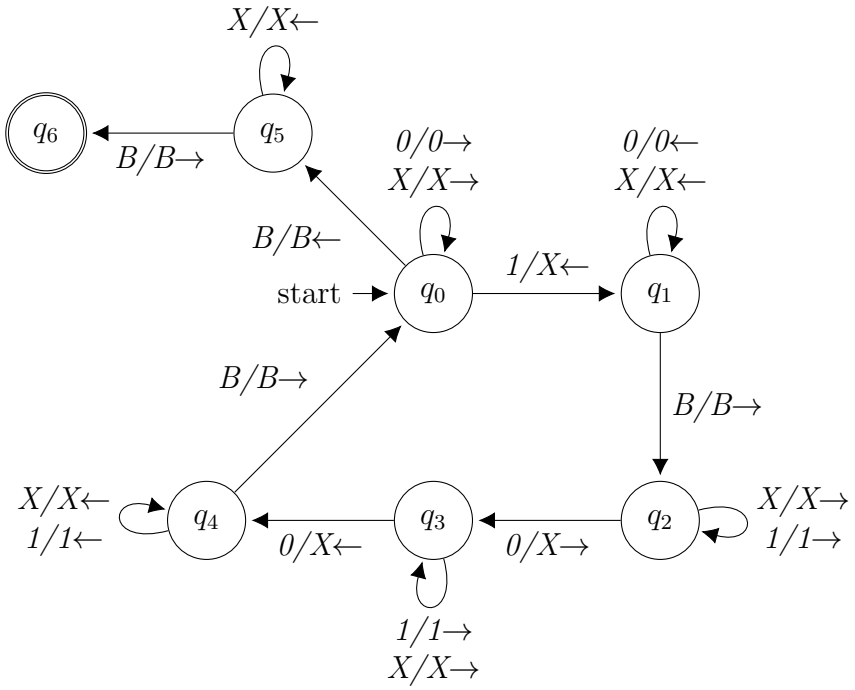
Exercício 1

Construa uma TM que permita reconhecer cadeias pertencentes à lingugem das cadeias binárias em que o número de zeros é o dobro do número de uns.

Novamente, para obtenção de uma solução começamos por delinear um algoritmo que permita resolver o problema, tratando depois da implementação. Como possível algoritmo para solução deste problema, podemos por exemplo realizar várias passagens pela cadeia, em que em cada passagem é feita a deteção de um 1, marcando-o como processado, e, voltando ao início, detetar dois 0's, marcando-os também. Novamente, será necessário no final garantir que caso não existam mais 1's também não existirão mais 0's.

Embora este algoritmo não seja muito eficiente, dado que regressa sempre ao início da cadeia, e obriga à deteção dos 1s e 0s na mesma ordem, torna-se mais simples de implementar.

Passando então para a implementação do algoritmo, podemos ter a seguinte TM como possível solução:



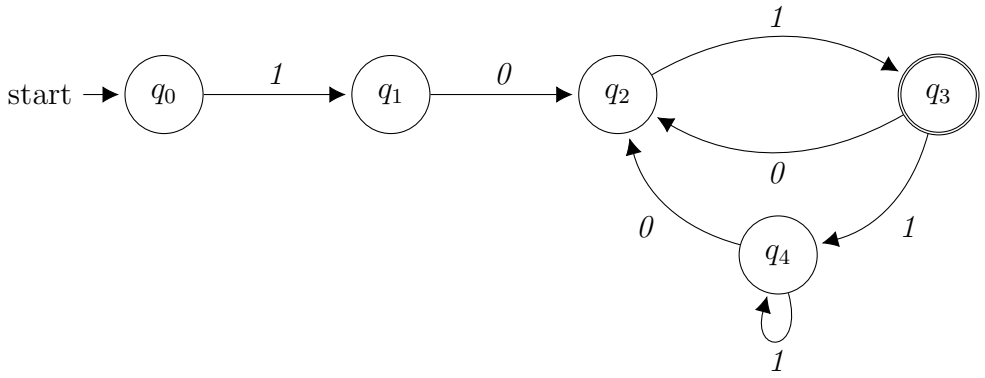
Podemos ver que começamos em q_0 por procurar o primeiro 1, avançando através dos possíveis 0's e X's existentes. Depois de identificar o 1, em q_1 volta-se ao início da cadeia, passando em q_2 a procurar o primeiro 0 não processado, e em q_3 o segundo 0, sendo que em q_4 voltamos novamente ao

início da cadeia, para prosseguir para a próxima iteração. No caso de não existência de mais 1's, passa-se de q_0 para q_5, onde se verifica que não sobraram 0's, passando a q_6 apenas nesse caso.

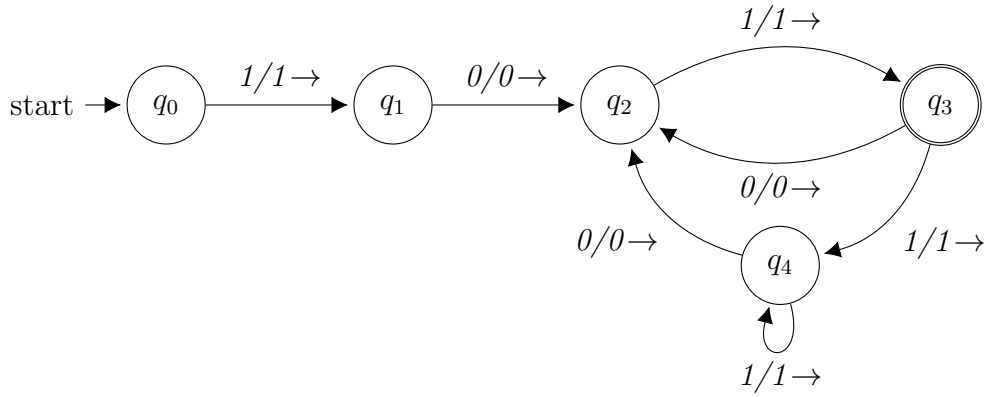
Exercício 2

Apresente uma Máquina de Turing que permita reconhecer cadeias binárias que iniciem em 10, terminem em 01, e não possuam nenhuma sequência com dois 0's.

A linguagem pedida neste exercício é uma Linguagem Regular, e como tal é possível definir um DFA para a reconhecer. De facto, este problema foi já visto no **Exercício 3** do segundo capítulo, em que foi produzido um DFA, que se recorda aqui (já sem o estado q_5, que não permite levar à aceitação):



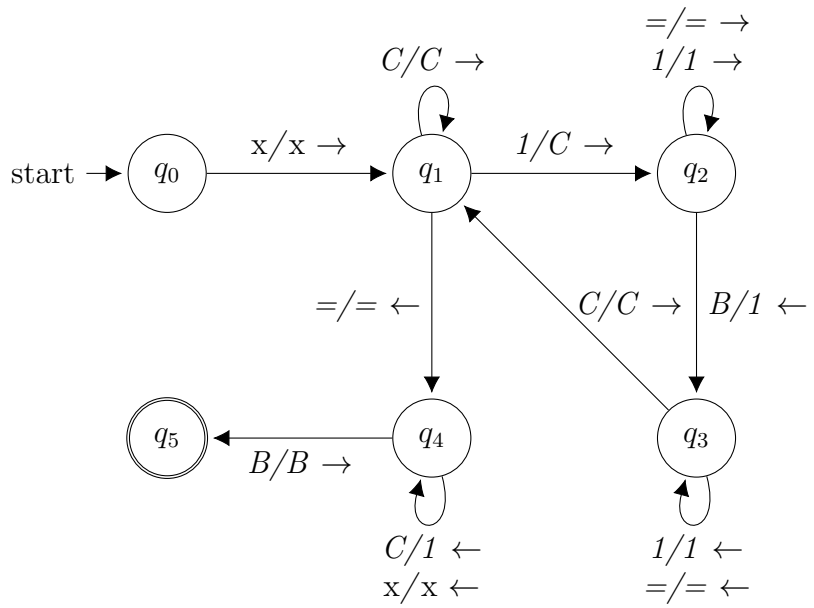
Uma TM para linguagens regulares pode ser vista como um DFA que mantém a cadeia na fita ao invés de a consumir. Então, modificando apenas as etiquetas das transições do diagrama do DFA, é possível obter o diagrama de uma TM para reconhecimento da mesma linguagem que o DFA original. Podemos então ficar com a seguinte TM:



Uma vez que a TM vai parar apenas quando não houver mais movimentos possíveis, isso acontecerá quando aparecer um 'Branco', sendo a cadeia aceite caso esteja em q_3 nesse momento, e não sendo aceite caso contrário.

Exercício 3 (Exame 2014/15)

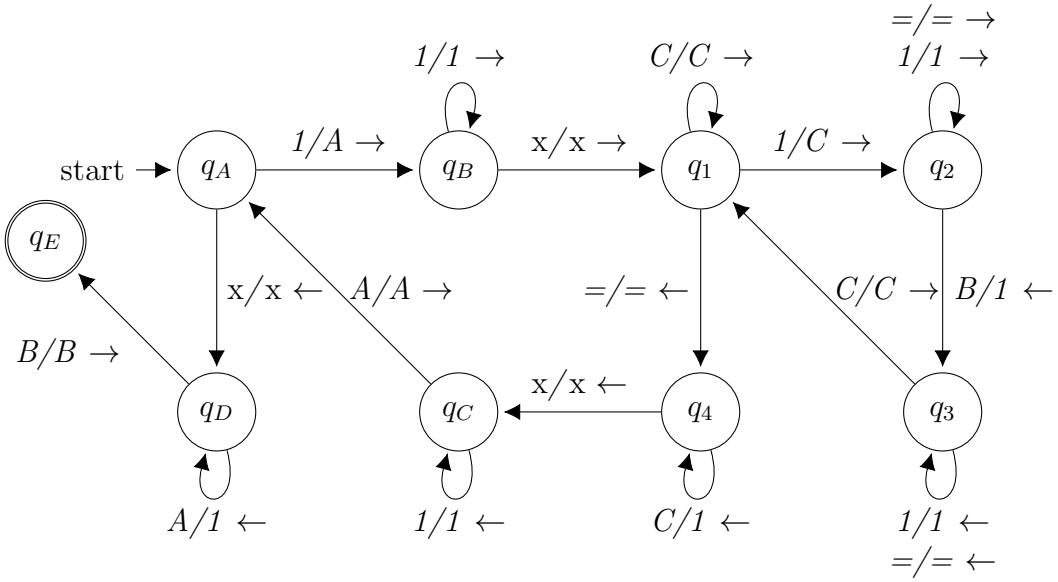
Pretende-se uma Máquina de Turing (TM) que realize a multiplicação de dois números em unário (em unário, os números são representados pela quantidade de 1's existentes; por exemplo, o número 3 em unário é representado por 111). A TM abaixo apresenta uma parte da solução.
a) Indique o que faz a TM apresentada e exemplifique, mostrando a sequência de descrições instantâneas da máquina quando a entrada na fita é x11=.
b) Baseando-se na TM apresentada, desenhe uma Máquina de Turing que implemente a multiplicação de dois números em unário. A entrada na fita é no formato N1xN2=, devendo no final a fita ficar com N1xN2=N3 em que N3 é a multiplicação de N1 por N2 em unário. Exemplo: 111x11= deverá resultar em 111x11=111111.



Começando então por analisar a máquina apresentada, podemos ver que ela inicia por reconhecer o x (interpretado aqui como sendo o sinal de multiplicação). É depois executado um ciclo em que são lidos C's em q_1 e o primeiro 1 é substituído por um C, passando em q_2 o resto da entrada até ao final (símbolos 1 e =), altura em que é inserido um novo 1 na cadeia, sendo que em q_3 se regressa até ao ponto em que o último 1 foi substituído por um C, voltando a q_1 e repetindo este ciclo até que não hajam mais 1's antes do sinal de igualdade. Nesta altura, passamos para q_4, percorrendo a cadeia para a esquerda e substituíndo os C's novamente por 1's.

Abstraíndo este funcionamento para uma linguagem de mais alto-nível, podemos dizer que esta TM copia os 1's entre o x e o = para depois do =. Vendo então o exemplo da cadeia x11=, o resultado será x11=11, como podemos ver pela sequência de descrições:
q_0x11= $\vdash$ x$q_1$11= $\vdash$ x$Cq_2$1= $\vdash$ xC1q_2= $\vdash$ xC1=q_2B $\vdash$ xC1q_3=1 $\vdash$ x$Cq_3$1=1 $\vdash$ xq_3C1=1 $\vdash$ x$Cq_1$1=1 $\vdash$ xCCq_2=1 $\vdash$ xCC=$q_2$1 $\vdash$ xCC=1q_2B $\vdash$ xCC=$q_3$11 $\vdash$ xCCq_3=11 $\vdash$ xCq_3C=11 $\vdash$ xCCq_1=11 $\vdash$ xCq_4C=11 $\vdash$ xq_4C1=11 $\vdash$ q_4x11=11 $\vdash$ q_4Bx11=11 $\vdash$ q_5x11=11

Em relação à TM completa, e de forma a reaproveitar a TM apresentada, podemos pensar em utilizá-la como uma rotina a invocar por cada 1 presente na parte à esquerda do sinal de multiplicação, ou seja, repetindo o segundo operando tantas vezes quantas o número de 1's no primeiro operando.



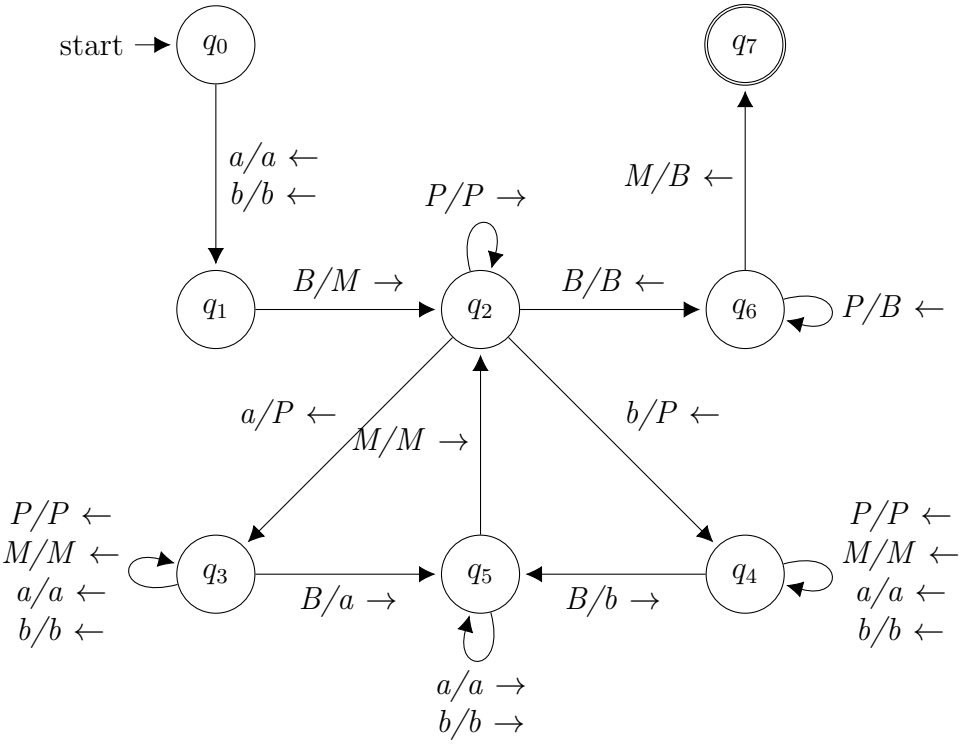
Repare-se que os estados q_0 e q_5 foram substituídos por q_B e q_C, respetivamente, com alterações ligeiras nas suas ligações, dada a necessidade de fazer a ponte entre a sub-rotina e a parte restante da TM, e para que a primeira seja reutilizável no contexto da TM geral.

Podemos ver entre q_A, q_B e q_C o mesmo padrão de ciclo, sendo 'invocada' a sub-TM em cada ciclo, ou seja, uma vez por cada 1 presente à esquerda do x. Finalmente, quando termina este ciclo, e é detetado um x em q_A passa-se para q_D, onde é feita a substituição dos A's de volta para 1's, deixando no final a cadeia inalterada.

Exercício 4 (TPC 2010/11)

Considere uma máquina de Turing que inverta uma sequência de símbolos no alfabeto Σ={a,b} existente na fita. Ex: BbabaB → BababB (B representa células vazias).
a) Desenhe a Máquina de Turing correspondente.
b) Apresente o traço de computação quando a entrada na fita é *bba*.

Para inverter a sequência na fita, podemos ir copiando cada um dos símbolos para a esquerda da cadeia original, em ordem inversa, apagando no final a cadeia original. Uma TM que poderá fazer isso poderá ser a seguinte:



Olhando para a TM acima, podemos ver que começa por andar para a esquerda, inserindo (em q_1) um marcador (símbolo M) para separar a parte original da cadeia da sua réplica invertida. Voltando depois para a direita, em q_2 verifica-se qual o próximo símbolo não processado (a ou b), marcando-o como processado (símbolo P) e transitando para q_3 ou q_4 consoante o símbolo lido. Em ambos estes estados, percorre-se a cadeia para a esquerda até encontrar uma célula vazia, inserindo nesse local o símbolo lido anteriormente e transitando para q_5. Percorre-se novamente a cadeia para a direita até chegar ao marcador de separação das cadeias, transitando novamente para q_2, e repetindo o ciclo até que não hajam mais símbolos por processar. Nessa altura, transita-se para q_6, percorrendo a cadeia original e apagando os símbolos P e M, terminando em q_7 em caso de sucesso.

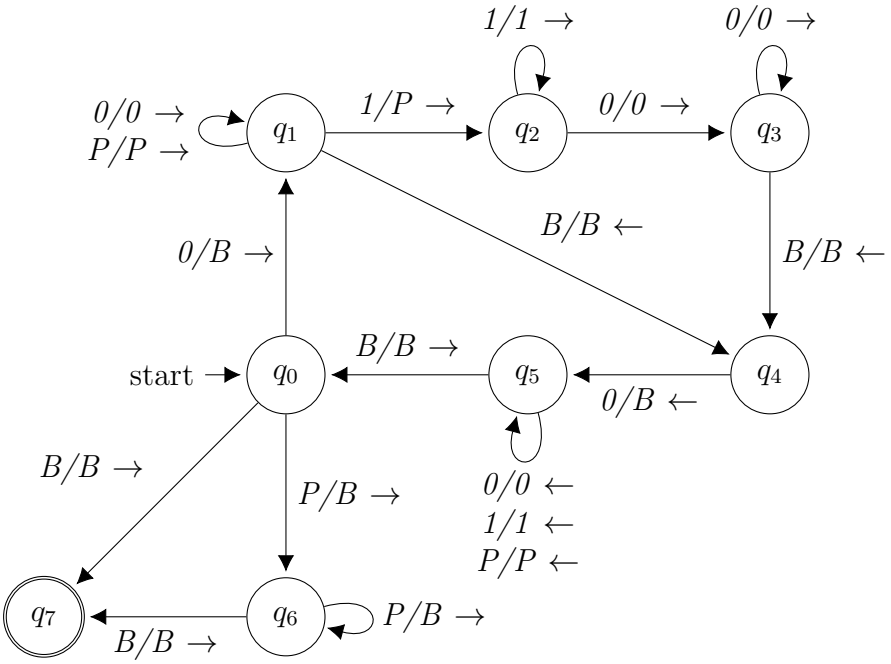
Para a entrada bba, teremos a seguinte sequência de descrições instantâneas:
$q_0bba \vdash q_1Bbba \vdash Mq_2bba \vdash q_4MPba \vdash q_4BMPba \vdash bq_5MPba \vdash bMq_2Pba \vdash bMPq_2ba \vdash bMq_4PPa \vdash bq_4MPPa \vdash q_4bMPPa \vdash q_4BbMPPa \vdash bq_5bMPPa \vdash bbq_5MPPa \vdash bbMq_2PPa \vdash bbMPq_2Pa \vdash bbMPPq_2a \vdash bbMPq_3PP \vdash bbMq_3PPP \vdash bbq_3MPPP \vdash bq_3bMPPP \vdash q_3bbMPPP \vdash q_3BbbMPPP \vdash aq_5bbMPPP \vdash abq_5bMPPP \vdash abbq_5MPPP \vdash abbMq_2PPP \vdash abbMPq_2PP \vdash abbMPPq_2P \vdash abbMPPPq_2B \vdash abbMPPq_6P \vdash abbMPq_6P \vdash abbMq_6P \vdash abbq_6M \vdash abq_7b$

Como podemos ver, o traço de computação é algo longo, mesmo para uma cadeia relativamente curta, dado que são necessárias várias passagens para fazer a cópia da cadeia.

Exercício 5 (TPC 2013/14)

Considere a linguagem $L = \{0^m 1^n 0^m, m \geq 0, n \leq m\}$. Desenhe uma Máquina de Turing capaz de reconhecer a linguagem, apresentando também a sua representação formal e tabular. Apresente ainda o traço de computação para a cadeia 00100.

Para reconhecer esta linguagem, podemos usar uma estratégia semelhante à vista acima para a linguagem $a^n b^n c^n, n \geq 1$, percorrendo a cadeia e marcando a cada iteração os zeros e uns processados. Como $n \leq m$, tem de ser possível percorrer a cadeia sem ler qualquer 1 no meio. Por outro lado, temos de ter também em atenção que a cadeia vazia pode ser aceite.



Como podemos ver, começando em q_0, se a cadeia for vazia, transita diretamente para q_7, aceitando. No caso de existirem zeros, o primeiro é eliminado, prosseguindo-se com a leitura da cadeia. Caso a cadeia tenha apenas zeros, a TM transitando diretamente de q_1 para q_4 ao chegar ao final

da cadeia, apagando o último zero e regressando ao início da cadeia, voltando a q_0. No caso de existirem uns, é usado o ciclo exterior, marcando o primeiro um como processado (símbolo P), e passando pelos uns e zeros até ao final da cadeia. Tendo as duas opções em q_1, a TM irá processar os uns enquanto estes existirem, pelo ciclo externo, usando depois o ciclo interno para processar os restantes zeros. No final, quando já não existirem zeros no início da cadeia, a transição para q_6 e a leitura de símbolos P irá assegurar que a cadeia está efetivamente no formato especificado, garantindo que não sobraram 1's por processar e que não existem 0's no meio dos 1's.

Usando a notação formal:
$TM = (\{q_0, q_1, q_2, q_3, q_4, q_5, q_6, q_7\}, \{0, 1\}, \{0, 1, P, B\}, \delta, q_0, B, \{q_7\})$, sendo δ definido da seguinte forma:
$\delta(q_0, 0) = (q_1, B, R)$; $\delta(q_0, P) = (q_6, B, R)$; $\delta(q_0, B) = (q_7, B, R)$; $\delta(q_1, 0) = (q_1, 0, R)$; $\delta(q_1, P) = (q_1, P, R)$; $\delta(q_1, 1) = (q_2, P, R)$; $\delta(q_1, B) = (q_4, B, L)$; $\delta(q_2, 1) = (q_2, 1, R)$; $\delta(q_2, 0) = (q_3, 0, R)$; $\delta(q_3, 0) = (q_3, 0, R)$; $\delta(q_3, B) = (q_4, B, L)$; $\delta(q_4, 0) = (q_5, B, L)$; $\delta(q_5, 0) = (q_5, 0, L)$; $\delta(q_5, 1) = (q_5, 1, L)$; $\delta(q_5, P) = (q_5, P, L)$; $\delta(q_5, B) = (q_0, B, R)$; $\delta(q_6, P) = (q_6, B, R)$; $\delta(q_6, B) = (q_7, B, R)$

Em forma tabular:

	0	1	P	B
$\rightarrow q_0$	(q_1, B, R)		(q_6, B, R)	(q_7, B, R)
q_1	$(q_1, 0, R)$	(q_2, P, R)	(q_1, P, R)	(q_4, B, L)
q_2	$(q_3, 0, R)$	$(q_2, 1, R)$		
q_3	$(q_3, 0, R)$			(q_4, B, L)
q_4	(q_5, B, L)			
q_5	$(q_5, 0, L)$	$(q_5, 1, L)$	(q_5, P, L)	(q_0, B, R)
q_6			(q_6, B, R)	(q_7, B, R)
$*q_7$				

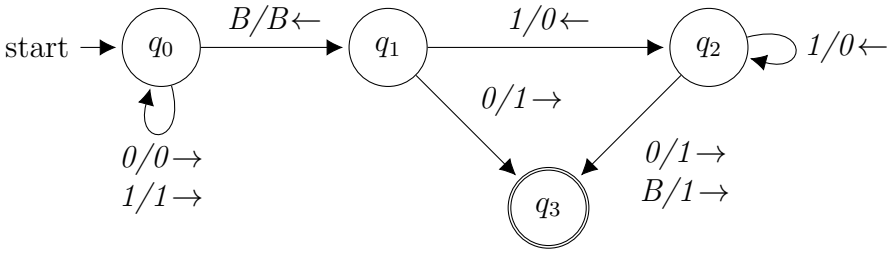
Relativamente ao traço de computação para a cadeia 00100, temos então a seguinte sequência:
$q_000100 \vdash q_10100 \vdash 0q_1100 \vdash 0Pq_200 \vdash 0P0q_30 \vdash 0P00q_3B \vdash 0P0q_40 \vdash 0Pq_50 \vdash 0q_5P0 \vdash q_50P0 \vdash q_5B0P0 \vdash q_00P0 \vdash q_1P0 \vdash Pq_10 \vdash P0q_1B \vdash Pq_40 \vdash q_5P \vdash q_5BP \vdash q_0P \vdash q_6B \vdash q_7B$

Exercício 6 (Exercício 2015/16)

Pretene-se obter uma Máquina de Turing que adicione 1 ao número inteiro representado em binário na fita. Exemplos: $0010 \rightarrow 0011$; $1 \rightarrow 10$
a) Desenhe uma Máquina de Turing correspondente.
b) Apresente o traço de computação quando a entrada na fita é 101.

Para resolver este problema, podemos pensar numa forma de somar 1 em binário, o que pode ser conseguido percorrendo o número da direita para a esquerda e:
- caso o primeiro bit (bit mais à direita) seja 0, mudar para 1
- caso o primeiro bit seja 1, mudar para 0 e continuar para a esquerda, mudando todos os 1s para 0s até aparecer o primeiro 0 (ou um espaço branco à esquerda do número), o qual se muda para 1.

Desenhando então uma TM que use esta estratégia:



Como podemos ver, em q_0 a cadeia é percorrida para a direita até ao final, sendo em q_1 tomada a decisão sobre qual das opções tomar: caso seja um 0, troca-se para 1, passando diretamente para q_3; caso seja 1, passa-se para q_2, continuando a trocar os 1's até aparecer o primeiro 0 ou uma casa vazia, transitando nesse momento para q_3.

O taço de computação para a cadeia 101 será então:
$q_0101 \vdash 1q_001 \vdash 10q_01 \vdash 101q_0B \vdash 10q_11 \vdash 1q_200 \vdash 11q_30$