

**José João Gonçalves Morais**

**Obtenção de expressões regulares  
pequenas a partir de autómatos  
finitos**



Departamento de Ciéncia de Computadores  
Faculdade de Ciéncias da Universidade do Porto  
**2004**

José João Gonçalves Morais

# Obtenção de expressões regulares pequenas a partir de autómatos finitos



*Tese submetida à Faculdade de Ciências da  
Universidade do Porto para obtenção do grau de Mestre  
em Informática, ramo de Ciência de Computadores*

Departamento de Ciência de Computadores  
Faculdade de Ciências da Universidade do Porto  
2004

**À minha mãe**

# Resumo

Há dois algoritmos *standard* para converter um autómato finito numa expressão regular equivalente, que são o algoritmo de Kleene e o algoritmo da eliminação de estados, que consiste em eliminar em cada passo um estado e actualizar as transições do autómato de forma a que a linguagem reconhecida não se altere. O primeiro algoritmo produz, regra geral, expressões de tamanho superior às produzidas pelo segundo. Na especificação do segundo algoritmo nada é dito quanto ao estado que deve ser eliminado, a não ser que deve ser diferente de dois estados especiais que não são removíveis em passo algum do algoritmo. A ordem em que os estados são removidos determina o tamanho da expressão obtida e assim, numa aplicação deste algoritmo sem uma preocupação com a escolha do estado a ser removido, podem produzir-se expressões de tamanho muito superior ao de uma expressão de tamanho mínimo obtinível por este algoritmo para o autómato em causa.

O problema que nos propomos tratar neste trabalho é o de obter, a partir de um autómato e baseados no algoritmo da eliminação de estados, uma expressão regular equivalente com o menor tamanho. A solução por *força bruta* para este problema, que consiste em aplicar o algoritmo em todas as ordens de remoção possíveis e escolher de entre as expressões obtidas uma de menor tamanho, tem complexidade  $\mathcal{O}(n!)$ , uma vez que para um autómato com  $n$  estados há  $n!$  ordens de remoção possíveis.

Neste trabalho apresentamos um resultado que caracteriza um tipo de autómatos para os quais o problema pode ser resolvido de forma eficiente, bem como duas heurísticas que permitem, nos outros casos, uma aproximação em tempo razoável à solução óptima do problema.

# Índice

<b>Resumo</b>	<b>vii</b>
<b>1 Introdução</b>	<b>1</b>
1.1 Definições Fundamentais . . . . .	3
1.1.1 Expressões Regulares . . . . .	4
1.1.1.1 Medidas de Complexidade de Expressões Regulares . .	5
1.1.1.2 Equivalência de Expressões Regulares . . . . .	6
1.1.2 Algumas Noções Elementares da Teoria de Grafos . . . . .	7
1.1.3 Autómatos Finitos . . . . .	8
1.1.3.1 Autómatos Finitos Determinísticos . . . . .	8
1.1.3.2 Autómatos Finitos não Determinísticos . . . . .	10
1.1.3.3 Autómatos Finitos não Determinísticos com Transições- $\epsilon$ . . . . .	11
1.1.3.4 Autómatos Finitos Estendidos . . . . .	11
1.1.4 Autómatos e Grafos . . . . .	12
1.1.5 Equivalência dos Modelos Apresentados . . . . .	13
1.1.6 Autómatos Mínimos . . . . .	14
1.1.7 Algoritmo da Eliminação de Estados . . . . .	14

1.1.7.1	Implementação . . . . .	15
1.1.8	Algoritmo de Kleene . . . . .	18
1.2	Descrição do Problema . . . . .	18
<b>2</b>	<b>Autómatos nard</b>	<b>23</b>
2.1	Introdução . . . . .	23
2.2	Autómatos nard . . . . .	24
2.3	Implementação . . . . .	31
2.4	Autómatos de Thompson . . . . .	35
2.5	Conclusões . . . . .	38
<b>3</b>	<b>Uma heurística para o caso geral</b>	<b>43</b>
3.1	Pesos . . . . .	43
3.2	Heurística . . . . .	44
3.3	Alguns Resultados Experimentais . . . . .	45
3.4	Implementação . . . . .	48
3.5	Conclusões . . . . .	49
<b>4</b>	<b>Uma heurística para o caso acíclico</b>	<b>55</b>
4.1	Motivação . . . . .	55
4.2	Heurística . . . . .	56
4.3	Exemplo . . . . .	62
4.4	Alguns Resultados Experimentais . . . . .	65
4.5	Implementação . . . . .	67
4.6	Conclusões . . . . .	71

<b>5 Conclusões</b>	<b>75</b>
<b>Referências</b>	<b>79</b>
<b>Índice de Tabelas</b>	<b>83</b>
<b>Índice de Figuras</b>	<b>85</b>
<b>Lista de Acrónimos</b>	<b>89</b>
<b>Índice Remissivo</b>	<b>91</b>
<b>A Código</b>	<b>93</b>
A.1 FAtoRatExpX . . . . .	93
A.2 NardFAtoRatExp . . . . .	99
A.3 FAtoRatExp . . . . .	107
A.4 AcyclicFAtoRatExp . . . . .	111

Este trabalho é destinado a servir de introdução à teoria de linguagens formais. A teoria de linguagens formais é uma área fundamental da ciência de computadores, que estuda os sistemas de geração e reconhecimento de linguagens formais. Ela tem aplicações em muitas áreas, como programação, processamento de linguagem natural, análise de circuitos, entre outras.

# Capítulo 1

## Introdução

A teoria de linguagens formais é uma área fundamental da ciência de computadores. Aquilo a que hoje se chama teoria de linguagens formais tem diversas origens, entre as quais alguns problemas de combinatória e de álgebra de semigrupos. Outra área que esteve na origem da teoria de linguagens formais foi a lógica, sendo aqui de destacar o trabalho, na área da computabilidade, de Alan Turing em meados da década de 30 do século XX. A *linguística* também contribuiu para o desenvolvimento da teoria de linguagens formais, em especial o trabalho de Noam Chomsky no estudo de gramáticas, tendo este autor apresentado uma hierarquia de linguagens formais baseada em tipos de gramáticas.

Da hierarquia de Chomsky [HU01] iremos lidar neste trabalho apenas com linguagens regulares. O estudo de linguagens regulares e autômatos finitos data do início da década de 40 do século XX, em que autômatos finitos eram usados para modelar redes neurais por McCulloch e Pitts [MP43]. Resultados da investigação inicial são, por exemplo, o teorema de Kleene que estabelece a equivalência entre expressões regulares e autômatos finitos [Kle56], a introdução de autômatos com *output* por Mealy e Moore [Moo66], a introdução de autômatos finitos não determinísticos por Rabin e Scott [RS59] e a caracterização de linguagens regulares por congruências de índice finito por Myhill [Myh57] e Nerode [Ner58].

As linguagens regulares e os autômatos finitos têm uma grande variedade de aplicações, das quais se destacam a análise léxica em compilação de linguagens de programação, o desenho de circuitos, a edição de texto e a pesquisa de padrões em texto. Em anos

mais recentes surgiram aplicações nos domínios do processamento paralelo, geração e compressão de imagem, teoria de tipos para linguagens de programação orientadas a objectos, computações de ADN, etc [Yu97].

Neste trabalho, dentro do tema das linguagens regulares, iremos focar-nos no problema de procurar, para uma dada linguagem deste tipo, uma sua representação por uma expressão regular com o *menor tamanho*. O tema da simplificação de expressões regulares foi abordado por alguns investigadores, nomeadamente os referidos em [Mit03]. Estes trabalhos consistem em manipulações algébricas de expressões regulares. A nossa abordagem difere destas no sentido de que tentamos obter, a partir de um autómato e baseados num algoritmo conhecido, uma expressão regular equivalente com o *menor tamanho*. Nas nossas pesquisas, encontrámos muito pouco trabalho neste sentido, sendo esta abordagem apenas referida em *Regular Expressions: New Results and Open Problems* [ESW02], onde se diz, a propósito da conversão de autómatos finitos em expressões regulares, que "Dividir para conquistar, juntamente com uma heurística para a separação de grafos ou um algoritmo de aproximação serão certamente técnicas poderosas para a obtenção de expressões regulares relativamente pequenas a partir de autómatos arbitrários..."; em *A characterization of Thompson digraphs* [GPW99], onde se caracterizam os digrafos de Thompson (aqueles que são obtidos a partir de uma expressão regular pelo algoritmo de Thompson) e para estes se apresenta um algoritmo linear que produz uma expressão de tamanho linear no tamanho do digrafo, em que o tamanho do digrafo é o número das suas transições e o tamanho de uma expressão regular é o número total de ocorrências na expressão de símbolos de  $\Sigma \cup \{\epsilon, \emptyset\}$ ; e em *Characterization of Glushkov automata* [CZ00] onde algo semelhante é feito mas para autómatos de Glushkov.

Na secção seguinte apresentaremos as definições necessárias à leitura deste trabalho. No Capítulo 2 apresentamos uma caracterização de autómatos acíclicos para os quais pode ser encontrada de forma eficiente uma expressão regular que representa a linguagem reconhecida pelo autómato e tendo tantos símbolos como o número de transições do autómato. No Capítulo 3 descrevemos uma heurística que tenta escolher, em cada passo do algoritmo da eliminação de estados, o estado cuja remoção conduzirá à obtenção de uma expressão regular *pequena* para um autómato arbitrário. Apresentamos também tabelas que comparam o tamanho das expressões obtidas usando esta heurística e as obtidas pelo algoritmo da eliminação de estados sem uma

escolha apropriada do estado a ser removido, pelas quais se verifica que a heurística produz melhores resultados. No Capítulo 4 apresentamos uma heurística com o mesmo objectivo que a anterior, mas concebida especificamente para o caso de autómatos acíclicos e que, nestes casos, produz melhores resultados que a heurística anterior. No Capítulo 5 delineamos a continuação deste trabalho que gostaríamos de fazer no futuro. Em apêndice apresentamos o código das várias funções implementadas.

## 1.1 Definições Fundamentais

Apresentamos em seguida as definições fundamentais para a leitura deste trabalho.

Chamamos *alfabeto* a um conjunto finito e não vazio de símbolos. Uma *palavra* sobre um alfabeto  $\Sigma$  é uma sequência finita de símbolos de  $\Sigma$ . Denotamos por  $\epsilon$  a *palavra vazia*, i.e., a palavra que contém 0 símbolos.

A *concatenação* de duas palavras é a palavra formada pela justaposição das duas, i.e., a primeira imediatamente seguida pela segunda. Por exemplo, se considerarmos o alfabeto  $\Sigma = \{a, b\}$  e as palavras  $x = aa$  e  $y = ab$  sobre  $\Sigma$ , a concatenação de  $x$  e  $y$ , denotada por  $x \cdot y$  é a palavra  $aa \cdot ab$ . Omitiremos, em geral, o símbolo  $\cdot$  quando nos referirmos à concatenação.

A potência  $w^n$  de uma palavra é indutivamente definida por:  $w^0 = \epsilon \wedge w^{n+1} = w^n w$ .

Denotamos por  $\Sigma^*$  o conjunto de todas as palavras sobre o alfabeto  $\Sigma$ . O *comprimento* de uma palavra  $x$ , que denotamos por  $|x|$ , é o número de símbolos que constituem  $x$ .

Uma *linguagem*  $L$  sobre  $\Sigma$  é um conjunto de palavras sobre  $\Sigma$ . A *linguagem vazia* é denotada por  $\emptyset$ . A *concatenação de duas linguagens*  $L_1$  e  $L_2$  é definida do seguinte modo:  $L_1 \cdot L_2 = \{w \mid w = w_1 w_2 \wedge w_1 \in L_1 \wedge w_2 \in L_2\}$ .

Dada uma linguagem  $L$  definem-se indutivamente as suas potências por:  $L^0 = \{\epsilon\}$ ,  $L^1 = L$  e  $L^{n+1} = L^n \cdot L$ . O *fecho de Kleene* de uma linguagem  $L$  é definido do seguinte modo:  $L^* = \bigcup_{i \geq 0} L^i$ .

O conjunto das *linguagens regulares* sobre um alfabeto  $\Sigma$  é definido como o menor conjunto tal que:

- a linguagem vazia  $\emptyset$  é uma linguagem regular,
- a linguagem  $\{\epsilon\}$ , constituída pela palavra vazia, é uma linguagem regular,
- para cada  $a \in \Sigma$ , a linguagem  $\{a\}$  é uma linguagem regular,
- se  $L_1$  e  $L_2$  são linguagens regulares, então  $L_1 \cup L_2$ ,  $L_1 \cdot L_2$  e  $L_1^*$  são linguagens regulares.

### 1.1.1 Expressões Regulares

Expressões regulares são uma forma sequencial de especificar uma linguagem regular e foram originalmente introduzidas por Kleene [Kle56].

Definimos indutivamente uma expressão regular  $r$  sobre um alfabeto  $\Sigma$  e a linguagem que ela representa  $L(r)$  do seguinte modo:

1.  $r = \emptyset$  é uma expressão regular e denota a linguagem  $L(r) = \emptyset$ .
2.  $r = \epsilon$  é uma expressão regular e denota a linguagem  $L(r) = \{\epsilon\}$ .
3.  $r = a$  com  $a \in \Sigma$  é uma expressão regular e denota a linguagem  $L(r) = \{a\}$ .

Sejam  $r_1$  e  $r_2$  duas expressões regulares, então

4.  $r = (r_1 + r_2)$  é uma expressão regular e denota a linguagem  $L(r) = L(r_1) \cup L(r_2)$ .
5.  $r = (r_1 \cdot r_2)$  é uma expressão regular e denota a linguagem  $L(r) = L(r_1)L(r_2)$ .
6.  $r = r_1^*$  é uma expressão regular e denota a linguagem  $L(r) = (L(r_1))^*$ .

Assumimos que  $*$  tem maior precedência que  $\cdot$  e  $+$  e que  $\cdot$  tem maior precedência que  $+$ . Os parêntesis serão omitidos sempre que isso não seja ambíguo e omitiremos o símbolo  $\cdot$  na escrita de expressões regulares.

**Exemplo 1.1.1** A linguagem constituída por todas as palavras que começam por  $aa$  e terminam com  $bb$  sobre o alfabeto  $\Sigma = \{a, b\}$  pode ser representada pela expressão regular  $aa(a + b)^*bb$ .

### 1.1.1.1 Medidas de Complexidade de Expressões Regulares

Na literatura há várias medidas de complexidade de uma expressão regular:

1. comprimento: o número total de símbolos, incluindo parêntesis e operadores, que ocorrem na expressão,
2. comprimento da expressão convertida para notação polaca (sem parêntesis) e
3. o número total de ocorrências de símbolos do alfabeto na expressão.

Uma expressão regular  $r$  diz-se *colapsável* [ESW02] se e só se:

1.  $r$  contém o símbolo  $\emptyset$  e o número de símbolos em  $r$  for superior a 1 ou
2.  $r$  contém uma subexpressão da forma  $r_1r_2$  ou  $r_2r_1$ , em que  $L(r_1) = \{\epsilon\}$  ou
3.  $r$  contém uma subexpressão da forma  $r_1 + r_2$  ou  $r_2 + r_1$ , em que  $L(r_1) = \{\epsilon\}$  e  $\epsilon \in L(r_2)$ .

Todas estas medidas são equivalentes, a menos de um factor constante, se as expressões forem não colapsáveis [ESW02], pois podemos sempre acrescentar um número arbitrário de parêntesis ou  $\epsilon$  a uma expressão regular sem modificar a linguagem por ela descrita e desta forma deixariamos de ter uma relação definida a menos de um factor constante entre as diferentes medidas. A terceira medida apresentada tem a particularidade de estar relacionada com o número de estados de um autómato obtido a partir de uma expressão regular: se o tamanho de uma dada expressão regular  $r$  for  $n$  (segundo a medida 3), então existe

- um autómato finito não determinístico com no máximo  $n + 1$  estados e que reconhece a linguagem representada por  $r$
- um autómato finito determinístico com no máximo  $2^n + 1$  estados e que reconhece a linguagem representada por  $r$  [ESW02].

Por outro lado, se  $L$  é uma linguagem regular aceite por um autómato finito (ver página 8)  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  em que  $|Q| = n$  e  $|\Sigma| = k$ , então  $L$  pode ser representada

por uma expressão regular de tamanho (segundo a medida 3) não superior a  $nk4^n$  [ESW02].

Na nossa implementação das heurísticas apresentadas neste trabalho, à semelhança de [TA98], definimos o *tamanho* de uma expressão regular  $r$  sobre  $\Sigma$ ,  $tamanho(r)$ , como sendo a soma do número de ocorrências em  $r$  de símbolos do alfabeto e de ocorrências de  $\epsilon$ . No entanto, neste trabalho, consideraremos todas as expressões regulares na sua forma não colapsável. Optámos por esta medida pelo facto de, considerando expressões não colapsáveis, as ocorrências de  $\epsilon$  não serem elimináveis (no sentido de que são necessárias à descrição da linguagem em causa) e como tal adquirirem um significado que equivale ao de um símbolo do alfabeto.

### 1.1.1.2 Equivalência de Expressões Regulares

Seja  $R_\Sigma$  o conjunto das expressões regulares sobre o alfabeto  $\Sigma$ .

Duas expressões regulares  $r$  e  $s$  são equivalentes se representam a mesma linguagem ( $L(r) = L(s)$ ), denotando-se por  $r = s$ . Podem ser definidos vários sistemas formais para expressões regulares que são consistentes e completos a respeito da equivalência de expressões regulares [ST99] e que são a base da simplificação destas expressões em abordagens algébricas. Apresentamos em seguida um desses sistemas [Koz94].

Considerando uma lógica de primeira ordem, um sistema dedutivo e os seguintes axiomas:

$$1. \ r + (s + t) = (r + s) + t$$

$$2. \ r + s = s + r$$

$$3. \ r + \emptyset = r$$

$$4. \ r + r = r$$

$$5. \ r(st) = (rs)t$$

$$6. \ \epsilon r = r$$

$$7. \ r\epsilon = r$$

$$8. r(s+t) = rs + rt$$

$$9. (r+s)t = rt + st$$

$$10. \emptyset r = \emptyset$$

$$11. r\emptyset = \emptyset$$

$$12. \epsilon + rr^* \leq r^*$$

$$13. \epsilon + r^*r \leq r^*$$

$$14. s + rx \leq x \rightarrow r^*s \leq x$$

$$15. s + xr \leq x \rightarrow sr^* \leq x$$

onde  $r \leq s \equiv r + s = s$ ,

temos que, se duas quaisquer expressões regulares  $r$  e  $s$  são equivalentes, então  $r = s$  é um teorema do sistema (o sistema é completo) e se  $r = s$  é um teorema do sistema, então  $r$  e  $s$  são equivalentes (o sistema é consistente).

O problema de saber se duas expressões regulares são equivalentes é PSPACE-completo [GJ03].

### 1.1.2 Algumas Noções Elementares da Teoria de Grafos

Um *grafo* é um par ordenado  $G = (V, E)$ , em que  $V$  é um conjunto de vértices e  $E$  é um conjunto de pares não ordenados (chamados *arestas*) de vértices de  $V$ .

Um *grafo dirigido*, ou *digrafo*, é um par ordenado  $G = (V, E)$ , em que  $V$  é um conjunto de vértices e  $E$  é um conjunto de pares ordenados (chamados *arcos*) de vértices de  $V$ . Chamamos *lacete* a um arco da forma  $(u, u)$ .

Neste trabalho, apenas consideraremos digrafos, pelo que, sempre que escrevermos "grafo", estaremos a referir-nos a um digrafo.

Um *caminho* num grafo  $G$  é uma sequência de vértices,  $v_1 - \dots - v_n$ , tais que  $(v_i, v_{i+1}) \in E$ ,  $1 \leq i < n$ . Chamamos *ciclo* a um caminho em que o primeiro e o último vértices são o mesmo.

Definimos o *grau de entrada* de um vértice  $v$ , num grafo  $G$ ,  $ge(v)$ , como o número de arcos  $(u, v)$  de  $G$ , com  $u \neq v$ . Analogamente, definimos o *grau de saída* de um vértice  $v$ , num grafo  $G$ ,  $gs(v)$ , como o número de arcos  $(v, u)$  de  $G$ , com  $u \neq v$  [BJG01]. Uma *ordem topológica*  $\mathcal{T}$  num grafo acíclico  $G = (V, E)$  com  $|V| = n$  é uma função de  $V$  em  $\{1, \dots, n\}$  tal que se  $(u, v) \in E$  então  $\mathcal{T}(u) < \mathcal{T}(v)$ .

É usual representar um grafo  $G = (V, E)$  de uma das seguintes formas:

- *lista de adjacências*: uma lista em que o  $i$ -ésimo elemento é a lista dos vértices  $v$  tais que  $(i, v) \in E$  e
- *matriz de adjacências*: uma matriz  $A_{n \times n}$  com  $n = |V|$ , tal que  $A_{i,j} = 1$  se  $(i, j) \in E$  e  $A[i, j] = 0$  caso contrário.

### 1.1.3 Autómatos Finitos

Nesta subsecção iremos apresentar quatro tipos de autómatos finitos (AF): autómatos finitos determinísticos (AFD), autómatos finitos não determinísticos (AFND), autómatos finitos não determinísticos com transições- $\epsilon$  (AFNDTE) e autómatos finitos estendidos (AFE).

#### 1.1.3.1 Autómatos Finitos Determinísticos

Um autómato finito consiste num número finito de estados e um conjunto de regras que definem a mudança de estado quando o autómato lê um símbolo do seu *input*. Se o estado seguinte for unicamente determinado pelo estado actual e pelo símbolo lido, dizemos que o autómato é determinístico.

Formalmente, definimos um AFD  $\mathcal{A}$  como sendo um quíntuplo  $(Q, \Sigma, \delta, q_0, F)$  onde

- $Q$  é o conjunto finito de estados,
- $\Sigma$  é o alfabeto de *input*,
- $\delta : Q \times \Sigma \rightarrow Q$  é a função de transição,
- $q_0 \in Q$  é o estado inicial e

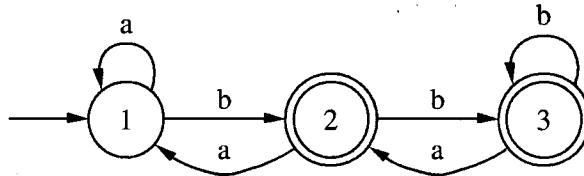
- $F \subseteq Q$  é o conjunto de estados finais.

Note-se que em geral não requeremos que a função de transição  $\delta$  seja total, i.e., que esteja definida para todos os pares em  $Q \times \Sigma$ . Se  $\delta$  for total dizemos que  $\mathcal{A}$  é um AFD *completo*.

**Exemplo 1.1.2** Seja  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  um AFD onde  $Q = \{1, 2, 3\}$ ,  $\Sigma = \{a, b\}$ ,  $q_0 = 1$ ,  $F = \{2, 3\}$  e  $\delta$  definida do seguinte modo:

$$\begin{aligned}\delta(1, a) &= 1, & \delta(1, b) &= 2, \\ \delta(2, a) &= 1, & \delta(2, b) &= 3, \\ \delta(3, a) &= 2, & \delta(3, b) &= 3.\end{aligned}$$

Na Figura que se segue encontra-se uma representação gráfica deste autómato, em que existe o arco  $(v_1, v_2)$  etiquetado por  $a$  se  $\delta(v_1, a) = v_2$ , o estado inicial é indicado por uma seta com a cabeça nesse estado e a cauda em nenhum estado e os estados finais são representados com círculos duplos.



Uma *configuração* de  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  é uma palavra de  $Q\Sigma^*$ , i.e., um estado  $q \in Q$  seguido de uma palavra  $x \in \Sigma^*$ , onde  $q$  é o estado actual do autómato e  $x$  é a parte do *input* que falta ser lida. A *configuração inicial* de  $\mathcal{A}$  com *input*  $x \in \Sigma^*$  é  $q_0x$ . Uma *configuração de aceitação* é definida por  $f\epsilon$  com  $f \in F$ .

Um *passo de computação* de  $\mathcal{A}$  é uma transição de uma configuração  $\alpha$  para uma configuração  $\beta$ , denotando-se por  $\alpha \vdash_{\mathcal{A}} \beta$  e é uma relação binária no conjunto de configurações de  $\mathcal{A}$ . A relação  $\vdash_{\mathcal{A}}$  é definida por: para  $px, qy \in Q\Sigma^*$ ,  $px \vdash_{\mathcal{A}} qy$  se  $x = ay$  para algum  $a \in \Sigma$  e  $\delta(p, a) = q$ . Usaremos a notação  $\alpha \vdash \beta$ , em vez de  $\alpha \vdash_{\mathcal{A}} \beta$ , se não houver lugar para dúvidas quanto ao autómato em causa.

A potência  $k$  de  $\vdash$ , denotada por  $\vdash^k$ , é definida indutivamente por  $\alpha \vdash^0 \alpha$  para todas as configurações  $\alpha \in Q\Sigma^*$  e  $\alpha \vdash^k \beta$ , para  $k > 0$  e  $\alpha, \beta \in Q\Sigma^*$ , se existe  $\gamma \in Q\Sigma^*$

tal que  $\alpha \vdash^{k-1} \gamma$  e  $\gamma \vdash \beta$ . O fecho transitivo e o fecho reflexivo e transitivo de  $\vdash$  são denotados por  $\vdash^+$  e  $\vdash^*$  respectivamente. Uma palavra  $x \in \Sigma^*$  é *aceite* por um AFD  $\mathcal{A}$  se e só se  $q_0x \vdash_{\mathcal{A}}^* f\epsilon$  com  $f \in F$ . A linguagem aceite por um AFD  $\mathcal{A}$  é o conjunto das palavras aceites por  $\mathcal{A}$  e designa-se por  $L(\mathcal{A})$ .

### 1.1.3.2 Autómatos Finitos não Determinísticos

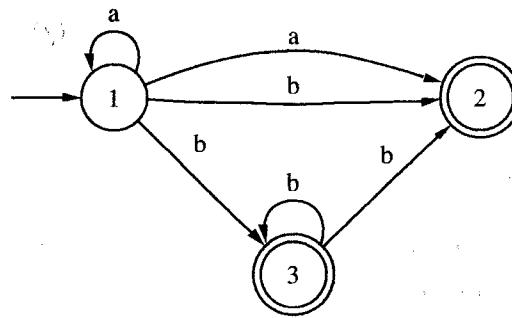
Os autómatos finitos não determinísticos são uma generalização do caso determinístico em que, para um dado estado e um símbolo de *input*, o número de transições possíveis pode ser superior a um.

Formalmente, um AFND  $\mathcal{A}$  é um quíntuplo  $(Q, \Sigma, \delta, q_0, F)$  onde  $Q, \Sigma, q_0$  e  $F$  são definidos exactamente como no caso determinístico e  $\delta : Q \times \Sigma \rightarrow 2^Q$  é a função de transição, onde  $2^Q$  representa o conjunto dos subconjuntos de  $Q$ .

**Exemplo 1.1.3** Seja um AFND  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  onde  $Q = \{1, 2, 3\}$ ,  $\Sigma = \{a, b\}$ ,  $q_0 = 1$ ,  $F = \{2, 3\}$  e  $\delta$  definida do seguinte modo:

$$\begin{array}{ll} \delta(1, a) = \{1, 2\}, & \delta(1, b) = \{2, 3\}, \\ \delta(2, a) = \emptyset, & \delta(2, b) = \emptyset, \\ \delta(3, a) = \emptyset, & \delta(3, b) = \{3, 2\}. \end{array}$$

Na Figura seguinte encontra-se uma representação gráfica deste autómato.



Um passo de computação de um AFND  $\mathcal{A}$  é uma relação binária definida em  $Q\Sigma^*$ ,  $\vdash_{\mathcal{A}} : Q\Sigma^* \times Q\Sigma^*$  tal que  $px \vdash_{\mathcal{A}} qy$  se  $x = ay$  e  $q \in \delta(p, a)$  com  $p, q \in Q, x, y \in \Sigma^*$  e  $a \in \Sigma$ . O fecho transitivo e o fecho reflexivo e transitivo de  $\vdash$  são denotados por  $\vdash^+$  e  $\vdash^*$  respectivamente. Uma palavra  $x \in \Sigma^*$  é aceite por um AFND se e só se  $q_0x \vdash^* f\epsilon$

com  $f \in F$ . A linguagem aceite por um AFND é o conjunto das palavras aceites pelo autómato em causa.

### 1.1.3.3 Autómatos Finitos não Determinísticos com Transições- $\epsilon$

Um autómato finito não determinístico com transições- $\epsilon$  (AFNDTE)  $\mathcal{A}$  é um quíntuplo  $(Q, \Sigma, \delta, q_0, F)$  em que  $Q, \Sigma, q_0$  e  $F$  são como no caso de um AFND e a função de transição é  $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ .

Um passo de computação de um AFNDTE  $\mathcal{A}$  é uma relação binária definida em  $Q\Sigma^*$ ,  $\vdash_{\mathcal{A}} : Q\Sigma^* \times Q\Sigma^*$  tal que  $px \vdash_{\mathcal{A}} qy$  se  $x = ay$  e  $q \in \delta(p, a)$  ou se  $x = y$  e  $q \in \delta(p, \epsilon)$  com  $p, q \in Q, x, y \in \Sigma^*$  e  $a \in \Sigma$ . Uma palavra  $x \in \Sigma^*$  é aceite por um AFND se e só se  $q_0x \vdash^* f\epsilon$  com  $f \in F$ . A linguagem aceite por um AFNDTE é o conjunto das palavras aceites pelo autómato em causa.

### 1.1.3.4 Autómatos Finitos Estendidos

Um autómato finito estendido é semelhante a um AFNDTE com a diferença de que as etiquetas das transições podem ser, não apenas símbolos do alfabeto ou  $\epsilon$ , mas expressões regulares gerais.

Seja  $R_\Sigma$  o conjunto das expressões regulares sobre o alfabeto  $\Sigma$ . Formalmente, um AFE  $\mathcal{A}$  é um quíntuplo  $(Q, \Sigma, \delta, q_0, F)$  onde

- $Q$  é o conjunto finito de estados,
- $\Sigma$  é o alfabeto de *input*,
- $\delta : Q \times Q \rightarrow R_\Sigma$  é a função que descreve as transições do autómato,
- $q_0$  é o estado inicial e
- $F \subseteq Q$  é o conjunto de estados finais.

Assumimos que  $\delta(p, q) = \emptyset$  se a transição de  $p$  para  $q$  não está presente.

Uma palavra  $x \in \Sigma^*$  é aceite por  $\mathcal{A}$  se  $x = x_1 \dots x_n$  com  $x_1, \dots, x_n \in \Sigma^*$  e existe uma sequência de estados  $q_0, q_1, \dots, q_n$  com  $q_n \in F$  tal que  $x_1 \in L(\delta(q_0, q_1)), \dots, x_n \in$

$L(\delta(q_{n-1}, q_n))$ . A linguagem aceite por um AFE é o conjunto das palavras aceites pelo autómato em causa.

Seja  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  um AFE. Definimos a função  $\delta^+ : Q \times Q \rightarrow R_\Sigma$  tal que, para cada caminho  $c_i = u_1 - \dots - u_n$  temos  $\delta(u, u_1) = r_1, \dots, \delta(u_{n-1}, u_n) = r_n$  e  $r_{c_i} = r_1 \dots r_n$ , sendo  $\delta^+(u_1, u_n) = r_{c_1} + \dots + r_{c_m}$  em que  $c_1, \dots, c_m$  são os caminhos de  $u_1$  para  $u_n$ . No capítulo seguinte usaremos esta função, escrevendo  $\delta^+(p, q) \neq \emptyset$  para indicar que existe um caminho de  $p$  para  $q$ .

Assumimos também que num AFE não há duas transições de um estado  $p$  para um estado  $q$  pois é fácil concluir que a linguagem aceite pelo AFE em causa não se altera se substituirmos estas duas transições por uma única, etiquetada pela união das expressões que etiquetavam as transições iniciais. Um caminho de um vértice  $v_1$  para um vértice  $v_n$  no grafo subjacente a um autómato finito estendido  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  fica assim unicamente definido pela sequência  $v_1 - \dots - v_n$  de vértices tais que  $\delta(v_i, v_{i+1}) \neq \emptyset, 1 \leq i < n$ .

#### 1.1.4 Autómatos e Grafos

O *grafo subjacente* a um AF  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  é o grafo  $G = (Q, E)$ , em que  $E = \{(u, v) \mid u, v \in Q \text{ e } \exists a \in \Sigma \text{ tal que } \delta(u, a) = v \text{ no caso de AFNDs ou } \delta(u, v) = a \text{ no caso de AFEs}\}$ .

No grafo subjacente a um autómato finito, chamaremos *vértices iniciais* aos vértices correspondentes aos estados iniciais do autómato, *vértices finais* aos vértices correspondentes aos estados finais e *vértices intermédios* a todos os outros.

Chamamos *autómato útil* a um autómato em cujo grafo subjacente passa por todo o vértice algum caminho entre o vértice inicial e um vértice final e para cada vértice final *fin* existe um caminho do vértice inicial para *fin*. Dizemos que um autómato é *acíclico* se não contém nenhum ciclo.

Dada a sua íntima relação, usaremos a terminologia para autómatos e grafos indistintamente.

### 1.1.5 Equivalência dos Modelos Apresentados

Dois AFs  $\mathcal{A}$  e  $\mathcal{A}'$  dizem-se *equivalentes* se  $L(\mathcal{A}) = L(\mathcal{A}')$ . Os AFDs são casos particulares de AFNDs e é possível transformar um AFND ou um AFNDTE num AFD equivalente (ver [ASU86]).

Um AF *aceita* um conjunto de palavras (no sentido de que dada uma palavra  $w$  e um autómato  $\mathcal{A}$  é possível saber se  $w \in L(\mathcal{A})$ ), ao passo que uma expressão regular representa um conjunto de palavras (no sentido de que a expressão regular é um *padrão* ao qual obedecem todas as palavras pertencentes à linguagem representada pela expressão).

As expressões regulares e os AFs têm o mesmo poder descritivo, ou seja, dada uma expressão regular  $r$  é possível construir um AF  $\mathcal{A}$  tal que  $L(r) = L(\mathcal{A})$  e vice-versa. Um dos algoritmos mais comuns para converter uma expressão regular num AF equivalente é o algoritmo de Thompson [Tho68]. Este algoritmo produz um AFNDTE com tamanho linear no tamanho da expressão regular e fá-lo em tempo linear [GPW98]. Uma das características destes autómatos (cuja caracterização se pode encontrar em [GPW99]) é o facto de terem apenas um estado final. Uma taxonomia dos métodos de conversão de uma expressão regular num autómato equivalente pode ser encontrada em [Wat94b]. Esta conversão encontra-se extensivamente estudada na literatura devido às suas múltiplas aplicações, embora o mesmo não aconteça com a conversão inversa.

Assim também vemos que um AFE pode ser transformado num AFNDTE equivalente, substituindo o arco entre dois estados  $p$  e  $q$  pelo AFNDTE que resulta da aplicação do algoritmo de Thompson à expressão  $\delta(p, q)$ , uma vez que o autómato de Thompson resultante tem apenas um estado final. Claramente, um AFNDTE pode ser visto como um AFE, resultando a equivalência entre AFEs e AFNDTEs.

A conversão de um AFND (ou AFNDTE) numa expressão regular equivalente pode ser feita usando o algoritmo da eliminação de estados, descrito na Secção 1.1.7. Além deste algoritmo, existe outro, que é o algoritmo apresentado por Kleene [Kle56] para converter AFNDs em expressões regulares e descrito na Secção 1.1.8.

Este segundo algoritmo produz, regra geral, expressões de tamanho superior às produzidas pelo algoritmo da eliminação de estados.

### 1.1.6 Autómatos Mínimos

Dado um AFD  $\mathcal{A}$  é possível encontrar um AFD  $\mathcal{A}'$  tal que  $L(\mathcal{A}) = L(\mathcal{A}')$  e  $\mathcal{A}'$  é mínimo no número de estados. Um dos algoritmos para a minimização encontra-se descrito em [ASU86] e uma taxonomia destes algoritmos pode ser encontrada em [Wat94a]. O AFD mínimo para uma dada linguagem regular é único a menos de uma renomeação dos estados [ASU86], o que permite testar se dois autómatos aceitam a mesma linguagem.

Um AFND mínimo no número de estados pode não ser único no conjunto dos AFNDs que lhe são equivalentes, o mesmo acontecendo com as expressões regulares. Em termos de espaço, a representação de uma linguagem regular por um AFND pode oferecer uma vantagem exponencial em relação a um AFD [HK02]. O problema de converter um AFD num AFND equivalente e minimal é PSPACE-completo [JR93]. O problema da minimização de AFNDs é PSPACE-completo (excepto no caso de alfabetos unários, em que o problema é NP-completo) [GJ03].

### 1.1.7 Algoritmo da Eliminação de Estados

O algoritmo da eliminação de estados (AEE) permite obter uma expressão regular que representa a linguagem reconhecida por um dado AF. O AEE consiste em remover em cada passo um vértice do AFE associado ao autómato finito dado e na actualização do AFE resultante de forma a que a linguagem reconhecida não se altere. O algoritmo termina quando restarem apenas dois vértices *ini* e *fin* (que não são removíveis e que são, respectivamente, um novo estado inicial do qual parte uma transição por  $\epsilon$  para o estado inicial, que deixa de ser inicial e um novo estado final para o qual há uma transição por  $\epsilon$  de cada estado final, deixando estes de ser finais) e a expressão regular de *output* é  $\delta(\text{ini}, \text{fin})$ .

Seja  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  um AFE e  $r_{pq}$  a expressão regular  $\delta(p, q)$ , i.e., a etiqueta da transição entre  $p$  e  $q$ ,  $p, q \in Q$ . Seja  $q$  um estado tal que  $q \neq q_0$  e  $q \notin F$ . Um AFE  $\mathcal{A}' = (Q', \Sigma, \delta', q_0, F)$  equivalente a  $\mathcal{A}$  tal que  $Q' = Q - \{q\}$ , i.e., em que  $q$  foi removido de  $\mathcal{A}$ , é definido do seguinte modo: para cada par de estados  $p$  e  $s$  pertencentes a  $Q'$ ,

$$\delta'(p, s) = r_{ps} + r_{pq}r_{qq}^*r_{qs}. \quad (1.1)$$

A descrição completa do algoritmo é a seguinte (ver [Yu97]): Seja  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  um AFE,

1. Se o estado inicial for também final ou se tiver grau de entrada superior a 0, então adiciona-se a  $\mathcal{A}$  um novo estado  $ini$  não removível e inicial e uma transição por  $\epsilon$  de  $ini$  para o estado inicial e este deixa de ser inicial; se o estado inicial não for final e tiver grau de entrada 0, então não é necessário acrescentar um novo estado e toma-se o estado não removível,  $ini$ , como sendo o estado inicial. Se houver mais que um estado final ou se o único estado final tiver grau de saída superior a 0, então acrescenta-se um novo estado ao autómato,  $fin$  não removível e final e uma transição por  $\epsilon$  de cada estado final para  $fin$ , deixando estes de ser finais; se houver apenas um estado final e este tiver grau de saída 0, então não é necessário acrescentar um novo estado e toma-se o estado não removível,  $fin$ , como sendo o estado final.

Seja  $\mathcal{A}' = (Q', \Sigma, \delta', ini, fin)$  o autómato assim obtido.

2. Se  $Q'$  consiste apenas em  $ini$  e  $fin$ , então a expressão regular resultado é  $\delta'(ini, fin)$  e o algoritmo termina. Caso contrário executa-se o passo seguinte.
3. Escolhe-se  $q \in Q'$  tal que  $ini \neq q \neq fin$ , remove-se  $q$  de  $\mathcal{A}'$  como indicado acima, actualizando  $\delta'$  como na Equação (1.1) e executa-se o passo 2.

### 1.1.7.1 Implementação

O GAP é um sistema de *software* livre sob GPL (*GNU Public License*) para álgebra computacional discreta e o seu nome é um acrónimo de *Groups, Algorithms and Programming*. Este sistema fornece uma linguagem de programação imperativa. Na Secção A.1 (página 93) apresentamos uma implementação do AEE escrita em GAP [GAP04]. Esta implementação fez, inicialmente, parte de um pacote sobre autómatos [DLM04] desenvolvido em colaboração com M. Delgado (Universidade do Porto) e S. Linton (Universidade de St. Andrews) e aceite para integrar futuras distribuições do GAP. Esta implementação foi posteriormente substituída pela heurística descrita no Capítulo 3.

Neste pacote, as transições de um AFND  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  são representadas por uma matriz  $\Sigma \times Q$  em que cada entrada  $[a, q], a \in \Sigma, q \in Q$  é a lista dos estados  $s$

tais que do estado  $q$  há uma transição para  $s$  etiquetada por  $a$ . No código chamamos *GTG* - *Generalized Transition Graph* [MMP<sup>+</sup>95] ao AFE associado ao autómato de *input*. A função `computeGTG` calcula o AFE associado ao AF dado e as funções `destroyLoop` e `destroyState` implementam, respectivamente, a eliminação de um estado pela eliminação primeiro do lacete, se existir e em seguida a eliminação do estado propriamente dita. A função `RatExpOnnLetters(n, op, list)` cria uma expressão regular sobre um alfabeto  $\Sigma$  de  $n$  símbolos, em que  $op$  é  $[]$ , `product`, `union` ou `star` conforme se trate de um símbolo do alfabeto (ou  $\epsilon$  ou  $\emptyset$ ), ou uma concatenação, ou uma união ou o fecho de Kleene, respectivamente;  $list$  é uma lista de expressões regulares (as subexpressões) se se trata de uma concatenação ou união,  $[n]$  se se trata do  $n$ -ésimo símbolo de  $\Sigma$  no caso de a expressão ser um símbolo de  $\Sigma$  (considera-se uma ordem em  $\Sigma$ ) ou uma expressão regular se se trata do fecho de Kleene.

Na Figura 1.1 apresentamos o corpo "principal" desta implementação.

```
FAtoRatExpX := function(A)

    # ... Definições das funções locais ...
    # Q é o número de estados do AFE
    computeGTG(); # Converte o autómato inicial
                    # no AFE equivalente
    while Q > 0 do
        destroyLoop(Q); # Elimina lacete do estado Q
        destroyState(Q); # Elimina o estado Q
    od;
    # Devolve a expressão resultado
    # T[p][q] é a etiqueta do arco (p,q)
    if T[1][2] = 0 then
        return(RatExpOnnLetters(A!.alphabet, [], "empty_set"));
    else
        return(T[1][2]);
    fi;
end;
```

Figura 1.1: Algoritmo da Eliminação de Estados

A função `computeGTG` é executada uma única vez e tem complexidade  $\mathcal{O}(kn^3)$  no caso de o *input* ser um AFND ou um AFNDTE e  $\mathcal{O}(kn^2)$  no caso de um AFD, em que  $n$  é o número de estados do AF de *input* e  $k = |\Sigma|$ . Como se pode ver pelas seguintes linhas de código da função `computeGTG` a complexidade é  $\mathcal{O}(kn^3)$ :

```

for p in [1 .. A!.states] do
    ...
    for q in [1 .. A!.states] do
        ...
            for a in [1 .. A!.alphabet] do
                if q in A!.transitions[a][p] then
                    ...
                    fi;
                od;
            ...
        od;
    ...
od;

```

em que `A!.states` é o número de estados do autómato inicial, `A!.alphabet`= $k$  e `A!.transitions[a][p]` é a lista de estados para os quais há uma transição do estado  $p$  pelo símbolo do alfabeto  $a$ . No caso de um AFD esta lista encontra-se reduzida a um único estado, pelo que a complexidade neste caso é  $\mathcal{O}(kn^2)$ .

A função `destroyLoop` é executada  $n$  vezes e tem complexidade  $\mathcal{O}(kn4^n)$ . Esta complexidade está interligada com a complexidade da função `destroyState` (estas duas funções implementam a remoção de um estado em dois passos, removendo primeiro o lacete e depois o estado) e deve-se ao facto de em cada chamada destas funções o tamanho das expressões regulares envolvidas quadriplicar e ter um tamanho máximo inicial de  $k$  (no autómato inicial pode haver no máximo  $k$  transições entre quaisquer dois estados, sendo, no AFE inicial, reduzidas a uma única transição etiquetada por uma expressão de tamanho  $k$ ). A função `destroyState` é executada  $n$  vezes e tem complexidade  $\mathcal{O}(n^2k4^n)$ . Esta complexidade deve-se às razões indicadas atrás e ao facto de a actualização das etiquetas das transições ser feita dentro do ciclo

```
for p in [1 .. n-1] do
```

```

...
for q in [1 .. n-1] do
... Actualização das etiquetas das transições ...
od;
od;

```

em que  $n$  é o estado a ser removido. Por tudo isto o algoritmo tem complexidade  $\mathcal{O}(n^3 k 4^n)$ .

### 1.1.8 Algoritmo de Kleene

Em seguida descrevemos o algoritmo de Kleene para converter um AFND numa expressão regular equivalente [Aho04]:

*Input:* AFND  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ , em que  $Q = \{1, 2, \dots, n\}$ . Representamos por  $L[i, j]$  a etiqueta do arco  $(i, j)$  (se este arco não existir, temos  $L[i, j] = \emptyset$ ; se existir mais que um arco, então  $L[i, j]$  é a expressão que representa a união das etiquetas destes arcos) e por  $\epsilon$  representamos  $\epsilon$ .

*Output:* Matriz  $C_{n \times n}$  em que  $C_{i,j}$  é uma expressão regular que descreve todos os caminhos de  $i$  para  $j$  e constrói-se calculando sucessivos  $C_k[i, j]$ , em que  $C_k[i, j]$  é uma expressão regular que descreve todos os caminhos de  $i$  para  $j$  que não passam por nenhum vértice cuja numeração seja superior a  $k$  (com a possível excepção do primeiro e último vértices desses caminhos). A expressão resultado será  $\bigcup_{q \in F} C_{q_0, q}$ . A descrição apresentada na Figura 1.2 é conceptual e não corresponde a uma implementação numa linguagem de programação concreta.

## 1.2 Descrição do Problema

O problema que abordamos neste trabalho é o de, dado um autómato finito, encontrar, pelo algoritmo da eliminação de estados, uma expressão regular equivalente com o menor tamanho. Quando aplicamos a um autómato o algoritmo da eliminação de estados, a ordem de remoção dos vértices influencia o tamanho da expressão regular obtida. Assim, a obtenção de uma expressão de menor tamanho pode ser feita através de uma escolha adequada da ordem de remoção dos vértices.

```

procedure Kleene(A){
    for i := 1 to n do
        for j := 1 to n do
            C_0[i,j] := L[i,j];
        od;
    od;
    for i := 1 to n do
        C_0[i,i] := @ + C_0[i,i];
    od;
    for k := 1 to n do
        for i := 1 to n do
            for j := 1 to n do
                C_k[i,j] := C_{k-1}[i,j] +
                    C_{k-1}[i,k].(C_{k-1}[k,k]).*.C_{k-1}[k,j];
            od;
        od;
    od;
    for i := 1 to n do
        for j := 1 to n do
            C[i,j] := C_n[i,j];
        od;
    od;
    return(C);
}

```

Figura 1.2: Algoritmo de Kleene

Consideremos o autómato da Figura 1.3.

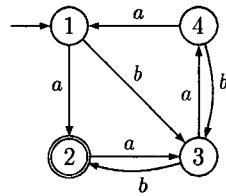


Figura 1.3: Autómato  $\mathcal{A}$

Se neste autómato removermos os vértices na ordem 2, 1, 4, 3, (ver Figuras 1.4 a 1.7 em que  $I$  é *ini* e  $F$  é *fin*), obtemos a expressão regular  $r = a + (b+aa)(ba+a(b+a(b+aa)))^*(b+aaa)$  em que  $tamanho(r) = 16$ ; se, por outro lado, removermos os vértices na ordem 3, 4, 2, 1 (ver Figuras 1.8 a 1.11), obtemos a expressão regular  $s = (ba(ba)^*a + (a+bb+ba(ba)^*bb)(ab+aa(ba)^*bb)^*aa(ba)^*a)^*(a+bb+ba(ba)^*bb)(ab+aa(ba)^*bb)^*$  em que  $tamanho(s) = 44$ .

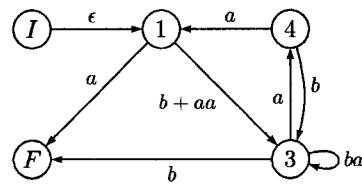


Figura 1.4: Autómato  $\mathcal{A}$  depois de removido o vértice 2

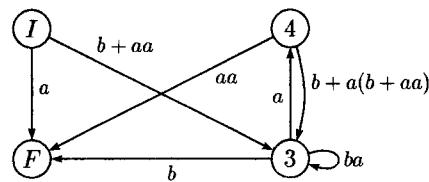


Figura 1.5: Autómato  $\mathcal{A}$  depois de removidos os vértices 2 e 1

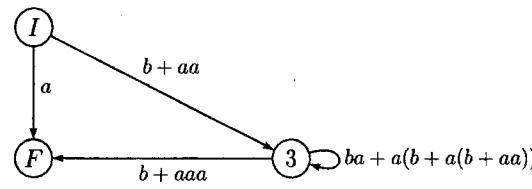


Figura 1.6: Autómato  $\mathcal{A}$  depois de removidos os vértices 2, 1 e 4

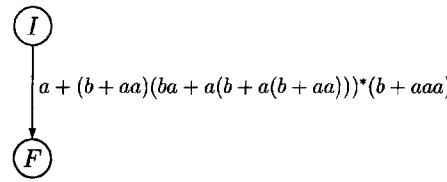


Figura 1.7: Autómato  $\mathcal{A}$  depois de removidos os vértices 2, 1, 4 e 3

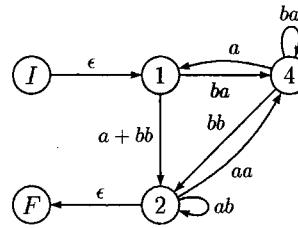


Figura 1.8: Autómato  $\mathcal{A}$  depois de removido o vértice 3

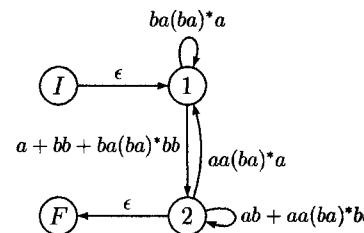


Figura 1.9: Autómato  $\mathcal{A}$  depois de removidos os vértices 3 e 4

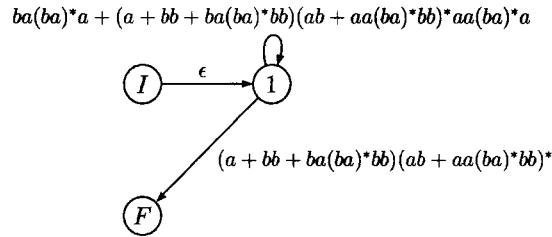


Figura 1.10: Autómato  $\mathcal{A}$  depois de removidos os vértices 3, 4 e 2

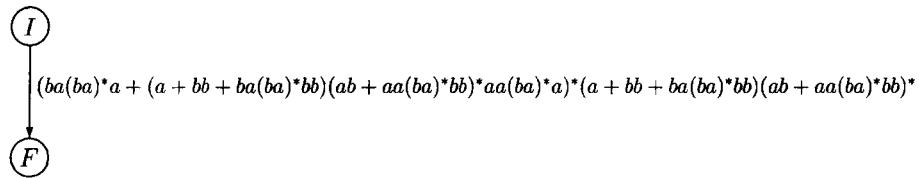


Figura 1.11: Autómato  $\mathcal{A}$  depois de removidos os vértices 3, 4, 2 e 1

No caso geral, encontrar uma ordem óptima (no sentido de ser uma que produz uma das expressões de menor tamanho) de remoção dos vértices é um problema computacionalmente difícil. O algoritmo por "força bruta", que consiste em aplicar o AEE em todas as ordens de remoção possíveis, tem complexidade  $\mathcal{O}(n!)$ , uma vez que para um autómato de  $n$  estados há  $n!$  ordens de remoção diferentes.

Como veremos no Capítulo 5, o AEE, mesmo quando aplicado ao AFD mínimo de uma dada linguagem, pode não produzir uma expressão mínima para essa linguagem.

# Capítulo 2

## Autómatos nard

Neste capítulo apresentamos uma caracterização de autómatos acíclicos para os quais a ordem óptima de remoção dos vértices pode ser encontrada de forma eficiente. Neste capítulo e nos seguintes consideraremos apenas autómatos úteis, uma vez que as etiquetas dos arcos  $(u, v)$  em que  $u$  ou  $v$  são vértices *não úteis* (vértices que não fazem parte de nenhum caminho do vértice inicial para um vértice final) não estarão presentes como subexpressões do resultado, não contribuindo assim para o aumento do tamanho da expressão regular final.

Neste capítulo consideraremos apenas autómatos com um único estado final, uma vez que o caso de autómatos com mais do que um estado final pode ser reduzido ao caso de um único estado final, pela introdução de uma transição por  $\epsilon$  de cada estado final (deixando estes de ser finais) para um novo estado, sendo este o único estado final.

Para os autómatos considerados neste capítulo verifica-se que o tamanho da expressão regular obtida pelo AEE modificado para este caso é igual ao número de transições do autómato inicial.

### 2.1 Introdução

Neste capítulo tratamos de autómatos a que demos o nome de *nard*, que é um acrônimo de *não ambíguo (relativamente às) regras da distributividade*, sendo as regras da distributividade as regras números 8 e 9 do sistema formal apresentado

na Secção 1.1.1.2 (ver página 6).

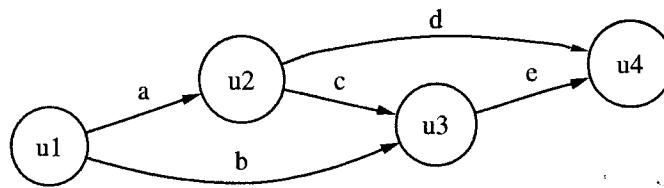


Figura 2.1: Grafo Etiquetado

Considerando o grafo da Figura 2.1 (que não é nard como veremos), se removermos primeiro, numa aplicação do AEE, o vértice  $u_2$  e em seguida o vértice  $u_3$  obtemos a expressão regular  $ad + (ac + b)e$ . Se, em alternativa, removermos primeiro o vértice  $u_3$  e em seguida  $u_2$ , obtemos a expressão  $be + a(ce + d)$ . Podemos notar que no primeiro caso o símbolo  $a$  é o único que ocorre duas vezes na expressão, ao passo que no segundo caso é o símbolo  $e$ , ou seja, no primeiro caso aplicámos a regra da distributividade à direita a  $e$  e no segundo, à esquerda, a  $a$ . No caso geral isto causa a ambiguidade de sabermos se devemos remover primeiro um ou outro vértice, uma vez que, no sentido de obter uma expressão de menor tamanho, devemos remover aquele vértice que causa a aplicação da regra da distributividade à etiqueta de maior tamanho, o que nem sempre é trivial de determinar.

No caso da aplicação do AEE a um autómato nard este problema não se põe, uma vez que, como veremos, será sempre removido um vértice  $v$  tal que  $ge(v) = 1 = gs(v)$ , não havendo portanto ocorrências múltiplas na expressão final da etiqueta de uma dada transição.

## 2.2 Autómatos nard

Começamos por introduzir algumas definições e usamos a função definida na Secção 1.1.3.4, escrevendo  $\delta^+(p, q) \neq \emptyset$  para indicar que existe um caminho de  $p$  para  $q$ .

**Definição 2.2.1 (Forma nard)** *Dizemos que um autómato acíclico útil, com apenas um estado inicial (que denotaremos sempre por ini) e um estado final (que denotaremos*

sempre por fin) se encontra na forma nard se no seu grafo subjacente não existirem vértices intermédios  $v_1$  e  $v_2$  ( $v_1 \neq v_2$ ) tais que:

1. existe um caminho de  $ini$  para  $v_1$  que não passa por  $v_2$  e um caminho de  $ini$  para  $v_2$  que não passa por  $v_1$  e
2. existe um caminho de  $v_1$  para  $fin$  que não passa por  $v_2$  e um caminho de  $v_2$  para  $fin$  que não passa por  $v_1$  e
3. existe um caminho de  $v_1$  para  $v_2$  e
4. não existe nenhum vértice intermédio  $u$  tal que: existem os caminhos  $v_1 - \dots - u$  e  $u - \dots - v_2$  e todos os caminhos de  $ini$  para  $v_2$  passam por  $u$ .

Na Figura 2.2 temos um autómato que não se encontra na forma nard porque se considerarmos  $v_1$  o vértice 2 e  $v_2$  o vértice 3, então 1 – 2 é um caminho de  $ini$  para  $v_1$  que não passa por  $v_2$ , 1 – 3 é um caminho de  $ini$  para  $v_2$  que não passa por  $v_1$ , 2 – 4 é um caminho de  $v_1$  para  $fin$  que não passa por  $v_2$ , 3 – 4 é um caminho de  $v_2$  para  $fin$  que não passa por  $v_1$  e 2 – 3 é um caminho de  $v_1$  para  $v_2$  e não existe nenhum vértice intermédio  $u$  tal que existem os caminhos  $v_1 - \dots - u$  e  $u - \dots - v_2$  e todos os caminhos de  $ini$  para  $v_2$  passam por  $u$ .

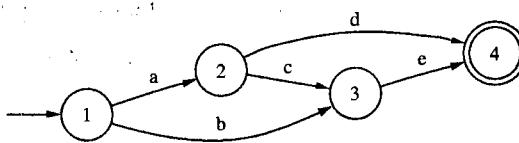


Figura 2.2: Autómato não nard

Na Figura 2.3 temos um exemplo de um autómato em forma nard.

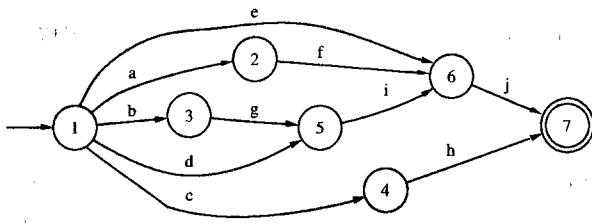


Figura 2.3: Autómato nard

**Definição 2.2.2 (Homeomorfismo de Grafos)** Dois grafos são homeomorfos se existir um terceiro a partir do qual ambos podem ser obtidos por uma sequência de operações em que um arco  $(u, v)$  é subdividido nos dois arcos  $(u, q)$  e  $(q, v)$ , sendo  $q$  um novo vértice [Har69].

**Definição 2.2.3 (Grafo  $D$ )** Definimos o grafo  $D$  como sendo  $D = (\{u_1, u_2, u_3, u_4\}, \{(u_1, u_2), (u_1, u_3), (u_2, u_3), (u_2, u_4), (u_3, u_4)\})$ .

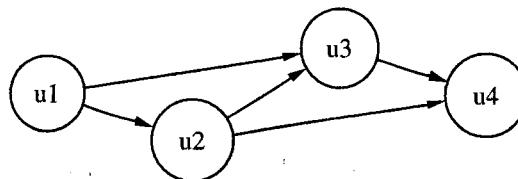


Figura 2.4: Grafo  $D$

**Proposição 2.2.1** Um autómato acíclico útil, com apenas um estado final é nard se e só se o seu grafo subjacente não contiver como subgrafo um grafo homeomorfo a  $D$ .

**Demonstração:** Seja  $G$  o grafo subjacente a um autómato acíclico útil, com apenas um estado final. Suponhamos que  $G$  contém como subgrafo um grafo homeomorfo a  $D$ . Claramente em  $G$  existem os vértices  $v_1$  e  $v_2$  da Definição 2.2.1, que satisfazem as condições 1 a 3 da mesma definição, nomeadamente, os vértices  $v_1$  e  $v_2$  da Definição 2.2.1 seriam respectivamente os vértices  $u_2$  e  $u_3$  da Definição 2.2.3. Se em  $G$  existir um ou mais vértices  $u$  nas condições da condição 4 da Definição 2.2.1, então, tomando  $v_2$  como sendo o vértice  $u$  de menor valor numa ordem topológica em  $G$ , temos satisfeita a Definição 2.2.1. Provámos assim que se um autómato se encontra em forma nard, então ele não contém como subgrafo um grafo homeomorfo a  $D$ .

Vamos agora provar que se  $G$  contém como subgrafo um grafo homeomorfo a  $D$  então não se encontra em forma nard. Suponhamos que em  $G$  existem vértices  $v_1$  e  $v_2$  nas condições da Definição 2.2.1. Tomemos o subgrafo de  $G$  definido por  $ini$  e vértices e arcos no caminho  $ini - \dots - v_1$  que não passa por  $v_2$ ,  $ini$  e vértices e arcos no caminho  $ini - \dots - v_2$  que não passa por  $v_1$ ,  $v_1$  e vértices e arcos no caminho  $v_1 - \dots - v_2$ ,  $v_1$  e vértices e arcos no caminho  $v_1 - \dots - fin$  que não passa por  $v_2$  e  $v_2$  e vértices e arcos no caminho  $v_2 - \dots - fin$  que não passa por  $v_1$ . Podemos concluir que este subgrafo é homeomorfo a  $D$ , nomeadamente, identificando, respectivamente,  $ini$ ,  $v_1$ ,  $v_2$  e  $fin$  com

$u_1, u_2, u_3$  e  $u_4$  da Definição 2.2.3. Estando provadas as implicações nos dois sentidos segue a veracidade da proposição.  $\square$

Foi feita uma contagem dos grafos nard de entre os grafos de até 7 vértices, acíclicos, com apenas um vértice  $u$  tal que  $ge(u) = 0$  e um vértice  $v$  tal que  $gs(v) = 0$  e úteis no sentido de que todos os vértices fazem parte de algum caminho de  $u$  para  $v$  e os resultados são apresentados na Tabela 2.1.

Tabela 2.1: Contagem de Grafos Nard

Vértices	Total	Não nards	Nards	Percentagem de nards
1	1	0	1	100%
2	1	0	1	100%
3	2	0	2	100%
4	8	2	6	75%
5	64	42	22	34,4%
6	1024	942	82	8%
7	32768	32470	298	0,9%

Vamos mostrar que se um autómato é nard então contém um estado  $v$  tal que  $ge(v) = 1 = gs(v)$ . Seja  $\mathcal{A} = (Q, \Sigma, \delta, ini, \{fin\})$  um autómato nard tal que  $gs(ini) > 1$  e  $ge(fin) > 1$ . Sejam  $u_1, \dots, u_m$  os vértices tais que  $\delta(ini, u_i) \neq \emptyset, 1 \leq i \leq m$  e  $U = \{u_1, \dots, u_m\}$ . Seja  $X_{u_i} = \{u_i\} \cup \{x \mid x \in Q \setminus \{fin\} \wedge \delta^+(u_i, x) \neq \emptyset\}$ . Seja  $R$  uma relação binária em  $U \times U$  tal que para  $u_1, u_2 \in U$  se tem  $u_1Ru_2$  se e só se  $X_{u_1} \cap X_{u_2} \neq \emptyset$ .

**Proposição 2.2.2** A relação  $R$  é uma relação de equivalência.

**Demonstração:** A reflexividade e simetria de  $R$  são triviais. Sejam  $u_1, u_2, u_3 \in U$  e tais que  $u_1Ru_2$  e  $u_2Ru_3$ . Então, pela definição de  $R$ , existe  $x \in Q \setminus \{fin\}$  tal que  $\delta^+(u_1, x) \neq \emptyset$  e  $\delta^+(u_2, x) \neq \emptyset$  (ou  $x = u_1$  ou  $x = u_2$ ) e existe  $y \in Q \setminus \{fin\}$  tal que  $\delta^+(u_2, y) \neq \emptyset$  e  $\delta^+(u_3, y) \neq \emptyset$  (ou  $y = u_2$  ou  $y = u_3$ ). Tomemos estes  $x$  e  $y$  como os dois vértices com menor valor numa ordem topológica  $\mathcal{T}$  em  $\mathcal{A}$  de entre os que obedecem

às mesmas condições. Vamos mostrar que tem de se ter  $\delta^+(x, y) \neq \emptyset$  ou  $\delta^+(y, x) \neq \emptyset$ , ou  $x = y$ , provando assim que  $X_{u_1} \cap X_{u_3} \neq \emptyset$ , ou seja,  $u_1 R u_3$ .

Se  $x = y$  ou  $x = u_2 = y$  é fácil concluir que  $X_{u_1} \cap X_{u_3} \neq \emptyset$ , nomeadamente,  $x \in X_{u_1} \cap X_{u_3}$ . Se  $x = u_2$  e  $x \neq y$  e por  $y \in X_{u_2} \cap X_{u_3}$  temos que  $\delta^+(x, y) \neq \emptyset$ , ou seja  $y \in X_{u_1} \cap X_{u_3}$ . Suponhamos, nos outros casos ( $x \neq u_2$ ), que  $\delta^+(x, y) = \emptyset$  e  $\delta^+(y, x) = \emptyset$  e  $x \neq y$ . Nestas condições, existe um caminho  $ini - \dots - x$  (o que passa por  $u_1$ ) que não passa por  $u_2$  por termos  $x \neq u_2$  e o valor de  $x$  em  $\mathcal{T}$  ser mínimo. O caminho  $ini - u_2$  não passa por  $x$ . De  $x$  existe um caminho para  $fin$  que não passa por  $u_2$  por termos  $\delta^+(u_2, x) \neq \emptyset$  e  $\mathcal{A}$  ser acíclico. Por  $u_2 R u_3$  temos um caminho  $u_2 - \dots - y - \dots - fin$  que não passa por  $x$  por termos  $\delta^+(x, y) = \emptyset$  e  $\delta^+(y, x) = \emptyset$ . Tomando respectivamente  $u_2$  e  $x$  como os vértices  $v_1$  e  $v_2$  da Definição 2.2.1, temos que, não existe nenhum vértice  $u$  nas condições do ponto 4 da Definição 2.2.1, pois se existisse, todos os caminhos de  $u_1$  e  $u_2$  para  $x$  teriam de passar por  $u$  e assim teríamos  $u \in X_{u_1} \cap X_{u_2}$  e  $u$  com menor valor em  $\mathcal{T}$  do que  $x$ , o que é uma contradição. Assim  $u_2$  e  $x$  seriam os vértices  $v_1$  e  $v_2$  da Definição 2.2.1, o que contradiz o facto de o autómato ser nard.  $\square$

**Proposição 2.2.3** *Seja, num autómato nard  $\mathcal{A}$ , tal que  $gs(ini) > 1$  e  $ge(fin) > 1$ , uma classe de equivalência  $[u_1]$  de  $R$  com mais que um elemento, então,  $\bigcap_{u \in [u_1]} X_u \neq \emptyset$*

**Demonstração:** Se  $|[u_1]| = 2$ , temos  $X_{u_1} \cap X_{u_2} \neq \emptyset$ , com  $u_1, u_2 \in [u_1]$ , pela definição de  $R$ . Suponhamos que a proposição é verdadeira para todas as classes de equivalência  $[u_1]$  tais que  $|[u_1]| < n$ . Seja  $[u_1]$  com  $n$  elementos. Tomemos  $n - 1$  elementos de  $[u_1]$ ,  $u_1, \dots, u_{n-1}$  e o subgrafo  $\mathcal{A}'$  de  $\mathcal{A}$  definido pelos vértices em  $X_{u_1} \cup \dots \cup X_{u_{n-1}} \cup \{ini, fin\}$  e arcos com ambas as extremidades neste conjunto.  $\mathcal{A}'$  tem apenas um estado inicial ( $ini$ ) e um estado final ( $fin$ ), é acíclico, por ser subgrafo de um grafo acíclico e por isto e por ser subgrafo de um grafo nard, também é nard. Neste subgrafo,  $u_1, \dots, u_{n-1}$  constituem a única classe de equivalência ( $[u_1]'$ ) de  $R$  e esta classe tem  $n - 1$  elementos, pelo que, por hipótese de indução,  $\bigcap_{u \in [u_1]'} X_u \neq \emptyset$ . Por se tratar de um subgrafo de  $\mathcal{A}$  temos que, em  $\mathcal{A}$ ,  $X = X_{u_1} \cap \dots \cap X_{u_{n-1}} \neq \emptyset$ . Por  $u_n \in [u_1]$  e pela definição de  $R$  temos  $X_{u_n} \cap X_{u_i} \neq \emptyset, 1 \leq i \leq n - 1$ . Seja  $v' \in X_{u_n} \cap X_{u_1}$  e  $v \in X$ . Em particular,  $v \in X_{u_1} \cap X_{u_2}$  e na prova da transitividade de  $R$  mostrámos que temos de ter  $\delta^+(v, v') \neq \emptyset$  ou  $\delta^+(v', v) \neq \emptyset$ . Se for  $\delta^+(v, v') \neq \emptyset$ , temos que  $v' \in \bigcap_{u \in [u_1]} X_u$  (pela transitividade de  $\delta^+$ ) e se for  $\delta^+(v', v) \neq \emptyset$  temos que  $v \in \bigcap_{u \in [u_1]} X_u$  (pela transitividade de  $\delta^+$ )

de  $\delta^+$ ), pelo que  $\bigcap_{u \in [u_1]} X_u \neq \emptyset$ . □

**Proposição 2.2.4** *Seja, num autómato nard  $\mathcal{A}$ , tal que  $gs(ini) > 1$  e  $ge(fin) > 1$ , uma classe de equivalência  $[u_1]$  de  $R$  com mais que um elemento e  $q \in \bigcap_{u \in [u_1]} X_u$  o estado com menor valor numa ordem topológica  $\mathcal{T}$  em  $\mathcal{A}$ , de entre os que obedecem às mesmas condições. Seja  $Z_q = \{z \mid z \in \bigcup_{u \in [u_1]} X_u \text{ e } \delta^+(z, q) \neq \emptyset\}$ . Então, para todo  $z \in Z_q$ , se  $c = z - s_1 - \cdots - s_n - fin$  é um caminho de  $z$  para  $fin$ , então existe  $1 \leq i \leq n$  tal que  $s_i = q$ , ou seja, todos os caminhos  $z - \cdots - fin$  passam por  $q$ .*

**Demonstração:** O caminho  $c = z - s_1 - \cdots - s_n - fin$ , se para  $1 \leq i \leq n, s_i \neq q$ , seria o caminho de  $v_1$  para  $fin$  sem passar por  $v_2$  da Definição 2.2.1 (tomando  $z$  como  $v_1$  e  $q$  como  $v_2$ ). Este caminho é a única condição da Definição 2.2.1 que falta satisfazer para que  $\mathcal{A}$  seja não nard, porque existe caminho de  $ini$  para  $z$  que não passa por  $q$ , pela definição de  $Z_q$  e por  $\mathcal{A}$  ser acíclico ( $z$  é antecessor de  $q$ ). Se todos os caminhos de  $ini$  para  $q$  passassem por  $z$  teríamos  $z \in \bigcap_{u \in [u_1]} X_u$  e o valor de  $z$  em  $\mathcal{T}$  seria menor que o de  $q$ , o que é uma contradição, pelo que, existe um caminho de  $ini$  para  $q$  que não passa por  $z$ . Por  $\mathcal{A}$  ser acíclico e por  $z \in Z_q$  ( $z$  é antecessor de  $q$ ) temos que, não existe nenhum caminho de  $q$  para  $z$ , e por  $\mathcal{A}$  ser útil, existe um caminho de  $q$  para  $fin$  que não passa por  $z$ . Se existisse um vértice  $v$  nas condições do ponto 4 da Definição 2.2.1, então teríamos  $v \in \bigcap_{u \in [u_1]} X_u$  e o valor de  $v$  em  $\mathcal{T}$  seria menor que o de  $q$ , o que seria uma contradição. Assim, o caminho  $c$ , se para  $1 \leq i \leq n, s_i \neq q$ , tornaria  $\mathcal{A}$  num autómato não nard, o que é uma contradição. □

**Proposição 2.2.5** *Num autómato nard existe pelo menos um vértice  $v$  com  $ge(v) = 1 = gs(v)$ .*

**Demonstração:** Os casos de autómatos com menos de 4 vértices são triviais. Seja  $\mathcal{A}$  um autómato nard com 4 vértices, só há seis e encontram-se enumerados na Figura 2.5 (ver página 32) e é fácil verificar que todos contêm um vértice  $v$  tal que  $ge(v) = 1 = gs(v)$ . Suponhamos que a proposição é verdadeira para todos os autómatos nard com menos de  $n$  estados. Seja  $\mathcal{A}$  um autómato nard com  $n$  estados. Se  $gs(ini) = 1$  e sendo  $v$  tal que o arco  $(ini, v)$  existe, então o subautómato que se obtém pela remoção de  $ini$  é nard (considerando  $v$  o novo  $ini$ ) e tem  $n - 1$  estados, donde, por hipótese de indução se conclui que este subautómato contém um vértice nas condições da proposição. O

autómato  $\mathcal{A}$  contém portanto um estado nas condições da proposição (o mesmo). Por um raciocínio análogo se pode concluir que se em  $\mathcal{A}$ ,  $ge(fin) = 1$ , então  $\mathcal{A}$  contém um estado nas condições da proposição. Suponhamos, portanto, que  $gs(ini) = m$  e  $ge(fin) = n$ , com  $m, n > 1$ .

Sejam  $u_1, \dots, u_m$ ,  $U$ ,  $X_{u_i}$  e  $R$  como definidos acima. Seja  $[u_1]$  uma classe de equivalência de  $R$  e analisemos os seguintes casos:

1.  $[u_1]$  tem apenas um elemento  $u_1$ . Seja  $Y = X_{u_1} \cup \{fin\}$ . Seja  $\mathcal{A}'$  o subautómato definido pelos vértices em  $Y$  (sendo  $u_1$  o estado inicial de  $\mathcal{A}'$  e  $fin$  o estado final).
  - (a) Por  $u_1$  ser o único elemento de  $[u_1]$  (significa que  $\delta^+(u_i, v) = \emptyset$ , para  $u_i \in U \setminus \{u_1\}$  e  $v \in X_{u_1}$ ), temos que, em  $\mathcal{A}'$ , apenas  $fin$  e  $u_1$  podem ter grau de entrada menor do que em  $\mathcal{A}$ .
  - (b) Por  $u_1$  ser o único elemento de  $[u_1]$  (significa que  $\delta^+(u_1, v) = \emptyset$ , para  $u_i \in U \setminus \{u_1\}$  e  $v \in X_{u_1}$ ), temos que, em  $\mathcal{A}'$ , nenhum estado tem grau de saída menor do que em  $\mathcal{A}$ .

Pela definição de  $X_{u_1}$  conclui-se que  $\mathcal{A}'$  é útil; pode igualmente concluir-se que  $\mathcal{A}'$  é acíclico e também que  $\mathcal{A}'$  é nard (por ser subgrafo de um grafo nard não são acrescentados caminhos).  $\mathcal{A}'$  é um autómato nard com menos de  $n$  estados pelo que, por hipótese de indução, tem um estado nas condições da proposição e este estado existe em  $\mathcal{A}$  com o mesmo grau de entrada e saída, por (a) e (b) (se  $\mathcal{A}'$  tiver apenas os estados  $u_1$  e  $fin$ , conclui-se que em  $\mathcal{A}$ ,  $u_1$  está nas condições da proposição).

2.  $[u_1]$  tem mais do que um elemento. Seja  $q \in \bigcap_{u \in [u_1]} X_u$  o estado com menor valor numa ordem topológica  $\mathcal{T}$  em  $\mathcal{A}$ , de entre os que obedecem às mesmas condições. Seja  $Z_q$  como na Proposição 2.2.4 e  $Y = Z_q \cup \{ini, q\}$ . Seja  $\mathcal{A}'$  o subautómato definido pelos vértices em  $Y$ , sendo  $ini$  o estado inicial de  $\mathcal{A}'$  e  $q$  o seu estado final. Pela definição de  $X_u$  e pela Proposição 2.2.4 pode concluir-se que  $\mathcal{A}'$  é útil; é fácil concluir que  $\mathcal{A}'$  é acíclico e, sendo subgrafo de um grafo nard,  $\mathcal{A}'$  é nard. Temos ainda que:
  - (a) Se existisse  $u_i \in U \setminus [u_1]$  tal que  $\delta^+(u_i, v) \neq \emptyset$  com  $v \in Y$  então ter-se-ia  $u_i \in [u_1]$  o que é absurdo, pelo que em  $\mathcal{A}'$ , nenhum estado tem grau de entrada menor do que em  $\mathcal{A}$ .

(b) Pela Proposição 2.2.4 temos que, em  $\mathcal{A}'$ , apenas  $ini$  e  $q$  podem ter grau de saída menor do que em  $\mathcal{A}$ , uma vez que em  $\mathcal{A}$  todos os caminhos, de qualquer antecessor de  $q$ , para  $fin$  passam por  $q$ .

$\mathcal{A}'$  é um autómato nard com menos de  $n$  estados, pelo que, por hipótese de indução, tem um estado nas condições da proposição e este estado existe em  $\mathcal{A}$  com o mesmo grau de entrada e saída, por (a) e (b).

□

**Teorema 2.2.1** *Seja  $\mathcal{A} = (Q, \Sigma, ini, \delta, fin)$  um autómato em forma nard. É possível aplicar a  $\mathcal{A}$  o algoritmo da eliminação de estados, removendo em cada passo um estado  $q$  tal que  $ge(q) = 1 = gs(q)$  e estes autómatos são os únicos para os quais esta propriedade se verifica.*

**Demonstração:** Pela Proposição 2.2.5,  $\mathcal{A}$  tem pelo menos um estado  $q$  nas condições do teorema. Se removermos  $q$  obtemos um autómato  $\mathcal{A}'$  que ainda se encontra em forma nard pois não acrescentamos em  $\mathcal{A}'$  nenhum caminho que possa formar um grafo homeomorfo a  $D$ . Por  $\mathcal{A}'$  ainda se encontrar em forma nard ele possui um vértice  $q'$  nas condições do teorema ou tem apenas dois estados, terminando o processo. Podemos aplicar o mesmo raciocínio a  $\mathcal{A}'$  e concluir a veracidade da propriedade indicada no teorema.

Suponhamos agora que aplicamos o AEE a um autómato que tenha um subgrafo homeomorfo a  $D$ . Em algum passo do AEE teremos de eliminar um dos vértices  $u_1, u_2, u_3$  ou  $u_4$  da definição do grafo  $D$  (Definição 2.2.3) e qualquer um destes vértices tem grau de entrada ou de saída superior a 1 e este grau não se reduz a 1 pela eliminação de outros vértices que não  $u_1, u_2, u_3$  ou  $u_4$ , pois, para que a linguagem reconhecida não se altere, terão de continuar a existir os caminhos entre os vértices  $u_1, u_2, u_3$  e  $u_4$  da Definição 2.2.3, pelo que os autómatos nard são os únicos para os quais a propriedade indicada se verifica. □

## 2.3 Implementação

Nesta secção, nas complexidades apresentadas,  $n$  significa sempre o número de vértices do grafo em causa. Seja `IsAcyclicGraph` uma função GAP para decidir se um grafo

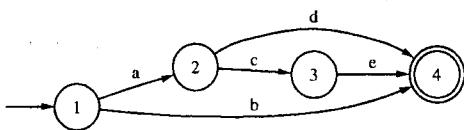
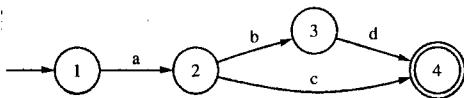
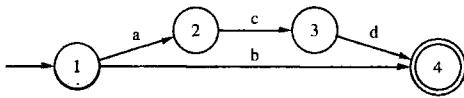
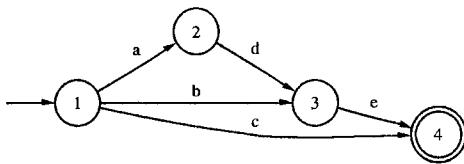
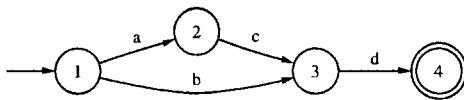
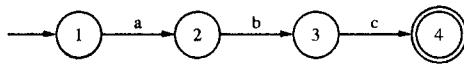


Figura 2.5: Autómatos nard de 4 estados

dado pela sua lista de adjacências é acíclico [KR99]. Esta função tem complexidade  $\mathcal{O}(n^2)$ . Seja `ExistsPathv1toV3Withoutv2` uma função GAP para decidir se num grafo dado pela sua lista de adjacências existe um caminho de um vértice  $v_1$  para um vértice  $v_3$  que não passe por um vértice  $v_2$ . Esta função tem complexidade  $\mathcal{O}(n)$ . Seja `DescendentsOfVertex` uma função GAP que, para um grafo dado pela sua lista de adjacências e um vértice  $v_1$  devolve a lista dos descendentes de  $v_1$ . Esta função tem complexidade  $\mathcal{O}(n^2)$ . Seja `ReverseOfGraph` uma função GAP que, para um grafo dado pela sua lista de adjacências devolve a lista de adjacências do grafo que tem os sentidos dos arcos invertidos. Esta função tem complexidade  $\mathcal{O}(n^2)$ .

Na Figura 2.6 apresentamos uma função GAP que, para o grafo, dado pela sua lista de adjacências, subjacente a um autómato com apenas um estado final, decide se o grafo é nard. Esta função, além do grafo subjacente, tem como *input* o estado inicial e o estado final do autómato. Pelas linhas de código seguintes podemos ver que esta função tem complexidade  $\mathcal{O}(n^5)$ .

```

for i in list do
    for j in Difference(list, [i]) do
        ...
        if ForAll(inter, x -> ExistsPathv1toV3Withoutv2(G,ini,j,x)) then
            ...
            fi;
        ...
    od;
od;

```

em que `list` é a lista de vértices do grafo excepto `ini` e `fin`, `Difference(list, [i])` é `list` sem o vértice `i`, `inter` é a lista dos vértices que são descendentes comuns dos vértices `i` e `j` e para todos os vértices `x` em `inter` é feito o teste `ExistsPathv1toV3Withoutv2(G,ini,j,x)`.

Na Secção A.2 (ver página 99) apresentamos uma versão do AEE específica para o caso de o AF de `input` ser nard e esta função tem complexidade  $\mathcal{O}(kn^3)$ , em que  $n$  é o número de estados do AF de `input` e  $k = |\Sigma|$ . Este valor é dominado pela conversão do AF inicial no seu AFE associado e esta complexidade deve-se às razões apresentadas na Secção 1.1.7.1 (ver página 17) para a complexidade da função `computeGTG`. Se o

```

IsNardGraph := function(G,ini,fin)
  local v, n, i, j, k, list, inter, Gr, flag;

  flag := false;
  if IsAcyclicGraph(G) then
    n := Length(G);
    list := Difference([1..n], [ini,fin]);
    for i in list do
      for j in Difference(list, [i]) do
        if ExistsPathvitov3Withoutv2(G,i,j,0) and
          ExistsPathvitov3Withoutv2(G,ini,i,j) and
          ExistsPathvitov3Withoutv2(G,ini,j,i) and
          ExistsPathvitov3Withoutv2(G,i,fin,j) and
          ExistsPathvitov3Withoutv2(G,j,fin,i) then
          Gr := ReverseOfGraph(G);
          inter := Intersection(
            DescendentsOfVertex(G,i),
            DescendentsOfVertex(Gr,j));
          if ForAll(inter, x ->
            ExistsPathvitov3Withoutv2(G,ini,j,x)) then
            return(false);
          fi;
        fi;
      od;
      return(true);
    else
      return(false);
    fi;
  end;

```

Figura 2.6: Verifica se um grafo é nard

*input* fosse já o AFE, então a função teria complexidade  $\mathcal{O}(kn^2)$ , pois este é tamanho máximo das expressões calculadas. A expressão resultado tem tamanho linear no número transições do autómato inicial.

Na Figura 2.7 apresentamos o corpo ”principal” desta implementação.

```
NardFAtoRatExp := function(A)

    # ... Definições das funções locais ...
    # Q é o número de estados do AFE
    computeGTG();  # Converte o autómato inicial
                    # no AFE equivalente

    while Q > 0 do
        for i in [3..Q] do
            if Nin[i] = 0 or Nout[i] = 0 or
                (Nin[i] = 1 and Nout[i] = 1) then
                destroyState(i);
                break;
            fi;
        od;
    od;
    # Devolve a expressão resultado ...
end;
```

Figura 2.7: NardFAtoRatExp

## 2.4 Autómatos de Thompson

Os autómatos de Thompson são aqueles que são obtidos pelo algoritmo de Thompson [Tho68] a partir de uma expressão regular e cuja caracterização se encontra em [GPW99]. Os autómatos de Thompson têm, entre outras características, apenas um estado inicial  $I$  com  $ge(I) = 0$  e um estado final  $F$  com  $gs(F) = 0$ . Os autómatos de Thompson de  $\emptyset$ ,  $\epsilon$  e  $a \in \Sigma$  são apresentados nas Figuras 2.8, 2.9 e 2.10 respectivamente.



Figura 2.8: Autómato de Thompson de  $\emptyset$

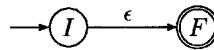


Figura 2.9: Autómato de Thompson de  $\epsilon$

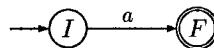


Figura 2.10: Autómato de Thompson de  $a \in \Sigma$

Os autómatos de Thompson que representam a união e concatenação de duas expressões  $r_1$  e  $r_2$  são apresentados nas Figuras 2.11 e 2.12, respectivamente e em que os estados  $I_1$  e  $F_1$  ( $I_2$  e  $F_2$ ) são respectivamente o estado inicial e final do autómato de Thompson que representa a expressão  $r_1$  ( $r_2$ ).

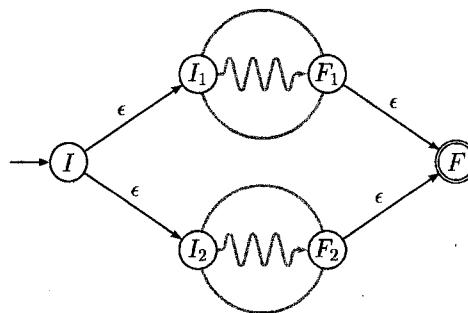


Figura 2.11: Autómato de Thompson representando uma união

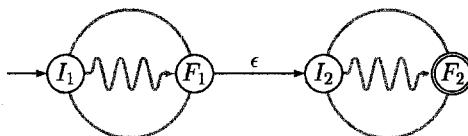


Figura 2.12: Autómato de Thompson representando uma concatenação

O autómato de Thompson que representa o fecho de Kleene de uma expressão regular  $r_1$  encontra-se representado na Figura 2.13.

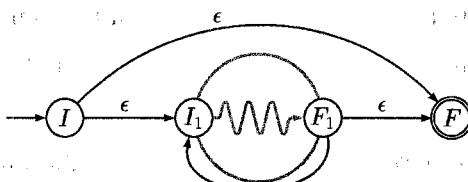


Figura 2.13: Autómato de Thompson representando o fecho de Kleene

A construção do autómato de Thompson que representa uma expressão regular  $r$  procede do seguinte modo [ASU86]:

1. se  $r = \emptyset$  ou  $r = \epsilon$  ou  $r = a, a \in \Sigma$  então constrói-se o autómato da Figura 2.8 ou 2.9 ou 2.10 respectivamente;
2. se  $r = r_1 + r_2$  então constroem-se os autómatos para  $r_1$  e  $r_2$  e usa-se o autómato da Figura 2.11 para representar  $r$ , em que  $I_1$  e  $F_1$  ( $I_2$  e  $F_2$ ) são respectivamente o estado inicial e final do autómato que representa  $r_1$  ( $r_2$ );
3. se  $r = r_1r_2$  então constroem-se os autómatos para  $r_1$  e  $r_2$  e usa-se o autómato da Figura 2.12 para representar  $r$ , em que  $I_1$  e  $F_1$  ( $I_2$  e  $F_2$ ) são respectivamente o estado inicial e final do autómato que representa  $r_1$  ( $r_2$ );
4. se  $r = r_1^*$  então constrói-se o autómato para  $r_1$  e usa-se o autómato da Figura 2.13 para representar  $r$ , em que  $I_1$  e  $F_1$  são respectivamente o estado inicial e final do autómato que representa  $r_1$ .

**Proposição 2.4.1** Os autómatos de Thompson de linguagens finitas são nard.

**Demonstração:** As linguagens finitas são representadas por expressões regulares que não contêm o fecho de Kleene. Vamos provar a proposição por indução no número de estados do autómato de Thompson. Os casos base são os das Figuras 2.8 a 2.10 e estes são nard. Suponhamos agora que a proposição é válida para todos os autómatos de Thompson, que representam linguagens finitas, com menos de  $n$  estados e seja  $\mathcal{A}$  um autómato de Thompson, de uma linguagem finita, com  $n$  estados. Por a linguagem ser finita, no último passo da construção de  $\mathcal{A}$  foi usado o autómato da Figura 2.11 ou 2.12 para juntar num só autómato dois autómatos de Thompson  $\mathcal{A}_1$  e  $\mathcal{A}_2$ . Por hipótese de indução, os subautómatos  $\mathcal{A}_1$  e  $\mathcal{A}_2$  são nard (têm menos de  $n$  estados). Ao agrupar estes dois autómatos num único que representa  $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$ , usando a construção da Figura 2.11, vemos que não é acrescentado nenhum arco, no autómato resultante, que forme um subgrafo homeomorfo a  $D$ . Igualmente, para o autómato que representa  $L(\mathcal{A}_1)L(\mathcal{A}_2)$ , usando a construção da Figura 2.12, não é acrescentado nenhum arco, no autómato resultante, que forme um subgrafo homeomorfo a  $D$ .  $\square$

## 2.5 Conclusões

Se durante a aplicação do AEE removermos um vértice  $v$  tal que  $ge(v) = 1$  e  $gs(v) = 2$  estamos a duplicar a ocorrência, na expressão regular final, da etiqueta do arco que entra em  $v$  (ver Figura 2.14), porque, seja  $u$  tal que  $\delta(u, v) \neq \emptyset$  e  $p$  e  $q$  tais que  $\delta(v, p) \neq \emptyset$  e  $\delta(v, q) \neq \emptyset$ , com a remoção de  $v$  estamos a substituir  $v$  e os arcos que nele incidem (entram e saem) pelos arcos tais que  $\delta(u, p) = \delta(u, v)\delta(v, p)$  e  $\delta(u, q) = \delta(u, v)\delta(v, q)$  (ver Figura 2.15).

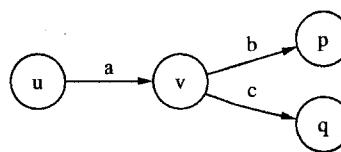


Figura 2.14: Remoção de um vértice  $v$  tal que  $ge(v) = 1$  e  $gs(v) = 2$

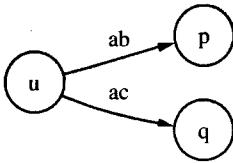


Figura 2.15: Após a remoção de  $v$

O único caso em que não acontece uma repetição de duas ou mais vezes de etiquetas de arcos é o caso da remoção de um vértice  $v$  tal que  $ge(v) = 1 = gs(v)$  (Figuras 2.16 e 2.17).

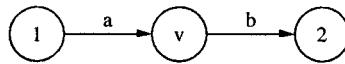


Figura 2.16:  $ge(v) = 1 = gs(v)$

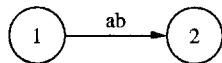


Figura 2.17: Após a remoção de  $v$

Em qualquer passo do AEE o tamanho da expressão final será sempre superior ou igual à soma dos tamanhos das etiquetas dos arcos presentes nesse passo, uma vez que a aplicação de um passo do AEE não faz diminuir a soma anterior à remoção de um vértice  $v$ , fazendo-a aumentar sempre, excepto no caso em que  $ge(v) = 1 = gs(v)$ , como veremos na Secção 3.1.

A versão do AEE para o caso nard (Secção A.2, ver página 99) tem como *output* a expressão regular equivalente e de menor tamanho obtinível pelo AEE para o autómato em causa, uma vez que em nenhum passo foi duplicada a etiqueta de algum arco. Neste caso o tamanho da expressão final seria igual ao número de arcos do autómato de *input* e é neste sentido que dizemos que a ordem de remoção é óptima.

O problema de encontrar uma expressão mínima obtinível pelo AEE para um autómato nard é resolvido em tempo linear no número de transições do AF inicial (assumindo que o *input* é já o AFE associado ao AF dado, uma vez que esta conversão

tem complexidade  $\mathcal{O}(kn^3)$  usando as representações GAP que temos vindo a referir) e é neste sentido que dizemos que a ordem óptima de remoção pode ser encontrada, neste caso, de forma eficiente.

Para terminar, consideremos o autómato não nard da Figura 2.18.

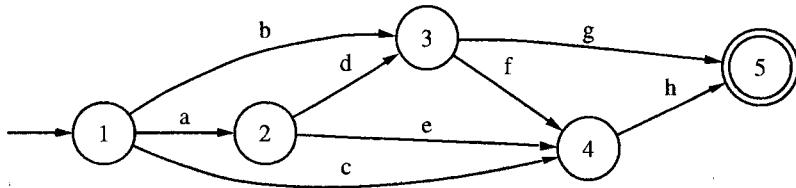


Figura 2.18: Autómato não nard

Neste autómato, consideramos  $ini = 1$  e  $fin = 5$  e a única ordem óptima de remoção é  $2, 4, 3$ . A expressão obtida pelo AEE para esta ordem de remoção é  $r = (ad + b)(fh + g) + (ae + c)h$ .

O autómato da Figura 2.19 é um autómato nard equivalente ao autómato que estamos a considerar e é mínimo no número de transições de entre os outros autómatos nard equivalentes, porque foi construído a partir de uma expressão minimal obtinível pelo AEE para o autómato da Figura 2.18, usando o algoritmo de Thompson e simplificando as transições por  $\epsilon$ .

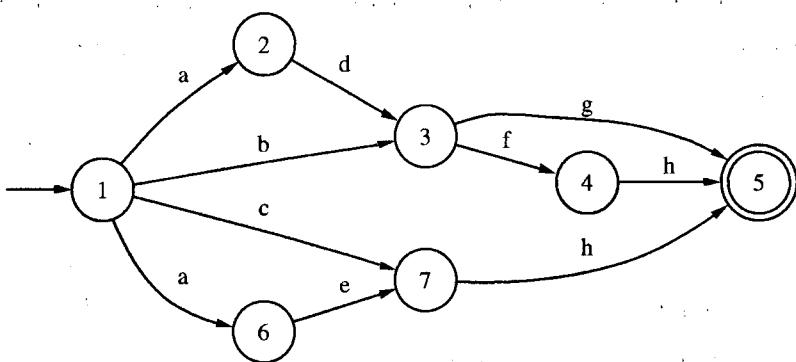


Figura 2.19: Autómato nard equivalente

Pelo que foi dito antes, deste autómato nard é possível encontrar de forma eficiente a ordem óptima de remoção para obter a expressão  $r$ , pelo que, em trabalho futuro,

gostaríamos de tentar resolver o Problema 2.5.1

**Problema 2.5.1 (Autómato nard mínimo equivalente)**

Dado um autómato  $\mathcal{A}$ , acíclico e com apenas um estado final, qual é um autómato nard  $\mathcal{A}'$  tal que  $L(\mathcal{A}') = L(\mathcal{A})$  e  $\mathcal{A}'$  é mínimo no número de transições?

Este problema é equivalente ao de encontrar uma expressão regular de tamanho mínimo que represente uma dada linguagem regular, pois, dada uma expressão regular é possível encontrar um autómato nard tal que, removendo em cada passo do AEE um estado com grau de entrada e saída 1 se obtém a expressão inicial. Este autómato nard terá um número de transições igual ao tamanho da expressão, portanto, se conseguirmos obter um autómato nard mínimo em transições que reconheça uma dada linguagem regular, conseguiremos obter a partir dele em tempo linear uma expressão que será de tamanho mínimo, pois se houvesse uma menor, existiria um autómato nard que reconheceria a mesma linguagem e teria menos transições.

que é o menor número de remoções necessárias para obter um AFE com apenas os estados *ini* e *fin*. Isto é, é o menor número de remoções necessárias para obter um AFE regular.

## Capítulo 3

# Uma heurística para o caso geral

Neste capítulo apresentamos uma heurística para o problema de encontrar uma ordem óptima de remoção dos vértices para o caso de um autómato arbitrário. Esta heurística surgiu no decurso dos trabalhos de estágio da licenciatura e foi apresentada em [DM04].

### 3.1 Pesos

Introduzimos a noção de *peso de um AFE*  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  como sendo

$$\mathcal{P}(\mathcal{A}) = \sum_{i=1}^Q \sum_{j=1}^Q \text{tamanho}(\delta(i, j)) - E$$

em que  $E$  é o número de transições etiquetadas por  $\epsilon$  (não contamos estas etiquetas, pois, sendo sempre feita a simplificação  $r\epsilon = r$  este peso nem sempre estará presente na expressão final) e de *peso de um vértice*  $v$  como sendo

$$\mathcal{P}(v) = \mathcal{P}(\mathcal{A}_1) - \mathcal{P}(\mathcal{A})$$

em que  $\mathcal{A}_1$  é o AFE resultante da remoção de  $v$  em  $\mathcal{A}$  pelo AEE, donde se conclui que

$$\mathcal{P}(\mathcal{A}_1) = \mathcal{P}(\mathcal{A}) + \mathcal{P}(v)$$

e em particular podemos notar que o tamanho da expressão regular, resultado do AEE, será  $\mathcal{P}(\mathcal{A}_n)$  em que  $\mathcal{A}_n$  é o AFE com apenas os estados *ini* e *fin*.

Como vimos na Secção 2.5, os únicos vértices  $v$  tais que  $\mathcal{P}(v) = 0$  são aqueles para os quais  $ge(v) = 1 = gs(v)$ . Mais precisamente, o peso de um vértice  $v$  pode ser calculado pela fórmula:

$$\begin{aligned}\mathcal{P}(v) = & \sum_{p \in E_v} (f(p, v) \times (gs(v) - 1)) + \\ & \sum_{q \in S_v} (f(v, q) \times (ge(v) - 1)) + \\ & h(v) \times (ge(v) \times gs(v) - 1),\end{aligned}\quad (3.1)$$

em que:

- $E_v = \{u \mid u \in Q \wedge \delta(u, v) \neq \emptyset\}$ ,
- $S_v = \{u \mid u \in Q \wedge \delta(v, u) \neq \emptyset\}$ ,
- $f : Q \times Q \rightarrow \mathbb{N}$  tal que  $f(p, q) = \text{tamanho}(\delta(p, q))$  e
- $h : Q \rightarrow \mathbb{N}$  tal que  $h(q) = \text{tamanho}(\delta(q, q))$ .

## 3.2 Heurística

A heurística consiste em eliminar, em cada passo do AEE, um vértice  $v$  que minimize  $\mathcal{P}(v)$ , uma vez que o tamanho da expressão regular resultado será igual à soma do peso do AFE inicial com o peso com que cada vértice é removido.  $\mathcal{P}(v)$  cresce com  $ge(v)$ ,  $gs(v)$  e os tamanhos das etiquetas. A eliminação de um vértice pode diminuir, preservar ou aumentar o número de arcos do AFE resultante, em relação ao AFE anterior. Seja  $v$  o vértice a eliminar e  $\mathcal{A}_1$  o AFE resultante da remoção de  $v$  de  $\mathcal{A}$ , o número de arcos em  $\mathcal{A}_1$  é

$$Narcos(\mathcal{A}_1) = Narcos(\mathcal{A}) - ge(v) - gs(v) + ge(v) \times gs(v). \quad (3.2)$$

Em face da Equação 3.2, vemos que o número de arcos se mantém apenas se

$$ge(v) \times gs(v) = ge(v) + gs(v),$$

ou seja, só acontece a igualdade se  $ge(v) = 2$  e  $gs(v) = 2$ .

Resumindo, se

- $ge(v) = 1$  (ou  $gs(v) = 1$ ) então o número de arcos vai diminuir com a remoção do vértice;
- $ge(v) = 2$  e  $gs(v) = 2$  então o número de arcos mantém-se;
- $ge(v) > 2$  e  $gs(v) > 1$  (ou vice-versa) então o número de arcos vai aumentar com a remoção do vértice.

A heurística favorece a escolha de um vértice que minimize o número de arcos criados pela sua remoção, fazendo com que o peso de alguns vértices possa diminuir após a remoção do vértice escolhido. De notar que o tamanho da expressão final será a soma do peso do AFE inicial com o peso de cada vértice quando removido.

### 3.3 Alguns Resultados Experimentais

Começámos por aplicar o AEE sem heurística a 512 autómatos diferentes, úteis, gerados aleatoriamente de 4, 5, 6 e 7 estados, obtendo os resultados da Tabela 3.1

Tabela 3.1: Alguns Resultados sem Heurística

Estados	N	Devolveu a min.	Média de erro	Média do tamanho min.
4	512	19,5%	13	25
5	512	3,3%	54	55
6	512	0,78%	219	115
7	512	0%	963	269

em que as colunas têm o seguinte significado:

1. N: número de autómatos;
2. Devolveu a min.: percentagem de casos em que foi devolvida uma expressão de tamanho mínimo;

3. Média de erro: média da diferença entre o tamanho da expressão devolvida e o tamanho mínimo, considerada apenas nos casos em que há diferença; e
4. Média do tamanho min.: média do tamanho da expressão mínima nos casos considerados em 3.

Na Tabela 3.2 temos os valores para os mesmos autómatos, mas usando a heurística descrita.

Tabela 3.2: Alguns Resultados com Heurística

Estados	N	Devolveu a min.	Média de erro	Média do tamanho min.
4	512	64,3%	4	25
5	512	44,5%	8	54
6	512	29,5%	14	111
7	512	24,0%	32	256

Apresentamos ainda, na Tabela 3.3, resultados obtidos com autómatos obtidos a partir do grafo de Cayley de alguns monóides de transformações. O grafo de Cayley de um monóide  $M$  é um grafo em que, para  $x, y \in M$  o arco  $(x, y)$  está presente se e só se existir um gerador  $a$  de  $M$  tal que  $x \cdot a = y$  e no autómato construído a partir do grafo de Cayley de  $M$  a etiqueta do arco  $(x, y)$  é  $a$ .

Sejam  $k_1, k_2, \dots, k_n \in \{1, \dots, n\} \cup \{\#\}$ . Denotamos por  $[k_1, k_2, \dots, k_n]$  a função parcial  $f$  de  $\{1, \dots, n\}$  em si mesmo, tal que para todo  $i \in \{1, \dots, n\}$ ,  $f(i) = k_i$  se  $k_i \neq \#$  e  $f(i)$  não está definida caso contrário.

Consideraremos os seguintes monóides:

- $\mathcal{POL}_4$ : gerado por  $[\#, 1, 2, 3], [1, 2, 4, \#], [1, 3, \#, 4]$  e  $[2, \#, 3, 4]$ ;
- $\mathcal{POL}_5$ : gerado por  $[\#, 1, 2, 3, 4], [1, 2, 3, 5, \#], [1, 2, 4, \#, 5], [1, 3, \#, 4, 5], [2, \#, 3, 4, 5]$ ;

- $\mathcal{POPI}_4$ : gerado por  $[2, 3, 4, 1]$  e  $[1, 2, 4, \#]$ ;
- $\mathcal{POPI}_5$ : gerado por  $[2, 3, 4, 5, 1]$  e  $[1, 2, 3, 5, \#]$ .

Na Tabela 3.3,  $Min(M[q_0, f])$  é o autómato determinístico mínimo equivalente ao autómato que tem como grafo subjacente o grafo de Cayley do monóide  $M$  e considerando o estado  $q_0$  como inicial e o estado  $f$  como o único estado final. Os tempos dos resultados obtidos, sem e com heurística, são da mesma ordem de grandeza.

Tabela 3.3: Tamanhos das expressões para alguns Grafos de Cayley

Autómato	Estados	Alfabeto	Sem heurística	Com heurística
$\mathcal{POI}_4[1, 20]$	70	4	12657	870
$Min(\mathcal{POI}_4[1, 20])$	16	4	1807	491
$\mathcal{POI}_5[1, 125]$	252	5	89456164	47532
$Min(\mathcal{POI}_5[1, 125])$	32	5	107438	8602
$\mathcal{POPI}_4[1, 60]$	141	2	5027	1344
$Min(\mathcal{POPI}_4[1, 60])$	33	2	8381	704
$\mathcal{POPI}_5[1, 70]$	631	2	1810719	43319
$Min(\mathcal{POPI}_5[1, 70])$	81	2	398620	11528

Foi também desenvolvida uma versão melhorada da heurística e cujos resultados para os autómatos mínimos da tabela anterior apresentamos na Tabela 3.4. Nesta tabela  $lk$  e  $nv$  são parâmetros da versão melhorada [DM04]. Consideremos uma árvore em que a raiz é o AFE inicial e cada descendente directo da raiz é o AFE obtido pela remoção de um vértice diferente, do AFE inicial. Cada descendente directo de cada um dos nós anteriores é o AFE obtido pela remoção de um vértice diferente, do AFE representado pelo nó pai. As folhas desta árvore representam o AFE com apenas os estados *ini* e *fin* para todas as ordens de remoção possíveis e se fosse possível calculá-la completamente, bastaria escolher, de entre as folhas, aquela que representasse a expressão regular de menor tamanho, sendo esta um expressão de tamanho mínimo obtenível pelo AEE. Dada a intratabilidade do cálculo de toda esta árvore, os parâmetros  $lk$  e  $nv$  permitem tratar uma árvore de menor tamanho, da seguinte forma:

1.  $lk$ : representa o número de níveis da árvore que iremos calcular e
2.  $nv$ : em vez de cada nó da árvore ter tantos descendentes directos como vértices tem o AFE representado por esse nó, este parâmetro permite definir que cada nó terá apenas  $nv$  descendentes directos; serão os definidos pela remoção dos  $nv$  vértices de menor peso.

Ao fim de  $lk$  níveis da árvore será escolhido para remoção o vértice cuja remoção deu origem ao AFE representado por um descendente directo da raiz da árvore que pertence a um caminho da raiz a uma folha que representa um AFE de menor peso.

Os tempos foram medidos num *Pentium 2,6GHz*.

Tabela 3.4: Tamanhos das expressões com a heurística melhorada

Autómato	Heurística melhorada	lk	nv	Tempo (segundos)
$Min(\mathcal{POI}_4[1, 20])$	286	7	4	4
$Min(\mathcal{POI}_5[1, 125])$	5269	7	4	21
$Min(\mathcal{POPI}_4[1, 60])$	423	5	4	3
$Min(\mathcal{POPI}_5[1, 70])$	7874	5	4	81

## 3.4 Implementação

Na Secção A.3 (ver página 107) apresentamos a função GAP que implementa a heurística descrita neste capítulo.

Na Figura 3.1 apresentamos o corpo "principal" desta implementação.

A função `computeGTG` é semelhante à apresentada na Secção 1.1.7.1, apenas com a diferença de que aqui são calculadas duas listas, `Invertices` e `Outvertices` que guardam, para cada estado  $v$ , os estados  $u$  tais que  $\delta(u, v) \neq \emptyset$  e  $\delta(v, u) \neq \emptyset$  respectivamente. Esta função é executada uma única vez e tem complexidade  $\mathcal{O}(kn^3)$ , pelas razões apresentadas na Secção 1.1.7.1 (ver página 17), no caso de o *input* ser um AFND ou um AFNDTE e  $\mathcal{O}(kn^2)$  no caso de um AFD, em que  $n$  é o

```

FAToRatExp := function(A)

    # ... Definições das funções locais ...
    # Q é o número de estados do AFE
    computeGTG(); # Converte o autómato inicial
                    # no AFE equivalente

    # ... Código para a remoção imediata dos
    # estados v tais que ge(v) = 1 = gs(v) ...
    while Q > 0 do
        V := computeBest(); # Calcula o "melhor"
                            # vértice a ser removido

        # ... Código para actualizar estruturas de dados ...
        destroyLoop(V); # Elimina o lacete do estado V
        destroyState(V); # Elimina o estado V
    od;
    # ... Devolve a expressão resultado ...
end;

```

Figura 3.1: AEE com heurística para o caso geral

número de estados do AF inicial e  $k = |\Sigma|$ . A função `destroyLoop` é executada  $n$  vezes e tem complexidade  $\mathcal{O}(kn4^n)$ , pelas razões apresentadas na Secção 1.1.7.1. A função `destroyState` é executada  $n$  vezes e tem complexidade  $\mathcal{O}(n^2k4^n)$ , pelas razões apresentadas na Secção 1.1.7.1. A função `computeWeight` é executada no máximo  $n^2$  vezes e tem complexidade  $\mathcal{O}(n)$ . A função `computeBest` é executada  $n$  vezes e tem complexidade  $\mathcal{O}(n^2)$ , pelo que o algoritmo tem complexidade  $\mathcal{O}(n^3k4^n)$ .

### 3.5 Conclusões

Nem sempre a heurística apresentada conduz à obtenção da expressão regular de tamanho mínimo obtinível pelo AEE. Consideremos de novo o autómato apresentado na Figura 3.2. Na Tabela 3.5 apresentamos o peso do AFE e de cada vértice.

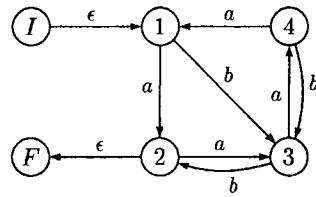


Figura 3.2: Autómato  $\mathcal{A}$

Tabela 3.5: Pesos Iniciais

	Peso
AFE	7
1	3
2	3
3	7
4	1

Nas Figuras 3.3 a 3.6 apresentamos os vários passos do AEE removendo em cada passo um dos vértices de menor peso e nas Tabelas 3.6 a 3.8 os pesos associados.

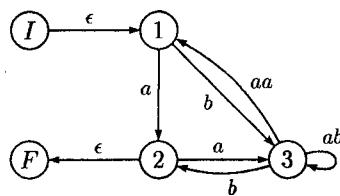


Figura 3.3: Autómato  $\mathcal{A}$  depois de removido o vértice 4

Tabela 3.6: Pesos no autómato  $\mathcal{A}$  depois de removido o vértice 4

	Peso
AFE	8
1	4
2	3
3	11

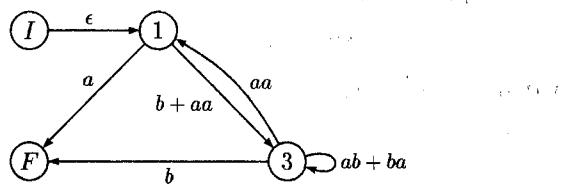


Figura 3.4: Autómato  $\mathcal{A}$  depois de removidos os vértices 4 e 2

Tabela 3.7: Pesos no autómato  $\mathcal{A}$  depois de removidos os vértices 4 e 2

	Peso
AFE	11
1	6
3	7

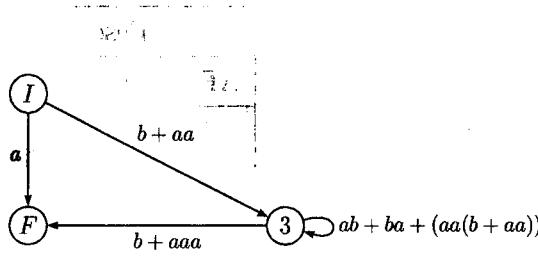


Figura 3.5: Autómato  $\mathcal{A}$  depois de removidos os vértices 4, 2 e 1

Tabela 3.8: Pesos no autómato  $\mathcal{A}$  depois de removidos os vértices 4, 2 e 1

	Peso
AFE	17
3	0

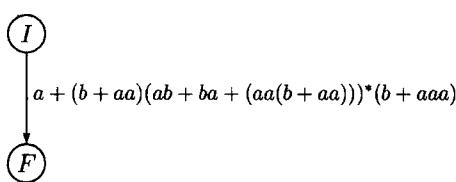


Figura 3.6: Autómato  $\mathcal{A}$  depois de removidos os vértices 4, 2, 1 e 3

Reparamos assim que o tamanho da expressão obtida usando a heurística apresentada tem tamanho 17. A expressão mínima obtinível pelo AEE para este autómato tem tamanho 16 e a sua obtenção foi ilustrada nas Figuras 1.4 a 1.7 (ver página 20). Apresentamos nas Tabelas 3.9 a 3.11 os pesos associados em cada passo.

Tabela 3.9: Pesos no autómato  $\mathcal{A}$  depois de removido o vértice 2

	Peso
AFE	10
1	5
3	12
4	1

Tabela 3.10: Pesos no autómato  $\mathcal{A}$  depois de removidos os vértices 2 e 1

	Peso
AFE	15
3	16
4	1

Tabela 3.11: Pesos no autómato  $\mathcal{A}$  depois de removidos os vértices 2, 1 e 4

	Peso
AFE	16
3	0

Podemos ver na Tabela 3.12 os pesos com que cada vértice foi removido em cada uma das duas ordens de remoção anteriores.

Tabela 3.12: Peso com que cada vértice foi removido

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

Representamos por  $\mathcal{P}_v(u)$  o peso do vértice  $u$  depois de removido o vértice  $v$ . Pode, assim, dar-se o caso de, para um AFE no decurso do AEE, existirem vértices  $v$ ,  $u$  e  $z$  tais que  $\mathcal{P}(v) < \mathcal{P}(u) < \mathcal{P}(z)$  e em que após a remoção de  $v$  tenhamos  $\mathcal{P}_v(u) = \mathcal{P}(u)$  e  $\mathcal{P}_v(z) = \mathcal{P}(z) + m$  e após a remoção de  $u$  tenhamos  $\mathcal{P}_u(v) = \mathcal{P}(v)$  e  $\mathcal{P}_u(z) = \mathcal{P}(z) + n$  e estes valores tais que  $\mathcal{P}(v) + \mathcal{P}_v(u) + \mathcal{P}_{vu}(z) > \mathcal{P}(u) + \mathcal{P}_u(v) + \mathcal{P}_{uv}(z)$ .

## Capítulo 4

# Uma heurística para o caso acíclico

Neste capítulo apresentamos uma heurística para a escolha do melhor vértice a ser removido em cada passo do AEE, que é específica para o caso de autómatos acíclicos e que tem resultados substancialmente melhores que os da heurística para o caso geral.

### 4.1 Motivação

Se aplicarmos o AEE a um autómato sem preocupação com a ordem de remoção dos vértices podemos obter como resultado uma expressão regular de tamanho muito superior ao tamanho mínimo obtinível pelo AEE. Por exemplo, para o autómato de 16 estados (ver Secção 3.3, página 45)  $\text{Min}(\mathcal{POI}_6[1, 56])$  em que  $\mathcal{POI}_6$  é gerado por  $[1, 2, 3, 4, 5, 6, \#]$ ,  $[\#, 1, 2, 3, 4, 5, \#]$ ,  $[1, 2, 3, 4, 6, \#, \#]$ ,  $[1, 2, 3, 5, \#, 6, \#]$ ,  $[1, 2, 4, \#, 5, 6, \#4]$ ,  $[1, 3, \#, 4, 5, 6, \#]$  e  $[2, \#, 3, 4, 5, 6, \#]$ , o programa AMoRE [MMP<sup>+</sup>95], que não inclui uma escolha especial do vértice a remover, produz uma expressão com tamanho 157675, ao passo que a heurística melhorada (apresentada no fim da Secção 3.3) com parâmetros  $lk = 7$  e  $nv = 4$  produz uma expressão equivalente de tamanho 394 (em 3 segundos)!

A heurística apresentada neste capítulo tenta obter subexpressões da expressão final da forma  $r_1(r_2+r_3)$  ou  $(r_1+r_2)r_3$  em vez de, respectivamente,  $r_1r_2+r_1r_3$  ou  $r_1r_3+r_2r_3$  e estas duas simplificações são as únicas que ocorrem no caso acíclico (além de  $r_1+r_2 = r_2$  se  $r_1 \subseteq r_2$ , que não tratámos dada a complexidade da verificação da inclusão), pois nestes casos não há a presença do operador  $*$ .

## 4.2 Heurística

A versão do AEE que implementámos para o caso acíclico começa por remover todos os estados  $q$  tais que  $ge(q) = 1 = gs(q)$ . Em seguida, se ainda houver vértices por remover, aplica a heurística para escolher o próximo vértice a ser removido. Nesta heurística, a função `computeWeight(q)` calcula o peso do estado  $q$  de acordo com a Equação (3.1) (ver página 44), que denotamos por  $\mathcal{P}(q)$ . A função `computeWDist(q)` calcula o peso que será "poupado" pela remoção de  $q$ , ou seja, com a remoção de  $q$  estamos a construir uma subexpressão da expressão final da forma  $(r_1 + \dots + r_n)(s_1 + \dots + s_m)$  em vez de  $r_1s_1 + \dots + r_ns_1 + \dots + r_ms_m$ . Esta função identifica, para um estado  $q$ , estados  $i$  e  $j$  tais que os arcos  $(i, q), (q, j)$  e  $(i, j)$  existem. Para cada par  $i, j$  encontrado, adiciona a um contador  $w$  o tamanho da etiqueta de cada arco que entra em  $i$  e em seguida o tamanho das etiquetas dos arcos que partem de  $j$  e retorna o valor deste contador. Na Figura 4.1 vemos um grafo etiquetado nestas condições.

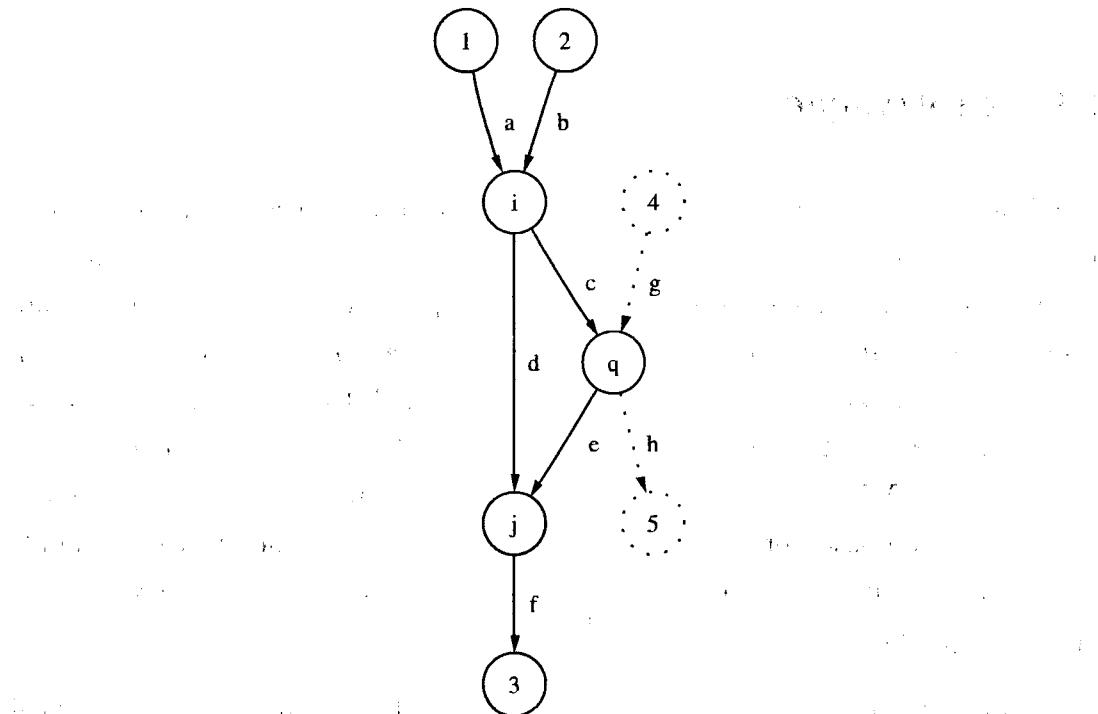


Figura 4.1: Grafo Etiquetado

Se, neste grafo, removermos  $q$  (supondo que os estados 4 e 5 não estão presentes),

passamos a ter  $gs(i) = 1$ , como se pode ver na Figura 4.2.

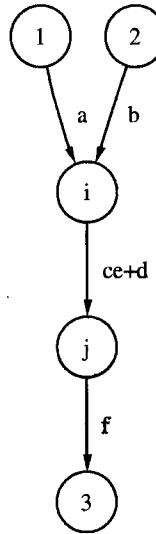


Figura 4.2: Grafo Etiquetado depois de removido  $q$

Nestas condições estaremos a obter a subexpressão  $(a + b)(ce + d)$  em vez de, se removéssemos o vértice  $i$ ,  $ad + ac + bd + bc$ .

Na função `computeWDist(q)` é também calculado o valor de um bónus, que se pretende máximo. Este bónus reflecte o facto de, ao remover  $q$ , estarmos a reduzir o peso de outro vértice a 0, ou seja, "ganhamos" o peso desse vértice antes da remoção de  $q$ , pois como já vimos, a remoção de um vértice de peso 0 não faz aumentar o tamanho da expressão final. No grafo da Figura 4.1 suponhamos que o estado 4 não está presente, ou seja,  $ge(q) = 1$ , neste caso, o bónus de  $q$  seria aumentado de  $\mathcal{P}(j)$ , pois com a remoção de  $q$  teríamos, no autómato resultante,  $\mathcal{P}(j) = 0$ , ou seja, o vértice  $j$  seria removido com peso 0, o que, como já vimos, é o ideal para a obtenção de uma expressão com o menor tamanho.

Formalmente, seja

$$X_q = \{(i, j) \mid i, j \in Q, \delta(i, q) \neq \emptyset, \delta(q, j) \neq \emptyset \text{ e } \delta(i, j) \neq \emptyset\},$$

para cada  $(i, j) \in X_q$  sejam

$$I_{(i,j)} = \{u \mid u \in Q \text{ e } \delta(u, i) \neq \emptyset\},$$

$$SI_{(i,j)} = \sum_{u \in I_{(i,j)}} \text{tamanho}(\delta(u, i)),$$

$$O_{(i,j)} = \{v \mid v \in Q \text{ e } \delta(j, v) \neq \emptyset\}$$

e

$$SO_{(i,j)} = \sum_{v \in O_{(i,j)}} \text{tamanho}(\delta(j, v)),$$

então

$$WD(q) = \sum_{(i,j) \in X_q} SI_{(i,j)} + SO_{(i,j)}.$$

Na Figura 4.3 apresentamos o código da função `computeWDist()`.

Podemos notar que, removendo o vértice  $q$  (Figura 4.1) diminuimos de 1 o número de arcos que partem de  $i$  (se `Nout[q] = 1`, ou seja,  $gs(q) = 1$ ). Esta diminuição faz com que para  $\mathcal{P}(i)$  não contribua a quantidade  $SI_{(i,j)}$ .

Simetricamente, removendo  $q$ , diminuimos em 1 o número de arcos que entram em  $j$  (se `Nin[q] = 1`, ou seja,  $ge(q) = 1$ ), pelo que para  $\mathcal{P}(j)$  não contribui a  $SO_{(i,j)}$ .

Assim, aparentemente, faz sentido que se tente escolher para remoção, um vértice que maximize  $WD(q)$ , pois isso pode significar diminuição no número de subexpressões repetidas no resultado, uma vez que podemos estar a diminuir  $gs(i)$  e/ou  $ge(j)$ .

Na função `computeDistGain(q)`, para estados  $q$  e  $t$  tais que existe o arco  $(q, t)$  e se para todos os vértices  $p$  tais que se o arco  $(p, q)$  existe, então existe o arco  $(p, t)$ , associa-se ao estado  $t$  o valor de  $gs(t)$ , pois se removermos  $q$  nestas condições, o grau de entrada de  $t$  reduz-se em uma unidade, diminuindo assim o peso de  $t$ . O valor assim associado por esta função ao estado  $t$  é guardado em `dist_gain[t]`. Formalmente, sejam

$$E_q = \{u \mid u \in Q \text{ e } \delta(u, q) \neq \emptyset\},$$

$$S_q = \{u \mid u \in Q \text{ e } \delta(q, u) \neq \emptyset\},$$

então, para cada  $t \in S_q$

$$\text{dist\_gain}[t] = \text{dist\_gain}[t] + gs(t)$$

se  $\forall s \in E_q (t \in S_s)$  e, inicialmente,  $\text{dist\_gain}[t] = 0$  para todos os  $t \in Q$ . Simetricamente, para cada  $s \in E_q$

$$\text{dist\_gain}[s] = \text{dist\_gain}[s] + ge(s)$$

se  $\forall t \in S_q (s \in E_t)$ .

```

computeWDist := function(q)
    local i, j, k, w;

    w := 0;
    # Invertices[q] é a lista de estados p tais que o arco
    #           (p,q) existe
    # Outvertices[q] é a lista de estados p tais que o arco
    #           (q,p) existe
    # Nin[q]      é |Invertices[q]| = ge(q)
    # Nout[q]     é |Outvertices[q]| = gs(q)
    # WT[k][i]    é tamanho(delta(k,i))

    for i in Invertices[q] do
        for j in Outvertices[q] do
            if j in Outvertices[i] then
                if Nout[j] = 1 and Nin[j] = 2 and Nin[q] = 1 then
                    bonus[q] := bonus[q] + computeWeight(j);
                fi;
                if Nout[i] = 2 and Nin[i] = 1 and Nout[q] = 1 then
                    bonus[q] := bonus[q] + computeWeight(i);
                fi;
                for k in Invertices[i] do
                    w := w + WT[k][i];
                od;
                for k in Outvertices[j] do
                    w := w + WT[j][k];
                od;
            fi;
        od;
    od;
    return(w);
end;

```

Figura 4.3: computeWDist()

```

computeDistGain := function(q)
    local s, t;

    dist_gain := List([1..Q], x->0);
    for t in Outvertices[q] do
        if ForAll(Outvertices[q], x -> t in Outvertices[x]) then
            dist_gain[t] := dist_gain[t] + Nout[t];
        fi;
    od;
    for s in Invertices[q] do
        if ForAll(Outvertices[q], x -> s in Invertices[x]) then
            dist_gain[s] := dist_gain[s] + Nin[s];
        fi;
    od;
end;

```

Figura 4.4: computeDistGain()

Na Figura 4.4 apresentamos o código da função `computeDistGain()`.

Para a escolha do estado a ser removido é construída, na função `computeBest()`, uma lista  $l = \{(val, i) \mid i \in Q \text{ e } val = WD(i) - \mathcal{P}(i) + bonus[i]\}$ . Esta lista é então ordenada por valor crescente do primeiro elemento de cada par membro da lista (se dois pares tiverem o primeiro elemento igual então a ordenação é feita pelo segundo elemento, ou seja, o número do estado). Sejam  $s$  o segundo elemento do penúltimo par desta lista e  $t$  o segundo elemento do último par desta lista. Em seguida é chamada a função `computeDistGain(s)` e  $dg$  é a lista `dist_gain` assim calculada. Como passo seguinte é chamada a função `computeDistGain(t)`, ficando calculada nova lista `dist_gain`. Se com a remoção de  $s$  o valor de  $dg[t]$  for superior ao valor de  $dist\_gain[s]$  se removido o vértice  $t$ , então é removido o vértice  $s$ , caso contrário é removido o vértice  $t$ .

A função de escolha é apresentada na Figura 4.5, onde  $Q$  é o número de estados do AFE.

```

computeBest := function()
    local i, j, l, s, t, dg;

    if Q = 2 then
        return(1);
    fi;
    bonus := List([1..Q], x->0);
    l := [];
    j := 0;
    for i in [3..Q] do
        AddSet(l, [computeWDist(i)-computeWeight(i) + bonus[i], i]);
        j := j + 1;
    od;
    if j > 1 then
        s := l[j-1][2];
        t := l[j][2];
        computeDistGain(s);
        dg := StructuralCopy(dist_gain);
        computeDistGain(t);
        if dg[t] > dist_gain[s] then
            return(s);
        else
            return(t);
        fi;
    else
        return(l[j][2]);
    fi;
end;

```

Figura 4.5: computeBest()

### 4.3 Exemplo

Para o autómato da Figura 4.6 (o número 847 refere-se a uma enumeração feita por nós) a única ordem óptima de remoção é 5, 3, 4, 6.

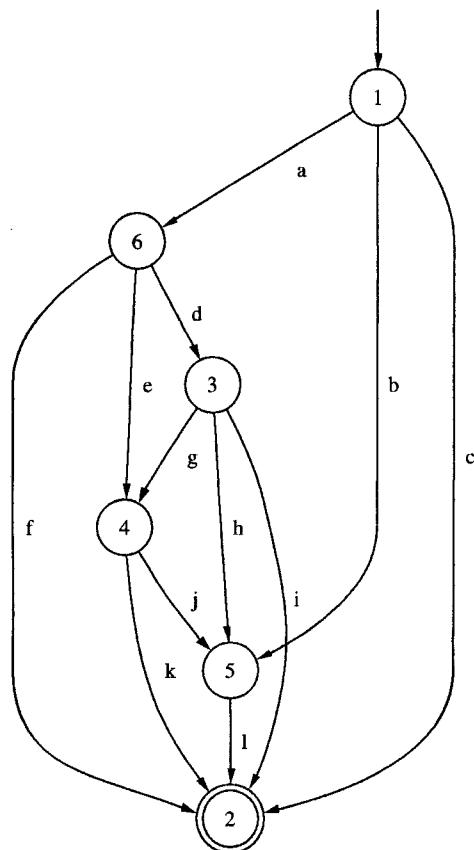


Figura 4.6: Autómato 847

A heurística apresentada neste capítulo basear-se-ia nos valores apresentados nas Tabelas 4.1 e 4.2 para escolher o primeiro vértice a ser removido.

Tabela 4.1: Valores para escolha do primeiro vértice a ser removido

i	WD(i)	$\mathcal{P}(i)$	bonus[i]	WD(i) - $\mathcal{P}(i)$ + bonus[i]
3	4	2	0	2
4	4	4	0	0
5	3	2	0	1
6	0	2	0	-2

4

Tabela 4.2:  $dist\_gain[i]$

i	dist-gain[i]
3	1
5	0

A escolha feita seria o vértice 5, uma vez que 3 e 5 são os dois vértices com maior valor na última coluna da Tabela 4.1 e  $dist\_gain[5] < dist\_gain[3]$ .

O AFE resultante da remoção do estado 5 é o da Figura 4.7 e para este AFE mostramos nas Tabelas 4.3 e 4.4 os valores que orientam a escolha do segundo vértice a ser removido.

Tabela 4.3: Valores para escolha do segundo vértice a ser removido

i	WD(i)	$\mathcal{P}(i)$	bonus[i]	WD(i) - $\mathcal{P}(i)$ + bonus[i]
3	5	1	3	7
4	2	3	1	0
6	0	2	0	-2

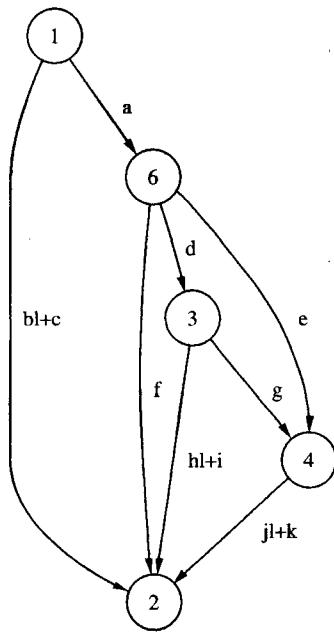


Figura 4.7: Autómato 847 depois de removido o estado 5

Tabela 4.4:  $dist\_gain[i]$

i	$dist\_gain[i]$
3	1
4	1

Aqui seria escolhido o estado 3 para remoção pois o valor da última coluna da Tabela 4.3 é máximo para este estado e entre 3 e o estado com segundo maior valor nessa coluna (4) temos  $dist\_gain[3] = dist\_gain[4]$ .

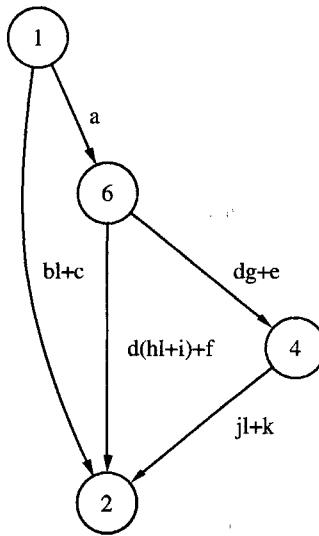


Figura 4.8: Autómato 847 depois de removidos os estados 5 e 3

O AFE obtido depois de remover o estado 3 é o da Figura 4.8. A partir deste AFE não seriam feitas mais escolhas porque o estado 4 seria eliminado imediatamente por se ter  $ge(4) = 1 = gs(4)$  e em seguida seria removido o vértice 6 pelo mesmo motivo, tendo sido assim conseguida a ordem óptima de remoção 5, 3, 4, 6.

Neste autómato, a heurística para o caso geral removeria os vértices na ordem 3, 6, 4, 5, resultando numa expressão com mais 1 símbolo que a mínima. Variações desta heurística para o caso acíclico foram tentadas, mas com piores resultados.

#### 4.4 Alguns Resultados Experimentais

Para todos os grafos subjacentes de autómatos de 1, 2, 3, 4, 5, 6 e 7 estados, acíclicos, úteis e com apenas um estado final foram aplicadas as duas heurísticas descritas, sendo os resultados apresentados nas Tabelas 4.5 e 4.6, onde as colunas 1 a 6 indicam a percentagem de casos em que a expressão devolvida tem mais 1, ... ou 6 símbolos, respectivamente, do que a expressão mínima e a coluna N indica o número de autómatos considerados.

Tabela 4.5: Heurística para o caso geral

estados	N	Atingiu min	1	2	3	4	5	6
1	1	100%						
2	1	100%						
3	2	100%						
4	8	100%						
5	64	93,8%	6,2%					
6	1024	76,4%	20,5%	2,7%	0,4%			
7	32768	52,0%	30,7%	11,8%	4,2%	1,0%	0,2%	0,1%

Tabela 4.6: Heurística para o caso acíclico

estados	N	Atingiu min	1	2	3	4	5	6
1	1	100%						
2	1	100%						
3	2	100%						
4	8	100%						
5	64	100%						
6	1024	97,5%	2,5%					
7	32768	85,5%	12,8%	1,6%	0,1%			

## 4.5 Implementação

Na Secção A.4 (ver página 111) apresentamos a função GAP que implementa a heurística descrita neste capítulo.

Na Figura 4.9 apresentamos o corpo "principal" desta implementação.

```
AcyclicFAToRatExp := function(A)

    # ... Definições das funções locais ...
    # Q é o número de estados do AFE

    computeGTG();
    removeImmediate();      # Remove os estados v tais que
                            # ge(v) = 1 = gs(v)

    while Q > 2 do
        q := computeBest(); # Calcula o "melhor"
                            # vértice a ser removido
        destroyState(q);   # Elimina o estado q
        removeImmediate();
    od;
    for i in [1..Q] do      # Remove os últimos 2 estados
        destroyState(1);
    od;
    # ... Devolve a expressão resultado ...
end;
```

Figura 4.9: AEE com heurística para o caso acíclico

A função `computeGTG` é executada uma única vez e tem complexidade  $\mathcal{O}(kn^3)$  no caso de o *input* ser um AFND ou um AFNDTE, pelas razões apresentadas na Secção 1.1.7.1 e  $\mathcal{O}(kn^2)$  no caso de um AFD, em que  $n$  é o número de estados do AF de *input* e  $k = |\Sigma|$ . A função `destroyState` é executada  $n$  vezes e a sua complexidade é determinada pelas linhas de código seguintes:

```
for p in Invertices[n] do
```

```

for q in Outvertices[n] do
    ... Actualização das etiquetas das transições ...
od;
od;

```

pelo que, para determinar a complexidade desta função é necessário majorar o tamanho das expressões regulares envolvidas. A maior expressão, obtenível pelo AEE, que representa a linguagem reconhecida por um autómato acíclico consiste numa união de concatenações, que é exactamente a união das expressões que representam um caminho do estado inicial para um estado final. Esta expressão terá tamanho máximo, para um autómato acíclico de  $n$  estados, se o seu grafo subjacente tiver todos os arcos possíveis, pois assim é garantido que o número de caminhos do estado inicial para qualquer estado final é máximo. A linguagem reconhecida por um autómato de  $n$  estados e com vários estados finais pode ser reconhecida por um autómato de  $n + 1$  estados em que, às transições existentes se acrescenta uma transição por  $\epsilon$  de cada estado final para o novo estado, passando este a ser o único estado final, pelo que nos restringiremos ao caso de apenas um estado final.

Chamemos GAUC - grafo acíclico, útil (no sentido de que há apenas um vértice  $s$  tal que  $ge(s) = 0$  e apenas um vértice  $t$  tal que  $gs(t) = 0$  e qualquer outro vértice faz parte de algum caminho de  $s$  para  $t$ ) e completo (no sentido de que existem todos os arcos possíveis).

**Proposição 4.5.1** *Num GAUC de  $n$  vértices o número de caminhos de  $s$  para  $t$  é  $2^{n-2}$ .*

**Demonstração:** Vamos provar a proposição por indução no número de vértices do GAUC. O caso base é o caso em que  $n = 2$  e neste caso é trivial verificar que há apenas um caminho, ou seja,  $2^{2-2} = 1$ , de  $s$  para  $t$ . Suponhamos que a proposição é verdadeira para todos os GAUCs com menos de  $n+1$  vértices. Seja  $G_n = (\{1, \dots, n\}, E_n)$  o GAUC de  $n$  vértices em que os vértices são nomeados de acordo com a única ordem topológica. O GAUC  $G_{n+1}$  de  $n+1$  vértices é o GAUC em que se adiciona um vértice  $n+1$  a  $G_n$  e um arco de cada vértice de  $G_n$  para o vértice  $n+1$  sendo este novo vértice o vértice  $t$  de  $G_{n+1}$ . Seja  $NC_G(p, q)$  o número de caminhos em  $G$  do vértice  $p$  para o vértice  $q$ . Em  $G_{n+1}$  o número de caminhos de  $s$  para  $t$  será

$$NC_{G_{n+1}}(1, n+1) = 1 + \sum_{i=2}^n NC_{G_n}(1, i). \quad (4.1)$$

A parcela 1 na Equação (4.1) deve-se ao arco  $(1, n+1)$ . Vamos mostrar que cada  $NC_{G_n}(1, i)$  na Equação (4.1) representa o número de caminhos de  $s$  para  $t$  num CAUC de  $i$  vértices. Seja  $G_i = (\{1, \dots, i\}, E_i)$  o subgrafo de  $G_n$  definido pelos vértices em  $\{1, \dots, i\}$  com  $2 \leq i \leq n$ . Seja  $\mathcal{T}_n$  a ordem topológica em  $G_n$  e  $\mathcal{T}_i$  a ordem topológica em  $G_i$ . Para todo  $u \in \{1, \dots, i\}$  temos  $\mathcal{T}_i(u) = \mathcal{T}_n(u)$ . Em  $G_i$  temos  $t = i$ .  $G_i$  é um GAUC de  $i$  vértices pois para quaisquer  $u, v \in \{1, \dots, i\}$  tais que  $\mathcal{T}_i(u) < \mathcal{T}_i(v)$  temos  $(u, v) \in E_i$  por  $\mathcal{T}_n(u) < \mathcal{T}_n(v)$  e  $G_n$  ser um GAUC. Em  $G_i$  o número de caminhos de  $s$  para  $t$  é exactamente  $NC_{G_n}(1, i)$  pois todos os caminhos de  $s$  para  $t$  em  $G_i$  passam apenas por vértices  $u$  tais que  $\mathcal{T}_n(u) < \mathcal{T}_n(i)$ . Por isto e por  $G_i$  ser um GAUC com menos de  $n+1$  vértices temos que, usando a hipótese de indução,

$$NC_{G_n}(1, i) = NC_{G_i}(s, t) = 2^{i-2},$$

podendo então reescrever-se a Equação (4.1) como

$$NC_{G_{n+1}}(1, n+1) = 1 + \sum_{i=2}^n 2^{i-2}, \quad (4.2)$$

ou seja,  $NC_{G_{n+1}}(1, n+1) = 1 + S_i$  em que  $S_i = 1 + 2 + 4 + 8 + \dots + 2^{i-2}$  com  $2 \leq i \leq n$ .  $S_i$  é uma progressão geométrica com razão 2. A fórmula para a soma dos primeiros  $n$  termos de uma progressão geométrica de razão  $r$  é [NV90]

$$S_n = u_1 \times \frac{1 - r^n}{1 - r},$$

em que  $u_1$  representa o primeiro termo da sucessão, pelo que

$$S_i = 1 \times \frac{1 - 2^{n-1}}{1 - 2},$$

em que o expoente  $n-1$  se deve ao facto de o primeiro termo de  $S_i$  ser para  $i=2$  e portanto estarmos a considerar de facto os primeiros  $n-1$  termos. Portanto  $S_i = 2^{n-1} - 1$ , donde  $NC_{G_{n+1}}(1, n+1) = 1 + 2^{n-1} - 1 = 2^{n-1} = 2^{(n+1)-2}$ , como queríamos mostrar.  $\square$

**Proposição 4.5.2** Dados um autómato  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  acíclico de  $n$  estados com apenas um estado final e uma expressão regular  $r$ , obtida por aplicação do AEE a  $\mathcal{A}$ , tem-se que  $tamanho(r) \leq k(n - 1)2^{n-2}$  em que  $k = |\Sigma|$ .

**Demonstração:** Para um autómato  $\mathcal{A}$  acíclico de  $n$  estados e apenas um estado final,  $r$  será de tamanho máximo se o grafo subjacente de  $\mathcal{A}$  for um GAUC de  $n$  estados e  $r$  for uma união das expressões que representam um caminho do estado inicial para o estado final. Pela Proposição 4.5.1 há exactamente  $2^{n-2}$  caminhos e portanto  $r$  será uma união de  $2^{n-2}$  expressões em que cada uma destas tem tamanho menor ou igual a  $k(n - 1)$ . O factor  $k$  deve-se ao facto de no AFE associado a  $\mathcal{A}$  cada transição ser etiquetada por uma expressão de tamanho menor ou igual a  $k$ . O factor  $n - 1$  deve-se ao facto de em  $\mathcal{A}$  o maior caminho ter tamanho menor ou igual a  $n - 1$ . Neste caso,

□

Pela Proposição 4.5.2 podemos concluir que a função `destroyState` tem complexidade  $\mathcal{O}(n^3k2^n)$ . Iremos agora determinar a complexidade da função `computeBest` (ver Figura 4.5), analisando a complexidade das funções por ela chamadas. A função `computeWeight` tem complexidade  $\mathcal{O}(n)$  e é executada no máximo  $n$  vezes em cada execução de `computeBest`. A função `computeWDist` é chamada por `computeBest` no máximo  $n$  vezes e tem complexidade  $\mathcal{O}(n^4)$  que se deve às seguintes linhas de código:

```

for i in Invertices[q] do
    for j in Outvertices[q] do
        if j in Outvertices[i] then
            ...
            for k in Invertices[i] do
                ...
            od;
            ...
        fi;
    od;
od;

```

A função `computeDistGain` é chamada por `computeBest` 2 vezes e tem complexidade  $\mathcal{O}(n^3)$  que se deve às linhas de código seguintes:

```

for t in Outvertices[q] do
    if ForAll(Invertices[q], x -> t in Outvertices[x]) then
        ...
    fi;
od;

```

em que, para um estado  $q$  e para cada estado  $t \in Outvertices[q]$  se verifica se para todos os estados  $x \in Invertices[q]$  se tem  $t \in Outvertices[x]$ . A função `computeBest` tem, portanto, complexidade  $\mathcal{O}(n^5)$  que se deve às seguintes linhas de código:

```

for i in [3..Q] do
    AddSet(l, [computeWDist(i)-computeWeight(i) + bonus[i], i]);
    ...
od;

```

em que  $Q$  é o número de estados do autómato e para cada um destes estados é adicionado, de forma ordenada, à lista  $l$  a lista  $[computeWDist(i)-computeWeight(i) + bonus[i], i]$ , em que `computeWDist(i)` tem complexidade  $\mathcal{O}(n^4)$ . A função `removeImmediate` tem complexidade  $\mathcal{O}(n)$  e é executada no máximo  $n$  vezes. Por isto, a heurística para o caso acíclico tem complexidade  $\mathcal{O}(n^4k2^n)$ .

## 4.6 Conclusões

Podemos ver pelas tabelas apresentadas na Secção 4.4 que a heurística para o caso acíclico produz resultados melhores que os da heurística para o caso geral, se aplicadas a autómatos acíclicos.

Para terminar, apresentamos um exemplo em que a heurística para o caso acíclico não consegue produzir a expressão mínima. Consideremos o autómato da Figura 4.10.

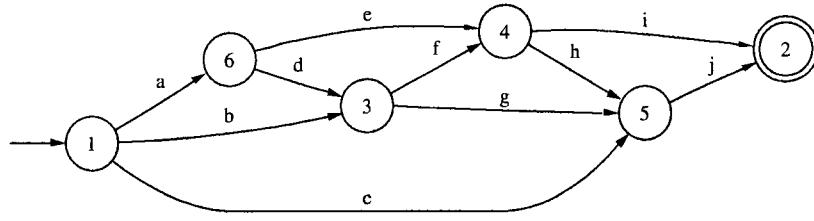


Figura 4.10: Autómato 663

Considerando neste autómato  $ini = 1$  e  $fin = 2$  temos que a única ordem óptima de remoção é 6, 3, 5, 4, obtendo-se a expressão  $r = ((ad+b)f+ae)(hj+i)+((ad+b)g+c)j$  em que  $tamanho(r) = 15$ .

Para a escolha do primeiro vértice a ser removido, a heurística deste capítulo basearia-se-ia nos valores das Tabelas 4.7 e 4.8.

Tabela 4.7: Valores para escolha do primeiro vértice a ser removido

i	WD(i)	$\mathcal{P}(i)$	bonus[i]	$WD(i) - \mathcal{P}(i) + bonus[i]$
3	4	4	0	0
4	3	4	0	-1
5	2	2	0	0
6	2	1	0	1

Tabela 4.8:  $dist\_gain[i]$

i	$dist\_gain[i]$
6	0
5	0

Assim, o primeiro vértice a ser removido seria o 6 e o AFE resultante é o da Figura 4.11.

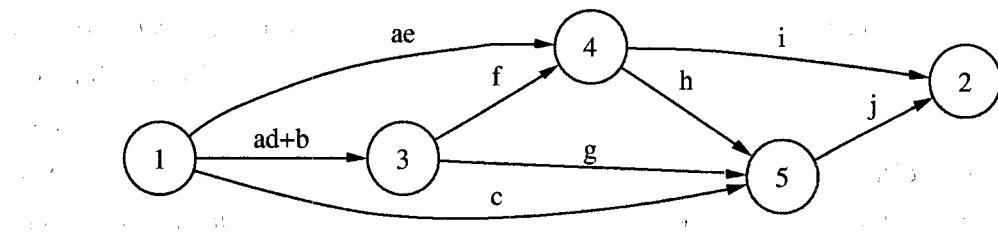


Figura 4.11: Autómato 663 depois de removido o vértice 6

Para a escolha do segundo vértice a ser removido temos os valores das Tabelas 4.9 e 4.10.

Tabela 4.9: Valores para escolha do segundo vértice a ser removido

i	WD(i)	$\mathcal{P}(i)$	bonus[i]	WD(i) - $\mathcal{P}(i)$ + bonus[i]
3	3	3	0	0
4	5	5	0	0
5	3	2	0	1

Tabela 4.10:  $dist\_gain[i]$

i	dist_gain[i]
4	2
5	1

O segundo vértice removido seria portanto o 5 e o AFE resultante é o da Figura 4.12.

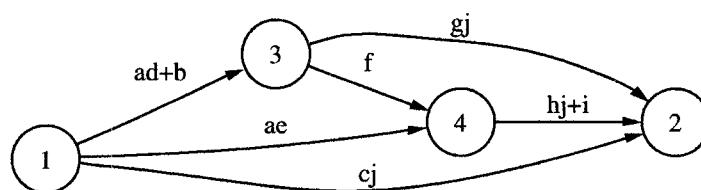


Figura 4.12: Autómato 663 depois de removidos os vértices 6 e 5

Vemos assim que na escolha do segundo vértice já há diferença em relação à ordem óptima. Notemos aqui que o estado 5 é o único com valor máximo na última coluna da Tabela 4.9 mas tanto o estado 4 como o 3 seriam elegíveis para comparação do valor de `computeDistGain(i)`. A escolha do estado 4 para figurar na Tabela 4.10 é de certo modo arbitrária, uma vez que é feita, em caso de igualdade, pelo índice maior. Se em vez do estado 4 tivessemos considerado o estado 3, em vez da Tabela 4.10 teríamos a Tabela 4.11

Tabela 4.11: `dist_gain[i]`

i	<code>dist_gain[i]</code>
3	0
5	1

e neste caso já seria escolhido correctamente (no sentido de corresponder à ordem óptima) o vértice 3. No entanto, foi tentada uma variação da heurística que reuniria neste caso, para comparação do valor de `computeDistGain(i)`, os estados 3, 4 e 5, e embora, para este autómato, o resultado fosse óptimo, no caso geral obtivemos piores resultados.

A partir daqui, os vértices removidos seriam o 4 e depois o 3, obtendo-se a expressão final  $s = (ad + b)(f(hj + i) + gj) + ae(hj + i) + cj$  em que  $tamanho(s) = 16$ , ou seja, a expressão obtida tem mais 1 símbolo que a mínima ( $r = ((ad + b)f + ae)(hj + i) + ((ad + b)g + c)j$ ).

que é o resultado da aplicação do AEE ao autómato  $A$ . O resultado é uma expressão regular que descreve a mesma linguagem que a linguagem gerada pelo autómato  $A$ . No Capítulo 1 mostrámos que a expressão regular obtida é equivalente ao menor autómato que gera a mesma linguagem.

## Capítulo 5

### Conclusões

Neste capítulo começaremos por fazer um resumo do trabalho apresentado e em seguida um delineamento do trabalho futuro. No Capítulo 1 apresentámos as noções necessárias à leitura deste trabalho, bem como a descrição do problema de encontrar, a partir de um autómato e usando o algortimo da eliminação de estados, uma expressão regular equivalente com o menor tamanho. Em particular tentámos salientar a dificuldade do problema, fazendo notar que a ordem de remoção dos estados, na aplicação do AEE, determina o tamanho da expressão regular obtida.

No Capítulo 2 apresentámos uma caracterização de autómatos acíclicos para os quais pode ser obtida pelo AEE uma expressão regular de tamanho linear no número de transições e em tempo linear também no número de transições do autómato em causa. Estes autómatos são aqueles para os quais o seu grafo subjacente não contém como subgrafo um grafo homeomorfo ao grafo  $D$  apresentado na Definição 2.2.3 (ver página 26) e para estes autómatos é possível eliminar em cada passo do AEE um estado  $q$  com  $ge(q) = 1 = gs(q)$  o que garante que a expressão regular obtida terá tamanho igual ao número de transições do autómato inicial. Apresentámos também um algoritmo para determinar se um dado autómato obedece à caracterização apresentada, bem como uma versão do AEE, que pode ser melhorada, específica para este tipo de autómatos. Mostrámos que os autómatos de Thompson de linguagens finitas obedecem à caracterização apresentada e deixámos em aberto o problema de encontrar, para uma dada linguagem, um autómato, do tipo caracterizado, que seja mínimo no número de transições.

No Capítulo 3 apresentámos uma heurística para o problema de encontrar, a partir de um autómato e usando o algortimo da eliminação de estados, uma expressão regular equivalente com o menor tamanho, que foi elaborada no decurso dos trabalhos de estágio da licenciatura. Esta heurística baseia-se na noção de peso de um vértice apresentada na Secção 3.1 (ver página 43) e consiste em eliminar primeiro os estados com menor peso. De notar que o tamanho da expressão final será igual à soma do peso do AFE inicial (definido na Secção 3.1) com o peso de cada estado quando removido. Esta heurística produz resultados consideravelmente melhores que os produzidos por uma aplicação do AEE que não tenha em consideração uma escolha adequada do estado a ser removido em cada passo. Apresentámos tabelas com alguns resultados obtidos e terminámos com o exemplo de um autómato para o qual a heurística não consegue produzir uma expressão de tamanho mínimo obtenível pelo AEE.

No Capítulo 4 apresentámos uma heurística com o mesmo objectivo que a descrita no Capítulo 3, mas específica para o caso de autómatos acíclicos. Para estes autómatos tentamos que na expressão final figurem subexpressões da forma  $a(b + c)$  ou  $(a + b)c$  em vez de  $ab + ac$  ou  $ac + bc$ , respectivamente, por uma escolhada adequada da ordem de remoção dos estados. Dado que nas expressões obtidas a partir de autómatos acíclicos não há a presença do operador \* estas simplificações são as únicas possíveis (exceptuando  $a + a = a$ , que não implementámos devido ao custo do teste da inclusão de uma linguagem regular noutra). Apresentámos tabelas que comparam os resultados obtidos pela heurística para o caso geral e esta e em que se verifica que os resultados obtidos pela heurística específica são substancialmente melhores. Terminámos o capítulo apresentando o exemplo de um autómato para o qual a heurística específica não consegue produzir uma expressão de tamanho mínimo obtenível pelo AEE.

Em seguida apresentamos o trabalho que gostaríamos de fazer no futuro. Consideremos a linguagem  $L(r)$  em que  $r = (a + b)^*b$ , sendo  $tamanho(r) = 3$ . O AFD mínimo para esta linguagem encontra-se desenhado na Figura 5.1.

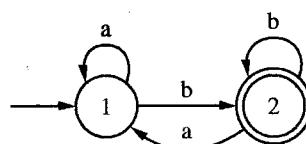


Figura 5.1: Autómato que reconhece  $L((a + b)^*b)$

Na Tabela 5.1 apresentamos as expressões obtidas pelo AEE para as duas ordens de remoção possíveis.

Tabela 5.1: Expressões obtidas para as duas ordens de remoção possíveis

Ordem	Expressão	Tamanho
1,2	$a^*b(aa^*b + b)^*$	6
2,1	$(bb^*a + a)^*bb^*$	6

Como se pode verificar, com nenhuma das ordens foi possível obter a expressão mínima  $r = (a+b)^*b$ . Num autómato útil, todas as etiquetas das transições contribuem para o tamanho da expressão final, no sentido de que nessa expressão as etiquetas aparecerão como subexpressões possivelmente com multiplicidade superior a 1. Isto faz-nos querer investigar se a Proposição 5.0.1 será verdadeira.

**Proposição 5.0.1** *Uma expressão regular mínima para uma dada linguagem  $L$  é obtenível pelo AEE quando aplicado a um AFNDTE que aceite  $L$  e seja mínimo no número de transições, se forem feitas, na aplicação do AEE, as simplificações  $a\epsilon = a$  e  $r + \emptyset = r$ .*

No artigo [Joh04] é apresentado um algoritmo de complexidade  $\mathcal{O}(2^n)$  para a minimização do número de transições de um AFNDTE. De acordo com o algoritmo apresentado neste artigo, o autómato mínimo em transições equivalente ao apresentado na Figura 5.1 seria o que se encontra na Figura 5.2, onde o símbolo  $@$  representa  $\epsilon$ .

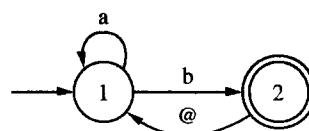


Figura 5.2: Autómato mínimo em transições que reconhece  $L((a + b)^*b)$

Na Tabela 5.2 apresentamos as expressões obtidas pelo AEE para as duas ordens de remoção possíveis, para este autómato mínimo.

Tabela 5.2: Expressões obtidas para as duas ordens de remoção possíveis

Ordem	Expressão	Tamanho
1,2	$a^*b(a^*b)^*$	4
2,1	$(a + b)^*b$	3

Vemos assim que a ordem de remoção 2,1 produz a expressão mínima  $r = (a + b)^*b$ .

Se a Proposição 5.0.1 for verdadeira temos assim um método (ainda que de complexidade exponencial) para calcular uma expressão mínima para uma dada linguagem.

Assumindo que a Proposição 5.0.1 é verdadeira e, numa tentativa de reduzir a complexidade do algoritmo apresentado em [Joh04], temos como objecto de estudo futuro o Problema 5.0.1.

**Problema 5.0.1 (Transição por  $\epsilon$  que não altera a linguagem reconhecida)**

Dados um AFND  $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$  e estados  $p$  e  $q$ , adicionar ao autómato a transição  $\delta(p, \epsilon) = q$  altera a linguagem reconhecida?

# Referências

- [Aho04] Aho. Computer representation of graphs. In Jonathan L. Gross and Jay Yellen, editors, *Handbook of Graph Theory*. CRC Press, 2004.
- [ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison Wesley, 1986.
- [BJG01] J. Bang-Jensen and G. Gutin. *Digraphs: Theory, Algorithms, and Applications*. Monographs in Mathematics (SMM). Springer, 2001.
- [CZ00] Pascal Caron and Djelloul Ziadi. Characterization of glushkov automata. *Theoret. Comput. Sci.*, 233:75–90, 2000.
- [DLM04] M. Delgado, S. Linton, and J. Morais. *Automata*. A GAP package on automata, version 1.01 2004. <http://www.gap-system.org/~gap/Packages/automata.html>.
- [DM04] Manuel Delgado and José Morais. Approximation to the smallest regular expression for a given regular language. In *CIAA 2004 - Ninth International Conference on Implementation and Application of Automata*, Lecture Notes in Computer Science. Springer-Verlag, 2004. poster.
- [ESW02] K. Ellul, J. Shallit, and Ming-wei Wang. Regular expressions: New results and open problems. In *Descriptional Complexity of Formal Systems*, Ontario, 2002. <http://www.math.uwaterloo.ca/~m2wang/papers/jalc.ps>.
- [GAP04] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.4*, 2004. (<http://www.gap-system.org>).

- [GJ03] Michael R. Garey and David S. Johnson. *Computers and Intractability / A Guide to the Theory of NP-Completeness*. W.H. Freeman & Company, New York, 2003.
- [GPW98] Dora Giannarasi, Jean-Luc Ponty, and Derick Wood. A re-examination of the Glushkov and Thompson constructions, 1998. <http://citeseer.ist.psu.edu/giannarasi98reexamination.html>.
- [GPW99] D. Giannarasi, J. Ponty, and D. Wood. A characterization of Thompson digraphs, 1999. [citeseer.ist.psu.edu/giannarasi99characterization.html](http://citeseer.ist.psu.edu/giannarasi99characterization.html).
- [Har69] Frank Harary. *Graph Theory*. Addison-Wesley, 1969.
- [HK02] Markus Holzer and Martin Kutrib. Nondeterministic descriptional complexity of regular languages, September 2002. <http://citeseer.ist.psu.edu/holzer02nondeterministic.html>.
- [HU01] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA, USA, second edition, 2001.
- [Joh04] Sebastian John. Minimal unambiguous  $\epsilon$ NFA. Technical report, Technical University Berlin, April 2004.
- [JR93] Tao Jiang and B. Ravikumar. Minimal NFA problems are hard. *SIAM J. Comput.*, 22(6):1117–1141, 1993.
- [Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, 1956.
- [Koz94] Dexter Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.
- [KR99] Khuller and Raghavachari. Basic graph algorithms. In Mikhail J. Atallah, editor, *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.
- [Mit03] Neil Mitchell. Regular expression simplification without finite state automata. Technical report, University of York, 2003.

[http://www.nmitchell.co.uk/download/\regular\\_expression\\_simplification.pdf](http://www.nmitchell.co.uk/download/\regular_expression_simplification.pdf).

- [MMP<sup>+</sup>95] O. Matz, A. Miller, A. Potthoff, W. Thomas, and E. Valkema. Report on the program AMoRe. Technical Report 9507, Christian Albrechts Universität, Kiel, 1995.
- [Moo66] E. F. Moore. Gedanken experiments on sequential machines. In *Automata Studies*. Princeton Univ Press, 1966. pp. 129–153.
- [MP43] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *Bull. Math. Biophysics*, 5:115–133, 1943.
- [Myh57] J. Myhill. Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson AFB, Ohio, 1957. pp. 112–137.
- [Ner58] A. Nerode. Linear automata transformations. In *Proceedings of AMS 9*, pages 541–544, 1958.
- [NV90] Maria Neves and Maria Vieira. *Acesso ao ensino superior: matemática*. Porto Editora, 1990.
- [RS59] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res*, 3: 2:115–125, 1959.
- [ST99] D. A. Simovici and R. L. Tenney. *Theory of formal languages with applications*. World Scientific, 1999.
- [TA98] Alejandro A. R. Trejo Ortiz and Guillermo Fernández Anaya. Regular expression simplification. *Math. Comput. Simul.*, 45(1-2):59–71, 1998.
- [Tho68] K. Thompson. Regular expression search algorithm. *Commun. ACM*, 11(6):410–422, 1968.
- [Wat94a] B. W. Watson. A taxonomy of finite automata minimization algorithms. Technical report, Eindhoven University of Technology, 1994. [citeseer.ist.psu.edu/article/watson94taxonomy.html](http://citeseer.ist.psu.edu/article/watson94taxonomy.html).

- [Wat94b] Bruce W. Watson. A taxonomy of finite automata construction algorithms. Technical Report Computing Science Note 93/43, Eindhoven University of Technology, The Netherlands, May 1994. cite-seer.ist.psu.edu/watson94taxonomy.html.
- [Yu97] Sheng Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1. Springer-Verlag Berlin Heidelberg, 1997.

# Índice de Tabelas

2.1	Contagem de Grafos Nard . . . . .	27
3.1	Alguns Resultados sem Heurística . . . . .	45
3.2	Alguns Resultados com Heurística . . . . .	46
3.3	Tamanhos das expressões para alguns Grafos de Cayley . . . . .	47
3.4	Tamanhos das expressões com a heurística melhorada . . . . .	48
3.5	Pesos Iniciais . . . . .	50
3.6	Pesos no autómato $\mathcal{A}$ depois de removido o vértice 4 . . . . .	50
3.7	Pesos no autómato $\mathcal{A}$ depois de removidos os vértices 4 e 2 . . . . .	51
3.8	Pesos no autómato $\mathcal{A}$ depois de removidos os vértices 4, 2 e 1 . . . . .	51
3.9	Pesos no autómato $\mathcal{A}$ depois de removido o vértice 2 . . . . .	52
3.10	Pesos no autómato $\mathcal{A}$ depois de removidos os vértices 2 e 1 . . . . .	52
3.11	Pesos no autómato $\mathcal{A}$ depois de removidos os vértices 2, 1 e 4 . . . . .	52
3.12	Peso com que cada vértice foi removido . . . . .	52
4.1	Valores para escolha do primeiro vértice a ser removido . . . . .	63
4.2	$dist\_gain[i]$ . . . . .	63
4.3	Valores para escolha do segundo vértice a ser removido . . . . .	63
4.4	$dist\_gain[i]$ . . . . .	64

4.5	Heurística para o caso geral . . . . .	66
4.6	Heurística para o caso acíclico . . . . .	66
4.7	Valores para escolha do primeiro vértice a ser removido . . . . .	72
4.8	<i>dist_gain[i]</i> . . . . .	72
4.9	Valores para escolha do segundo vértice a ser removido . . . . .	73
4.10	<i>dist_gain[i]</i> . . . . .	73
4.11	<i>dist_gain[i]</i> . . . . .	74
5.1	Expressões obtidas para as duas ordens de remoção possíveis . . . . .	77
5.2	Expressões obtidas para as duas ordens de remoção possíveis . . . . .	78

# Índice de Figuras

1.1	Algoritmo da Eliminação de Estados . . . . .	16
1.2	Algoritmo de Kleene . . . . .	19
1.3	Autómato $\mathcal{A}$ . . . . .	20
1.4	Autómato $\mathcal{A}$ depois de removido o vértice 2 . . . . .	20
1.5	Autómato $\mathcal{A}$ depois de removidos os vértices 2 e 1 . . . . .	20
1.6	Autómato $\mathcal{A}$ depois de removidos os vértices 2, 1 e 4 . . . . .	21
1.7	Autómato $\mathcal{A}$ depois de removidos os vértices 2, 1, 4 e 3 . . . . .	21
1.8	Autómato $\mathcal{A}$ depois de removido o vértice 3 . . . . .	21
1.9	Autómato $\mathcal{A}$ depois de removidos os vértices 3 e 4 . . . . .	21
1.10	Autómato $\mathcal{A}$ depois de removidos os vértices 3, 4 e 2 . . . . .	22
1.11	Autómato $\mathcal{A}$ depois de removidos os vértices 3, 4, 2 e 1 . . . . .	22
2.1	Grafo Etiquetado . . . . .	24
2.2	Autómato não nard . . . . .	25
2.3	Autómato nard . . . . .	25
2.4	Grafo $D$ . . . . .	26
2.5	Autómatos nard de 4 estados . . . . .	32
2.6	Verifica se um grafo é nard . . . . .	34

2.7	NardFAtoRatExp . . . . .	35
2.8	Autómato de Thompson de $\emptyset$ . . . . .	36
2.9	Autómato de Thompson de $\epsilon$ . . . . .	36
2.10	Autómato de Thompson de $a \in \Sigma$ . . . . .	36
2.11	Autómato de Thompson representando uma união . . . . .	36
2.12	Autómato de Thompson representando uma concatenação . . . . .	37
2.13	Autómato de Thompson representando o fecho de Kleene . . . . .	37
2.14	Remoção de um vértice $v$ tal que $ge(v) = 1$ e $gs(v) = 2$ . . . . .	38
2.15	Após a remoção de $v$ . . . . .	39
2.16	$ge(v) = 1 = gs(v)$ . . . . .	39
2.17	Após a remoção de $v$ . . . . .	39
2.18	Autómato não nard . . . . .	40
2.19	Autómato nard equivalente . . . . .	40
3.1	AEE com heurística para o caso geral . . . . .	49
3.2	Autómato $\mathcal{A}$ . . . . .	50
3.3	Autómato $\mathcal{A}$ depois de removido o vértice 4 . . . . .	50
3.4	Autómato $\mathcal{A}$ depois de removidos os vértices 4 e 2 . . . . .	51
3.5	Autómato $\mathcal{A}$ depois de removidos os vértices 4, 2 e 1 . . . . .	51
3.6	Autómato $\mathcal{A}$ depois de removidos os vértices 4, 2, 1 e 3 . . . . .	51
4.1	Grafo Etiquetado . . . . .	56
4.2	Grafo Etiquetado depois de removido $q$ . . . . .	57
4.3	computeWDist() . . . . .	59
4.4	computeDistGain() . . . . .	60

4.5	computeBest()	61
4.6	Autómato 847	62
4.7	Autómato 847 depois de removido o estado 5	64
4.8	Autómato 847 depois de removidos os estados 5 e 3	65
4.9	AEE com heurística para o caso acíclico	67
4.10	Autómato 663	72
4.11	Autómato 663 depois de removido o vértice 6	73
4.12	Autómato 663 depois de removidos os vértices 6 e 5	73
5.1	Autómato que reconhece $L((a + b)^*b)$	76
5.2	Autómato mínimo em transições que reconhece $L((a + b)^*b)$	77

# **Lista de Acrónimos**

AEE, algoritmo da eliminação de estados

AF, autómato finito

AFD, autómato finito determinístico

AFE, autómato finito estendido

AFND, autómato finito não determinístico

AFNDTE, autómato finito não determinístico com transições- $\epsilon$

$ge(v)$ , grau de entrada do vértice  $v$

$gs(v)$ , grau de saída do vértice  $v$

# Índice Remissivo

- adjacências
    - lista de, 8
    - matriz de, 8
  - AEE, 14
  - AFD, 8
    - completo, 9
  - AFE, 11
  - AFND, 10
  - AFNDTE, 11
  - alfabeto, 3
  - algoritmo
    - eliminação de estados, 14
    - Kleene, 18
  - arcos, 7
  - arestas, 7
  - autómato
    - útil, 12
    - acíclico, 12
  - caminho, 7
  - ciclo, 7
  - colapsável, 5
  - comprimento, 3
  - concatenação, 3
    - de linguagens, 3
  - configuração, 9
  - configuração de aceitação, 9
  - configuração inicial, 9
  - digrafo, 7
- equivalentes
    - autómatos, 13
    - expressões regulares, 6
  - Expressões Regulares, 4
  - grafo, 7
    - D*, 26
    - dirigido, 7
    - homeomorfo, 26
    - subjacente, 12
  - grau de entrada, 8
  - grau de saída, 8
  - GTG, 16
  - Kleene
    - fecho de, 3
  - lacete, 7
  - linguagem, 3
    - vazia, 3
  - linguagens regulares, 3
  - nard, 24
  - ordem topológica, 8
  - palavra, 3
    - vazia, 3
  - passo de computação, 9
  - peso
    - de um AFE, 43
    - de um vértice, 43

tamanho

de uma expressão regular, 6

vértices

finais, 12

iniciais, 12

intermédios, 12

não úteis, 23

# Apêndice A

## Código

### A.1 FAtoRatExpX

```
FAtoRatExpX := function(A)
local T,          # Transition function of the GTG
    Q,          # The number of states of the GTG without
               # counting q0 and qf
    q0, qf,      # The two extra states of the GTG
    computeGTG,
    destroyLoop,
    destroyState,
    i;

if not IsAutomatonObj(A) then
    Error("The argument to FAtoRatExp must be an automaton");
fi;
T := []; # Transition function of the GTG
Q := A!.states;
q0 := A!.states + 1;
qf := A!.states + 2;

## Function that computes the GTG (generalized transition graph)
## associated with the given FA.
computeGTG := function()
    local list, # List of alphabet symbols
        list2,
        l,
        X,
        a,      # Alphabet symbol
        c,      # Counter
        p, q; # States

    if A!.type = "nondet" then
```

```

for p in [1 .. A!.states] do
    T[p] := [];
    for q in [1 .. A!.states] do
        list := [];
        c := 0;
        for a in [1 .. A!.alphabet] do
            if q in A!.transitions[a][p] then
                Add(list, a);
                c := c + 1;
            fi;
        od;
        if list = [] then
            T[p][q] := 0; # 0 means the empty_set
        elif IsBound(list[2]) then
            T[p][q] :=
                RatExpOnnLetters(
                    ShallowCopy(FamilyObj(A)!.alphabet),
                    "union",
                    List(list, a -> RatExpOnnLetters(
                        ShallowCopy(FamilyObj(A)!.alphabet),
                        [], [a])));
        else
            T[p][q] :=
                RatExpOnnLetters(
                    ShallowCopy(FamilyObj(A)!.alphabet),
                    [], list);
        fi;
    od;
od;
elif A!.type = "epsilon" then
    if IsString(FamilyObj(A)!.alphabet) then
        l := ShallowCopy(FamilyObj(A)!.alphabet);
        Unbind(l[Length(l)]);
    fi;
    for p in [1 .. A!.states] do
        T[p] := [];
        for q in [1 .. A!.states] do
            list := [];
            c := 0;
            for a in [1 .. A!.alphabet-1] do
                if q in A!.transitions[a][p] then
                    Add(list, a);
                    c := c + 1;
                fi;
            od;
            a := A!.alphabet;
            if q in A!.transitions[a][p] then
                Add(list, 0);
            fi;
            if list = [] then
                T[p][q] := 0; # 0 means the empty_set
            elif IsBound(list[2]) then

```

```

list2 := [];
for x in list do
    if x = 0 then
        if IsString(FamilyObj(A)!.alphabet) then
            Add(list2, RatExp0nnLetters(l, [], []));
        else
            Add(list2, RatExp0nnLetters(
                FamilyObj(A)!.alphabet-1, [], []));
        fi;
    else
        if IsString(FamilyObj(A)!.alphabet) then
            Add(list2, RatExp0nnLetters(
                l, [], [x]));
        else
            Add(list2, RatExp0nnLetters(
                FamilyObj(A)!.alphabet-1, [], [x]));
        fi;
    fi;
od;
if IsString(FamilyObj(A)!.alphabet) then
    T[p][q] := RatExp0nnLetters(
        l, "union", list2);
else
    T[p][q] := RatExp0nnLetters(
        FamilyObj(A)!.alphabet-1,
        "union", list2);
fi;
else
    if list = [0] then
        if IsString(FamilyObj(A)!.alphabet) then
            T[p][q] := RatExp0nnLetters(l, [], []);
        else
            T[p][q] := RatExp0nnLetters(
                FamilyObj(A)!.alphabet-1, [], []);
        fi;
    else
        if IsString(FamilyObj(A)!.alphabet) then
            T[p][q] := RatExp0nnLetters(
                l, [], list);
        else
            T[p][q] := RatExp0nnLetters(
                FamilyObj(A)!.alphabet-1, [], list);
        fi;
    fi;
fi;
od;
else
    for p in [1 .. A!.states] do
        T[p] := [];
        for q in [1 .. A!.states] do
            list := [];

```

```

c := 0;
for a in [1 .. A!.alphabet] do
    # The only difference to the
    nondeterministic case
    if q = A!.transitions[a][p] then
        Add(list, a);
        c := c + 1;
    fi;
od;
if list = [] then
    T[p][q] := 0; # 0 means the empty_set
elif IsBound(list[2]) then
    T[p][q] := RatExpOnnLetters(
        ShallowCopy(FamilyObj(A)!.alphabet),
        "union", List(
            list, a -> RatExpOnnLetters(
                ShallowCopy(FamilyObj(A)!.alphabet),
                [], [a])));
else
    T[p][q] := RatExpOnnLetters(
        ShallowCopy(FamilyObj(A)!.alphabet),
        [], list);
fi;
od;
od;
fi;
T[q0] := []; T[qf] := [];
for p in [1 .. A!.states] do
    T[p][q0] := 0;
    if p in A!.accepting then
        if A!.type = "epsilon" then
            if IsString(FamilyObj(A)!.alphabet) then
                T[p][qf] := RatExpOnnLetters(1, [], []);
            else
                T[p][qf] := RatExpOnnLetters(
                    FamilyObj(A)!.alphabet-1, [], []);
            fi;
        else
            T[p][qf] := RatExpOnnLetters(
                ShallowCopy(FamilyObj(A)!.alphabet), [], []);
        fi;
    else
        T[p][qf] := 0;
    fi;
od;
for q in [1 .. A!.states] do
    if q in A!.initial then
        if A!.type = "epsilon" then
            if IsString(FamilyObj(A)!.alphabet) then
                T[q0][q] := RatExpOnnLetters(1, [], []);
            else
                T[q0][q] := RatExpOnnLetters(

```

```

        FamilyObj(A)!.alphabet-1, [], [] );
    fi;
else
    T[q0][q] := RatExpOnnLetters(
        ShallowCopy(FamilyObj(A)!.alphabet), [], [] );
    fi;
else
    T[q0][q] := 0;
    fi;
    T[qf][q] := 0;
od;
T[q0][q0] := 0; T[q0][qf] := 0; T[qf][q0] := 0; T[qf][qf] := 0;
end;
## End of computeGTG() --


## Function that deletes a loop in the GTG
destroyLoop := function(n)
    local q;

    if not T[n][n] = 0 then
        for q in [1 .. n-1] do
            if T[n][q] = 0 then
            elif T[n][q]!.list_exp = [] then
                T[n][q] := StarRatExp(T[n][n]);
            else
                T[n][q] := ProductRatExp(
                    StarRatExp(T[n][n]), T[n][q]);
            fi;
        od;
        for q in [n+1 .. Q+2] do
            if T[n][q] = 0 then
            elif T[n][q]!.list_exp = [] then
                T[n][q] := StarRatExp(T[n][n]);
            else
                T[n][q] := ProductRatExp(
                    StarRatExp(T[n][n]), T[n][q]);
            fi;
        od;
        T[n][n] := 0;
    fi;
end;
## End of destroyLoop() --


## Function that deletes a state in the GTG
destroyState := function(n)
    local p, q;

    for p in [1 .. n-1] do
        if not T[p][n] = 0 then
            for q in [1 .. n-1] do

```

```

        if not T[n][q] = 0 then
            if T[p][q] = 0 then
                T[p][q] := ProductRatExp(T[p][n], T[n][q]);
            else
                T[p][q] := UnionRatExp(
                    ProductRatExp(T[p][n], T[n][q]), T[p][q]);
            fi;
        fi;
    od;
    for q in [n+1 .. Q+2] do
        if not T[n][q] = 0 then
            if T[p][q] = 0 then
                T[p][q] := ProductRatExp(T[p][n], T[n][q]);
            else
                T[p][q] := UnionRatExp(
                    ProductRatExp(T[p][n], T[n][q]), T[p][q]);
            fi;
        fi;
    od;
    fi;
od;
for p in [n+1 .. Q+2] do
    if not T[p][n] = 0 then
        for q in [1 .. n-1] do
            if not T[n][q] = 0 then
                if T[p][q] = 0 then
                    T[p][q] := ProductRatExp(T[p][n], T[n][q]);
                else
                    T[p][q] := UnionRatExp(
                        ProductRatExp(T[p][n], T[n][q]), T[p][q]);
                fi;
            fi;
        od;
        for q in [n+1 .. Q+2] do
            if not T[n][q] = 0 then
                if T[p][q] = 0 then
                    T[p][q] := ProductRatExp(T[p][n], T[n][q]);
                else
                    T[p][q] := UnionRatExp(
                        ProductRatExp(T[p][n], T[n][q]), T[p][q]);
                fi;
            fi;
        od;
    fi;
od;
for p in [1 .. n-1] do
    for q in [n .. Q+1] do
        T[p][q] := T[p][q+1];
    od;
od;
for p in [n .. Q+1] do
    T[p] := T[p+1];

```

```

        for q in [n .. Q+1] do
            T[p][q] := T[p][q+i];
        od;
    od;
    Q := Q - 1;
end;
## End of destroyState() --


computeGTG();
while Q > 0 do
    destroyLoop(Q);
    destroyState(Q);
od;
if T[1][2] = 0 then
    return(RatExpOnnLetters(A!.alphabet, [], "empty_set"));
else
    return(T[1][2]);
fi;
end;

```

## A.2 NardFAToRatExp

```

NardFAToRatExp := function(A)
local T,           # Transition function of the GTG
      WT,          # Weighted transition function of the GTG
      Nout,         # Number of edges going out to distinct vertices
      Nin,          # Number of edges coming in from distinct
                  # vertices
      Invertices,   # The numbers of the vertices that have a
                  # transition into i
      Outvertices,  # The numbers of the vertices that have a
                  # transition from i
      Q,            # The number of states of the GTG without
                  # counting q0 and qf
      q0, qf,       # The two extra states of the GTG
      best,
      computeGTG,
      destroyState,
      flag,
      perm,
      i, j, k, q, r, l, p;

if not IsAutomatonObj(A) then
    Error("The argument to NardFAToRatExp must be an automaton");
fi;

perm := [1..A!.states];

```

```

i := A!.initial[1];
j := A!.accepting[1];
perm[i] := 1;
perm[1] := i;
p := Position(perm, 2);
k := perm[j];
perm[j] := 2;
perm[p] := k;
A := PermutatedAutomaton(A, perm);

T := □; # Transition function of the GTG
WT := □;
Nout := □;
Nin := □;
Q := A!.states;
q0 := A!.states + 1;
qf := A!.states + 2;
for i in [1 .. qf] do
    Nin[i] := 0;
od;
Nout[q0] := 0; # Value not used but saves tests
Nout[qf] := 0; # Value not used but saves tests
Invertices := □;
Outvertices := □;
for i in [1 .. Q+2] do
    Invertices[i] := [];
    Outvertices[i] := [];
od;

## Function that computes the GTG (generalized transition graph)
## associated with the given FA.
computeGTG := function()
    local list, # List of alphabet symbols
          list2,
          x,
          a,      # Alphabet symbol
          c,      # Counter
          p, q;  # States

    if A!.type = "nondet" then
        for p in [1 .. A!.states] do
            T[p] := □;
            WT[p] := □;
            Nout[p] := 0;
            for q in [1 .. A!.states] do
                list := [];
                c := 0;
                for a in [1 .. A!.alphabet] do
                    if q in A!.transitions[a][p] then
                        Add(list, a);
                        c := c + 1;
                    if not q = p then

```

```

        AddSet(Invvertices[q], p);
        AddSet(Outvertices[p], q);
    fi;
fi;
od;
if list = [] then
    T[p][q] := 0; # 0 means the empty_set
    WT[p][q] := -1;
elif IsBound(list[2]) then
    T[p][q] := RatExpOnnLetters(
        ShallowCopy(FamilyObj(A)!.alphabet), "union",
        List(list, a -> RatExpOnnLetters(
            ShallowCopy(FamilyObj(A)!.alphabet),
            [], [a])));
    WT[p][q] := c;
    if not p = q then
        Nout[p] := Nout[p] + 1;
        Nin[q] := Nin[q] + 1;
    fi;
else
    T[p][q] := RatExpOnnLetters(
        ShallowCopy(FamilyObj(A)!.alphabet), [], list);
    WT[p][q] := c;
    if not p = q then
        Nout[p] := Nout[p] + 1;
        Nin[q] := Nin[q] + 1;
    fi;
fi;
od;
od;
od;
elif A!.type = "epsilon" then
    if IsString(FamilyObj(A)!.alphabet) then
        l := ShallowCopy(FamilyObj(A)!.alphabet);
        Unbind(l[Length(l)]);
    fi;
    for p in [1 .. A!.states] do
        T[p] := [];
        WT[p] := [];
        Nout[p] := 0;
        for q in [1 .. A!.states] do
            list := [];
            c := 0;
            for a in [1 .. A!.alphabet-1] do
                if q in A!.transitions[a][p] then
                    Add(list, a);
                    c := c + 1;
                    if not q = p then
                        AddSet(Invvertices[q], p);
                        AddSet(Outvertices[p], q);
                    fi;
                fi;
            od;
        od;
    od;
fi;

```

```

a := A!.alphabet;
if q in A!.transitions[a][p] then
    Add(list, 0);
    if not q = p then
        AddSet(Outvertices[q], p);
        AddSet(Outvertices[p], q);
    fi;
fi;
if list = [] then
    T[p][q] := 0; # 0 means the empty_set
    WT[p][q] := -1;
elif IsBound(list[2]) then
    list2 := [];
    for x in list do
        if x = 0 then
            if IsString(FamilyObj(A)!.alphabet) then
                Add(list2, RatExpOnnLetters(1, [], []));
            else
                Add(list2, RatExpOnnLetters(
                    FamilyObj(A)!.alphabet-1, [], []));
            fi;
        else
            if IsString(FamilyObj(A)!.alphabet) then
                Add(list2, RatExpOnnLetters(
                    1, [], [x]));
            else
                Add(list2, RatExpOnnLetters(
                    FamilyObj(A)!.alphabet-1, [], [x]));
            fi;
        fi;
    od;
    if IsString(FamilyObj(A)!.alphabet) then
        T[p][q] := RatExpOnnLetters(
            1, "union", list2);
    else
        T[p][q] := RatExpOnnLetters(
            FamilyObj(A)!.alphabet-1, "union", list2);
    fi;
    WT[p][q] := c;
    if not p = q then
        Nout[p] := Nout[p] + 1;
        Nin[q] := Nin[q] + 1;
    fi;
else
    if list = [0] then
        if IsString(FamilyObj(A)!.alphabet) then
            T[p][q] := RatExpOnnLetters(1, [], []);
        else
            T[p][q] := RatExpOnnLetters(
                FamilyObj(A)!.alphabet-1, [], []);
        fi;
    else

```

```

        if IsString(FamilyObj(A)!.alphabet) then
            T[p][q] := RatExpOnnLetters(1, [], list);
        else
            T[p][q] := RatExpOnnLetters(
                FamilyObj(A)!.alphabet-1, [], list);
            fi;
        fi;
        WT[p][q] := c;
        if not p = q then
            Nout[p] := Nout[p] + 1;
            Nin[q] := Nin[q] + 1;
        fi;
        fi;
    od;
    od;
else
    for p in [1 .. A!.states] do
        T[p] := [];
        WT[p] := [];
        Nout[p] := 0;
        for q in [1 .. A!.states] do
            list := [];
            c := 0;
            for a in [1 .. A!.alphabet] do
                # The only difference to the nondeterministic case
                if q = A!.transitions[a][p] then
                    Add(list, a);
                    c := c + 1;
                    if not q = p then
                        AddSet(Invertices[q], p);
                        AddSet(Outvertices[p], q);
                    fi;
                fi;
            od;
            if list = [] then
                T[p][q] := 0; # 0 means the empty_set
                WT[p][q] := -1;
            elif IsBound(list[2]) then
                T[p][q] := RatExpOnnLetters(ShallowCopy(
                    FamilyObj(A)!.alphabet), "union",
                    List(list, a -> RatExpOnnLetters(
                        ShallowCopy(FamilyObj(A)!.alphabet), [], [a])));
                WT[p][q] := c;
                if not p = q then
                    Nout[p] := Nout[p] + 1;
                    Nin[q] := Nin[q] + 1;
                fi;
            else
                T[p][q] := RatExpOnnLetters(ShallowCopy(
                    FamilyObj(A)!.alphabet), [], list);
                WT[p][q] := c;
                if not p = q then

```

```

        Nout[p] := Nout[p] + 1;
        Nin[q] := Nin[q] + 1;
    fi;
    fi;
od;
od;
fi;
T[q0] := []; T[qf] := [];
WT[q0] := []; WT[qf] := [];
for p in [1 .. A!.states] do
    T[p][q0] := 0;
    WT[p][q0] := -1;
    if p in A!.accepting then
        if A!.type = "epsilon" then
            if IsString(FamilyObj(A)!.alphabet) then
                T[p][qf] := RatExpOnnLetters(1, [], []);
            else
                T[p][qf] := RatExpOnnLetters(
                    FamilyObj(A)!.alphabet-1, [], []);
            fi;
        else
            T[p][qf] := RatExpOnnLetters(ShallowCopy(
                FamilyObj(A)!.alphabet), [], []);
        fi;
    else
        T[p][qf] := RatExpOnnLetters(ShallowCopy(
            FamilyObj(A)!.alphabet), [], []);
    fi;
    WT[p][qf] := 0;
    Nout[p] := Nout[p] + 1;
    AddSet(Invvertices[qf], p);
    AddSet(Outvertices[p], qf);
else
    T[p][qf] := 0;
    WT[p][qf] := -1;
fi;
od;
for q in [1 .. A!.states] do
    if q in A!.initial then
        if A!.type = "epsilon" then
            if IsString(FamilyObj(A)!.alphabet) then
                T[q0][q] := RatExpOnnLetters(1, [], []);
            else
                T[q0][q] := RatExpOnnLetters(
                    FamilyObj(A)!.alphabet-1, [], []);
            fi;
        else
            T[q0][q] := RatExpOnnLetters(ShallowCopy(
                FamilyObj(A)!.alphabet), [], []);
        fi;
    else
        T[q0][q] := RatExpOnnLetters(ShallowCopy(
            FamilyObj(A)!.alphabet), [], []);
    fi;
    WT[q0][q] := 0;
    Nin[q] := Nin[q] + 1;
    AddSet(Invvertices[q], q0);
    AddSet(Outvertices[q0], q);
else
    T[q0][q] := 0;

```

```

        WT[q0][q] := -1;
    fi;
    T[qf][q] := 0;
    WT[qf][q] := -1;
od;
T[q0][q0] := 0; T[q0][qf] := 0; T[qf][q0] := 0; T[qf][qf] := 0;
WT[q0][q0] := -1; WT[q0][qf] := -1; WT[qf][q0] := -1;
WT[qf][qf] := -1;
end;
## End of computeGTG() --
```

## Function that deletes a state in the GTG

```

destroyState := function(n)
    local p, q;

    for p in Invertices[n] do
        for q in Outvertices[n] do
            if T[p][q] = 0 then
                T[p][q] := ProductRatExp(T[p][n], T[n][q]);
                WT[p][q] := WT[p][n] + WT[n][q];
            else
                T[p][q] := UnionRatExp(ProductRatExp(T[p][n],
                    T[n][q]), T[p][q]);
                if WT[p][q] = 0 then
                    WT[p][q] := WT[p][n] + WT[n][q] + 1;
                else
                    WT[p][q] := WT[p][n] + WT[n][q] + WT[p][q];
                fi;
            fi;
        od;
    od;
    for p in [1 .. n-1] do
        for q in [n .. Q+1] do
            T[p][q] := T[p][q+1];
            WT[p][q] := WT[p][q+1];
        od;
    od;
    for p in [n .. Q+1] do
        T[p] := T[p+1];
        WT[p] := WT[p+1];
        for q in [n .. Q+1] do
            T[p][q] := T[p][q+1];
            WT[p][q] := WT[p][q+1];
        od;
    od;
    Q := Q - 1;
    for p in [1 .. Q+2] do
        Nout[p] := 0; Nin[p] := 0;
        Invertices[p] := []; Outvertices[p] := [];
    od;
    for p in [1 .. Q+2] do
```

```

        for q in [1 .. p-1] do
            if WT[p][q] >= 0 then
                Nout[p] := Nout[p] + 1;
                Nin[q] := Nin[q] + 1;
                AddSet(Invvertices[q], p);
                AddSet(Outvertices[p], q);
            fi;
        od;
        for q in [p+1 .. Q+2] do
            if WT[p][q] >= 0 then
                Nout[p] := Nout[p] + 1;
                Nin[q] := Nin[q] + 1;
                AddSet(Invvertices[q], p);
                AddSet(Outvertices[p], q);
            fi;
        od;
    od;
end;
## End of destroyState() --

-----
----- Main Code -----
-----

```

```

computeGTG();
while Q > 0 do
    for i in [3..Q] do
        if Nin[i] = 0 or Nout[i] = 0 or
           (Nin[i] = 1 and Nout[i] = 1) then
            destroyState(i);
            break;
        fi;
    od;
od;

if T[1][2] = 0 then
    if A!.type = "epsilon" then
        r := RatExpOnnLetters(l, [], "empty_set");
    else
        r := RatExpOnnLetters(
            ShallowCopy(FamilyObj(A)!.alphabet), [], "empty_set");
    fi;
    Setter(SizeRatExp)(r,0);
    MinimalKnownRatExp(A)[1] := r;
    return(r);
else
    r := T[1][2];
    if WT[1][2] = 0 then
        Setter(SizeRatExp)(r,1);
    else

```

```

        Setter(SizeRatExp)(r,WT[1][2]);
    fi;
    return(r);
fi;
end;

```

### A.3 FAtoRatExp

```

FAtoRatExp := function(A)
local T,           # Transition function of the GTG
      WT,           # Weighted transition function of the GTG
      Nout,          # Number of edges going out to distinct vertices
      Nin,           # Number of edges coming in from distinct
                      # vertices
      Invertices,   # The numbers of the vertices that have a
                      # transition into i
      Outvertices,  # The numbers of the vertices that have a
                      # transition from i
      Inv,
      Outv,
      Q,            # The number of states of the GTG without
                      # counting q0 and qf
      q0, qf,       # The two extra states of the GTG
      WL,
      V,
      computeGTG,
      destroyLoop,
      destroyState,
      computeWeight,
      computeBest,  # Try to choose the best vertex
      flag,
      i, q, r, l;

if not IsAutomatonObj(A) then
    Error("The argument to FAtoRatExp must be an automaton");
fi;
T := []; # Transition function of the GTG
WT := [];
WL := [];
Nout := [];
Nin := [];
Q := A!.states;
q0 := A!.states + 1;
qf := A!.states + 2;
for i in [1 .. qf] do
    Nin[i] := 0;
od;
Nout[q0] := 0; # Value not used but saves tests
Nout[qf] := 0; # Value not used but saves tests

```

```

Invertices := [];
Outvertices := [];
for i in [1 .. Q+2] do
  Invertices[i] := [];
  Outvertices[i] := [];
od;

## Function that computes the GTG (generalized transition graph)
## associated with the given FA.
computeGTG := function()
... Código igual ao da função NardFAtoRatExp ...
end;
## End of computeGTG() --


## Function that deletes a loop in the GTG
destroyLoop := function(n)
... Código igual ao da função NardFAtoRatExp ...
end;
## End of destroyLoop() --


## Function that deletes a state in the GTG
destroyState := function(n)
... Código igual ao da função NardFAtoRatExp ...
end;
## End of destroyState() --


computeWeight := function(i)
  local w, p, q;

  w := 0;
  for p in Invertices[i] do
    w := w + WT[p][i] * (Nout[i] - 1);
  od;
  if WT[i][i] = -1 then # Do nothing, there is no loop
  else
    w := w + WT[i][i] * (Nout[i] * Nin[i] - 1);
  fi;
  for q in Outvertices[i] do
    w := w + WT[i][q] * (Nin[i] - 1);
  od;
  return(w);
end;
## End of computeWeight() --


computeBest := function()
  local p, q, i, w;

```

```

if WL = [] then
    for i in [1 .. Q] do
        AddSet(WL, [computeWeight(i),i]);
    od;
else
    for i in Inv do
        AddSet(WL, [computeWeight(i),i]);
    od;
    for i in Outv do
        AddSet(WL, [computeWeight(i),i]);
    od;
fi;
return(WL[1][2]);
end;
## End of computeBest() --

computeGTG();
flag := true;
while flag do
    flag := false;
    for i in [1 .. Q] do
        if Nout[i] = 0 then
            destroyLoop(i);
            destroyState(i);
            flag := true;
            break;
        fi;
    od;
    od;
    flag := true;
    while flag do
        flag := false;
        for i in [1 .. Q] do
            if Nin[i] = 0 then
                destroyLoop(i);
                destroyState(i);
                flag := true;
                break;
            fi;
        od;
    od;
    while Q > 0 do
        V := computeBest();
        i := 1;
        while IsBound(WL[i]) do
            if WL[i][2] = V then
                Unbind(WL[i]);
            elif WL[i][2] > V then
                if WL[i][2] in Invertices[V] or
                    WL[i][2] in Outvertices[V] then

```

```

        Unbind(WL[i]);
    else
        WL[i][2] := WL[i][2] - 1;
    fi;
else
    if WL[i][2] in Invertices[V] or
        WL[i][2] in Outvertices[V] then
        Unbind(WL[i]);
    fi;
fi;
i := i + 1;
od;
WL := Compacted(WL);
Inv := Difference(Invertices[V], [Q+1,Q+2]);
Outv := Difference(Outvertices[V], [Q+1,Q+2]);
i := 1;
while IsBound(Inv[i]) do
    if Inv[i] > V then
        Inv[i] := Inv[i] - 1;
    fi;
    i := i + 1;
od;
i := 1;
while IsBound(Outv[i]) do
    if Outv[i] > V then
        Outv[i] := Outv[i] - 1;
    fi;
    i := i + 1;
od;
destroyLoop(V);
destroyState(V);
od;

if T[1][2] = 0 then
    if A!.type = "epsilon" then
        r := RatExpOnnLetters(l, [], "empty_set");
    else
        r := RatExpOnnLetters(ShallowCopy(
            FamilyObj(A)!.alphabet), [], "empty_set");
    fi;
    Setter(SizeRatExp)(r,0);
    return(r);
else
    r := T[1][2];
    if WT[1][2] = 0 then
        Setter(SizeRatExp)(r,1);
    else
        Setter(SizeRatExp)(r,WT[1][2]);
    fi;
    return(r);
fi;
end;

```

## A.4 AcyclicFAtоРatExp

```

AcyclicFAtоРatExp := function(A)
    local T,           * Transition function of the GTG
    WT,              * Weighted transition function of the GTG
    Nout,            * Number of edges going out to distinct vertices
    Nin,             * Number of edges coming in from distinct
                      vertices
    Invertices,      * The numbers of the vertices that have a
                      transition into i
    Outvertices,     * The numbers of the vertices that have a
                      transition from i
    Q,               * The number of states of the GTG without
                      counting q0 and qf
    q0, qf,          * The two extra states of the GTG
    best,
    computeGTG,
    destroyState,
    computeWeight,
    computeWDist,
    computeDistGain, dist_gain,
    computeBest,    # Try to choose the best vertex
    removeImmediate,
    flag,
    perm, bonus,
    i, j, k, q, r, l, p;

    if not IsAutomatonObj(A) then
        Error("The argument to AcyclicFAtоРatExp must be an automaton");
    fi;

    perm := [1..A!.states];
    i := A!.initial[1];
    j := A!.accepting[1];
    perm[i] := i;
    perm[1] := i;
    p := Position(perm, 2);
    k := perm[j];
    perm[j] := 2;
    perm[p] := k;
    A := PermutatedAutomaton(A, perm);

    T := □; * Transition function of the GTG
    WT := □;
    Nout := □;
    Nin := □;
    Q := A!.states;
    q0 := A!.states + 1;
    qf := A!.states + 2;
    for i in [1 .. qf] do
        Nin[i] := 0;

```

```

od;
Nout[q0] := 0; # Value not used but saves tests
Nout[qf] := 0; # Value not used but saves tests
Invertices := [];
Outvertices := [];
for i in [1 .. Q+2] do
    Invertices[i] := [];
    Outvertices[i] := [];
od;

## Function that computes the GTG (generalized transition graph)
## associated with the given FA.
computeGTG := function()
... Código igual ao da função NardFAToRatExp ...
end;
## End of computeGTG() --


## Function that deletes a state in the GTG
destroyState := function(n)
... Código igual ao da função NardFAToRatExp ...
end;
## End of destroyState() --


computeWeight := function(i)
... Código igual ao da função FAToRatExp ...
end;
## End of computeWeight() --


computeWDist := function(q)
local i, j, k, w;

w := 0;

for i in Invertices[q] do
    for j in Outvertices[q] do
        if j in Outvertices[i] then
            if Nout[j] = 1 and Nin[j] = 2 and Nin[q] = 1 then
                bonus[q] := bonus[q] + computeWeight(j);
            fi;
            if Nout[i] = 2 and Nin[i] = 1 and Nout[q] = 1 then
                bonus[q] := bonus[q] + computeWeight(i);
            fi;
            for k in Invertices[i] do
                w := w + WT[k][i];
            od;
            for k in Outvertices[j] do
                w := w + WT[j][k];
            od;
        fi;
    fi;

```

```

        od;
    od;
    return(w);
end;
## End of computeWDist() --


computeDistGain := function(q)
local s, t;

dist_gain := List([1..Q], x->0);
for t in Outvertices[q] do
    if ForAll(Outvertices[q], x -> t in Outvertices[x]) then
        dist_gain[t] := dist_gain[t] + Nout[t];
    fi;
od;
for s in Invertices[q] do
    if ForAll(Outvertices[q], x -> s in Invertices[x]) then
        dist_gain[s] := dist_gain[s] + Nin[s];
    fi;
od;
end;
## End of computeDistGain() --


computeBest := function()
local i, j, l, s, t, dg;

if Q = 2 then
    return(1);
fi;
bonus := List([1..Q], x->0);
l := [];
j := 0;
for i in [3..Q] do
    AddSet(l, [computeWDist(i)-computeWeight(i) + bonus[i], i]);
    j := j + 1;
od;
if j > 1 then
    s := l[j-1][2];
    t := l[j][2];
    computeDistGain(s);
    dg := StructuralCopy(dist_gain);
    computeDistGain(t);
    if dg[t] > dist_gain[s] then
        return(s);
    else
        return(t);
    fi;
else
    return(l[j][2]);
fi;

```

```

end;
## End of computeBest() --


removeImmediate := function()
local i, flag;

flag := false;
for i in [3..Q] do
  if Nin[i] = 0 or Nout[i] = 0 or
    (Nin[i] = 1 and Nout[i] = 1) then
    flag := true;
    destroyState(i);
    break;
  fi;
od;
if flag then
  removeImmediate();
fi;
end;
## End of removeImmediate() --


-----  

----- Main Code -----  

-----  




computeGTG();
removeImmediate();
while Q > 2 do
  q := computeBest();
  destroyState(q);
  removeImmediate();
od;
for i in [1..Q] do
  destroyState(i);
od;

if T[1][2] = 0 then
  if A!.type = "epsilon" then
    r := RatExpOnLetters(l, [], "empty_set");
  else
    r := RatExpOnLetters(
      ShallowCopy(FamilyObj(A)!.alphabet), [], "empty_set");
  fi;
  Setter(SizeRatExp)(r,0);
  MinimalKnownRatExp(A)[1] := r;
  return(r);
else
  r := T[1][2];
  if WT[1][2] = 0 then

```

```
    Setter(SizeRatExp)(r,1);
else
    Setter(SizeRatExp)(r,WT[1][2]);
fi;
return(r);
fi;
end;
```

