Universidade do Minho
Escola de Ciências

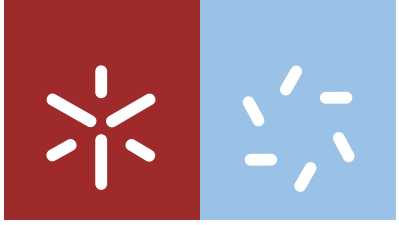
Gerson Benjamim Hungulu

Problemas de decisão em teoria de linguagens regulares

outubro de 2019

Problemas de decisão em teoria de linguagens regulares
Gerson Benjamim Hungulu

UMinho | 2019



Universidade do Minho
Escola de Ciências

Gerson Benjamim Hungulu

Problemas de decisão em teoria de linguagens regulares

Dissertação de Mestrado
Mestrado em Matemática e Computação

Trabalho efetuado sob a orientação do
Professor Doutor José Carlos Espírito Santo

outubro de 2019

Agradecimentos

A Deus, pelo dom da vida, inspiração, força e sabedoria necessária para a efetivação dos meus objetivos.

Aos meus queridos pais, Gonçalves Francisco Hungulu e Isabel Ngueve Tchitico Mucuambi Hungulu, que possibilitaram a realização do presente trabalho, me encaminhando à vida académica, pelo incentivo, cuidado, atenção, por me apoiarem em tudo quanto tem sido necessário e por tudo que têm feito em prol do meu sucesso.

A todas as forças do bem que emanaram energias positivas possibilitando inspiração e segurança para a concretização deste trabalho.

Ao meu orientador, Professor Doutor José Carlos Espírito Santo, pela sábia e instrutiva orientação, pela paciência que teve no decorrer da execução do trabalho, pelo comprometimento, responsabilidade, apoio e estímulo que foram fundamentais para a conclusão deste trabalho.

Aos demais professores do curso de Mestrado em Matemática e Computação, pelos preciosos ensinamentos fundamentais para o desenvolvimento do curso, bem como a concretização dos respetivos objetivos. Aos meus colegas, que estiveram sempre comigo nesta jornada, contribuindo direta ou indiretamente para que esse trabalho fosse concretizado. Aos que me receberam desde o primeiro momento que me fiz presente na universidade, orientando-me para que então pudesse iniciar o curso.

A todos os meus familiares e amigos, que torceram pelo meu sucesso.

DECLARAÇÃO DE INTEGRIDADE

Declaro ter actuado com integridade na elaboração do presente trabalho académico e confirmo que não recorri à prática de plágio nem a qualquer forma de utilização indevida ou falsificação de informações ou resultados em nenhuma das etapas conducente à sua elaboração.

Mais declaro que conheço e que respeitei o Código de Conduta Ética da Universidade do Minho.

Resumo

Um problema de decisão consiste num conjunto de perguntas cujas respostas são "sim" ou "não" . A solução para um problema de decisão é um procedimento completo, mecânico e determinista, sendo assim (muitas vezes) chamado de procedimento efetivo. Um problema de decisão é indecidível se não houver nenhum procedimento/algoritmo que resolva o problema, caso contrário, é decidível.

A capacidade das máquinas de Turing de retornar respostas afirmativas e negativas, torna-as um sistema matemático adequado para a construção de soluções para problemas de decisão. As máquinas de Turing que se limitam a ler uma determinada palavra podem ser vistas como autómatos finitos. O autómato finito é um modelo matemático da computação de um sistema com entradas e saídas discretas.

As linguagens regulares são as linguagens representáveis por uma expressão regular, constituindo o nível mais elementar da hierarquia do linguista Noam Chomsky. Kleene demonstrou que as linguagens regulares são precisamente as linguagens reconhecidas pelos autómatos finitos, fundando assim a teoria das linguagens regulares e dos autómatos finitos.

Neste trabalho estudam-se alguns problemas de decisão envolvendo linguagens regulares, quando estas são representadas por autómatos finitos ou expressões regulares. Especificamente, apresentamos e analisamos algoritmos para o problema de decidir se uma dada linguagem é vazia, bem como o problema de decidir se uma dada linguagem é infinita.

Palavras chaves: Problemas de decisão, expressões regulares, autómatos finitos, algoritmo.

Abstract

A decision problem is a set of questions that are answered "yes" or "no". The solution to a decision problem is a complete, mechanical and deterministic procedure, so it is (often) called an effective procedure. A decision problem is undecidable if there is no procedure/algorithm that solves the problem, otherwise it is decidable.

The ability of Turing machines to return affirmative and negative answers makes them a suitable mathematical system for building solutions to decision problems. Turing machines that merely read a certain word can be seen as finite automata. The finite automaton is a mathematical model of the computation of a system with discrete inputs and outputs.

Regular languages are the languages representable by a regular expression, constituting the most elementary level of the linguist Noam Chomsky's hierarchy. Kleene demonstrated that the regular languages are precisely the languages recognized by finite automata, thus founding the theory of regular languages and finite automata.

In this work we study some decision problems for regular languages, when languages are given both by finite automata and regular expressions. Specifically we present and analyze algorithms for the problem of deciding if the given language is empty, as well as for the problem of deciding if the given language is infinite.

Keywords: Decision problems, regular expressions, finite automata, algorithm.

Conteúdo

1 Introdução **1**

2 Requisitos **5**

2.1 Definições Básicas 5

2.2 Autómatos Finitos 10

2.3 Algoritmos sobre grafos 21

3 Problemas de Decisão **33**

3.1 Problema da linguagem vazia 34

3.2 Problema da linguagem infinita 37

3.3 Relacionando AFN com os algoritmos propostos 42

3.4 Quando a linguagem é dada por uma expressão regular 43

4 Conclusão **60**

Bibliografia **62**

Siglas

AFD Autómato finito determinista. 12–15, 18, 33, 35, 39, 42, 43, 61

AFN Autómato finito não determinista. 15–18, 42, 43

BFS Breadth-first search. 23, 25, 26, 29, 35, 36

bft Breadth-first tree. 25–27

dff Depth-first forest. 29, 31, 32

DFS Depth-first search. 28–32, 41

dft Depth-first trees. 29, 32

ER Expressões regulares. 8

Capítulo 1

Introdução

Um dos tópicos que frequentemente são tratados em ciências da computação é o problema de decisão.

Um **problema de decisão** consiste num conjunto de perguntas cujas respostas são "sim" ou "não" .

Um problema de decisão é definido, também, como uma função com uma saída de um bit (sim ou não). Para tornar mais específico um problema de decisão, deve-se especificar:

- o conjunto A de entradas possíveis, e
- o subconjunto $B \subseteq A$ de instâncias "sim" .

Por exemplo, para decidir se um determinado grafo está conectado, o conjunto de entradas possíveis deve ser o conjunto de todos os grafos, e o subconjunto das instâncias "sim" devem corresponder aos grafos conectados. Para decidir se um determinado número é primo, o conjunto de entradas possíveis devem ser o conjunto de todos os números naturais, e o subconjunto das instâncias "sim" deve corresponder ao conjunto dos números primos.

Segundo Sudkamp 1997, uma solução para um problema de decisão é um procedimento efetivo que determina a resposta para cada questão do conjunto. Um problema de decisão é **decidível** se houver algum algoritmo que seja solução do problema, caso contrário, é **indecidível**. Um algoritmo que resolve um problema de decisão deve ser

- Completo: Produz uma resposta, positiva ou negativa, a cada questão no domínio do problema;

- Mecânico: Consiste em uma sequência finita de instruções, cada uma das quais pode ser executada sem necessidade de conhecimento, habilidade ou suposição;

- Determinista: Quando recebe uma entrada idêntica (a uma anterior), sempre produz o mesmo resultado.

Um procedimento que satisfaça as propriedades supracitadas é, muitas vezes, chamado efetivo.

A capacidade das máquinas de Turing de retornar respostas afirmativas e negativas, torna-as num sistema matemático adequado para a construção de soluções para problemas de decisão. A tese Church-Turing afirma que as máquinas de Turing podem ser projetadas para resolver qualquer problema de decisão que seja resolúvel por qualquer procedimento efetivo. As máquinas de Turing que se limitam a ler uma determinada palavra podem ser vistas como **autómatos finitos** e, portanto, este é também um sistema matemático apropriado para solucionar alguns problemas de decisão, e, tal como as máquinas de Turing, este é um procedimento mecânico e determinista.

A teoria dos autómatos é o estudo de dispositivos abstratos de computação ou "máquinas". Antes dos computadores, na década dos anos 30 do século XX, A. Turing estudou uma máquina abstrata que possuía todas as capacidades dos computadores atuais. O objetivo de Turing era descrever precisamente a fronteira entre o que uma máquina de computação poderia fazer e o que não poderia; as suas conclusões aplicam-se não apenas às máquinas abstratas de Turing, mas às máquinas reais de hoje.

Nas décadas de 40 e 50 do século XX, tipos mais simples de máquinas, que hoje chamamos de **autómatos finitos**, foram estudados por vários pesquisadores. Esses autómatos, originalmente propostos para modelar a função cerebral, acabaram por ser extremamente úteis para uma variedade de outros propósitos (citaremos alguns, já a seguir).

O autómato finito, ou máquina de estados finitos, é um modelo matemático de computação de um sistema com entradas e saídas discretas. O mecanismo de controle de um elevador é um bom exemplo de um sistema de estados finitos. Na ciência da computação encontramos muitos exemplos de sistemas de estados finitos, e a teoria de autómatos finitos é uma ferramenta de design útil para esses sistemas, tipos importantes de hardware e software, como por exemplo (segundo Hopcroft, Motwani e Ullman 2006):

1. Software para projetar e verificar o comportamento de circuitos digitais.

2. O "analisador léxico" de um compilador típico, ou seja, o componente do compilador que divide o texto de entrada em unidades lógicas, como palavras-chaves de identificadores e pontuação.

3. Software para verificação de sistemas de todos os tipos que possuem um número finito de estados distintos, como protocolos de comunicação ou protocolos para troca segura de informações.

4. Software para digitalizar grandes volumes de texto, como coleções de páginas da Web, para localizar ocorrências de palavras, frases ou outros padrões.

Ainda na década dos anos 50, quando procuravam modelar certas propriedades das línguas naturais, linguistas tais como Noam Chomsky introduziram a formalização matemática das noções de palavra, linguagem e gramática que são hoje utilizadas. Chomsky estabeleceu uma hierarquia de linguagens naturais cujo nível mais elementar é constituído pela classe das "linguagens que podem ser descritas por expressões regulares", ou seja, **linguagens regulares**; e todas as linguagens desta hierarquia estão associadas a alguma máquina de Turing. Os níveis menos elevados da hierarquia de Chomsky são, também, reconhecidos por outros modelos de máquinas abstratas menos poderosas que as máquinas de Turing. Entre estes modelos, os mais elementares são os autómatos finitos. Kleene demonstrou, em 1954, que as linguagens regulares são precisamente as linguagens reconhecidas pelos autómatos finitos, e este resultado é considerado como o fundador da teoria das linguagens regulares e dos autómatos finitos.

O presente trabalho tem como objetivo estudar alguns problemas de decisão envolvendo linguagens regulares, quando estas são representadas por autómatos finitos ou expressões regulares, mais precisamente os problemas de decidir se a linguagem aceite por um dado autómato é vazia ou infinita; e se uma dada expressão regular representa a linguagem vazia, ou representa a linguagem infinita.

Deste propósito resulta o título deste trabalho: **problemas de decisão em teoria de linguagens regulares**. O mesmo está estruturado em quatro capítulos a contar com a introdução. O capítulo 2, dividido em 3 secções, trata de assuntos teóricos relevantes para o capítulo a seguir. O capítulo 3, com quatro secções, trata da parte prática deste trabalho,

pois o seu objetivo é de apresentar respostas para os problemas de decisão acima citados. O capítulo 4, o último capítulo do trabalho, cinge-se às principais conclusões a que se chegou com a execução desta dissertação.

Capítulo 2

Requisitos

Neste capítulo, trataremos de aspectos relevantes para a melhor compreensão do que será tratado a seguir, e para tal, tivemos em conta o que já foi abordado por Coelho e Neto 2010; Hopcroft e Ullman 1979; Cormen et al. 2001; Costa 2004; Sipser 2012.

2.1 Definições Básicas

Notação assintótica

Uma das preocupações quando se faz um algoritmo é a de garantir a sua eficiência, isto é, que possa ser executado em um curto espaço de tempo, classificando-o com base em sua ordem de crescimento de tempo de execução. A ordem de crescimento de tempo de execução de um algoritmo fornece uma caracterização simples da eficiência do algoritmo e também nos permite comparar o desempenho relativo de algoritmos alternativos.

Nesta subsecção, focaremos na notação que se usará para enfatizar a velocidade do tempo de execução de um algoritmo .

Sejam $f, g : \mathbb{N}_0 \to \mathbb{R}_0^+$ funções. Diz-se que $g(n)$ é de ordem $f(n)$ e escreve-se $g(n) \in \mathcal{O}(f(n))$ ou $g(n) = \mathcal{O}(f(n))$, se existem constantes positivas c e n_0 tais que

$$\forall n \geq n_0,\ 0 \leq g(n) \leq cf(n).$$

Isto é,

$$\mathcal{O}(f(n)) = \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \exists c \in \mathbb{R}_0^+,\ \exists n_0 \in \mathbb{N}_0,\ \forall n \geq n_0,\ 0 \leq g(n) \leq cf(n)\}$$

é o conjunto de funções que são limitadas superiormente pela função $f(n)$.

Exemplo 2.1.1. Sejam $g(n) = an^2 + bn + c$ e $f(n) = n^2$, onde $a, b, c \in \mathbb{R}_0^+$.
$g(n) \in \mathcal{O}(f(n))$, pois existem $d \in \mathbb{R}_0^+,\ n_0 \in \mathbb{N}_0$ tais que

$$\forall n \geq n_0,\ an^2 + bn + c \leq dn^2.$$

Basta tomar $d = a + b + 1$ e n_0 tal que $n_0^2 \geq c$. Com efeito: seja $n \geq n_0$

$$\begin{aligned} an^2 + bn + c &\leq an^2 + bn^2 + c && (\text{ por } n \leq n^2) \\ &\leq an^2 + bn^2 + n^2 && (\text{ por } n \geq n_0 \Rightarrow n^2 \geq n_0^2 \Rightarrow n^2 \geq c) \\ &= (a + b + 1)n^2 \\ &= dn^2 \end{aligned}$$

Se $g(n) \in \mathcal{O}(f(n))$ e $f(n) \in \mathcal{O}(g(n))$, então $f(n)$ e $g(n)$ devem crescer ao mesmo ritmo. Neste caso, dizemos que $f(n) \in \Theta(g(n))$ (e também $g(n) \in \Theta(f(n))$).

$\Theta(f(n))$ é o conjunto de funções que são limitadas inferiormente e superiormente pela função $f(n)$, ou seja,

$$\Theta(f(n)) = \{g : \mathbb{N}_0 \to \mathbb{R}_0^+ \mid \exists c_1, c_2 \in \mathbb{R}_0^+,\ \exists n_0 \in \mathbb{N}_0,\ \forall n \geq n_0,\ 0 \leq c_1 f(n) \leq g(n) \leq c_2 f(n)\}.$$

Palavras e Linguagens

Um "símbolo" é uma entidade abstrata que não será definida formalmente. Letras e dígitos são exemplos de símbolos frequentemente utilizados. Um **alfabeto** é um conjunto finito de símbolos, como por exemplo, o alfabeto da língua portuguesa ($\{a,\ b,\ c, ...,\ x,\ y,\ z,$
$A,\ B,\ C, ...,\ X,\ Y,\ Z\}$), o alfabeto binário ($\{0, 1\}$).

Seja A um alfabeto. Uma sequência finita de símbolos de A diz-se uma **palavra** em A.

É permitida a sequência vazia, que se diz **palavra vazia** e é denotada por ϵ. Por exemplo, a, b e c são símbolos e $abcb$ é uma palavra. A^* é o conjunto de todas as palavras em A. A^+ é o conjunto de todas as palavras não vazias em A. Por exemplo, se $A = \{a\}$, então $A^* = \{\epsilon, a, aa, aaa, ...\}$ e $A^+ = \{a, aa, aaa, ...\}$. Se $A = \{0, 1\}$, então $A^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, ...\}$ e $A^+ = \{0, 1, 00, 01, 10, 11, 000, ...\}$.

Sejam A alfabeto, $n \in \mathbb{N}_0$, $a_1, a_2, ..., a_n \in A$ e $u = a_1a_2...a_n$ palavra de A^*. O **comprimento** de u, denotada por $|u|$, é o número de símbolos que compõem a palavra u, isto é, $|u| = n$. Por exemplo, $abcb$ tem comprimento 4 ($|abcb| = 4$). A palavra ϵ tem comprimento 0 ($|\epsilon| = 0$). Sejam ainda $m \in \mathbb{N}_0$, $b_1, b_2, ..., b_m \in A$ e $v = b_1b_2...b_m$ uma palavra em A. Então $u = v$ se, e só se: (a) tiverem o mesmo comprimento: $n = m$; (b) serem iguais letra a letra (para qualquer que seja $1 \leq i \leq n$, $a_i = b_i$). A palavra $a_1a_2...a_nb_1b_2...b_m$ diz-se a **concatenação** de u e v, é denotada por $u.v$ ou uv. Por exemplo, a concatenação de $casa$ e $mento$ é $casamento$. A $n-$ésima **potência** de u, denotada por u^n, define-se por recursão em n do seguinte modo

$$u^n = \begin{cases} \epsilon & \text{se } n = 0 \\ u^{n-1}u & \text{se não.} \end{cases}$$

Por exemplo, $(abcb)^2 = (abcb)^1.abcb = (abcb)^0.abcb.abcb = \epsilon.abcb.abcb = abcbabcb$. A palavra vazia é a identidade do operador de concatenação, ou seja, $\epsilon v = v\epsilon = v$ para cada palavra v.

A palavra **inversa** de u, denotada por u^I, define-se por recursão em u do seguinte modo

$$u^I = \begin{cases} \epsilon & \text{se } u = \epsilon \\ av^I & \text{se } u = va \text{ com } v \in A^* \text{ e } a \in A. \end{cases}$$

Por exemplo, $(a_1a_2...a_n)^I = a_n...a_2a_1$. Se $u = u^I$ então u diz-se uma capicua ou um palíndromo.

Sejam A um alfabeto, u e v duas palavras de A^*. Diz-se que u é um: (1) fator de v se existem $x, y \in A^*$ tais que $xuy = v$; (2) prefixo ou fator esquerdo de v se existe $y \in A^*$ tal que $uy = v$; (3) sufixo ou fator direito de v se existe $x \in A^*$ tal que $xu = v$; (4) fator próprio (resp. prefixo próprio, sufixo próprio) de v se u é um fator (resp. prefixo, sufixo) de v e $u \neq v$.

Uma **linguagem** sobre um alfabeto A é um subconjunto de A^*. Por exemplo, se $A = \{a, b\}$, então $\emptyset$, $\{\epsilon\}$, $\{a\}$, A, $\{aa, aba, bbb, ababa\}$, $\{a^n ba^n \mid n \in \mathbb{N}\}$, A^+ e A^* são linguagens sobre A. $\mathcal{P}(A^*)$ é o conjunto de todas as linguagens sobre A.

Sejam A um alfabeto, L, L_1 e L_2 conjuntos de palavras de A^*. A concatenação de L_1 e L_2, denotado por $L_1.L_2$ ou L_1L_2, é o conjunto $\{uv \in A^* \mid u \in L_1 \text{ e } v \in L_2\}$. Define-se

$$\begin{aligned} L_1 \cup L_2 &= \{u \in A^* \mid u \in L_1 \text{ ou } u \in L_2\} \\ L_1 \cap L_2 &= \{u \in A^* \mid u \in L_1 \text{ e } u \in L_2\} \\ L_1 \setminus L_2 &= \{u \in A^* \mid u \in L_1 \text{ e } u \notin L_2\} \\ \overline{L} &= A^* \setminus L \\ &= \{u \in A^* \mid u \notin L\} \end{aligned}$$

Define-se $L^0 = \{\epsilon\}$ e $L^i = LL^{i-1}$ para $i \geq 1$. O **fecho de Kleene** (ou apenas fecho) de L, denotado por L^*, é o conjunto

$$L^* = \bigcup_{i=0}^{\infty} L^i$$

e o fecho positivo de L, denotado por L^+, é o conjunto

$$L^+ = \bigcup_{i=1}^{\infty} L^i.$$

Expressões regulares

As linguagens aceites por autómato finito são facilmente descritas por simples expressões chamadas expressões regulares (ER). Segue a definição de uma expressão regular.

Definição 2.1.1. Seja A um alfabeto. O conjunto das expressões regulares sobre A, denotado por $ER(A)$, é o conjunto das palavras sobre o alfabeto $A \cup \{\underline{\emptyset}, \underline{\epsilon}, (,), +, \cdot, ^*\}$ definido indutivamente por

(i) $\underline{\emptyset}, \underline{\epsilon} \in ER(A)$;

(ii) $\forall a \in A,\ a \in ER(A)$;

(iii) $\forall r, s \in ER(A),\ (r + s), (r.s), (r^*) \in ER(A)$.

Por exemplo, sendo $A = \{a, b\}$, são expressões regulares sobre A,

$$\underline{\emptyset},\ \underline{\epsilon},\ a,\ b,\ (a.b),\ ((a.b) + \epsilon),\ ((((a^*).b) + \emptyset)^*).$$

Para simplificar a notação das expressões regulares são usadas abreviaturas seguindo as regras seguintes:

1) escreve-se $\emptyset$ e ϵ em vez de $\underline{\emptyset}$ e $\underline{\epsilon}$, respetivamente;

2) omite-se o símbolo $\cdot$;

3) sendo $r \in ER(A)$ e $n \in \mathbb{N}$, define-se $r^0 = \underline{\epsilon}$, $r^n = (r.r^{n-1})$ e $r^+ = (r.r^*)$;

4) omitem-se parênteses inúteis usando associatividade para as operações $+$ e $\cdot$, e considerando que * tem a maior prioridade e que $\cdot$ tem a maior prioridade em relação a $+$.

Por exemplo, c^+b^3 é uma abreviatura de $(((c^*).c).((b.b).b))$ enquanto que $\emptyset^*c + (b + \epsilon)bb$ é uma abreviatura de $(((\emptyset^*).c) + ((b + \epsilon).(b.b)))$.

Definição 2.1.2. Seja A um alfabeto. A cada expressão regular r sobre A corresponde a linguagem $L(r)$ sobre A por ela representada, por intermédio da função

$$\begin{aligned} L : ER(A) &\to \mathcal{P}(A^*) \\ r &\mapsto L(r) \end{aligned}$$

que é definida recursivamente por:

(i) $L(\emptyset) = \emptyset$, $L(\epsilon) = \{\epsilon\}$;

(ii) Para cada $a \in A$, $L(a) = \{a\}$;

(iii) $\forall r, s \in ER(A)$, $L(r + s) = L(r) \cup L(s)$, $L(r.s) = L(r).L(s)$, $L(r^*) = (L(r))^*$.

Quando necessário, para distinguir entre uma expressão regular r e a linguagem representada por ela, usa-se $L(r)$ para o último. Quando não for possível confundi-los, usa-se r para ambos, expressão regular e a linguagem representada pela expressão regular.

Exemplo 2.1.2. Sendo $A = \{a, b\}$, tem-se:

1. a^* denota a linguagem $L(a^*) = (L(a))^* = \{a\}^* = \{a^n \mid n \in \mathbb{N}_0\}$;

2. $a(a+b)$ denota a linguagem $L(a(a+b)) = L(a)L(a+b) = \{a\}(L(a) \cup L(b)) = \{a\}(\{a\} \cup \{b\}) = \{a\}\{a,b\} = \{aa, ab\}$;

3. $a^*(a+b)$ denota a linguagem $L(a^*(a+b)) = \{a\}^*\{a,b\} = \{a^n \mid n \in \mathbb{N}\} \cup \{a^n b \mid n \in \mathbb{N}_0\}$.

Definição 2.1.3. Seja $L \subseteq A^*$. A linguagem L diz-se **regular** se existe $r \in ER(A)$ tal que $L(r) = L$.

2.2 Autómatos Finitos

Um autómato finito consiste num conjunto finito de estados (entre eles, temos um estado inicial e um subconjunto de estados finais/de aceitação) e num conjunto de transições de estado para estado com etiquetas de símbolos de um alfabeto A. Um grafo dirigido, chamado de diagrama de transição, está associado a um autómato finito da seguinte forma:

1. Os vértices do grafo correspondem aos estados do autómato.

2. Se existe uma transição do estado q_1 para o estado q_2, com etiqueta a, então existe uma aresta com etiqueta a, do estado q_1 para q_2.

3. O autómato finito aceita uma palavra u, se existir uma sequência de transições correspondente a u, que parte do estado inicial para um estado de aceitação.

Um exemplo de diagrama de transição de um autómato finito $\mathcal{A}$ é ilustrado na figura 2.1. Tem cinco estados, nomeados como q_0, q_1, q_2, q_3 e q_4. O estado inicial, q_0, é indicado por uma seta, o estado de aceitação (ou final), q_3, é indicado com círculo duplo. As setas que vão de um estado para outro são chamadas de transições. Quando este autómato $\mathcal{A}$ recebe uma palavra como 0001, processa essa palavra e produz uma saída. A saída pode ser aceite ou rejeitada. O processamento começa no estado inicial de $\mathcal{A}$. O autómato recebe os símbolos da palavra de entrada, um a um, da esquerda para a direita. Depois de ler cada símbolo, $\mathcal{A}$ move-se de um estado para outro ao longo da transição (ou aresta) que tem esse símbolo como etiqueta. Quando lê o último símbolo, $\mathcal{A}$ produz a sua saída. A saída é $aceite$, se $\mathcal{A}$ estiver em um estado de aceitação, e $rejeitada$, se não estiver. Por exemplo, quando introduzimos a palavra de entrada 0001 no autómato da figura 2.1, o processamento procede da seguinte forma:

1. Inicia no estado q_0;

2. Lê 0, segue a transição de q_0 para q_1;

3. Lê 0, segue a transição de q_1 para q_1;

4. Lê 0, segue a transição de q_1 para q_1;

5. Lê 1, segue a transição de q_1 para q_3.

6. Aceitar, porque $\mathcal{A}$ chegou a um estado de aceitação q_3 depois de ter lido o último símbolo da entrada.

O autómato $\mathcal{A}$ aceita todas as palavras que tenham um segmento inicial de 0's seguidas de uma única ocorrência de 1 (por exemplo 0001) e todas as palavras que tenham um segmento inicial de 1's seguidas por uma única ocorrência de 0 (por exemplo 1110), todas outras palavras que não tenham este formato, são rejeitadas. $\mathcal{A}$ aceita todas palavras u que pertencem a linguagem representada pela expressão regular $0^+1 + 1^+0$.

Vimos o que é um autómato finito informalmente, entendendo como se comporta o seu grafo de transição. Veremos a seguir sua definição formal, distinguindo autómato finito determinista e não determinista, terminaremos esta secção enunciado alguns resultados teóricos que descrevam suas potencialidades e limitações.

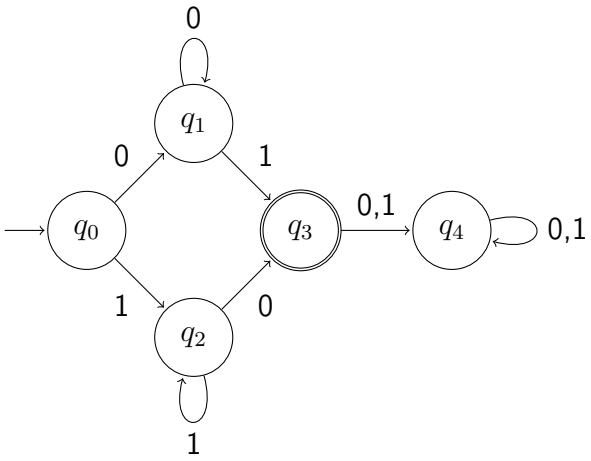


Figura 2.1: Diagrama de transição de um autómato finito

Autómato finito determinista (AFD)

Em teoria de autómatos, uma máquina de estado finito é chamada autómato finito determinista (AFD) se cada uma de suas transições é determinada exclusivamente pelo seu estado de origem e símbolo de entrada; ou seja, para cada letra de entrada existe exatamente uma transição para cada estado (possivelmente, de volta ao estado). A leitura de um símbolo de entrada é necessária para cada transição de estado, o estado inicial é denotado usualmente por i ou q_0. Segue sua definição formal:

Definição 2.2.1. Um AFD é um quíntuplo $\mathcal{A} = (Q, A, \delta, i, F)$ onde

- Q é um conjunto finito de estados;
- A é um alfabeto, chamado o alfabeto (de entrada) de $\mathcal{A}$;
- $\delta : Q \times A \to Q$ é uma função total, designada função de transição de $\mathcal{A}$. Isto é, $\delta(q, a)$ é o estado a que se chega após ler um símbolo a a partir de q;
- $i \in Q$ é o estado inicial de $\mathcal{A}$;
- $F \subseteq Q$ é o conjunto de estados finais/de aceitação de $\mathcal{A}$.

No grafo da figura 2.1, a transição

$$q_0 \xrightarrow{0} q_1$$

também é denotada por $(q_0, 0, q_1)$. Duas transições (p, a, q) e (p', a', q') de um autómato $\mathcal{A}$ dizem-se consecutivas se $q = p'$. Um caminho em $\mathcal{A}$ é uma sequência finita

$$(q_0, a_1, q_1), (q_1, a_2, q_2), ..., (q_{n-1}, a_n, q_n)$$

de transições consecutivas de $\mathcal{A}$, também denotado por

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \ . \ . \ . \ q_{m-1} \xrightarrow{a_m} q_m.$$

O estado q_0 é a origem do caminho e o estado q_m o seu término, e diz-se que o caminho sai de q_0 e chega a q_m. Diz-se também que o caminho passa pelos estados $q_0, q_1, ..., q_m$. A palavra $a_1a_2...a_m$ sobre A é chamada etiqueta do caminho.

Chama-se **caminho inicial** a um caminho que sai do estado inicial, e **caminho final**, a um caminho que chega a um estado final. Um caminho diz-se **bem sucedido** se é simultaneamente inicial e final. Um caminho que não é bem sucedido, diz-se **mal sucedido**.

Uma palavra $u \in A^*$ diz-se **aceite** pelo autómato $\mathcal{A}$ se u é a etiqueta de pelo menos um caminho bem sucedido em $\mathcal{A}$, caso contrário diz-se rejeitada por $\mathcal{A}$. A linguagem aceite pelo autómato $\mathcal{A}$ é o conjunto, representado por $L(\mathcal{A})$, das palavras aceites por $\mathcal{A}$.

Exemplo 2.2.1. Retomemos o autómato $\mathcal{A}$ da figura 2.1. Como já vimos, a palavra 0001 é aceite por $\mathcal{A}$. A palavra 1110, também é aceite por $\mathcal{A}$, pois

$$q_0 \xrightarrow{1} q_2 \xrightarrow{1} q_2 \xrightarrow{1} q_2 \xrightarrow{0} q_3$$

é um caminho bem sucedido em $\mathcal{A}$, cuja etiqueta é 1110. Já a palavra 11 (respetivamente 00) é rejeitada por $\mathcal{A}$, porque todo o caminho bem sucedido tem que ter a transição $(q_2, 0, q_3)$ (respetivamente $(q_1, 1, q_3)$).

Define-se alternativamente palavras (e linguagem) aceites por um autómato $\mathcal{A} = (Q, A, \delta, i, F)$, pela extensão da função de transição δ. A função $\delta : Q \times A \to Q$ estende-se a uma função $\delta^* : Q \times A^* \to Q$, sendo a função de transição aplicada a um estado e uma palavra, $\delta^*(q, w)$ é o estado em que o AFD estará após ler w começando no estado q. Colocado de outra forma, $\delta^*(q, w)$ é o único estado p tal que existe um caminho no diagrama de transição de q para p, com etiqueta w. Definindo δ^* por recorrência, temos

1) $\delta^*(q, \epsilon) = q$;

2) $\forall w \in A^*,\ \delta^*(q, aw) = \delta^*(\delta(q, a), w)$.

Assim $1)$ afirma que, sem ler um símbolo de entrada, o AFD não pode transitar para um outro estado, e $2)$ diz como encontrar o estado de uma palavra de entrada não vazia aw. Uma palavra $u \in A^*$ é dita ser aceite por um autómato finito $\mathcal{A} = (Q, A, \delta, i, F)$ se $\delta^*(i, u) = p$ para algum p em F.

$L(\mathcal{A})$ (a linguagem aceite por $\mathcal{A}$), é o conjunto $\{u \in A^* \mid \delta^*(i, u) \in F\}$. Uma linguagem diz-se um conjunto **reconhecível** (ou regular) se o conjunto é aceite por algum autómato finito.

Sejam $p, q \in Q$ (estados em $\mathcal{A}$); p diz-se **atingível** a partir de q, se existe um caminho em $\mathcal{A}$ de q para p, p diz-se **acessível** se p é atingível a partir do estado inicial i, e diz-se **co-acessível** se existe um caminho em $\mathcal{A}$ de p para um estado final. Em um AFD, para uma dada palavra de entrada w e estado q, haverá exatamente um caminho com etiqueta w a partir de q. O autómato $\mathcal{A}$ diz-se acessível (respetivamente co-acessível) se todos os seus estados são acessíveis (respetivamente co-acessíveis).

Exemplo 2.2.2. Seja $\mathcal{A} = (Q, A, \delta, i, F)$ o autómato cujo o grafo é o representado na figura 2.1, onde $Q = \{q_0\ , q_1,\ q_2,\ q_3\ q_4\}$, $A = \{0, 1\}$, $i = q_0$, $F = \{q_3\}$ e δ é a função definida pela tabela seguinte:

δ	q_0	q_1	q_2	q_3	q_4
0	q_1	q_1	q_3	q_4	q_4
1	q_2	q_3	q_2	q_4	q_4

Figura 2.2: Função δ para o AFD da figura 2.1

Se 1110 é a entrada para $\mathcal{A}$, então

$$\begin{aligned}
\delta^*(q_0, 1110) &= \delta^*(q_0, 1.110) \\
&= \delta^*(\delta(q_0, 1), 110) \\
&= \delta^*(q_2, 1.10) \\
&= \delta^*(\delta(q_2, 1), 1.0) \\
&= \delta^*(q_2, 1.0) \\
&= \delta^*(\delta(q_2, 1), 0.\epsilon) \\
&= \delta^*(q_2, 0.\epsilon) \\
&= \delta^*(\delta(q_2, 0), \epsilon) \\
&= \delta^*(q_3, \epsilon) \\
&= q_3 \in F
\end{aligned}$$

Ou seja, $\delta^*(q_0, 1110) \in F$. Se a entrada fosse 0001, $\delta^*(q_0, 0001) \in F$.
$L(\mathcal{A}) = \{u \in A^* \mid \delta^*(q_0, u) \in F\} = 0^+1 + 1^+0$.

Autómato finito não determinista (AFN)

Um autómato finito não determinista (AFN), ou máquina de estados finitos não determinista, não obedece necessariamente às restrições do AFD.

O AFD é um caso especial de AFN no qual, para cada estado, existe uma única transição em cada símbolo. Em um AFN pode haver vários (ou nenhum) caminhos etiquetado por uma palavra w, e todos devem ser verificados para ver se terminam em algum estado final.

Uma sequência de entradas $a_1a_2...a_n$ é aceite por um AFN se existe uma sequência de transições que conduz de um estado inicial para algum estado final.

Um exemplo de diagrama de transições para um AFN é mostrado na figura 2.3. Observe que há duas arestas com etiquetas 0 com origem q_0, uma voltando ao estado q_0 e outra indo para o estado q_3 (facto que não acontece em um AFD).

Exemplo 2.2.3. A palavra 0100 é aceite pelo AFN da figura 2.3 porque existe o caminho bem sucedido, $(q_0, 0, q_0), (q_0, 1, q_0), (q_0, 0, q_3), (q_3, 0, q_4)$. Este AFN, em particular, aceita todas palavras que tenham duas ocorrências consecutivas de 0 ou 1.

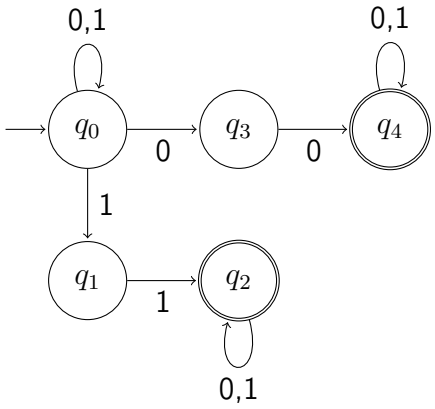


Figura 2.3: Diagrama de transição do AFN

A seguir, definimos formalmente um AFN.

Definição 2.2.2. Um autómato finito não determinista (AFN) é o quíntuplo $\mathcal{A} = (Q, A, \delta, i, F)$ onde Q, A, i e F (conjunto de estados, alfabeto, estado inicial e conjunto de estados finais) têm o mesmo significado como para o AFD mas $\delta : Q \times A \to \mathcal{P}(Q)$.

A intenção é que $\delta(q, a)$ seja o conjunto de todos estados p tais que existe uma transição com etiqueta a de q para p.

Exemplo 2.2.4. Seja $\mathcal{A} = (Q, A, \delta, i, F)$ o AFN cujo o grafo é o representado na figura 2.3, onde $Q = \{q_0, q_1, q_2, q_3, q_4\}$, $A = \{0, 1\}$, $i = q_0$, $F = \{q_2, q_4\}$, e a função δ é dada na figura 2.4.

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

Figura 2.4: Mapeamento de δ para o AFN da figura 2.3

A função δ estende-se para a função $\delta^* : Q \times A^* \to \mathcal{P}(Q)$, refletindo sequência das seguintes entradas:

1) $\delta^*(q, \epsilon) = \{q\}$

2) $\delta^*(q, wa) = \bigcup_{p \in \delta^*(q,w)} \delta(p, a)$, para cada $q \in Q$, $w \in A^*$ e $a \in A$.

Lema 2.2.1. $\delta^*(q, wv) = \bigcup_{p \in \delta^*(q,w)} \delta^*(p, v)$

Demonstração. Sejam $q \in Q$ e $w \in A^*$. Dado $v \in A^*$, seja $\Phi(v)$ a seguinte propriedade:

$$\delta^*(q, wv) = \bigcup_{p \in \delta^*(q,w)} \delta^*(p, v)$$

Vamos demonstrar que, para todo $v \in A^*$, $\Phi(v)$. Demonstração por indução em v.
Base da indução: $\Phi(\epsilon)$ se e só se

$$\delta^*(q, w\epsilon) = \bigcup_{p \in \delta^*(q,w)} \delta^*(p, \epsilon).$$

Ora

$$\begin{aligned} \delta^*(q, w\epsilon) &= \delta^*(q, w) && \text{(por } w\epsilon = w\text{)} \\ &= \bigcup_{p \in \delta^*(q,w)} \{p\} && \text{(def. de } \cup\text{)} \\ &= \bigcup_{p \in \delta^*(q,w)} \delta^*(p, \epsilon) && \text{(por } 1\text{))} \end{aligned}$$

Passo indutivo: Sejam $v \in A^*$ e $a \in A$. Suponhamos $\Phi(v)$.

Queremos mostrar $\Phi(va)$, ou seja

$$\delta^*(q, w(va)) = \bigcup_{p \in \delta^*(q,w)} \delta^*(p, va)$$

isto é, para todo $q' \in Q$

$$q' \in \delta^*(q, w(va)) \text{ sse } q' \in \bigcup_{p \in (\delta^*(q,w)} \delta^*(p, va)).$$

Ora,

$$\begin{aligned}
q' \in \delta^*(q, w(va)) \text{ sse } & q' \in \delta^*(q, (wv)a) && \text{(associativa da concatenação)} \\
\text{sse } & q' \in (\bigcup_{p' \in \delta^*(q,wv)} \delta(p', a)) && \text{(por 2))} \\
\text{sse } & \exists p' \in \delta^*(q, wv) \text{ tal que } q' \in \delta(p', a) && \text{(def. de } \cup) \\
\text{sse } & \exists p' \in (\bigcup_{p \in \delta^*(q,w)} \delta^*(p, v)) \text{ tal que } q' \in \delta(p', a) && \text{(por } \Phi(v)) \\
\text{sse } & \exists p \in \delta^*(q, w),\ \exists p' \in \delta^*(p, v) \text{ tal que } q' \in \delta(p', a) && \text{(def. de } \cup) \\
\text{sse } & \exists p \in \delta^*(q, w),\ \text{tal que } q' \in (\bigcup_{p' \in \delta^*(p,v)} \delta(p', a)) && \text{(def. de } \cup) \\
\text{sse } & \exists p \in \delta^*(q, w),\ \text{tal que } q' \in \delta^*(p, va) && \text{(por 2))} \\
\text{sse } & q' \in (\bigcup_{p \in \delta^*(q,w)} \delta^*(p, va)) && \text{(def. de } \cup)
\end{aligned}$$

□

Note que $\delta^*(q, a) = \delta(q, a)$ para um símbolo de entrada a.

Corolário 2.2.1. $\delta^*(q, av) = \bigcup_{p \in \delta(q,a)} \delta^*(p, v)$

No AFN, $L(\mathcal{A}) = \{u \in A^* \mid \delta^*(i, u) \cap F \neq \emptyset\}$.

Exemplo 2.2.5. Voltando ao AFN da figura 2.3

$$\begin{aligned}
\delta^*(q_0,0100) &= \bigcup_{p\in\delta(q_0,0)} \delta^*(p,100) = \bigcup_{p\in\{q_0,q_3\}} \delta^*(p,100) = \delta^*(q_0,100)\cup\delta^*(q_3,100)\\
&= (\bigcup_{p\in\delta(q_0,1)} \delta^*(p,00)) \cup (\bigcup_{p\in\delta(q_3,1)} \delta^*(p,00)) = (\bigcup_{p\in\{q_0,q_1\}} \delta^*(p,00)) \cup (\bigcup_{p\in\emptyset} \delta^*(p,00))\\
&= \delta^*(q_0,00)\cup\delta^*(q_1,00)\cup\emptyset = (\bigcup_{p\in\delta(q_0,0)} \delta^*(p,0)) \cup (\bigcup_{p\in\delta(q_1,0)} \delta^*(p,0))\\
&= (\bigcup_{p\in\{q_0,q_3\}} \delta^*(p,0)) \cup (\bigcup_{p\in\emptyset} \delta^*(p,0)) = \delta^*(q_0,0.\epsilon)\cup\delta^*(q_3,0.\epsilon)\cup\emptyset\\
&= (\bigcup_{p\in\delta(q_0,0)} \delta^*(p,\epsilon)) \cup (\bigcup_{p\in\delta(q_3,0)} \delta^*(p,\epsilon)) = (\delta^*(q_0,\epsilon)\cup\delta^*(q_3,\epsilon))\cup\delta^*(q_4,\epsilon)\\
&= (\{q_0\}\cup\{q_3\})\cup\{q_4\} = \{q_0,q_3\}\cup\{q_4\} = \{q_0,q_3,q_4\}
\end{aligned}$$

Isto é, $\delta^*(q_0,0100)\cap F\neq\emptyset$, concluindo que $0100\in L(\mathcal{A})$, conforme já foi dito no exemplo 2.2.3.

Teoremas

Equivalência entre AFD e AFN

Já se viu, que o AFD é um caso especial do AFN. No entanto, os AFDs e os AFNs reconhecem as mesmas linguagens. A prova depende de mostrar que, para cada AFN, pode-se construir um AFD equivalente (que aceita a mesma linguagem). A construção formal está incorporada no teorema a seguir:

Teorema 2.2.1. Seja L, um conjunto aceite por um AFN. Então existe um AFD que aceita L.

Demonstração. Seja $\mathcal{ND}=(Q,A,\delta,i,F)$ um AFN e $\mathcal{D}=(Q_0,A,\delta_0,i_0,F_0)$ o AFD definido do seguinte modo:

(1) $Q_0=\mathcal{P}(Q)$

(2) $\delta_0: Q_0\times A\to Q_0$ é a função definida, para cada $X\in Q_0$ e cada $a\in A$, por

$$\delta_0(X,a)=\bigcup_{q\in X}\delta(q,a)$$

(3) $i_0 = \{i\}$

(4) $F_0 = \{X \in Q_0 \mid X \cap F \neq \emptyset\}$

Suponhamos que $L(\mathcal{ND}) = L$. Queremos mostrar que $L(\mathcal{D}) = L$.

Lema 2.2.2. Para cada $u \in A^*$: $\delta_0^*(X, u) = \bigcup_{q \in X} \delta^*(q, u)$

Demonstração. Indução em u. □

Segue que, para cada $u \in A^*$, ${\delta_0}^*(i_0, u) = \delta^*(i, u)$.

Demonstração. De facto

$$\begin{aligned} {\delta_0}^*(i_0, u) &= {\delta_0}^*(\{i\}, u) && \text{(por (3))} \\ &= \bigcup_{q \in \{i\}} \delta^*(q, u) && \text{(pelo lema 2.2.2)} \\ &= \delta^*(i, u) \end{aligned}$$

□

Portanto

$$\begin{aligned} L(\mathcal{ND}) &= \{u \in A^* \mid \delta^*(i, u) \cap F \neq \emptyset\} && \text{(por def. de } L(\mathcal{ND})) \\ &= \{u \in A^* \mid {\delta_0}^*(i_0, u) \cap F \neq \emptyset\} && \text{(pela obs. anterior)} \\ &= \{u \in A^* \mid {\delta_0}^*(i_0, u) \in F_0\} && \text{(por (4))} \\ &= L(\mathcal{D}) && \text{(por def. de } L(\mathcal{D})) \\ &= L \end{aligned}$$

como o pretendido. □

Lema da Bombagem

O lema da bombagem é uma ferramenta poderosa para provar que certas linguagens não são regulares, ou seja, para provar que certas linguagens não são aceites por algum autómato finito. O lema também é útil no desenvolvimento de algoritmos para responder a certas questões concernente ao autómato, como se a linguagem aceite pelo autómato é finita ou infinita. Segue o lema com a sua devida prova:

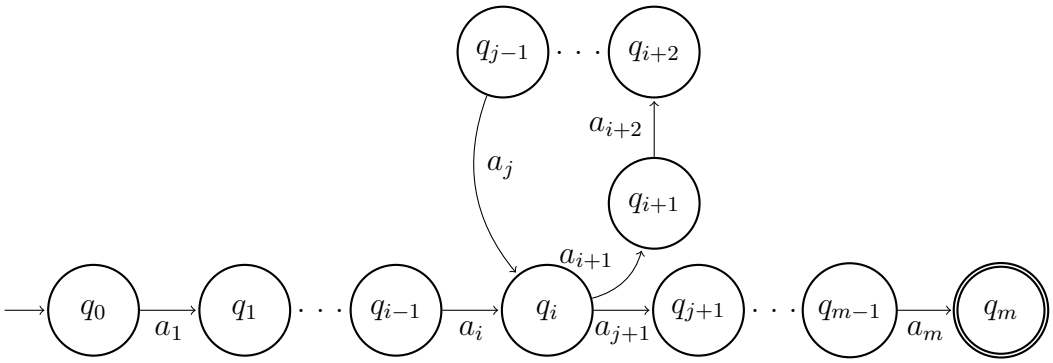
Lema 2.2.3. Seja L uma linguagem reconhecível infinita sobre um alfabeto A. Existe uma constante $n \in \mathbb{N}$ tal que, para toda a palavra $u \in L$ de comprimento superior ou igual a n, existem palavras x, y, $z \in A^*$ tais que:

1. $u = xyz$;
2. $|xy| \leq n$ e $y \neq \epsilon$;
3. $\forall t \in \mathbb{N}_0$, $xy^t z \in L$.

Demonstração. Seja $\mathcal{A} = (Q, A, \delta, i, F)$ um autómato finito com n estados tal que $L = L(\mathcal{A})$. Seja $u = a_1 a_2 ... a_m$ (com a_1, a_2, ..., $a_m \in A$) uma palavra aceite por $\mathcal{A}$, onde $m \geq n$ (tal palavra existe porque L é infinita). Mas $\mathcal{A}$ só tem n estados, por isso, algum dos $q \in Q$, aparece mais de uma vez em qualquer caminho etiquetado por $a_1 a_2 ... a_m$. Escolha-se um tal caminho bem sucedido:

$$i = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \cdots q_{m-1} \xrightarrow{a_m} q_m \in F$$

Consideremos inteiros i e j, com $0 \leq i < j \leq n$ tais que $q_i = q_j$. Tem-se:



Tomemos:

$$x = \begin{cases} \epsilon & \text{se } i = 0 \\ a_1 a_2 ... a_i & \text{se } i \neq 0 \end{cases}$$

$$y = a_{i+1} a_{i+2} ... a_j$$

$$z = \begin{cases} \epsilon & \text{se } j = m \\ a_{j+1} ... a_m & \text{se } j \neq m \end{cases}$$

Então tem-se $|xy| = j \leq n$, $|y| \geq 1$ (porque $i < j$) e, para todo $t \in \mathbb{N}_0$, $xy^t z \in L$, pois $xy^t z$ é etiqueta de um caminho bem sucedido.

□

2.3 Algoritmos sobre grafos

Esta secção apresenta métodos para representar e pesquisar um grafo. Pesquisar um grafo significa seguir sistematicamente as arestas do grafo de modo que visite cada vértice do grafo. Um algoritmo de busca de grafos pode descobrir muito sobre a estrutura de um grafo. Muitos algoritmos começam por pesquisar a sua entrada para obter esta informação estrutural. Outros algoritmos de grafos são organizados como elaborações simples de algoritmos básicos de pesquisa de grafos.

Inicialmente ver-se-á formas de representar um grafo e, a seguir, dois algoritmos de pesquisa de grafo. Os créditos do conteúdo abordado nesta secção são dados a Cormen et al. 2001.

Representação de grafos

Um grafo G é um par ordenado (V, E) onde V é um conjunto finito de vértices e $E \subseteq V \times V$. Os elementos de E dizem-se as arestas do grafo. Para X um conjunto qualquer, $\#X$ denota o número de elementos de X.

Existem duas formas padrão de representar um grafo $G = (V, E)$:

- como um conjunto de listas de adjacência , ou
- como uma matriz de adjacência .

Ambas as formas são aplicáveis a grafos dirigidos e não dirigidos. A representação por listas de adjacências é geralmente a mais usual, pois fornece uma maneira compacta de representar grafos esparsos, aqueles para os quais o número de arestas ($\#E$) é muito inferior $(\#V)^2$. A representação por matrizes de adjacência é mais usada quando o grafo é denso, aqueles para os quais $\#E$ está muito próximo de $(\#V)^2$.

A representação **por listas de adjacências** do grafo $G = (V, E)$ consiste em um vetor Adj de $\#V$ listas, uma para cada vértice em V. Para cada $u \in V$ a lista de adjacência $Adj[u]$ contem todos os vértices v tal que existe uma aresta $(u, v) \in E$, ou seja, $Adj[u]$ consiste em todos os vértices adjacentes a u em G. Os vértices em cada lista de adjacência são

normalmente armazenados em uma ordem arbitrária. A figura 2.5(b) é uma representação por listas de adjacências do grafo não dirigido da figura 2.5(a). Similarmente, a figura 2.6(b) é uma representação de lista de adjacências do grafo dirigido da figura 2.6(a).
Se G é um grafo dirigido, a soma dos comprimentos de todas as listas de adjacência é $\#E$, já que uma aresta da forma (u, v) é representada por ter v aparecendo em $Adj[u]$. Se G é um grafo não dirigido, a soma dos comprimentos de todas as listas de adjacências é $2(\#E)$, já que se (u, v) é uma aresta não dirigida, então u aparece na lista de adjacências de v e vice-versa.

A representação por listas de adjacências é bastante robusta na medida em que pode ser modificada para suportar muitas outras variantes de grafos. Uma desvantagem potencial da representação por listas de adjacências é que não há maneira mais rápida de determinar se uma determinada aresta (u, v) está num grafo, pois a única consiste em procurar v em $Adj[u]$.

Esta desvantagem pode ser contornada representando o grafo por uma matriz de adjacência. A representação por **matriz de adjacência** do grafo $G = (V, E)$, assume que os vértices são numerados $1, 2, ..., \#V$ de uma forma arbitrária. Então a representação matriz-adjacência de G consiste em uma matriz $(\#V) \times (\#V),\ A = (a_{ij})$, de tal forma que

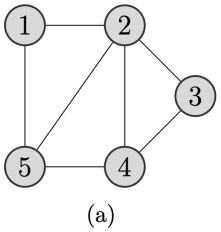
$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E, \\ 0 & \text{se não} \end{cases} .$$

As figuras 2.5 (c) e 2.6(c) são as matrizes de adjacências dos grafos não dirigido e dirigido das figuras 2.5(a) e 2.6(a), respetivamente.

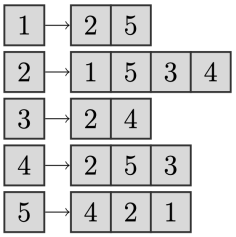
Se uma aresta não existir, um valor NIL pode ser armazenado como sua entrada na matriz correspondente, embora para muitos problemas seja conveniente usar um valor como 0 ou ∞.

A transposta da matriz $A = (a_{ij})$ é a matriz $A^T = (a_{ji})$. Como em um grafo não dirigido, (u, v) e (v, u) representam a mesma aresta, a matriz de adjacência A de um grafo não dirigido é sua própria transposta: $A = A^T$.

Embora a representação por listas de adjacências seja assintoticamente pelo menos tão eficiente quanto a representação por matriz de adjacência, a simplicidade de uma matriz de adjacência pode torná-la preferível quando os grafos são razoavelmente pequenos.



(a)

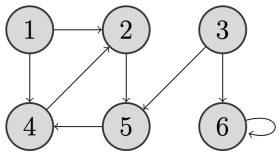


(b)

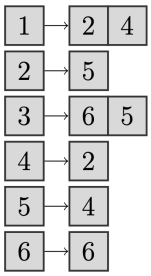
	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c)

Figura 2.5: Duas representações de um grafo não dirigido. **(a)** Um grafo G não dirigido com cinco vértices e sete arestas. **(b)** Uma representação de listas de adjacências de G. **(c)** Uma representação matriz de adjacência de G.



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Figura 2.6: Duas representações de um grafo dirigido. **(a)** Um grafo G com seis vértices e sete arestas. **(b)** Uma representação de listas de adjacências de G. **(c)** Uma representação matriz de adjacência de G.

Breadth-first search(BFS)

Com vista um pleno entendimento do algoritmo de grafos **BFS**, explicar-se-á antes o algoritmo sobre filas (em inglês: *Queues*) de espera.

As filas são conjuntos dinâmicos nos quais o elemento eliminado da fila pela operação DELETE é predefinido. Em uma fila, o elemento eliminado é sempre aquele que está na fila há mais tempo: a fila implementa uma política de primeira entrada, primeira saída, ou FIFO.

Chamou-se a operação inserir em uma fila **ENQUEUE**, e eliminar **DEQUEUE**. A fila tem uma cabeça (*head*) e uma cauda (*tail*). Quando um elemento é inserido, este toma seu lugar na cauda da fila, o elemento eliminado é sempre aquele que está na cabeça da fila. A figura 2.7 mostra uma maneira de implementar uma fila de no máximo $n-1$ elementos usando uma matriz Q[1 ... n]. A fila tem um atributo $head[Q]$ que indica ou aponta para sua cabeça. O atributo $tail[Q]$ indica o próximo local no qual um elemento recém-chegado será

inserido na fila. Os elementos na fila estão em locais $head[Q]$, $head[Q]+1, ..., tail[Q]-1$. Quando $head[Q] = tail[Q]$, a fila está vazia. Inicialmente, temos $head[Q] = tail[Q] = 1$. Quando a fila está vazia, uma tentativa de eliminar um elemento causa o sub-fluxo (em inglês: *underflow*) da fila. Quando $head[Q] = tail[Q]+1$, a fila está cheia, e uma tentativa de inserir um elemento causa sobre-fluxo (em inglês: *overflow*). No algoritmo 1, temos o pseudocódigo das operações ENQUEUE e DEQUEUE.

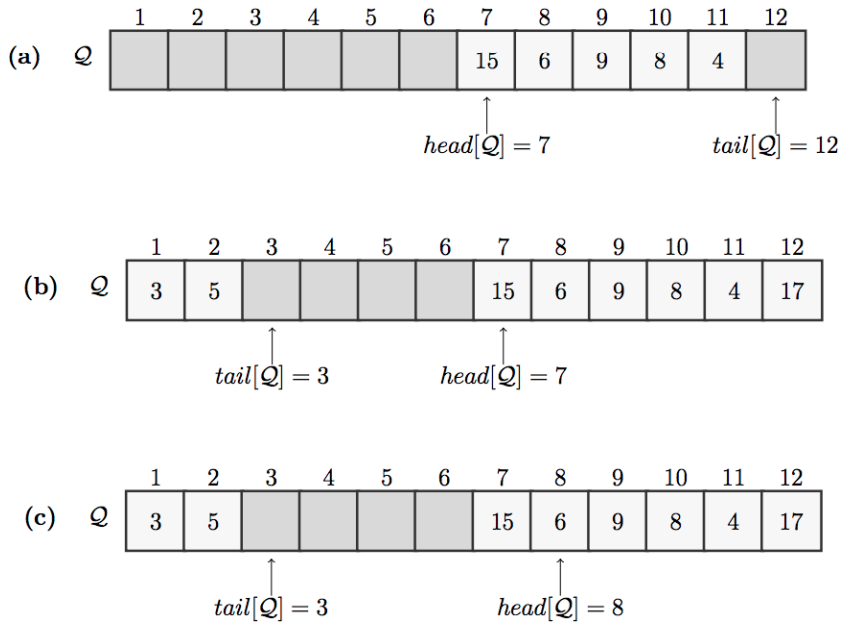


Figura 2.7: Implementação de uma fila utilizando uma matriz $Q[1...12]$. Os elementos da fila só aparecem nas posições ligeiramente sombreadas. **(a)** A fila tem 5 elementos, em locais $Q[7...11]$. **(b)** A configuração da fila após as chamadas ENQUEUE$(Q, 17)$, ENQUEUE$(Q, 3)$ e ENQUEUE$(Q, 5)$. **(c)** A configuração da fila após a chamada DEQUEUE(Q).

O algoritmo BFS também usa uma fila Q, portanto, usa as operações ENQUEUE e DEQUEUE no seu código. Segue-se os detalhes do funcionamento deste método de pesquisar um grafo.

Algoritmo 1 Pseudocódigo das operações ENQUEUE e DEQUEUE. Cada operação leva o tempo $\mathcal{O}(1)$.

```
1: procedure ENQUEUE(Q, x)
2:     Q[tail[Q]] ← x
3:     if tail[Q] = length[Q] then
4:         tail[Q] ← 1
5:     else
6:         tail[Q] ← tail[Q] + 1
1: procedure DEQUEUE(Q)
2:     x ← Q[head[Q]]
3:     if head[Q] = length[Q] then
4:         head[Q] ← 1
5:     else
6:         head[Q] ← head[Q] + 1
7:     return x
```

Breadth-first-search (BFS) é um dos algoritmos mais simples para pesquisar um grafo e o modelo para muitos algoritmos importantes de grafos. Dado um grafo $G = (V, E)$ e um vértice fonte s distinguido, BFS explora sistematicamente as arestas de G para "descobrir" cada vértice que é acessível a partir de s. Calcula a distância (menor número de arestas) de s a cada vértice acessível u e guarda-a no campo d. Também produz uma árvore com raiz s que contém todos os vértices alcançáveis, denominada *breadth-first tree* (bft). A árvore bft é definida pelo campo π em cada vértice.

O algoritmo *Breadth-first-search* (em português, pesquisa-primeiro-em-largura) é assim chamado porque expande a fronteira entre vértices descobertos e não descobertos uniformemente através da largura da fronteira. Ou seja, o algoritmo descobre todos os vértices à distância k do vértice de origem s, antes de descobrir qualquer vértice à distância $k+1$. Para acompanhar o progresso, o BFS pinta cada vértice branco (WHITE), de cinza(GRAY) ou de preto(BLACK). Todos os vértices começam em WHITE e podem mais tarde tornar-se GRAY e depois BLACK. Um vértice é descoberto na primeira vez em que é encontrado durante a pesquisa, altura em que se torna não branco. Vértices GRAY e BLACK, portanto, foram descobertos, mas o BFS distingue entre eles para garantir que a busca prossegue de forma ampla, em primeiro lugar. Se $(u, v) \in E$ e u um vértice BLACK, então o vértice v é ou GRAY ou BLACK; isto é, todos os vértices adjacentes aos vértices BLACK foram descobertos. Vértices GRAY podem ter alguns vértices adjacentes WHITE; eles representam a fronteira entre vértices descobertos

e não descobertos.

A árvore bft, construída pelo BFS, contém inicialmente apenas a sua raiz, que é o vértice de origem s. Sempre que um vértice v WHITE é descoberto no decurso da verificação da lista de adjacência de um vértice u já descoberto, o vértice v e a aresta (u, v) são adicionados à árvore. Dizemos que u é o predecessor de v na bft. Como um vértice é descoberto no máximo uma vez, ele tem no máximo um predecessor. As relações de antecessor e descendente na bft são definidas como de costume: se u está num caminho para o vértice v, então u é um antecessor de v e v é um descendente de u.

O procedimento BFS abaixo assume que o grafo de entrada $G = (V, E)$ é representado usando listas de adjacências. Este mantém várias estruturas de dados adicionais com cada vértice no grafo. A cor de cada vértice $u \in V$ é armazenada na variável $color[u]$, e o predecessor de u é armazenado na variável $\pi[u]$. Se u não tem nenhum antecessor (por exemplo, se $u = s$ ou u não foi descoberto), então $\pi[u] = NIL$. A distância da fonte s ao vértice u calculada pelo algoritmo é armazenada em $d[u]$.

Mais formalmente, definimos o sub-grafo predecessor de G como $G_\pi = (V_\pi, E_\pi)$, onde $V_\pi = \{v \in V \mid \pi[v] \neq NIL\} \cup \{s\}$ e $E_\pi = \{(\pi[v], v) \mid v \in (V_\pi - \{s\})\}$. Prova-se que o sub-grafo predecessor G_π é uma árvore, se V_π consiste nos vértices alcançáveis de s e, para todos $v \in V_\pi$, há um caminho simples único de s para v em G_π que é também um caminho mais curto de s para v em G.

O pseudocódigo a seguir é o algoritmo BFS, o grafo de entrada G pode ser dirigido ou não dirigido. O método BFS funciona da seguinte forma: as linhas $2 - 5$ pintam cada vértice WHITE, definem $d[u]$ para ser infinito para cada vértice u, e definem o antecessor de cada vértice para ser NIL. A linha 6 pinta como GRAY o vértice de origem s, uma vez que é considerado para ser descoberto quando o método começa. A linha 7 inicializa $d[s]$ para 0, e a linha 8 define o antecessor da fonte para ser NIL. As linhas $9 - 10$ inicializam a fila $\mathcal{Q}$, contendo apenas o vértice s. O ciclo *while* nas linhas $11 - 19$ itera enquanto existirem vértices GRAY, que são vértices descobertos que ainda não tiveram suas listas de adjacências completamente examinadas, pois o ciclo mantém o seguinte invariante: no teste da linha 11, a fila $\mathcal{Q}$ consiste no conjunto de vértices GRAY.

Os resultados do BFS podem depender da ordem em que os vizinhos de um determinado

vértice são visitados na linha 12: a árvore bft pode variar, mas as distâncias calculadas pelo algoritmo não variam.

Algoritmo 2 Pseudocódigo BFS de um grafo não dirigido. Dentro de cada vértice u é mostrado $d[u]$. A fila Q é mostrada no início de cada iteração do ciclo **while**.

```
1:  procedure BFS(G, s)
2:      for each vertex u ∈ V[G] − {s} do        ▷ "pinta" WHITE todos vértices excepto s
3:          color[u] ← WHITE
4:          d[u] ← ∞
5:          π[u] ← NIL
6:      color[s] ← GRAY                          ▷ "pinta" GRAY o vértice fonte
7:      d[s] ← 0
8:      π[s] ← NIL
9:      Q ← ∅                                    ▷ fila dos vértices já visitados
10:     ENQUEUE(Q, s)                            ▷ Insere o vértice fonte na fila
11:     while Q ≠ ∅ do                           ▷ Visita os vértices atingíveis a partir de s
12:         u ← DEQUEUE(Q)
13:         for each v ∈ Adj[u] do               ▷ Visita os vértices adjacentes à u
14:             if color[v] =WHITE then          ▷ o vértice v ainda não foi visitado
15:                 color[v] ← GRAY
16:                 d[v] ← d[u] + 1
17:                 π[v] ← u
18:                 ENQUEUE(Q, v)                ▷ insere o vértice adjacente na fila
19:         color[u] ← BLACK                     ▷ "pinta" BLACK o vértice u
```

Análise do tempo de execução

Após a inicialização, nenhum vértice é não branco e, portanto, o teste na linha 14 garante que cada vértice será inserido na fila no máximo uma vez e, portanto, executa a operação DEQUEUE no máximo uma vez. As operações ENQUEUE e DEQUEUE levam tempo $\mathcal{O}(1)$, de modo que o tempo total dedicado às operações na fila é $\mathcal{O}(\#V)$. Como a lista de adjacências de cada vértice é examinada somente quando o vértice é eliminado, cada lista de adjacências é percorrida no máximo uma vez. Uma vez que a soma dos comprimentos de todas as listas de adjacências é $\Phi(\#E)$ o tempo total gasto na verificação das listas de adjacências é $\mathcal{O}(\#E)$. A sobrecarga para inicialização é $\mathcal{O}(\#V)$, e portanto o tempo total de funcionamento do BFS é $\mathcal{O}(\#V + \#E)$. Assim, o método BFS é executado em tempo linear no tamanho da representação lista-adjacência de G.

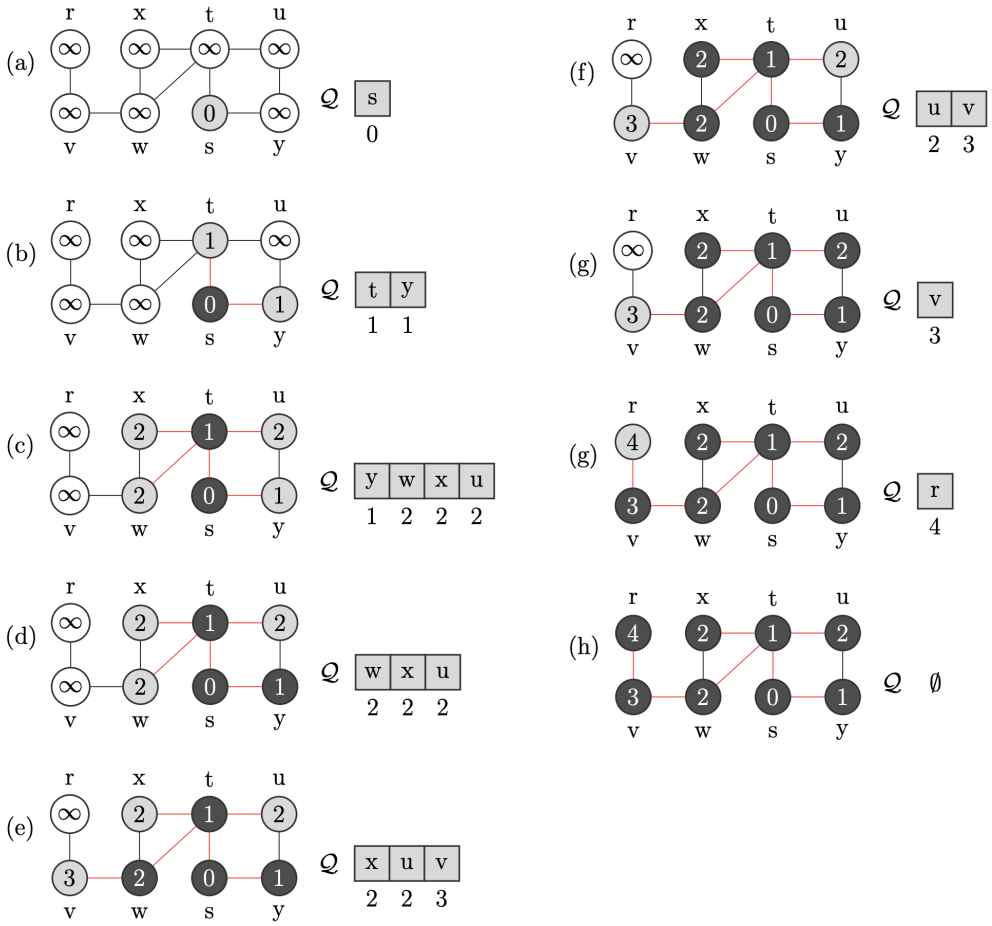


Figura 2.8: Ilustração do progresso do método BFS. As arestas são mostradas avermelhadas à medida que são produzidas pelo BFS. As distâncias dos vértices são mostradas ao lado dos vértices na fila.

Depth-first search(DFS)

A estratégia seguida pelo algoritmo *Depth-first search* (em português: pesquisa-primeiro-em-profun-

didade) é, como o próprio nome indica, pesquisar "mais a fundo" no grafo sempre que possível. No *Depth-first search* (DFS), as arestas são exploradas a partir do vértice v mais recentemente descoberto que ainda tem arestas inexploradas que partem de v. Quando forem exploradas todas as arestas de v, a pesquisa "retrocede" para explorar as arestas, que partem do vértice a partir do qual v foi descoberto. Esse processo continua até que tenhamos descoberto todos os vértices que são alcançáveis a partir do vértice fonte inicial. Se algum vértice não descoberto permanecer, então um deles é selecionado como uma nova fonte e a pesquisa é repetida a partir dessa fonte. Todo esse processo é repetido até que todos os vértices sejam descobertos.

Como no BFS, sempre que um vértice v é descoberto durante em uma leitura da lista de adjacência de um vértice u já descoberto, o DFS regista esse evento através da configuração do campo predecessor $\pi[v]$. Ao contrário do BFS, cujo sub-grafo predecessor forma uma árvore, o sub-grafo predecessor produzido pelo DFS pode ser composto de várias árvores (denominadas *depth-first trees* (dft)), pois a busca pode ser repetida a partir de múltiplas fontes e, portanto, o sub-grafo predecessor do DFS forma uma floresta (que denominaremos por *depth-first forest* (dff)), composta por várias dft. O sub-grafo predecessor do método DFS é, portanto, definido de forma ligeiramente diferente do BFS: seja $G_\pi = (V, E_\pi)$ onde $E_\pi = \{(\pi[v], v) \mid v \in V \text{ e } \pi[v] \neq NIL\}$. As arestas em E_π são chamadas de arestas de árvore.

Os vértices, tal como no BFS, são coloridos durante a pesquisa para indicar seu estado. Cada vértice é inicialmente branco (WHITE), ficando cinzento (GRAY) quando descoberto na pesquisa, e preto (BLACK) quando terminado, ou seja, quando a sua lista de adjacências for completamente examinada. Esta técnica garante que cada vértice termine em exatamente uma dft, para que estas árvores sejam disjuntas.

Além de criar uma dff, o DFS também marca o tempo em cada vértice. Cada vértice v tem duas marcas de tempo: a primeira marca, representado por $d[v]$, regista quando v é descoberto pela primeira vez (e v torna-se cinzento), e a segunda, representado por $f[v]$, regista quando a pesquisa termina de examinar a lista de adjacência de v (e v torna-se preto). Estas marcas de tempo são usadas em muitos algoritmos de grafos e são geralmente úteis no raciocínio sobre o comportamento do método DFS.

Estas marcas são inteiros entre 1 e $2(\#V)$, uma vez que existe um evento de descoberta e de finalização para cada um dos vértices V e o relógio é incrementado precisamente quando há um desses eventos.

Para cada vértice u, $d[u] < f[u]$. O vértice u é WHITE antes do tempo $d[u]$, GRAY entre o tempo $d[u]$ e o tempo $f[u]$, e BLACK depois disso.

O seguinte pseudocódigo é o algoritmo DFS básico. O grafo de entrada G pode ser dirigido ou não.

O procedimento DFS(G) funciona da seguinte forma: As linhas 2 - 4 pintam todos os vértices

Algoritmo 3 Pseudocódigo do método DFS

```
1: procedure DFS(G)
2:     for each vertex u ∈ V[G] do                ▷ "pinta" WHITE todos vértices
3:         color[u] ← WHITE
4:         π[u] ← NIL
5:     time ← 0
6:     for each vertex u ∈ V[G] do                ▷ descobre os vértices fontes u
7:         if color[u] = WHITE then
8:             DFS-VISIT(u)
1: procedure DFS-VISIT(u)
2:     color[u] ← GRAY                            ▷ o vértice WHITE u foi descoberto
3:     time ← time + 1
4:     d[u] ← time
5:     for each v ∈ Adj[u] do                     ▷ explora arestas (u, v)
6:         if color[v] = WHITE then
7:             π[v] ← u
8:             DFS-VISIT(v)
9:     color[u] ← BLACK                           ▷ "pinta" BLACK o vértice u
10:    time ← time + 1
11:    f[u] ← time
```

a WHITE e inicializam seus campos π para NIL. A linha 5 inicializa o contador de tempo. As linhas 6 - 8 verificam cada vértice em V e, se um vértice branco é encontrado, o vértice é visitado usando DFS-VISIT(u). Toda a vez que o DFS-VISIT(u) é chamado na linha 8, o vértice u torna-se a raiz de uma nova árvore na dff. Quando procedimento DFS(G) retorna, a cada vértice u foi atribuído um tempo de descoberta $d[u]$, e um tempo de finalização $f[u]$.

Em cada chamada DFS-VISIT(u), o vértice u é inicialmente WHITE. A linha 2 pinta u como GRAY, a linha 3 incrementa o tempo e a linha 4 regista o novo valor de tempo como o tempo de descoberta $d[u]$. As linhas 5 - 8 examinam cada vértice v adjacente a u e visitam recursivamente v, se v for WHITE. Como cada vértice $v \in Adj[u]$ é considerado na linha 5, dizemos que a aresta (u, v) é **explorada** pelo método DFS. Finalmente, depois de cada aresta que parte de u ter sido explorada, as linhas 9-10 pintam u como BLACK e registam o tempo de finalização em $f[u]$.

O DFS fornece informações valiosas sobre a estrutura de um grafo. Talvez a propriedade mais básica do DFS é que o sub-grafo antecessor G_π de fato forma uma floresta de árvores, já que a estrutura de uma dft reflete exatamente a estrutura das chamadas recursivas do DFS-VISIT. Ou seja, $u = \pi[v]$ se, e somente se, DFS-VISIT(v) foi chamado durante uma verificação da

lista de adjacências de u. Adicionalmente, o vértice v é um descendente do vértice u em dff se, e somente se, v for descoberto durante o tempo em que u é GRAY.

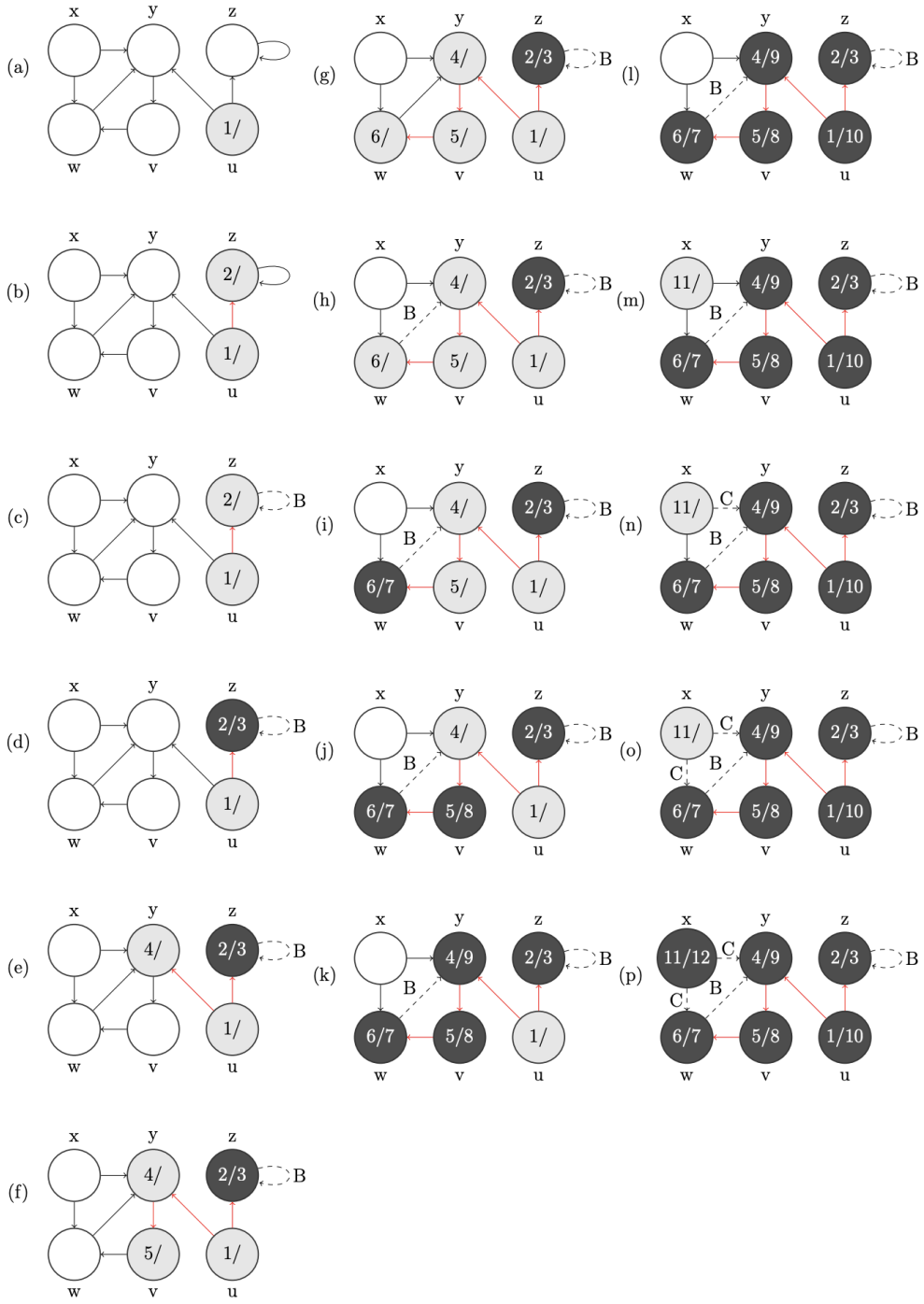


Figura 2.9: Progresso do método DFS em um grafo dirigido. Como as arestas são exploradas pelo algoritmo, elas são mostradas avermelhadas (se forem arestas de árvore) ou tracejadas (caso contrário). As arestas que não são de árvore são rotuladas como B ou C se forem arestas para trás (*back edges*) ou arestas cruzadas respectivamente (vai ser explicada mais a frente). Os vértices são marcados com tempo de descoberta/tempo de finalização.

O DFS visita os vértices de um grafo em uma ordem arbitrária, por isso, se um grafo for pelo

menos duas vezes argumento do DFS, o modo como este grafo for pesquisado em cada vez, pode variar. O garantido é que em cada vez, todos os vértices do grafo serão visitados.

Análise do tempo de execução

Os ciclos nas linhas 2 - 4 e 6 - 8 do DFS(G) levam tempo $\Theta(V)$, excluindo tempo de execução das chamadas ao DFS-VISIT(u). O procedimento DFS-VISIT(u) é chamado exatamente uma vez para cada vértice $u \in V$ já que DFS-VISIT é invocado apenas em vértices brancos e a primeira coisa que ele faz é pintar o vértice como GRAY. Durante uma execução do DFS-VISIT(v), o ciclo nas linhas 5 - 8 é executado $\#(Adj[v])$ vezes. Como

$$\sum_{v \in V} \#(Adj[v]) \in \Theta(\#E),$$

o custo total da execução das linhas 5 - 8 do DFS-VISIT é $\Theta(\#E)$. O tempo de execução do método DFS é portanto $\Theta(\#V + \#E)$.

Classificação das arestas

A pesquisa realizada pelo método DFS pode ser usada para classificar as arestas do grafo de entrada $G = (V, E)$. Esta classificação de arestas pode ser usada para obter informações importantes sobre um grafo. Podemos definir quatro tipos de arestas em termos da dff G_π produzido pelo DFS com entrada G:

1. **Arestas de árvore** são arestas na dff G_π. Aresta (u, v) é uma aresta de árvore se v foi descoberto pela primeira vez ao explorar a aresta (u, v);

2. **Arestas para trás** (em inglês: *back edges*) são aquelas arestas (u, v) que ligam um vértice u a um antecessor v numa dft. Os auto-ciclos, que podem ocorrer em grafos dirigidos, são considerados como arestas para trás;

3. **Arestas para frente** são as arestas (u, v) que não são arestas de árvores, conectando um vértice u a um descendente v em uma dft;

4. **Arestas cruzadas** são todas as outras arestas que não sejam 1., 2. e 3. Elas podem ir entre vértices na mesma dft, desde que um vértice não seja um ancestral do outro, ou podem ir entre vértices em dft diferentes.

Capítulo 3

Problemas de Decisão

Nesta secção, entraremos de concreto no tema desta dissertação. Tivemos em conta o que já foi tratado por Hopcroft e Ullman 1979; Hopcroft, Motwani e Ullman 2006; Lewis e Papadimitriou 1990; Cormen et al. 2001.

Consideramos algumas questões importantes sobre linguagens regulares, mas antes, devemos considerar o que significa fazer uma pergunta sobre uma linguagem. Seja L uma linguagem:

- L é uma linguagem vazia?
- Seja w uma palavra de A^* : w está em L?
- L é finita?

Diremos "sim" ou "não" a questões do género, por intermédio de algoritmos que decidem se L tem ou não determinada propriedade.

No entanto, para muitas questões que fazemos, os algoritmos existem apenas para a classe de linguagens regulares. Nos concentraremos em questões referentes a **aceitação da linguagem vazia**, **aceitação da linguagem infinita** bem como o problema de decidir se **uma expressão regular representa a linguagem vazia ou infinita** e assumimos (de início) que conjuntos regulares são representados por autómato finito. Antes, começaremos por definir o grafo de um autómato.

Definição 3.0.1. Seja $\mathcal{A} = (Q, A, \delta, i, F)$ um AFD. $\mathcal{G}(\mathcal{A}) = (V, E)$ é o grafo de $\mathcal{A}$ onde:

- $V = Q$,
- $E = \{(q, q') \in Q \times Q \mid \exists x \in A,\ \delta(q, x) = q'\}$.

$\mathcal{G}(\mathcal{A})$ é cíclico se no seu grafo existir pelo menos um ciclo.

3.1 Problema da linguagem vazia

Antes de tudo, enunciaremos alguns teoremas que poderão ser úteis, seguidos de sua respetiva demonstração.

Teorema 3.1.1. Sejam $\mathcal{A}$ um AFD e $n \in \mathbb{N}$ o número de estados de $\mathcal{A}$.
$L(\mathcal{A}) \neq \emptyset$ se, e só se, $\mathcal{A}$ aceita alguma palavra de comprimento menor que n.

Demonstração.
$\Leftarrow$
Trivial.

$\Rightarrow$
Suponhamos que $L(\mathcal{A}) \neq \emptyset$. Seja $u \in L(\mathcal{A})$ de comprimento mínimo. Pelo lema da Bombagem, $|u| < n$ porque, se $|u| \geq n$, pelo lema da Bombagem, existiriam $x, y, z \in A^*$ tais que,

$$\begin{cases} u = xyz,\ y \neq \epsilon \\ |xy| \leq n \\ xy^k z \in L(\mathcal{A}),\ \forall k \in \mathbb{N}_0 \end{cases}$$

donde $xz \in L(\mathcal{A})$ (tomando $k = 0$), mas $|xz| < |xyz| = |u|$, que nos leva a uma contradição, pois u é uma palavra de comprimento minimal em $L(\mathcal{A})$. Portanto $\mathcal{A}$ aceita palavras de comprimento menor que n. □

Teorema 3.1.2. Seja $\mathcal{A} = (Q, A, \delta, i, F)$ um AFD. Então $L(\mathcal{A}) \neq \emptyset$ se, e só se, existir em $\mathcal{A}$ um estado final acessível.

Demonstração. Existe $u \in A^*$ tal que $u \in L(\mathcal{A})$ se, e só se, existe $u \in A^*$ tal que $\delta^*(i, u) \in F$. Portanto, $L(\mathcal{A}) \neq \emptyset$ se e só se, existe $f \in F$ tal que $\delta^*(i, u) = f$, como o pretendido. □

Seja $\mathcal{A}$ um AFD. Como decidir se $\mathcal{A}$ aceita a linguagem vazia?!

Pelo teorema 3.1.1, o algoritmo para decidir se $\mathcal{A}$ aceita a linguagem vazia consistiria em averiguar se alguma palavra de comprimento até n (sendo n o número de estado de $\mathcal{A}$) é aceite por $\mathcal{A}$. Mas este algoritmo não seria eficiente, porque, no pior caso, teria que testar todas as palavras de comprimento não superior a n, num tempo de execução de ordem $\mathcal{O}(2^n)$.

No entanto, pode-se testar facilmente se $\mathcal{A}$ aceita o conjunto vazio, tomando o seu grafo e excluindo todos os estados que não são acessíveis. Se, depois disto, o grafo tiver pelo menos um estado de aceitação/final, a linguagem é não vazia. Também poderíamos dizer que, se $\mathcal{A}$ tiver um caminho bem sucedido, então a linguagem é não vazia. Enfatizar que esta alternativa é refletida no teorema 3.1.2.

Fizemos um algoritmo para testar se um dado AFD $\mathcal{A}$ aceita a linguagem vazia, tendo em conta o seu grafo, recorrendo ao algoritmo BFS. Segue o pseudocódigo deste algoritmo:

Algoritmo 4 Algoritmo para decidir se um dado AFD $\mathcal{A}$ aceita ou não a linguagem vazia

```
1:  procedure VAZIA(A)
2:      i := estado inicial de A
3:      F := conjunto de estados finais de A
4:      G := G(A)
5:      G' := BFS(G, i)
6:      V := conjunto de vértices BLACK de G'        ▷ vértices acessíveis a partir de i
7:      acessivel := falso
8:      while ( ¬ acessivel ∧ V ≠ ∅) do              ▷ procura um vértice acessível final
9:          v := escolher em V
10:         if v ∈ F then
11:             acessivel := true
12:         else
13:             V := V \ {v}
14:     return(acessivel)
```

O algoritmo VAZIA começa por obter o grafo do autómato $\mathcal{A}$ e em seguida submete este grafo ao BFS com vértice fonte o estado inicial i, resultando assim o grafo G'. G' é o grafo que contém todos vértices que são acessíveis (se existir) a partir do estado inicial, pois o BFS constrói uma árvore com raiz i. Com o ciclo *while* queremos saber se um dos vértices acessíveis é um estado final e quando assim for, o booleano $acessivel$ terá valor $true$. Este processo deve decorrer enquanto se puder escolher $v \in V$, a escolha é possível porque o grafo está

dado por listas de adjacências que é um vetor de listas, e por isso, temos uma ordenação de vértices permitindo assim escolher um primeiro vértice e depois passar para o seguinte e assim sucessivamente. No final de todo este processo, o algoritmo retorna o booleano $acessivel$.

Análise do tempo de execução

Sejam V vértices e E arestas de $\mathcal{G}(\mathcal{A})$. O algoritmo VAZIA($\mathcal{A}$) na linha 4, percorre o grafo do autómato $\mathcal{A}$, e no entanto, percorre todos seus estados ou vértices e todas as transições ou arestas, por isso, este processo seria em tempo $\mathcal{O}(\#V + \#E)$. Na linha 5, custa exatamente a unidade de tempo em que é executado o BFS, que também é $\mathcal{O}(\#V + \#E)$. As linhas correspondentes ao ciclo *while*, as linhas de 8 - 13, o tempo é consumido com base a cada iteração do ciclo terminando depois de observar os $\#V$ elementos; e em cada iteração, as linhas 9 e 13 consumirão uma unidade de tempo cada, a linha de 10 - 12, consumirá exatamente $\#F + 1$ unidades de tempo. Assim sendo, o tempo gasto nas linha de 8 - 13, é:

$$\begin{aligned}
\sum_{i=1}^{\#V}(\#F + 3) &= \#V \times (\#F + 3) \\
&\leq \#V \times (\#V + 3) && \text{(por } \#F \leq \#V\text{)} \\
&= (\#V)^2 + 3(\#V) \\
&\in \mathcal{O}((\#V)^2) \\
&\subseteq \mathcal{O}((\#Q)^2)
\end{aligned}$$

Como

$$\begin{aligned}
\mathcal{O}(\#V + \#E) &\subseteq \mathcal{O}(\#V + (\#Q)^2) \\
&\subseteq \mathcal{O}(\#Q)^2)
\end{aligned}$$

concluímos que o algoritmo VAZIA($\mathcal{A}$) é executado em tempo $\mathcal{O}((\#Q)^2)$.

3.2 Problema da linguagem infinita

Tal como na secção anterior, começamos enunciando os teoremas que serão úteis para os nossos propósitos para, posteriormente, falar sobre o problema da linguagem infinita.

Teorema 3.2.1. Sejam $\mathcal{A}$ um AFD e $n \in \mathbb{N}$, o número de estados de $\mathcal{A}$.
$L(\mathcal{A})$ é infinita se, e só se, $\mathcal{A}$ aceita alguma palavra de comprimento l, para algum l satisfazendo $n \leq l < 2n$.

Demonstração. Seja $L = L(\mathcal{A})$.

$\Leftarrow$

Suponhamos que existe $u \in A^*$ tal que $u \in L$ e $n \leq |u| < 2n$. Queremos mostrar que L é infinita.

Pelo lema da Bombagem, existem $x, y, z \in A^*$ tais que

$$\begin{cases} u = xyz,\ y \neq \epsilon \\ |xy| \leq n \\ xy^k z \in L,\ \forall k \in \mathbb{N}_0 \end{cases}$$

logo L é infinita.

$\Rightarrow$

Suponhamos que L é infinita. Com vista uma contradição, suponhamos que não existe $v \in L$ tal que $n \leq |v| < 2n$. Existem palavras em L de comprimento maior ou igual a $2n$ (porque L é infinita). Seja $u \in L$ tal que $|u| \geq 2n$ e u tem comprimento mínimo entre as palavras de L de comprimento maior ou igual a $2n$. Mas, pelo lema da Bombagem, existem $x, y, z \in A^*$ tais que,

$$\begin{cases} u = xyz,\ y \neq \epsilon \\ |xy| \leq n \\ xy^k z \in L,\ \forall k \in \mathbb{N}_0 \end{cases}$$

donde $|y| \leq n$ e $xy^0z = xz \in L$. Logo $n \leq |xz|$ (porque $|y| \leq n$ e $2n \leq |u| = |xyz|$) e $|xz| < 2n$ (porque $|xz| < |xyz| = |u|$ e u tem comprimento mínimo entre as palavras de L

de comprimento maior ou igual a $2n$). Então, existe v tal que $n \leq |v| < 2n$, o que é uma contradição. Portanto, $\mathcal{A}$ aceita alguma palavra de comprimento l, onde $n \leq l < 2n$. □

Teorema 3.2.2. Sejam $\mathcal{A} = (Q, A, \delta, i, F)$ e $\mathcal{A}' = (Q', A, \delta', i, F')$ AFD onde:

- $Q' = \{q \in Q \mid q \text{ é acessível e co-acessível em } \mathcal{A}\}$
- $F' = F \cap Q'$
- $\delta' = \delta|_{Q' \times A} = \delta \cap ((Q' \times A) \times Q')$

Qualquer que seja $u \in A^*,\ u \in L(\mathcal{A}')$ se, e só se, $u \in L(\mathcal{A})$

Demonstração. Ora vejamos que,

$$
\begin{aligned}
u \in L(\mathcal{A}') &\text{ sse } \delta'^*(i,u) \in F' && \text{(def. } L(\mathcal{A})\text{)}\\
&\text{sse } \delta'^*(i,u) \in (F \cap Q') && \text{(def. } F'\text{)}\\
&\text{sse } (\delta'^*(i,u) \in F) \text{ e } (\delta'^*(i,u) \in Q')\\
&\text{sse } \delta'^*(i,u) \in F) && (*)\\
&\text{sse } \delta^*(i,u) \in F) && (**)\\
&\text{sse } u \in L(\mathcal{A}) && \text{(def. } L(\mathcal{A})\text{)}
\end{aligned}
$$

$(*)$
- $\Downarrow$ Trivial
- $\Uparrow$ Seja $q = \delta'^*(i,u)$ e suponhamos que $q \in F$.
  Falta ver que q é acessível e co-acessível em $\mathcal{A}$.
  Como $\delta'^* \subseteq \delta^*,\ q = \delta^*(i,u)$. $\delta^*(i,u)$ é acessível em $\mathcal{A}$, pois há em $\mathcal{A}$ um caminho de i até $\delta^*(i,u)$ etiquetado por u.
  Como $q \in F,\ \delta^*(i,u)$ é trivialmente co-acessível em $\mathcal{A}$.

Lema 3.2.1. $\delta^*(q,u) = \delta'^*(q,u)$.

Demonstração. Indução em u. □

$(**)$
- $\Downarrow$ Trivial, pois $\delta'^* \subseteq \delta^*$.
- $\Uparrow$ Pelo lema 3.2.1, pois i é acessível em $\mathcal{A}$.

□

Teorema 3.2.3. Seja $\mathcal{A}$ AFD acessível e co-acessível. Seja $\mathcal{G}(\mathcal{A})$ o grafo do autómato $\mathcal{A}$. $L(\mathcal{A})$ é infinita se, e só se, $\mathcal{G}(\mathcal{A})$ é cíclico.

Demonstração.

$\Leftarrow$

Suponhamos que $\mathcal{G}(\mathcal{A})$ é cíclico. Queremos mostrar que $L(\mathcal{A})$ é infinita.
Por hipótese há um ciclo em $\mathcal{G}(\mathcal{A})$. Seja q um dos estado deste ciclo.
Por $\mathcal{A}$ ter todos seus estados acessíveis, segue que a partir do estado inicial chega-se a q.
Seja $u_1 \in A^*$ etiqueta do caminho pelo qual se chega a q, seja $u_2 \in A^*$ etiqueta do ciclo.
Então, por $\mathcal{A}$ ter todos os seus estados co-acessíveis, segue que existe $u_3 \in A^*$ etiqueta do caminho que parte de q e chega a um estado final. Logo, qualquer que seja $k \in \mathbb{N}_0$, $u_1 u_2^k u_3 \in L(\mathcal{A})$, porque $u_1 u_2^k u_3$ é a etiqueta de um caminho bem sucedido. Concluímos que $L(\mathcal{A})$ é infinita.

$\Rightarrow$

Suponhamos que $L(\mathcal{A})$ é infinita. Então $L(\mathcal{A})$ tem palavras de comprimento arbitrariamente grande.
Seja n o número de vértices em $\mathcal{G}(\mathcal{A})$, seja $u \in L(\mathcal{A})$ tal que $|u| \geq n$.
Então em $\mathcal{G}(\mathcal{A})$ existe um caminho etiquetado por u, no qual um vértice é visitado mais de uma vez, concluindo que $\mathcal{G}(\mathcal{A})$ é cíclico. □

Seja $\mathcal{A}$ um AFD. Como decidir se $\mathcal{A}$ aceita a linguagem infinita?

O teorema 3.2.1 sugere-nos um algoritmo, que consistiria em averiguar se $\mathcal{A}$ aceita alguma palavra de comprimento entre n e $2n$, onde n é o número de estado de $\mathcal{A}$. Mas tal algoritmo, no pior dos casos teria que testar todas as palavras de comprimento entre n e $2n$. Este processo seria executado num tempo de ordem $\mathcal{O}(2^n)$ e, portanto, o algoritmo não seria eficiente.

O teorema 3.2.3 auxiliado pelo teorema 3.2.2, sugerem um outro algoritmo, que basicamente consiste em apagar no grafo de $\mathcal{A}$ todos os estados não acessíveis e não co-acessíveis. Se no final deste processo, o grafo de $\mathcal{A}$ tiver um ciclo, então a resposta será sim ($\mathcal{A}$ aceita a linguagem infinita); se não, a resposta será não ($\mathcal{A}$ não aceita a linguagem infinita). Segue o pseudocódigo do último algoritmo sugerido.

Algoritmo 5 Algoritmo para decidir se um dado AFD $\mathcal{A}$ aceita ou não uma linguagem infinita

```
1: procedure INFINITA(A)
2:     G := G(A)
3:     i := estado inicial de A
4:     F := conjunto de estados finais de A
5:     G_0 := DFS-ACESSIVEL(G, {i})
6:     G_1 :=INVERTER(G_0)
7:     G_2 := DFS-ACESSIVEL(G_1, F)
8:     for each u ∈ V[G_2] do
9:         color[u] ←WHITE
10:    time ← 0
11:    for each u ∈ V[G_2] do
12:        if color[u] = WHITE then
13:            return(DFS-CICLICO(u))
```

```
1: procedure DFS-ACESSIVEL(G, Q')
2:     for each vertex u' ∈ V[G] do                ▷ "pinta" WHITE todos vértices
3:         color[u'] ← WHITE
4:         π[u'] ← NIL
5:     time ← 0
6:     for each q ∈ Q' do                          ▷ descobre os vértices fontes q
7:         if color[q] = WHITE then
8:             DFS-VISIT(q)
9:     for each v ∈ V[G] do
10:        if color[v] ≠ BLACK then                ▷ apaga vértices que não são BLACK
11:            V[G] := V[G] \ {v}
12:            for each w ∈ V[G] do
13:                if (v, w) ∈ E[G] then           ▷ apaga arestas (v, w)
14:                    E[G] := E[G] \ {(v, w)}
15:                if (w, v) ∈ E[G] then           ▷ apaga arestas (w, v)
16:                    E[G] := E[G] \ {(w, v)}
```

```
1: procedure DFS-CICLICO(u)
2:     back_edge := false
3:     color[u] ← GRAY                          ▷ o vértice WHITE u foi descoberto
4:     time ← time + 1
5:     d[u] ← time
6:     for each v ∈ Adj[u] do                   ▷ explora arestas (u, v)
7:         if color[v] = WHITE then
8:             π[v] ← u
9:             DFS-CICLICO(v)
10:        else
11:            if color[v] = GRAY then          ▷ identifica ciclo
12:                back_edge := true
13:                break
14:    color[u] ← BLACK                         ▷ "pinta" BLACK o vértice u
15:    time ← time + 1
16:    f[u] ← time
17:    return(back_edge)
```

```
1: procedure INVERTER(G)
2:     V[(G)^-1] := V[(G)]
3:     E[(G)^-1] := ∅
4:     (Adj)^-1 := ∅
5:     for u ∈ V[(G)^-1] do
6:         for v ∈ Adj[u] do
7:             E[(G)^-1] := E[(G)^-1] ∪ {(v, u)}
8:             (Adj[v])^-1 := (Adj[v])^-1 ∪ {u}
9:     G^-1 := (V[(G)^-1], E[(G)^-1], Adj^-1)
10:    return(G^-1)
```

O algoritmo INFINITA começa por obter o grafo do autómato $\mathcal{A}$ para então submeter ao procedimento DFS-ACESSIVEL juntamente com o conjunto singular cujo o elemento é o estado inicial. Nestas condições, o DFS-ACESSIVEL visita todos os vértices acessíveis a partir do estado inicial, pois, o DFS-ACESSIVEL é uma variante do DFS no qual os vértices fontes são elementos de um determinado subconjunto de estados/vértices, produzindo um novo grafo G_0. G_0 é submetido ao procedimento INVERTER, onde se altera o sentido das direções das arestas de G_0, produzindo o grafo G_1. G_1 juntamente com o conjunto de estados finais F são submetidos ao procedimento DFS-ACESSIVEL, onde se visita os vértices em G_1 acessíveis a partir de um dos elementos de F, produzindo o grafo G_2. Em seguida pinta-se todos os vértices de G_2 de branco (WHITE) para posteriormente percorrer as arestas de G_2 com vista a encontrar um ciclo, submetendo G_2 ao procedimento DFS-CICLICO. O procedimento DFS-CICLICO é uma variante do procedimento DFS-VISIT, pois ele funciona exatamente como o DFS-VISIT, porém, enquanto pesquisa as arestas de um grafo classifica-as como sendo ou

não, arestas para trás (*back edge*).

Análise do tempo de execução

Sejam V e E, respetivamente, vértices e arestas de $\mathcal{G}(\mathcal{A})$.

Para analisar a complexidade de tempo do algoritmo para decidir se um dado AFD $\mathcal{A}$ aceita ou não uma linguagem infinita, começaremos por ver o tempo de execução dos procedimentos DFS-ACESSIVEL(G, Q'), DFS-CICLICO(u) e INVERTER(G). O precedimento DFS-ACESSIVEL por ser uma versão restringida do algoritmo DFS, temos que $\Theta(\#Q' + \#E) \subseteq \Theta(\#V + \#E)$ e, portanto, o tempo de execução do DFS-ACESSIVEL é $\Theta(\#V + \#E)$. O tempo de execução do DFS-CICLICO, tal como o do DFS-VISIT, é $\Theta(\#E)$. No procedimento INVERTER(G) o tempo consumido nas linhas de 5 - 8, está refletido na seguinte expressão:

$$\begin{aligned}\sum_{i=1}^{\#V}\sum_{i=1}^{\#V} 2 &= \sum_{i=1}^{\#V} 2(\#V) \\ &= 2(\#V)(\#V) \\ &= 2(\#V)^2 \\ &\in \mathcal{O}((\#V)^2) \\ &\subseteq ((\#Q)^2)\end{aligned}$$

Como

$$\begin{aligned}\Theta(\#V + \#E) &\subseteq \Theta((\#Q)^2) \\ &\subseteq \mathcal{O}((\#Q)^2)\end{aligned}$$

concluímos que o tempo de execução do algoritmo INFINITA($\mathcal{A}$) é $\mathcal{O}((\#Q)^2)$.

3.3 Relacionando AFN com os algoritmos propostos

Na secção anterior estudou-se os problemas de aceitação da linguagem vazia e infinita para conjunto regular representado por um AFD. Como seria este estudo se o conjunto regular estiver representado por um AFN?

Já vimos que a partir de um AFN pode-se construir um AFD equivalente. Este processo, que é feito pelo teorema 2.2.1, sugere um algoritmo que pode receber um AFN e retornar um AFD, só que este (segundo Lewis e Papadimitriou 1990) seria exponencial. Uma vez concretizado este algoritmo, poderíamos então aplicar algoritmo VAZIA (respetivamente INFINITA) ao AFD resultante deste para decidir se este AFD aceita a linguagem vazia (respetivamente infinita) e por conseguinte, decidiríamos se o AFN equivalente aceita a linguagem vazia (respetivamente infinita), já que a linguagem aceite pelo AFD é exatamente a mesma aceite pelo AFN.

Outra alternativa seria o de submeter imediatamente o AFN aos algoritmos VAZIA e INFINITA, mas antes, tínhamos de saber se os algoritmos sugeridos pelos teoremas 3.1.2, 3.2.2 e 3.2.3 são válidos para os AFNs.

Os resultados que sugerem os algoritmos VAZIA e INFINITA podem ser reformulados para um dado AFN e a demonstração destes teoremas seria similar aos formulados para os AFDs. Por isso, os AFNs não podem ser submetidos diretamente aos algoritmos VAZIA e INFINITA, mas sim, a uma adaptação destes, tendo em conta o grafo de um AFN e os resultados reformulados.

3.4 Quando a linguagem é dada por uma expressão regular

E se o conjunto regular, em vez de estar representado por um autómato, estiver representado por uma expressão regular, como seria o processo de decisão da linguagem vazia, assim como da linguagem infinita?

Uma das formas de decidir se, dada uma expressão regular r, r representa a linguagem vazia, ou r representa a linguagem infinita, é construindo o AFD equivalente, para então ser testado nos algoritmos apresentados nas secções anteriores.

Lewis e Papadimitriou 1990, afirmam o seguinte:

(a) Existe um algoritmo polinomial que, dada uma expressão regular r, constrói um AFN equivalente;

(b) Existe um algoritmo exponencial que, dado um AFN, constrói um AFD equivalente.

E por isso, a alternativa sugerida, exigiria um algoritmo exponencial, já que, inicialmente, teríamos que construir o AFN equivalente à expressão regular dada, e a seguir, construir o AFD equivalente ao AFN.

A alternativa, mais eficiente, que adotamos tem em conta a definição indutiva das expressões regulares, definindo assim (recursivamente), os predicados (listados abaixo) que recebem uma expressão regular $r \in ER(A)$, e retornam um booleano, isto é, retornam V (verdadeiro) ou F (falso):

1. $vazia$;
2. $epsilon$;
3. $infinita$.

O leitor pode perguntar-se porquê que houve a necessidade de definir três predicados já que, só queremos decidir se $r \in ER(A)$ representa ou não a linguagem, vazia ou infinita. Esta necessidade, deve-se ao facto de precisarmos do predicado $epsilon$ na definição do predicado $infinita$, já que o fecho da linguagem $\{\epsilon\}$ nunca será uma linguagem infinita e o mesmo sucede ao fecho da linguagem $\emptyset$ e, portanto, na definição de $infinita$, também foi necessário o predicado $vazia$. Segue a definição dos respetivos predicados:

$$vazia : ER(A) \to \{V, F\}$$

$$vazia(\emptyset) = V$$

$$vazia(\epsilon) = F$$

$$vazia(a) = F$$

$$vazia(r_1 + r_2) = vazia(r_1) \wedge vazia(r_2)$$

$$vazia(r_1.r_2) = vazia(r_1) \vee vazia(r_2)$$

$$vazia(r^*) = F$$

$epsilon : ER(A) \to \{V, F\}$

$epsilon(\emptyset) = F$

$epsilon(\epsilon) = V$

$epsilon(a) = F$

$$epsilon(r_1 + r_2) = \begin{cases} epsilon(r_1) \vee epsilon(r_2) & \text{se } vazia(r_1) \vee vazia(r_2) \\ epsilon(r_1) \wedge epsilon(r_2) & \text{se não} \end{cases}$$

$epsilon(r_1.r_2) = epsilon(r_1) \wedge epsilon(r_2)$

$epsilon(r^*) = epsilon(r) \vee vazia(r)$

$infinita : ER(A) \to \{V, F\}$

$infinita(\emptyset) = F$

$infinita(\epsilon) = F$

$infinita(a) = F$

$infinita(r_1 + r_2) = infinita(r_1) \vee infinita(r_2)$

$$infinita(r_1.r_2) = \begin{cases} F & \text{se } vazia(r_1) \vee vazia(r_2) \\ infinita(r_1) \vee infinita(r_2) & \text{se não} \end{cases}$$

$$infinita(r^*) = \begin{cases} F & \text{se } vazia(r) \vee epsilon(r) \\ V & \text{se não} \end{cases}$$

Como formas de garantir a funcionalidade dos predicados propostos, enunciaremos teoremas que refletem os efeitos destes predicados, seguido de suas respetivas demonstrações:

Teorema 3.4.1. Qualquer que seja $r \in ER(A)$, $vazia(r) = V$ se, e só se, $L(r) = \emptyset$.

Demonstração.

$\Rightarrow$

Dado $r \in ER(A)$, seja $\Phi(r)$ a seguinte propriedade:

$$\text{se } vazia(r) = V \text{ então } L(r) = \emptyset$$

Demonstração por indução em r. Pelo princípio de indução estrutural associado a r basta

demonstrar:

1. $\Phi(\emptyset)$;

2. $\Phi(\epsilon)$;

3. $\Phi(a)$ para cada $a \in A$;

4. Para cada $r_1, r_2 \in ER(A)$, se $\Phi(r_1)$ e $\Phi(r_2)$ então $\Phi(r_1 + r_2), \Phi(r_1.r_2)$;

5. Para cada $r \in ER(A)$, se $\Phi(r)$ então $\Phi(r^*)$.

Dem. 1. $\Phi(\emptyset)$ sse, se $vazia(\emptyset) = V$ então $L(\emptyset) = \emptyset$.
Trivial pela definição da função L;

Dem. 2. $\Phi(\epsilon)$ sse, se $vazia(\epsilon) = V$ então $L(\epsilon) = \{\epsilon\}$.
Mas, pela definição de $vazia$, temos que $vazia(\epsilon) = F$ logo, $\Phi(\epsilon)$ é verdadeiro;

Dem. 3. $\Phi(a)$ sse, se $vazia(a) = V$ então $L(a) = \{a\}$.
Mas, pela definição de $vazia$, temos que $vazia(a) = F$ logo, $\Phi(a)$ é verdadeiro;

Dem. 4. Suponhamos $\Phi(r_1)$ e $\Phi(r_2)$.
Queremos mostrar:

(a) $\Phi(r_1 + r_2)$ ou seja, se $vazia(r_1 + r_2) = V$ então $L(r_1 + r_2) = \emptyset$;

(b) $\Phi(r_1.r_2)$ ou seja, se $vazia(r_1.r_2) = V$ então $L(r_1.r_2) = \emptyset$;

Dem. (a) Suponhamos que $vazia(r_1 + r_2) = V$.
Falta ver que $L(r_1 + r_2) = \emptyset$. Mas $L(r_1 + r_2) = L(r_1) \cup L(r_2)$. Por isso, falta ver que $L(r_1) = L(r_2) = \emptyset$.
Como $vazia(r_1 + r_2) = vazia(r_1) \wedge vazia(r_2)$, segue de $vazia(r_1 + r_2) = V$ que $vazia(r_1) = vazia(r_2) = V$.
De $vazia(r_1) = V$ e $\Phi(r_1)$ segue que $L(r_1) = \emptyset$.
De $vazia(r_2) = V$ e $\Phi(r_2)$ segue que $L(r_2) = \emptyset$.

Dem. (b) Suponhamos que $vazia(r_1.r_2) = V$.
Falta ver que $L(r_1.r_2) = V$.
Mas $L(r_1.r_2) = L(r_1).L(r_2)$. Por isso, falta ver que $L(r_1) = \emptyset$ ou $L(r_2) =$

$\emptyset$.

Como $vazia(r_1.r_2) = vazia(r_1) \vee vazia(r_2)$, segue de $vazia(r_1.r_2) = V$ que $vazia(r_1) = V$ ou $vazia(r_2) = V$.

1º caso $vazia(r_1) = V$: De $vazia(r_1) = V$ e $\Phi(r_1)$ segue que $L(r_1) = \emptyset$;

2º caso $vazia(r_2) = V$: De $vazia(r_2) = V$ e $\Phi(r_2)$ segue que $L(r_2) = \emptyset$.

Dem. 5. Suponhamos $\Phi(r)$.

Queremos mostrar $\Phi(r^*)$, ou seja, se $vazia(r^*) = V$ então $L(r^*) = \emptyset$.

Mas, por definição $vazia(r^*) = F$, logo, $\Phi(r^*)$ é verdadeiro.

$\Leftarrow$

Dado $r \in ER(A)$, seja $\Phi(r)$ a seguinte propriedade:

$$\text{se } L(r) = \emptyset \text{ então } vazia(r) = V$$

Demonstração por indução em r. Pelo princípio de indução estrutural associado a r basta demonstrar:

1. $\Phi(\emptyset)$;

2. $\Phi(\epsilon)$;

3. $\Phi(a)$ para cada $a \in A$;

4. Para cada $r_1, r_2 \in ER(A)$, se $\Phi(r_1)$ e $\Phi(r_2)$ então $\Phi(r_1 + r_2), \Phi(r_1.r_2)$;

5. Para cada $r \in ER(A)$, se $\Phi(r)$ então $\Phi(r^*)$.

Dem. 1. $\Phi(\emptyset)$ sse, se $L(\emptyset) = \emptyset$ então $vazia(\emptyset) = V$.

Trivial, pela definição de $vazia$.

Dem. 2. $\Phi(\epsilon)$ sse, se $L(\epsilon) = \emptyset$ então $vazia(\epsilon) = V$.

Mas, $L(\epsilon) = \{\epsilon\}$, logo, $\Phi(\epsilon)$ é verdadeiro.

Dem. 3. $\Phi(a)$ sse, se $L(a) = \emptyset$ então $vazia(a) = V$.

Mas, $L(a) = \{a\}$, logo, $\Phi(a)$ é verdadeiro.

Dem. 4. Suponhamos $\Phi(r_1)$ e $\Phi(r_2)$.

Queremos mostrar:

(a) $\Phi(r_1 + r_2)$ ou seja, se $L(r_1 + r_2) = \emptyset$ então $vazia(r_1 + r_2) = V$;

(b) $\Phi(r_1.r_2)$ ou seja, se $L(r_1.r_2) = \emptyset$ então $vazia(r_1.r_2) = V$;

Dem. (a) Suponhamos que $L(r_1 + r_2) = \emptyset$

Falta ver que $vazia(r_1 + r_2) = V$.

Mas, como $vazia(r_1+r_2) = vazia(r_1) \wedge vazia(r_2)$, falta ver que $vazia(r_1) = vazia(r_2) = V$.

Como $L(r_1 + r_2) = L(r_1) \cup L(r_2)$ de $L(r_1 + r_2) = \emptyset$ segue que $L(r_1) = L(r_2) = \emptyset$.

De $L(r_1) = \emptyset$ e $\Phi(r_1)$ segue que $vazia(r_1) = V$.

De $L(r_2) = \emptyset$ e $\Phi(r_2)$ segue que $vazia(r_2) = V$.

Dem. (b) Suponhamos que $L(r_1.r_2) = \emptyset$.

Falta ver que $vazia(r_1.r_2) = V$.

Mas, $vazia(r_1.r_2) = vazia(r_1) \vee vazia(r_2)$. Por isso falta ver que $vazia(r_1) = V$ ou $vazia(r_2) = V$.

Como $L(r_1.r_2) = L(r_1).L(r_2)$ de $L(r_1.r_2) = \emptyset$ segue que $L(r_1) = \emptyset$ ou $L(r_2) = \emptyset$.

1º caso $L(r_1) = \emptyset$: De $L(r_1) = \emptyset$ e $\Phi(r_1)$ segue que $vazia(r_1) = V$.

2º caso $L(r_2) = \emptyset$: De $L(r_2) = \emptyset$ e $\Phi(r_2)$ segue que $vazia(r_2) = V$.

Dem. 5. Suponhamos $\Phi(r)$.

Queremos mostrar $\Phi(r^*)$, ou seja, se $L(r^*) = \emptyset$ então $vazia(r^*) = V$.

Mas $L(r^*) = (L(r))^* \neq \emptyset$ para cada $r \in ER(A)$. Portanto $\Phi(r^*)$ é verdadeiro.

□

Teorema 3.4.2. Qualquer que seja $r \in ER(A)$, $epsilon(r) = V$ se, e só se, $L(r) = \{\epsilon\}$.

Demonstração.

$\Rightarrow$

Dado $r \in ER(A)$, seja $\Phi(r)$ a seguinte propriedade:

$$\text{se } epsilon(r) = V \text{ então } L(r) = \{\epsilon\}$$

Demonstração por indução em r. Pelo princípio de indução estrutural associado a r basta demonstrar:

1. $\Phi(\emptyset)$;

2. $\Phi(\epsilon)$;

3. $\Phi(a)$ para cada $a \in A$;

4. Para cada $r_1, r_2 \in ER(A)$, se $\Phi(r_1)$ e $\Phi(r_2)$ então $\Phi(r_1 + r_2), \Phi(r_1.r_2)$;

5. Para cada $r \in ER(A)$, se $\Phi(r)$ então $\Phi(r^*)$.

Dem. 1. $\Phi(\emptyset)$ sse, se $epsilon(\emptyset) = V$ então $L(\emptyset) = \{\epsilon\}$.
Trivial, já que $epsilon(\emptyset) = F$, portanto, $\Phi(\emptyset)$ é verdadeiro.

Dem. 2. $\Phi(\epsilon)$ sse, se $epsilon(\epsilon) = V$ então $L(\epsilon) = \{\epsilon\}$.
Trivial pela definição da função L.

Dem. 3. $\Phi(a)$ sse, se $epsilon(a) = V$ então $L(a) = \{\epsilon\}$.
Trivial, já que $epsilon(a) = F$, portanto, $\Phi(a)$ é verdadeiro.

Dem. 4. Suponhamos $\Phi(r_1)$ e $\Phi(r_2)$.
Queremos mostrar:

(a) $\Phi(r_1 + r_2)$, ou seja, se $epsilon(r_1 + r_2) = V$ então $L(r_1 + r_2) = \{\epsilon\}$.

(b) $\Phi(r_1.r_2)$, ou seja, se $epsilon(r_1.r_2) = V$ então $L(r_1.r_2) = \{\epsilon\}$.

Dem. (a) Suponhamos que $epsilon(r_1 + r_2) = V$.
Falta ver que $L(r_1 + r_2) = \{\epsilon\}$.

1º caso $vazia(r_1) = V$:
Pelo teorema 3.4.1, $L(r_1) = \emptyset$. Tem-se

$$V = epsilon(r_1 + r_2) \qquad \text{(por hipótese)}$$
$$= epsilon(r_1) \vee epsilon(r_2) \qquad (*)$$

$(*)$ pela definição de $epsilon$ e por $vazia(r_1) = V$.
Então $epsilon(r_1) = V$ ou $epsilon(r_2) = V$.
Mas $epsilon(r_1) \neq V$ (caso contrário, por $\Phi(r_1)$ seguiria $L(r_1) = \{\epsilon\}$,

o que é absurdo pois $L(r_1) = \emptyset$).

Então $epsilon(r_2) = V$. Por $\Phi(r_2)$ segue que $L(r_2) = \{\epsilon\}$.

Então

$$\begin{aligned} L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ &= \emptyset \cup \{\epsilon\} \\ &= \{\epsilon\}. \end{aligned}$$

2º caso $vazia(r_2) = V$:

Pelo teorema 3.4.1, $L(r_2) = \emptyset$. Tem-se

$$\begin{aligned} V &= epsilon(r_1 + r_2) && \text{(por hipótese)} \\ &= epsilon(r_1) \vee epsilon(r_2) && (*) \end{aligned}$$

$(*)$ pela definição de $epsilon$ e por $vazia(r_2) = V$.

Então $epsilon(r_1) = V$ ou $epsilon(r_2) = V$.

Mas $epsilon(r_2) \neq V$ (caso contrário, por $\Phi(r_2)$ seguiria $L(r_2) = \{\epsilon\}$, o que é absurdo pois $L(r_2) = \emptyset$).

Então $epsilon(r_1) = V$. Por $\Phi(r_1)$ segue que $L(r_1) = \{\epsilon\}$.

Então

$$\begin{aligned} L(r_1 + r_2) &= L(r_1) \cup L(r_2) \\ &= \{\epsilon\} \cup \emptyset \\ &= \{\epsilon\}. \end{aligned}$$

3º caso $vazia(r_1) = vazia(r_2) = F$:

Pelo teorema 3.4.1, $L(r_1) \neq \emptyset$ e $L(r_2) \neq \emptyset$. Tem-se

$$\begin{aligned} V &= epsilon(r_1 + r_2) && \text{(por hipótese)} \\ &= epsilon(r_1) \wedge epsilon(r_2) && (*) \end{aligned}$$

$(*)$ pela definição de $epsilon$ e porque $vazia(r_1) = vazia(r_2) = F$.

Então $epsilon(r_1) = V$ e $epsilon(r_2) = V$. Por $\Phi(r_1)$ e $\Phi(r_2)$, segue que $L(r_1) = \{\epsilon\}$ e $L(r_2) = \{\epsilon\}$.

Então

$$L(r_1 + r_2) = L(r_1) \cup L(r_2) = \{\epsilon\} \cup \{\epsilon\} = \{\epsilon\}.$$

Dem. (b) Suponhamos que $epsilon(r_1.r_2) = V$.

Falta ver que $L(r_1.r_2) = \{\epsilon\}$.

Mas, $L(r_1.r_2) = L(r_1).L(r_2)$, por isso basta ver que $L(r_1) = L(r_2) = \{\epsilon\}$.

Como $epsilon(r_1.r_2) = epsilon(r_1) \wedge epsilon(r_2)$ de $epsilon(r_1.r_2) = V$ segue que $epsilon(r_1) = epsilon(r_2) = V$.

De $epsilon(r_1) = V$ e $\Phi(r_1)$ segue que $L(r_1) = \{\epsilon\}$.

De $epsilon(r_2) = V$ e $\Phi(r_2)$ segue que $L(r_2) = \{\epsilon\}$.

Dem. 5. Suponhamos $\Phi(r)$.

Queremos mostrar $\Phi(r^*)$, ou seja, se $epsilon(r^*) = V$ então $L(r^*) = \{\epsilon\}$.

Suponhamos que $epsilon(r^*) = V$.

Falta ver que $L(r^*) = \{\epsilon\}$.

Como $epsilon(r^*) = epsilon(r) \vee vazia(r)$, de $epsilon(r^*) = V$ segue que $epsilon(r) = V$ ou $vazia(r) = V$.

1º caso $epsilon(r) = V$: De $epsilon(r) = V$ e $\Phi(r)$ segue que $L(r) = \{\epsilon\}$. Então:

$$\begin{aligned} L(r^*) &= (L(r))^* && \text{(por definição de } L\text{)} \\ &= \{\epsilon\}^* && (L(r) = \{\epsilon\}) \\ &= \{\epsilon\} \end{aligned}$$

2º caso $vazia(r) = V$: De $vazia(r) = V$ e pelo teorema 3.4.1 segue que $L(r) = \emptyset$. Então:

$$\begin{aligned} L(r^*) &= (L(r))^* && \text{(por definição de } L\text{)} \\ &= \emptyset^* && (L(r) = \emptyset) \\ &= \{\epsilon\} \end{aligned}$$

$\Leftarrow$

Dado $r \in ER(A)$, seja $\Phi(r)$ a seguinte propriedade:

$$\text{se } L(r) = \{\epsilon\} \text{ então } epsilon(r) = V$$

Demonstração por indução em r. Pelo princípio de indução estrutural associado a r basta demonstrar:

1. $\Phi(\emptyset)$;

2. $\Phi(\epsilon)$;

3. $\Phi(a)$ para cada $a \in A$;

4. Para cada $r_1, r_2 \in ER(A)$, se $\Phi(r_1)$ e $\Phi(r_2)$ então $\Phi(r_1 + r_2), \Phi(r_1.r_2)$;

5. Para cada $r \in ER(A)$, se $\Phi(r)$ então $\Phi(r^*)$.

Dem. 1. $\Phi(\emptyset)$ sse, se $L(\emptyset) = \{\epsilon\}$ então $epsilon(\emptyset) = V$.
Trivial, já que $L(\emptyset) = \emptyset$, portanto, $\Phi(\emptyset)$ é verdadeiro.

Dem. 2. $\Phi(\epsilon)$ sse, se $L(\epsilon) = \{\epsilon\}$ então $epsilon(\epsilon) = V$.
Trivial pela definição do predicado $epsilon$.

Dem. 3. $\Phi(a)$ sse, se $L(a) = \{\epsilon\}$ então $epsilon(a) = V$.
Trivial, já que $L(a) = \{a\}$, portanto, $\Phi(a)$ é verdadeiro.

Dem. 4. Suponhamos $\Phi(r_1)$ e $\Phi(r_2)$.
Queremos mostrar:

(a) $\Phi(r_1 + r_2)$, ou seja, se $L(r_1 + r_2) = \{\epsilon\}$ então $epsilon(r_1 + r_2) = V$.

(b) $\Phi(r_1.r_2)$, ou seja, se $L(r_1.r_2) = \{\epsilon\}$ então $epsilon(r_1.r_2) = V$.

Dem. (a) Suponhamos que $L(r_1 + r_2) = \{\epsilon\}$.
Falta ver que $epsilon(r_1 + r_2) = V$.

1º caso $vazia(r_1) = V$:
Pelo teorema 3.4.1, $L(r_1) = \emptyset$.
Então

$$\begin{aligned} \{\epsilon\} &= L(r_1 + r_2) && \text{(por hipótese)} \\ &= L(r_1) \cup L(r_2) && \text{(por definição de } L) \\ &= L(r_2) && \text{(porque } L(r_1) = \emptyset) \end{aligned}$$

Logo $L(r_2) = \{\epsilon\}$. Por $\Phi(r_2)$, $epsilon(r_2) = V$.
Então

$$\begin{aligned} epsilon(r_1 + r_2) &= epsilon(r_1) \vee epsilon(r_2) && (*) \\ &= V && (**) \end{aligned}$$

($*$) por definição de $epsilon$ e porque $vazia(r_1) = V$.

($**$) porque $epsilon(r_2) = V$.

2º caso $vazia(r_2) = V$:

Pelo teorema 3.4.1, $L(r_2) = \emptyset$.

Então

$$\begin{aligned} \{\epsilon\} &= L(r_1 + r_2) && \text{(por hipótese)} \\ &= L(r_1) \cup L(r_2) && \text{(por definição de } L) \\ &= L(r_1) && \text{(porque } L(r_2) = \emptyset) \end{aligned}$$

Logo $L(r_1) = \{\epsilon\}$. Por $\Phi(r_1)$, $epsilon(r_1) = V$.

Então

$$\begin{aligned} epsilon(r_1 + r_2) &= epsilon(r_1) \vee epsilon(r_2) && (*) \\ &= V && (**) \end{aligned}$$

($*$) por definição de $epsilon$ e porque $vazia(r_2) = V$.

($**$) porque $epsilon(r_1) = V$.

3º caso $vazia(r_1) = vazia(r_2) = F$:

Pelo teorema 3.4.1, $L(r_1) \neq \emptyset$ e $L(r_2) \neq \emptyset$. Mas, de novo, $\{\epsilon\} = (r_1 + r_2) = L(r_1) \cup L(r_2)$, donde $L(r_1) = L(r_2) = \{\epsilon\}$. Por $\Phi(r_1)$ e $\Phi(r_2)$ segue que $epsilon(r_1) = V$ e $epsilon(r_2) = V$.

Então

$$\begin{aligned} epsilon(r_1 + r_2) &= epsilon(r_1) \wedge epsilon(r_2) && (*) \\ &= V \quad \text{(porque } epsilon(r_1) = epsilon(r_2) = V) \end{aligned}$$

($*$) por definição de $epsilon$ e porque $vazia(r_1) = vazia(r_2) = F$.

Dem. (b) Suponhamos que $L(r_1.r_2) = \{\epsilon\}$.

Falta ver que $epsilon(r_1.r_2) = V$.

Mas, $epsilon(r_1.r_2) = epsilon(r_1) \wedge epsilon(r_2)$ por isso falta ver que $epsilon(r_1) = epsilon(r_2) = V$.

Como $L(r_1.r_2) = L(r_1).L(r_2)$ de $L(r_1.r_2) = \{\epsilon\}$ segue que $L(r_1) = L(r_2) = \{\epsilon\}$.

De $L(r_1) = \{\epsilon\}$ e $\Phi(r_1)$ segue que $epsilon(r_1) = V$.

De $L(r_2) = \{\epsilon\}$ e $\Phi(r_2)$ segue que $epsilon(r_2) = V$.

Dem. 5. Suponhamos $\Phi(r)$.

Queremos mostrar $\Phi(r^*)$, ou seja, se $L(r^*) = \{\epsilon\}$ então $epsilon(r^*) = V$.

Suponhamos $L(r^*) = \{\epsilon\}$.

Falta ver que $epsilon(r^*) = V$.

Mas, $epsilon(r^*) = epsilon(r) \vee vazia(r)$, por isso, basta ver que $epsilon(r) = V$ ou $vazia(r) = V$.

Como $L(r^*) = (L(r))^*$ de $L(r^*) = \{\epsilon\}$ segue que $L(r) = \{\epsilon\}$ ou $L(r) = \emptyset$.

1º caso $L(r) = \{\epsilon\}$: De $L(r) = \{\epsilon\}$ e $\Phi(r)$ segue que $epsilon(r) = V$;

2º caso $L(r) = \emptyset$: De $L(r) = \emptyset$ e pelo teorema 3.4.1 segue que $vazia(r) = V$.

□

Teorema 3.4.3. Qualquer que seja $r \in ER(A)$, $infinita(r) = V$ se, e só se, $L(r)$ é infinita.

Demonstração.

$\Rightarrow$

Dado $r \in ER(A)$, seja $\Phi(r)$ a seguinte propriedade:

$$\text{se } infinita(r) = V \text{ então } L(r) \text{ é infinita}$$

Demonstração por indução em r. Pelo princípio de indução estrutural associado a r basta demonstrar:

1. $\Phi(\emptyset)$;

2. $\Phi(\epsilon)$;

3. $\Phi(a)$ para cada $a \in A$;

4. Para cada $r_1, r_2 \in ER(A)$, se $\Phi(r_1)$ e $\Phi(r_2)$ então $\Phi(r_1 + r_2), \Phi(r_1.r_2)$;

5. Para cada $r \in ER(A)$, se $\Phi(r)$ então $\Phi(r^*)$.

Dem. 1. $\Phi(\emptyset)$ sse, se $infinita(\emptyset) = V$ então $L(\emptyset)$ é infinita.

Trivial, já que $infinita(\emptyset) = F$, portanto, $\Phi(\emptyset)$ é verdadeiro.

Dem. 2. $\Phi(\epsilon)$ sse, se $infinita(\epsilon) = V$ então $L(\epsilon)$ é infinita.

Trivial, já que $infinita(\epsilon) = F$, portanto, $\Phi(\epsilon)$ é verdadeiro.

Dem. 3. $\Phi(a)$ sse, se $infinita(a) = V$ então $L(a)$ é infinita.

Trivial, já que $infinita(a) = F$, portanto, $\Phi(a)$ é verdadeiro.

Dem. 4. Suponhamos $\Phi(r_1)$ e $\Phi(r_2)$.

Queremos mostrar:

(a) $\Phi(r_1 + r_2)$, ou seja, se $infinita(r_1 + r_2) = V$ então $L(r_1 + r_2)$ é infinita.

(b) $\Phi(r_1.r_2)$, ou seja, se $infinita(r_1.r_2) = V$ então $L(r_1.r_2)$ é infinita.

Dem. (a) Suponhamos $infinita(r_1 + r_2) = V$.

Falta ver que $L(r_1 + r_2)$ é infinita.

Mas, $L(r_1 + r_2) = L(r_1) \cup L(r_2)$. Por isso, falta ver que $L(r_1)$ é infinita ou $L(r_2)$ é infinita.

Como $infinita(r_1 + r_2) = infinita(r_1) \vee infinita(r_2)$ de $infinita(r_1 + r_2) = V$ segue que $infinita(r_1) = V$ ou $infinita(r_2) = V$.

1º caso $infinita(r_1) = V$: de $infinita(r_1) = V$ e $\Phi(r_1)$ segue que $L(r_1)$ é infinita.

2º caso $infinita(r_2) = V$: de $infinita(r_2) = V$ e $\Phi(r_2)$ segue que $L(r_2)$ é infinita.

Dem. (b) Suponhamos $infinita(r_1.r_2) = V$.

Falta ver que $L(r_1.r_2)$ é infinita.

Mas, $L(r_1.r_2) = L(r_1).L(r_2)$. Por isso, falta ver que $L(r_1)$ é infinita e $L(r_2) \neq \emptyset$ ou $L(r_2)$ é infinita e $L(r_1) \neq \emptyset$.

Da definição de $infinita$ e de $infinita(r_1.r_2) = V$ segue que $vazia(r_1) = vazia(r_2) = F$ e ($infinita(r_1) = V$ ou $infinita(r_2) = V$).

1º caso $infinita(r_1) = V$ e $vazia(r_2) = F$: De $vazia(r_2) = F$ e pelo teorema 3.4.1 segue que $L(r_2) \neq \emptyset$. De $infinita(r_1) = V$ e $\Phi(r_1)$ segue que $L(r_1)$ é infinita.

2º caso $infinita(r_2) = V$ e $vazia(r_1) = F$: De $vazia(r_1) = F$ e pelo teorema 3.4.1 segue que $L(r_1) \neq \emptyset$. De $infinita(r_2) = V$ e $\Phi(r_2)$ segue que $L(r_2)$ é infinita.

Dem. 5. Suponhamos $\Phi(r)$.

Queremos mostrar $\Phi(r^*)$, ou seja, se $infinita(r^*) = V$ então $L(r^*)$ é infinita.

Suponhamos $infinita(r^*) = V$.

Falta ver que $L(r^*)$ é infinita. Mas, $L(r^*) = (L(r))^*$, por isso falta ver que $(L(r))^*$ é infinita.

De $infinita(r^*) = V$ segue que $epsilon(r) = vazia(r) = F$.

De $vazia(r) = F$ e pelo teorema 3.4.1, segue que $L(r) \neq \emptyset$.

De $epsilon(r) = F$ e pelo teorema 3.4.2 segue que $L(r) \neq \{\epsilon\}$.

Logo $(L(r))^*$ é infinita, porque contém uma infinidade de palavras u, u^2, u^3, ...

$\Leftarrow$

Dado $r \in ER(A)$, seja $\Phi(r)$ a seguinte propriedade:

$$\text{se } L(r) \text{ é infinita então } infinita(r) = V$$

Demonstração por indução em r. Pelo princípio de indução estrutural associado a r basta demonstrar:

1. $\Phi(\emptyset)$;

2. $\Phi(\epsilon)$;

3. $\Phi(a)$ para cada $a \in A$;

4. Para cada $r_1, r_2 \in ER(A)$, se $\Phi(r_1)$ e $\Phi(r_2)$ então $\Phi(r_1 + r_2), \Phi(r_1.r_2)$;

5. Para cada $r \in ER(A)$, se $\Phi(r)$ então $\Phi(r^*)$.

Dem. 1. $\Phi(\emptyset)$ sse, se $L(\emptyset)$ é infinita então $infinita(\emptyset) = V$.

Trivial, já que $L(\emptyset) = \emptyset$, e $L(\emptyset)$ não é infinito.

Dem. 2. $\Phi(\epsilon)$ sse, se $L(\epsilon)$ é infinita então $infinita(\epsilon) = V$.

Trivial, já que $L(\epsilon) = \{\epsilon\}$, e $L(\epsilon)$ não é infinito.

Dem. 3. $\Phi(a)$ sse, se $L(a)$ é infinita então $infinita(a) = V$.

Trivial, já que $L(a) = \{a\}$, e $L(a)$ não é infinito.

Dem. 4. Suponhamos $\Phi(r_1)$ e $\Phi(r_2)$.

Queremos mostrar:

(a) $\Phi(r_1 + r_2)$, ou seja, se $L(r_1 + r_2)$ é infinita então $infinita(r_1 + r_2) = V$.

(b) $\Phi(r_1.r_2)$, ou seja, se $L(r_1.r_2)$ é infinita então $infinita(r_1.r_2) = V$.

Dem. (a) Suponhamos que $L(r_1 + r_2)$ é infinita.

Falta ver que $infinita(r_1 + r_2) = V$.

Mas $infinita(r_1 + r_2) = infinita(r_1) \vee infinita(r_2)$, por isso, falta ver que $infinita(r_1) = V$ ou $infinita(r_2) = V$.

Como $L(r_1 + r_2) = L(r_1) \cup L(r_2)$, de $L(r_1 + r_2)$ ser infinita segue que $L(r_1)$ é infinita ou $L(r_2)$ é infinita.

1º caso $L(r_1)$ infinita: Por $L(r_1)$ ser infinita e de $\Phi(r_1)$ segue que $infinita(r_1) = V$.

2º caso $L(r_2)$ infinita: Por $L(r_2)$ ser infinita e de $\Phi(r_2)$ segue que $infinita(r_2) = V$.

Dem. (b) Suponhamos que $L(r_1.r_2)$ é infinita.

Falta ver que $infinita(r_1.r_2) = V$, isto é, $vazia(r_2) = vazia(r_1) = F$ e ($infinita(r_1) = V$ ou $infinita(r_2) = V$) .

Como $L(r_1.r_2) = L(r_1).L(r_2)$, de $L(r_1.r_2)$ ser infinita segue que $L(r_1)$ é infinita e $L(r_2) \neq \emptyset$, ou $L(r_2)$ é infinita e $L(r_1) \neq \emptyset$.

1º caso $L(r_1)$ infinita e $L(r_2) \neq \emptyset$: De $L(r_2) \neq \emptyset$ e pelo teorema 3.4.1 segue que $vazia(r_2) = F$. Por $vazia(r_2) = F$, $L(r_1)$ ser infinita, e $\Phi(r_1)$ segue que $infinita(r_1) = V$.

2º caso $L(r_2)$ infinita e $L(r_1) \neq \emptyset$: De $L(r_1) \neq \emptyset$ e pelo teorema 3.4.1 segue que $vazia(r_1) = F$. Por $vazia(r_1) = F$, $L(r_2)$ ser infinita, e $\Phi(r_2)$ segue que $infinita(r_2) = V$.

Dem. 5. Suponhamos $\Phi(r)$.

Queremos mostrar $\Phi(r^*)$, ou seja, se $L(r^*)$ é infinita então $infinita(r^*) = V$.

Suponhamos que $L(r^*)$ é infinita.

Falta ver que $infinita(r^*) = V$.

Como $L(r^*) = (L(r))^*$, por $L(r^*)$ ser infinita segue que $L(r) \neq \emptyset$ e $L(r) \neq \{\epsilon\}$.

De $L(r) \neq \emptyset$ e pelo teorema 3.4.1 segue que $vazia(r) = F$.

De $L(r) \neq \{\epsilon\}$ e pelo teorema 3.4.2 segue que $epsilon(r) = F$.

De $epsilon(r) = vazia(r) = F$ e pela definição do predicado $infinita$ segue que $infinita(r^*) = V$.

□

Análise do tempo de execução

Seja $|r|$ o número de símbolos em uma expressão regular r. Uma expressão regular r, cujo $|r|$ é maior que um, pode ter as seguintes formas: (1) $r = (r_1)^*$; (2) $r = (r_1 + r_2)$ ou $r = (r_1.r_2)$ onde $r_1, r_2 \in ER(A)$.

No primeiro caso $|r_1| = |r| - 3$ e para este caso definimos a função $T : \mathbb{N}_0 \to \mathbb{R}$ do seguinte modo:

$$\begin{cases} T(0) = c & \text{onde } c \text{ é constante} \\ T(n+1) = T(n) + d & \text{onde } d \text{ é constante} \end{cases} \tag{3.4.1}$$

Seja $h(m) = m + d$. Prova-se que a função T definida em (3.4.1) satisfaz $T(n) = h^n(c)$. Assim

$$\begin{aligned} T(0) &= h^0(c) = c \\ T(1) &= h^1(c) = h(c) = c + d \\ T(2) &= h^2(c) = h(h(c)) = h(T(1)) = (c + d) + d = c + 2d \\ T(3) &= h^3(c) = h(T(2)) = (c + 2d) + d = c + 3d \\ &\ldots \\ T(n) &= c + nd \in \mathcal{O}(n) \end{aligned}$$

No segundo caso vamos assumir que r é "equilibrada" no sentido em que satisfaz $|r_1| < |r|/2$ e $|r_2| < |r|/2$ e para este caso a função $T : \mathbb{N}_0 \to \mathbb{R}$ tem a seguinte definição :

$$T(n) = 2T(\lceil n/2 \rceil) + d, \text{ onde } d \text{ é uma constante} \tag{3.4.2}$$

De acordo com a alínea 1 do teorema 4.1 (*Master theorem*) de Cormen et al. 2001[1], podemos concluir que $T(n) \in \Theta(n)$.

Como $\Theta(n) \subseteq \mathcal{O}(n)$ concluímos que o tempo de execução para decidir se, dada uma expressão regular r, r representa a linguagem vazia, ou r representa a linguagem infinita é $\mathcal{O}(n)$.

[1] Na alínea 1 do referido teorema, temos de escolher $a = b = 2$ e $\epsilon = 1$ e $f(n) = d$.

Lema 3.4.1. Para dado $r \in ER(A)$ existe $r' \in ER(A)$ tal que: $L(r') = L(r)$ e $|r'| \leq 2|r|$ e r' é equilibrada.

Demonstração. Ideia: equilibrar os dois ramos de $r_1 \bigcirc r_2$, onde $\bigcirc \in \{+, \cdot\}$, aplicando a transformação $s \mapsto s + \emptyset$. □

Teorema 3.4.4. Qualquer que seja $r \in ER(A)$, $T(|r|) \in \mathcal{O}(n)$.

Demonstração. Seja $n = |r|$, r' dado pelo lema 3.4.1 e $n' = |r'|$.

$$\begin{aligned} T(n) &\leq T(n') && (n \leq n' \text{ e } T \text{ monótona}) \\ &\in \mathcal{O}(n') && (\text{análise acima e } r' \text{ equilibrada}) \\ &\subseteq \mathcal{O}(2n) && (n' \leq 2n) \\ &= \mathcal{O}(n) \end{aligned}$$

□

Capítulo 4

Conclusão

Uma das questões mais abordadas em ciências da computação tem a ver com os problemas que podem ser resolvidos mecanicamente por processos algorítmicos. E, sendo que alguns dos problemas de decisão têm esta propriedade, constituem matéria de estudo em muitos temas da ciência da computação e, em particular, na teoria de linguagens regulares e consequentemente em autómatos finitos. Por conseguinte, torna-se notória a relevância do estudo dos problemas de decisão na teoria das linguagens regulares. Sendo assim, esta dissertação é mais um contributo para esta área de investigação pois fornece mais matéria de estudo para eventuais trabalhos futuros.

O principal objetivo desta investigação era o de apresentar algoritmos de decisão para os problemas da linguagem vazia e da linguagem infinita e, pelo tratado no capítulo três, afirmamos que o objetivo foi alcançado.

No princípio da execução deste trabalho, existiam algumas interrogações de como se poderiam alcançar os objetivos traçados. Assim, esta investigação constitui um instrumento através do qual é possível desenvolver competências, como por exemplo, o pensamento crítico e abstrato para criar algoritmos que resolvam certos problemas, principalmente os ligados à teoria de grafos. Também, com esta investigação, foi possível compreender que a correção de algoritmos depende de certos resultados matemáticos.

Em síntese, com o presente trabalho,

- foi possível constatar a importância da teoria de grafos, assim como os algoritmos que existem para pesquisar um grafo, pois os seus fundamentos tiveram grande utilidade na

execução deste trabalho;

- conclui-se que é mais eficiente decidir se uma determinada expressão regular representa a linguagem vazia, ou representa uma linguagem infinita (fazendo uso dos predicados propostos), do que decidir se um dado AFD aceita a linguagem vazia, ou aceita uma linguagem infinita, pelo facto de que num caso o tempo ser $\mathcal{O}(n)$ e no outro $\mathcal{O}(n^2)$.

Bibliografia

Coelho, Francisco e João Pedro Neto (2010). *Teoria da Computação. Computabilidade e Complexidade.* Escolar Editora.

Cormen, Thomas H. et al. (2001). *Introduction to Algorithms, 2nd ed.* The Massachusetts Institute of Technology.

Costa, José Carlos (2004). *Autómatos e Máquinas de Turing*. Departamento de Matemática da Universidade do Minho.

Hopcroft, John E., Rajeev Motwani e Jeffrey D. Ullman (2006). *Introduction to Automata theory Languages, and Computation, 3rd Edition*. Pearson.

Hopcroft, John E. e Jeffrey D. Ullman (1979). *Introduction to Automata theory, Languages and Computation*. Cambridge University Press.

Kozen, Dexter C. (1997). *Automata and computablity*. Springer.

Lewis, Harry R. e Christos H. Papadimitriou (1990). *Elements of the theory of computation. Second edition*. Alan Apt.

Sipser, Michael (2012). *Introduction to the Theory of Computation, Third Edition*. Cengage Learning.

Sudkamp, Thomas A. (1997). *Languages and machines: an introduction to the theory of computer, Second Edition*. Addison Wesley Longmam, Inc.