Projeto de Iniciação Científica Autômatos Finitos: Algoritmos e Estruturas de Dados

Aluno: Leo Kazuhiro Ueda
Orientadora: Nami Kobayashi
Bolsa: PIBIC - CNPq

Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

Relatório correspondente ao período de março/2001 a julho/2001
(com correções)

Resumo

A Teoria de Autômatos Finitos tem sido aplicada a áreas diversas, como, por exemplo, no desenvolvimento de analisadores léxicos e editores de texto, em Processamento de Imagens, Biologia Molecular Computacional e Lingüística Computacional. Para compreender e estudar essas aplicações, é importante conhecer os algoritmos e estruturas de dados fundamentais que utilizam autômatos finitos. O objetivo deste projeto foi iniciar um estudo de tais aspectos básicos da Teoria de Autômatos Finitos. Mais especificamente, foi realizado um estudo detalhado do algoritmo de Hopcroft para a minimização de autômatos finitos determinísticos, do algoritmo de Revuz para a minimização de autômatos finitos determinísticos acíclicos, e do algoritmo de Aho e Corasick para a busca de um conjunto finito de palavras em um texto.

Sumário

1 Introdução **3**

2 Metodologia **3**

3 Tópicos Estudados **4**
3.1 Notação e Definições Básicas 4
3.2 Minimização de Autômatos Finitos 4
3.2.1 Definição do Problema 5
3.2.2 O algoritmo de Hopcroft 5
3.2.3 O algoritmo de Revuz para autômatos acíclicos 11
3.3 Uso de autômatos finitos num problema de busca de padrões 16
3.3.1 Descrição do Problema 17
3.3.2 O algoritmo de Aho e Corasick 18

4 Observações e Conclusões **24**

1 Introdução

As aplicações de Autômatos Finitos vêm crescendo significativamente, em grande parte devido ao aumento da demanda por sistemas de processamento de quantidades imensas de dados. Como estrutura de dados, um autômato pode armazenar informações de forma a economizar espaço em memória, enquanto isso, os algoritmos já conhecidos para autômatos podem oferecer um processamento eficiente.

A Lingüística Computacional está entre as áreas de aplicação mais importantes da Teoria de Autômatos Finitos, em especial a área de Processamento de Línguas Naturais. Outra área que tem recebido destaque é a Biologia Computacional. As seqüências de DNA são geralmente vistas como palavras sobre o alfabeto $\{a, c, g, t\}$ de nucleotídeos, dessa maneira elas constituem objetos para a aplicação da Teoria de Autômatos Finitos.

Neste projeto, nós estudamos um dos algoritmos fundamentais e uma aplicação que utiliza autômatos de forma eficiente. O objetivo é criar a base para um estudo futuro sobre as inúmeras aplicações dessa teoria. Os tópicos estudados até o momento foram:

- Minimização do número de estados de autômatos finitos determinísticos
  - algoritmo de Jonh Hopcroft, que minimiza qualquer autômato finito determinístico;
  - algoritmo de Dominique Revuz, que minimiza um autômato finito determinístico acíclico;
- Uso de autômatos finitos num problema de busca de padrões
  - algoritmo de Alfred Aho e Margaret Corasick para a busca de um conjunto finito de palavras num texto.

2 Metodologia

As atividades foram voltadas basicamente para a compreensão e implementação de cada algoritmo, principalmente através da leitura de artigos e da realização de reuniões semanais. As reuniões envolveram discussões sobre o andamento do projeto, destacando:

- resolução de dúvidas e análise dos detalhes dos artigos lidos, como por exemplo, provas de corretude e de complexidade de espaço e de tempo;
- investigação de questões de implementação, que tem compromisso com as complexidades de espaço e tempo;
- definição dos passos seguintes do projeto;
- acompanhamento do desempenho acadêmico do aluno.

3 Tópicos Estudados

Nesta seção descreveremos o que foi estudado.

3.1 Notação e Definições Básicas

Um *autômato finito determinístico* $\mathcal{A}$ é uma quíntupla $(Q, \Sigma, \delta, q_0, F)$, onde:

- Q é um conjunto finito e não vazio de estados;
- Σ é o alfabeto (conjunto finito e não vazio de símbolos);
- δ é a função de transição, definida por $\delta : Q \times \Sigma \to Q$;
- $q_0 \in Q$ é o estado inicial;
- $F \subseteq Q$ é o conjunto dos estados finais.

Uma *palavra* é uma seqüência finita de símbolos; a *palavra vazia* é denotada por λ. O conjunto de todas as palavras, incluindo λ, é Σ^*.

A função δ será estendida para $\hat{\delta} : Q \times \Sigma^* \to Q$ da maneira usual:

$$\begin{cases} \hat{\delta}(q, \lambda) = q & \forall q \in Q \\ \hat{\delta}(q, x\sigma) = \delta(\hat{\delta}(q, x), \sigma) & \forall q \in Q,\ \forall x \in \Sigma^*,\ \forall \sigma \in \Sigma \end{cases} .$$

A partir de agora usaremos δ para denotar $\hat{\delta}$.

Diremos também que n é a cardinalidade de Q, ou seja, o número de estados de $\mathcal{A}$.

A linguagem reconhecida por $\mathcal{A}$, denotada por $L(\mathcal{A})$, é o conjunto das palavras $x \in \Sigma^*$ tais que $\delta(q_0, x) \in F$.

Seja xuy uma palavras em Σ^*, então x é um *prefixo*, u é um *fator* e y é um *sufixo* de xuy, para x, u e y em Σ^*.

O grafo associado a $\mathcal{A}$, denotado por $G(\mathcal{A})$, é o grafo orientado (V, A), onde:

- os vértices são os estados de $\mathcal{A}$: $V = Q$;
- os arcos são rotulados por elementos de Σ e representam as transições: $A = \{(p, \sigma, q) : p \in Q, \sigma \in \Sigma, q = \delta(p, \sigma)\}$.

3.2 Minimização de Autômatos Finitos

Estudamos dois algoritmos para minimização de autômatos finitos determinísticos, o de John Hopcroft, com complexidade de tempo $O(|\Sigma| n \log n)$, e de Dominique Revuz, para autômatos acíclicos e com complexidade $O(|\Sigma| n)$.

Começaremos com a definição do problema que queremos resolver, em seguida discutiremos os dois algoritmos.

3.2.1 Definição do Problema

Seja $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ um autômato finito determinístico.

Queremos um algoritmo que determine um autômato finito determinístico $\mathcal{B}$ com o menor número de estados e tal que $L(\mathcal{B}) = L(\mathcal{A})$. Tal autômato é denominado *minimal*.

Os métodos utilizados pelos dois algoritmos que apresentamos aqui se baseiam na seguinte definição de uma relação de equivalência sobre Q:

Definição 3.1 *Dois estados p e q são* equivalentes *se, e somente se, para todo $x \in \Sigma^*$, vale que $\delta(p, x) \in F \iff \delta(q, x) \in F$.*

De modo muito simplificado, a idéia é encontrar os estados equivalentes de $\mathcal{A}$. Mais especificamente, os algoritmos procuram pelas classes de equivalência em Q, conforme a relação definida acima. Sendo $[q]$ a classe de equivalência de $q \in Q$, o autômato $\mathcal{A}' = (Q', \Sigma, \delta', q_0', F')$ definido da seguinte forma é minimal:

- $Q' = \{[q] : q \in Q\}$;
- $\delta'([q], \sigma) = [\delta(q, \sigma)] \quad \forall q \in Q \ , \forall \sigma \in \Sigma$;
- $q_0' = [q_0]$;
- $F' = \{[q] : q \in F\}$.

3.2.2 O algoritmo de Hopcroft

Muitos textos básicos sobre autômatos finitos [9, 11] apresentam algoritmos para minimizar o número de estados de um autômato finito determinístico. Entretanto, a complexidade de tempo no pior caso desses algoritmos é $O(|\Sigma| n^2)$, ou pior.

Em [8], Hopcroft apresenta um algoritmo de minimização com complexidade de tempo $O(|\Sigma| n \log n)$. Em seguida, David Gries [7] descreve o mesmo algoritmo (com uma pequena modificação), porém, de forma mais clara.

A discussão a seguir é baseada na descrição de Gries.

O algoritmo básico

Analisaremos o algoritmo básico de forma a facilitar o entendimento do algoritmo final.

Definição 3.2 *Uma* partição *dos estados de $\mathcal{A}$ em blocos $B_1, B_2, \cdots, B_p$ é* aceitável *quando é verdade que:*

(a) nenhum bloco contém, ao mesmo tempo, um estado final e um não final; e

(b) se p e q são estados equivalentes, então eles estão no mesmo bloco.

O algoritmo manterá uma partição de Q, que no início (ou final) de cada iteração é *aceitável*. Queremos que o algoritmo devolva uma partição onde dois estados p e q são equivalentes se e somente se eles estão no mesmo bloco. Os próximos dois lemas formalizam esse objetivo, mostrando a situação inicial e final do algoritmo.

Lema 1 *A partição $B_1 = F, B_2 = Q - F$ é aceitável.*

Lema 2 *Uma partição $B_1, B_2, \cdots, B_p$ é a partição que fornece as classes de equivalência em Q se e somente se:*

(a) a partição é aceitável; e

(b) para cada par de blocos B_i, B_j e cada símbolo $\sigma \in \Sigma$,

$$\forall p, q \in B_i\,, \quad \delta(p,\sigma) \in B_j \Rightarrow \delta(q,\sigma) \in B_j\ . \tag{1}$$

Em outras palavras, pensando no grafo $G(\mathcal{A})$, todos os arcos com rótulo σ que saem (dos vértices) de B_i devem chegar a (vértices de) um único bloco.

Portanto, podemos começar o algoritmo na situação do Lema 1 e fazer com que ele chegue à situação do Lema 2. O próximo lema descreve como fazer isso, refinando uma partição aceitável de uma maneira induzida pelas condições do Lema 2.

Lema 3 *Seja $B_1, B_2, \cdots, B_p$ uma partição aceitável. Suponha que existem dois blocos B_i e B_j, um símbolo σ e dois estados p e q tais que:*

$$p, q \in B_i\,, \quad \delta(p,\sigma) \in B_j \quad \textit{mas} \quad \delta(q,\sigma) \notin B_j\ . \tag{2}$$

Então p e q não são estados equivalentes, e podemos obter uma nova partição aceitável substituindo B_i pelos blocos:

$$\overline{B_i} = \{s \in B_i : \delta(s,\sigma) \in B_j\} \quad e \quad \widetilde{B_i} = \{s \in B_i : \delta(s,\sigma) \notin B_j\}\ . \tag{3}$$

Essa substituição é na verdade uma *divisão* do bloco B_i em dois blocos. Considerando novamente $G(\mathcal{A})$, no bloco $\overline{B_i}$ todos os arcos com rótulo σ que saem dele chegam ao bloco B_j. Da mesma forma, no bloco $\widetilde{B_i}$, nenhum arco com rótulo σ chega ao bloco B_j. Note que esse é um refinamento do bloco B_i em direção ao nosso objetivo descrito no Lema 2. Chamaremos essa operação de *divisão do bloco B_i em relação ao par (B_j, σ)*, ou simplesmente *divisão de B_i em relação a (B_j, σ)*.

Usando os lemas vistos até agora, podemos escrever o Algoritmo 4.

O algoritmo termina, pois a cada iteração um bloco é necessariamente adicionado na partição, e não podemos ter mais do que n blocos.

Os lemas também podem mostrar que no final da execução do algoritmo, a partição obtida é a desejada.

Algoritmo 4

$B_1 \leftarrow F$; $B_2 \leftarrow Q - F$;
enquanto $\exists \sigma, B_i, B_j$ tais que vale (2) **faça**
  divida B_i *em relação a* (B_j, σ);
fim. (4)

- Pré-condição: no início da execução temos uma partição aceitável (Lema 1);
- Invariante: após cada iteração, a partição continua sendo aceitável (Lema 3);
- Pós-condição: no final da execução, temos uma partição aceitável e nela não existem dois blocos B_i e B_j, um símbolo σ e dois estados p e q tais que vale (2). Pelo Lema 2, a partição fornece as classes de equivalência de Q.

Melhorias no algoritmo básico: lista L

Note que a ordem em que o Algoritmo (4) faz as operações de divisão de blocos não importa para a corretude. Então em cada iteração podemos determinar todas as divisões em relação a (B_j, σ) e depois executar todas essas divisões no mesmo passo. Com isso teremos o Algoritmo 5.

Algoritmo 5

$B_1 \leftarrow F$; $B_2 \leftarrow Q - F$;
enquanto $\exists \sigma, B_i, B_j$ tais que vale (2) **faça**
  determine as divisões de todos o blocos *em relação a* (B_j, σ);
  divida cada bloco conforme determinado;
fim. (5)

Esse algoritmo não é muito eficiente, já que para verificar a existência da tripla (σ, B_i, B_j) da condição do *enquanto*, precisamos testar todas as triplas (σ, B_i, B_j) existentes. Para esse teste somente, o algoritmo precisa fazer pelo menos $|\Sigma|n^2$ operações.

A estratégia para melhorar a complexidade é justamente reduzir o tempo gasto na verificação da condição do *enquanto*. Faremos isso mantendo uma lista L dos pares (B_j, σ) em relação aos quais sabemos que existem blocos B_i que precisam ser divididos. Veremos a seguir como é possível manter tal lista L.

Considere a seguinte observação:

Após a divisão de todos blocos em relação a (B_j, σ)*, todos os blocos* B *da partição satisfazem* uma *das seguintes condições:*

$$\begin{array}{ll} (a) & \forall q \in B \quad \delta(q,\sigma) \in B_j\,, \quad \textit{ou} \\ (b) & \forall q \in B \quad \delta(q,\sigma) \notin B_j\,. \end{array} \tag{6}$$

Isso nos leva ao seguinte lema:

Lema 4 *Suponha que todos os blocos foram divididos em relação a* (B_j, σ)*. Então não será preciso dividir mais nenhum bloco em relação a* (B_j, σ)*.*

Com isso concluímos que um par (B_j, σ) só precisa entrar na lista no máximo uma vez.

O próximo lema descreve o fato que permite reduzir o número de operações de divisão na próxima versão do algoritmo, ou seja, permite diminuir o número de elementos que passam pela lista L.

Lema 5 *Suponha que em algum momento o bloco* B_j *foi dividido nos blocos* $\overline{B_j}$ *e* $\widetilde{B_j}$ *(em relação a algum par). Fixe um símbolo* σ*. Dividir todos os blocos em relação a quaisquer dois dos três pares* (B_j, σ)*,* $(\overline{B_j}, \sigma)$ *e* $(\widetilde{B_j}, \sigma)$ *resulta no mesmo que dividir todos os blocos em relação a todos os três pares.*

Lema 6 *Considere a situação inicial, onde, digamos,* $B = Q$*,* $B_1 = F$*,* $B_2 = Q - F$*. Então, para um dado símbolo* σ*, é necessário dividir todos os blocos apenas em relação a* (B_1, σ) *ou a* (B_2, σ)*.*

O Algoritmo 7 usa a lista L conforme discutido.

Comparando com o Algoritmo 5, observa-se que a diferença está na escolha do par (B_j, σ). Para essa escolha, o novo algoritmo manipula a lista L.

O Algoritmo 7 termina, pois o número de pares (B_j, σ) é limitado; cada par entra na lista L no máximo uma vez; e a cada iteração removemos um elemento da lista L. Precisamos então saber se esse o algoritmo continua correto, isto é, se a lista L fornece os pares certos. Para isso vamos atribuir um significado à lista L.

L é uma lista de pares (B_j, σ) *em relação aos quais deve-se tentar dividir todos os blocos para se obter as condições da situação (6). Se para o bloco* B_j *o par* $(B_j, \sigma) \notin L$ *para algum* σ*, então ou* (B_j, σ) *já passou pela lista* L *(isto é, valem as condições em (6)), ou em algum momento até o final da execução, as condições da situação (6) valerão para* (B_j, σ)*. Nesse último caso, ou* (B_j, σ) *entrará na lista* L *ou* (B_j, σ) *é um dos três pares descritos no Lema 5 para o qual não é preciso dividir nenhum bloco.* (8)

A corretude pode ser demonstrada a partir das seguintes condições:

- Pré-condição: a afirmação (8) é verdadeira pelo Lema 6;
- Invariante: a afirmação (8) se mantém verdadeira pelos Lemas 4 e 5;
- Pós-condição: a afirmação (8) continua verdadeira, pois vale o invariante.

Análise da complexidade no pior caso

Para analisar a complexidade do algoritmo, precisamos detalhar mais os passos c e e. Isso é feito no Algoritmo 9, visto mais adiante.

Considere a árvore binária onde

- os nós são os blocos;
- a raíz é o bloco com todos os estados, o conjunto Q, e os seus filhos são os blocos $B_1 = F$ e $B_2 = Q - F$;
- os filhos de um bloco B são os blocos $\overline{B}$ e $\widetilde{B}$ resultantes de sua divisão.

Temos que o número de nós folhas dessa árvore é limitado por n, portanto, o número total de blocos criados durante a execução do algoritmo é

Algoritmo 7

$B_1 \leftarrow F$; $B_2 \leftarrow Q - F$; $L = \emptyset$;
para cada $\sigma \in \Sigma$ **faça**
  se $|B_1| < |B_2|$ **então** $L \leftarrow L + (B_1, \sigma)$;
  senão $L \leftarrow L + (B_2, \sigma)$;
fim;
enquanto $L \neq \emptyset$ **faça**
  b: Pegue um par (B_j, σ) de L;
  c: Determine as divisões de todos o blocos *em relação a* (B_j, σ);
  d: $L \leftarrow L - (B_j, \sigma)$;
  e: Divida cada bloco conforme determinado;
  f: *Atualize L de acordo com as divisões que ocorreram:*
    para cada bloco B dividido em $\overline{B}$ e $\widetilde{B}$ **faça**
      para cada $\sigma \in \Sigma$ **faça**
        se $(B, \sigma) \in L$
          então $L \leftarrow L + (\overline{B}, \sigma) + (\widetilde{B}, \sigma) - (B, \sigma)$;
          senão se $|\overline{B}| < |\widetilde{B}|$
            então $L \leftarrow L + (\overline{B}, \sigma)$;
            senão $L \leftarrow L + (\widetilde{B}, \sigma)$;
      fim;
    fim;
fim.

(7)

Algoritmo 9

$B_1 \leftarrow F$; $B_2 \leftarrow Q - F$; $L = \emptyset$;
para cada $\sigma \in \Sigma$ **faça**
  se $|B_1| < |B_2|$ **então** $L \leftarrow L + (B_1, \sigma)$;
  senão $L \leftarrow L + (B_2, \sigma)$;
fim;
enquanto $L \neq \emptyset$ **faça**
  b: Pegue um par (B_j, σ) de L;
  c: *Determine as divisões de todos o blocos* em relação a (B_j, σ)*:*
    $D \leftarrow \emptyset$;
    para cada $q \in B_j$ **faça** $D \leftarrow D \cup \delta^{-1}(q, \sigma)$ **fim**;
  d: $L \leftarrow L - (B_j, \sigma)$;
  e: *Divida cada bloco conforme determinado:*
    para cada $q \in D$ **faça**
      $\overline{B} \leftarrow$ bloco em que q está;
      se $\delta(p, \sigma) \in B_j$ vale para $\forall p \in \overline{B}$
        então não faça nada;
        senão se $\overline{B}$ não tem ainda um irmão $\widetilde{B}$
            então gere $\widetilde{B}$; $\widetilde{B} \leftarrow \emptyset$;
            mova q de $\overline{B}$ para o seu irmão $\widetilde{B}$;
    fim;
  f: *Atualize L de acordo com as divisões que ocorreram:*
    para cada bloco B dividido em $\overline{B}$ e $\widetilde{B}$ **faça**
      para cada $\sigma \in \Sigma$ **faça**
        se $(B, \sigma) \in L$
          então $L \leftarrow L + (\overline{B}, \sigma) + (\widetilde{B_j}, \sigma) - (B, \sigma)$;
          senão se $|\overline{B}| < |\widetilde{B}|$
            então $L \leftarrow L + (\overline{B}, \sigma)$;
            senão $L \leftarrow L + (\widetilde{B}, \sigma)$;
      fim;
    fim;
fim. (9)

no máximo $2n$. Como cada par (B, σ) entra na lista L no máximo uma vez, o número máximo de iterações é $2|\Sigma|n$.

Além disso, a lista L pode ser implementada de modo que as operações b e d gastem tempo constante. Sendo assim, o tempo *total*[1] gasto por essas duas operações é $O(|\Sigma|n)$. Com a estrutura de dados adequada, isso também mostra que o tempo *total* gasto pela operação f é $O(|\Sigma|n)$.

Os tempos das operações c e e são claramente limitados inferiormente pelo número total de elementos que passam pelo conjunto D.

[1] Considerando a execução toda, não apenas uma iteração.

Para calcularmos esse número, vamos considerar uma transição particular $\delta(q,\sigma) = t$. Temos que o número de vezes que o par (B_j, σ), com $t \in B_j$, é escolhido em b é igual ao número de vezes que o estado q é adicionado a D por causa da transição $\delta(q,\sigma) = t$. Suponha então que t se encontra em um B_j, $(B_j, \sigma) \in L$, e o par (B_j, σ) é escolhido em b. Como já foi dito, q será adicionado a D por causa da transição. Na próxima vez em que isso se repetir, isto é, t se encontra em um B'_j e $(B'_j, \sigma) \in L$, teremos que $|B'_j| \leq |B_j|/2$. Isso é verdade porque (B_j, σ) nunca mais entrará em L, B'_j tem que ser o resultado de alguma divisão de B_j. Sejam $\overline{B_j}$ e $\widetilde{B_j}$ o resultado da divisão de B_j. Se o algoritmo insere $\overline{B_j}$ em L, então $\overline{B_j} \leq |B_j|/2 \leq \widetilde{B_j}$. Disso, segue que $|B'_j| \leq |B_j|/2$. Logo, o número de vezes que q é inserido em D por causa da transição $\delta(q,\sigma) = t$ é limitado por $\log n$. Como temos no máximo $|\Sigma|n$ transições, então o número máximo de elementos que passam por D é $O(|\Sigma| n \log n)$.

A prova de que o tempo de c é limitado superiormente por $|\Sigma| n \log n$ pode ser vista em [7]. Ela representa uma dificuldade porque $\delta^{-1}(q,\sigma)$ pode ser vazio, e com isso o número de elementos que passam por D pode ser menor do que o número de operações em c. O algoritmo original de Hopcroft [8] não necessita dessa demonstração, pois, para cada bloco B_i e símbolo σ, mantém um conjunto

$$B_i(\sigma) = \{q \in B_i : \delta^{-1}(q,\sigma) \neq \emptyset\} \ .$$

Isso facilita a demonstração, mas complica o algoritmo e aumenta o espaço em memória utilizado.

Concluindo, apresentamos na tabela seguinte as complexidades *totais* de cada passo do Algoritmo 9.

Passo	Complexidade
b	$O(\|\Sigma\| \, n)$
c	$O(\|\Sigma\| \, n \log n)$
d	$O(\|\Sigma\| \, n)$
e	$O(\|\Sigma\| \, n \log n)$
f	$O(\|\Sigma\| \, n)$
Total	$O(\|\Sigma\| \, n \log n)$

Nós implementamos o algoritmo na linguagem C, padrão ANSI, seguindo as sugestões de estruturas de dados encontradas no artigo de Gries [7]. O espaço em memória utilizado por essas estruturas é $O(|\Sigma|n)$.

3.2.3 O algoritmo de Revuz para autômatos acíclicos

Um autômato $\mathcal{A}$ é acíclico se o grafo $G(\mathcal{A})$ é acíclico. Autômatos acíclicos são bastante usados para representar dicionários [3] e manipular funções booleanas [4].

O algoritmo de minimização que veremos a seguir tem grande importância para essas aplicações, pois além de diminuir o espaço em memória utilizado e

agilizar as buscas com autômatos, ele tem complexidade de tempo $O(|\Sigma|n)$. Lembrando, o algoritmo clássico tem complexidade $O(|\Sigma|n^2)$ e o de Hopcroft $O(|\Sigma|n \log n)$, como já foi visto.

Seguiremos a descrição encontrada em [12].

Definições adicionais

Na *definição do problema* muda apenas a entrada, que é restrita a autômatos finitos determinísticos *acíclicos*. Portanto, a entrada é um autômato acíclico $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$. Com isso, o autômato não deve ser *completo*, isto é, δ é uma função parcial.

Como já foi mencionado, um autômato acíclico $\mathcal{A}$ é um autômato cujo grafo associado $G(\mathcal{A})$ é acíclico.

Seja $X = (x_1, x_2, \cdots, x_k)$ uma seqüência de palavras. Vamos definir o *prefixo comum de* X como sendo a seqüência $Y = (y_1, y_2, \cdots, y_k)$ definida da seguinte maneira. Seja

$$P_i = \{u : u \text{ é um prefixo comum entre } x_i \text{ e } x_j, \text{ para algum } j \neq i\} \quad .$$

Então y_i é o prefixo de x_i tal que $|y_i| = max\{|u| : u \in P_i\}$.

O comprimento do prefixo comum de X é a soma dos comprimentos das palavras em Y. Essa definição será útil para a análise de complexidade de tempo do algoritmo.

Exemplo 3.1 *Se* $X = (a, b, c)$*, então o prefixo comum de* X *será* $Y = (\lambda, \lambda, \lambda)$ *e o comprimento será* 0*. Se* $X = (abd, a, abc, bacd, bacddd, babc, abc)$*, então o prefixo comum será* $Y = (ab, a, abc, bacd, bacd, ba, abc)$*, o comprimento do prefixo comum será* $|ab|+|a|+|abc|+|bacd|+|bacd|+|ba|+|abc| = 19$*.*

A *altura* de um estado q do autômato acíclico $\mathcal{A}$, ou $h(q)$, é o tamanho do caminho de maior comprimento que começa em q e chega a um estado final. Mais formalmente, $h(q) = max\{|x| : \delta(q, x) \in F\}$.

Essa função *altura* induz uma partição $\Pi = \Pi_0, \Pi_1, \cdots, \Pi_{h(q_0)}$ de Q: Π_i será o conjunto ou bloco de estados de altura i. Diremos que conjunto Π_i é *distinto* se nenhum par de estados em Π_i é equivalente.

Idéia do algoritmo

A idéia do algoritmo é simples. Informalmente, dado o autômato acíclico $\mathcal{A}$, a partição Π é calculada e cada estado q é nomeado com uma palavra que descreve as suas transições $\delta(q, \sigma)$. Os nomes dos estados são dados de tal forma que, se dois estados possuem nomes iguais, eles são equivalentes. Portanto, o algoritmo percorre os blocos $\Pi_0, \Pi_1, \Pi_2, \cdots$ nessa ordem, aplicando uma ordenação lexicográfica nos nomes dos estados de cada bloco Π_i, e dessa forma encontrando os estados com nomes iguais. Os estados equivalentes são então unidos, formando um único estado. A nomeação dos estados é feita uma única vez em cada estado, e a ordenação é feita em

tempo linear, logo, a complexidade de tempo total é linear no número de transições ($O(|\Sigma|n)$).

A dificuldade maior está em criar os nomes de forma que o gasto total com as ordenações seja de fato linear. Embora o algoritmo de ordenação em si seja linear no tamanho da entrada, o tamanho dos nomes poderia não ser linear em relação ao tamanho do autômato.

O algoritmo

A seguinte propriedade relaciona a função *altura* com a equivalência dos estados.

Propriedade 7 *Se todo Π_j, com $j < i$, é distinto, então dois estados p e q em Π_i são equivalentes se e somente se para qualquer símbolo $\sigma \in \Sigma$ vale que $\delta(q, \sigma) = \delta(p, \sigma)$.*

Com essa propriedade, é possível construir um algoritmo simples. Primeiro devemos criar a partição Π induzida pelas alturas. Isso é feito percorrendo $G(\mathcal{A})$ como em uma busca em profundidade. A complexidade desse passo é $O(e)$, onde o número de transições e é no máximo $|\Sigma|n$. Em seguida, para cada nível encontramos os estados equivalentes através da ordenação lexicográfica. Veja o Algoritmo 10.

Algoritmo 10

```
calcule a partição Π = Π_0, Π_1, ···, Π_{h(q_0)} induzida pelas alturas;
para i ← 0 até h(q_0) faça
    ordene os estados de Π_i pelos seus arcos;
    una todos os estados equivalentes;
fim.
```

(10)

A Propriedade 7 pode mostrar que esse algoritmo é correto. Ademais, podemos executar o passo de união dos estados integrado ao passo da ordenação. Isso nos permite enunciar o seguinte lema:

Lema 8 *Usando uma ordenação com complexidade de tempo $O(f(n))$, o Algoritmo 10 minimiza um autômato acíclico em $O(\sum_{i=0}^{h(q_0)} f(|\Pi_i|))$. A complexidade total é $O(e + \sum_{i=0}^{h(q_0)} f(|\Pi_i|))$.*

A seguir detalharemos o passo de ordenação e união dos estados.

Ordenação lexicográfica

Para a ordenação, nomearemos cada estado com uma palavra e aplicaremos uma ordenação lexicográfica. Essa ordenação consiste de várias aplicações de um outro algoritmo conhecido como *bucket sort*. Note que o algoritmo

que usaremos é um pouco mais simples, pois queremos apenas distingüir os elementos.

- *Bucket sort*

  Ordena uma seqüência $a_1, a_2, \cdots, a_n$ de inteiros $1 \leq a_i \leq m$ executando os seguintes passos:

  1. Construa um vetor B de filas vazias: $B[1..m]$;
  2. Percorra a seqüência $a_1, a_2, \cdots, a_n$ inserindo o elemento a_i em $B[a_i]$, o a_i-ésimo *bucket*;
  3. A concatenação das filas $B[1]B[2]\cdots B[m]$ é a seqüencia ordenada.

  As complexidades de tempo e espaço são $O(n + m)$.

- *Ordenação lexicográfica*

  Ordena uma seqüência $S = (A_1, A_2, \cdots, A_n)$ de k-uplas $A_i = (a_{i1}, a_{i2}, \cdots, a_{ik})$, onde $1 \leq a_{ij} \leq m$. No primeiro passo, aplica-se o algoritmo *bucket sort* na seqüência S considerando o k-ésimo inteiro de cada k-upla, ou seja, o *bucket sort* considerará a seqüência $a_{1k}, a_{2k}, \cdots, a_{nk}$, mas ordenará na verdade a seqüência S. O próximo passo considerará o $(k-1)$-ésimo inteiro de cada A_i e a seqüência S já ordenada de acordo com o k-ésimo inteiro. Portanto, por indução, a seqüência final é a seqüência S ordenada.

  A complexidade de tempo é $O(k(n + m))$ e o espaço em memória é $O(n + m)$.

A generalização da ordenação lexicográfica para seqüências de k-uplas de tamanho variável, entre 1 e L_{max}, é feita ordenando-se primeiro pelo tamanho das k-uplas e em seguida aplicando-se as ordenações, com uma pequena diferença. Quando o algoritmo considera o l-ésimo inteiro apenas as k-uplas com pelo menos l inteiros são ordenadas.

Uma descrição completa desse algoritmo pode ser encontrada em [2]. Com um pouco mais de sofisticação, esse algoritmo pode ser melhorado aplicando-se os *bucket sorts* da esquerda para a direita. Não explicaremos os detalhes aqui, mas usaremos essa idéia no Algoritmo 11, mostrado mais adiante.

Eis o que o Algoritmo 11 faz.

Lema 9 *Seja uma seqüência de n k-uplas, com k variável, onde cada componente de uma k-upla está entre* 1 *e* m. *O Algoritmo 11 devolve a lista das k-uplas que são iguais a pelo menos uma outra k-upla da seqüência. O tempo gasto pelo algoritmo é* $O(n')$, *onde* n' *é o* comprimento do prefixo comum *da seqüência. O espaço em memória necessário é* $O(n + m)$.

O algoritmo final

Vamos tentar finalmente juntar o Algortimo 11 ao Algoritmo 10 para obtermos um algoritmo linear. Para isso é preciso discutir a nomeação dos estados. Cada estado q receberá um rótulo, definido da seguinte maneira.

$$\textit{rótulo}(q) = (F \textit{ ou } N,\, \sigma_1,\, p_1,\, \sigma_2,\, p_2,\, \cdots,\, \sigma_k,\, p_k)\ ,$$

onde F ou N diz se o estado q é final ou não, σ_i é o símbolo do i-ésimo arco, e p_i é um identificador (número) do estado apontado pelo i-ésimo arco. A subseqüência $(\sigma_1, \sigma_2, \cdots, \sigma_k)$ deve estar ordenada.

De acordo com o Lema 8, com esses rótulos nós obteremos uma complexidade de tempo $O(\sum_{i=0}^{h(q_0)} r'_i + e)$ no total, onde r'_i é o *comprimento do*

Algoritmo 11

Entrada: a seqüência $A_1, A_2, \cdots, A_n$ de k_i-uplas $(a_1, a_2, \cdots, a_{k_i}, \$)$, onde a_j está entre 1 e m.
Saída: LIGUAIS, a lista de k_i-uplas iguais.

```
coloque A_1, A_2, ..., A_n em LISTA;
insira LISTA em FILA2;
i ← 0;
repita
    mova FILA2 para FILA1;
    i++;
    enquanto FILA1 não vazia faça
      seja L a primeira lista em FILA1;
      remova L de FILA1;
      enquanto L não vazia faça
        seja A= (a_1, a_2, ..., a_k, $) a primeira k-upla em L;
        remova A de L;
        se B[a_i] vazio
          então adicione a_i a NAOVAZIOS;
        mova A para o bucket B[a_i];
      fim;
      para cada a ∈ NAOVAZIOS tal que a ≠ $
                    e B[a] tenha mais de um elemento faça
        mova os elementos de B[a] para uma lista LTEMP;
        insira LTEMP em FILA2;
      fim;
      descarte os buckets que contenham apenas uma k-upla;
      mova os elementos de B[$] para LIGUAIS;
    fim;
até FILA2 vazia;
```

(11)

prefixo comum dos rótulos dos estados de altura i. Portanto, para obtermos a linearidade, precisamos limitar o tamanho dos rótulos, que de certa forma dependem da representação dos valores p_i, os números dos estados. Por exemplo:

- Se representarmos esses números com dígitos, o comprimento da representação de cada um deles será limitado por $\log |Q|$. Assim, o comprimento dos rótulos crescerá por um fator $\log |Q|$, ou seja, o tempo não será linear.

- Podemos então enxergar os números como sendo letras, mas assim o tamanho do vetor B de filas (*buckets*) terá que ser $max\{|Q|, |\Sigma|\}$. Entretanto, note que desse modo, teremos que

$$\sum_{i=0}^{h(q_0)} r'_i = \sum_{i=0}^{h(q_0)} |E_i| \ ,$$

  no pior caso, onde E_i é o número de arcos que saem dos estados de Π_i. Então a complexidade de tempo será:

$$O(\sum_{i=0}^{h(q_0)} |E_i| + e) = O(e + e) = O(|\Sigma| n) \ .$$

Nós diminuiremos o limitante para o tamanho do vetor B com uma renumeração dos estados. A idéia é reutilizar os números a cada ordenação. Para isso, um estado só receberá um número se ele for usado. Então a cada estado estará associado um par (alt, num). O valor num é o número atribuído ao estado quando ele foi usado na ordenação do bloco Π_{alt}. Se estivermos ordenando o bloco Π_i e precisarmos do estado q, cujo par é $(h, número)$, teremos duas possibilidades:

- Se $h \neq i$, então *número* não é mais válido. O par de q é trocado pelo par $(i, novo_número)$ e o valor usado pelo *bucket sort* é *novo_número*.

- Se $h = i$, então *número* é usado.

O Algoritmo 12 é uma função que recebe um estado q, uma altura h e um número n, e devolve um número que representa o estado q num determinado momento da execução do algoritmo, conforme a discussão acima.

Agora o tamanho do vetor B está limitado por $max\{|\Sigma|, max\{E_i\}_{i=0}^{h(q_0)}\}$.

Finalmente, o Algoritmo 13 mostrado mais à frente é a versão final.

3.3 Uso de autômatos finitos num problema de busca de padrões

Neste projeto, estudamos também um problema particular de busca de padrões em textos. Trata-se do caso em que o padrão é um conjunto finito K de palavras.

Algoritmo 12

```
função renumera(estado q, altura h, número n)
  se (q.alt ≠ h) então
    q.alt ← h;
    q.num ← n;
    n++;
  devolva (q.num);
fim.
```
(12)

O nosso estudo foi centrado no algoritmo clássico de Alfred Aho e Margaret Corasick. Em [1] eles apresentam o algoritmo, tendo como motivação a otimização de um sistema de consulta a um banco de dados de referências bibliográficas. Os resultados em relação aos algoritmos convencionais da época foram excelentes. Outras descrições do algoritmo podem ser encontradas em [5, 6].

Vamos então descrever o problema e o modelo de solução, que envolve a construção de um autômato finito determinístico que representa as palavras em K. Para essa construção foram aplicadas algumas idéias do algoritmo KMP, de Knuth, Morris e Pratt [10]. Esse algoritmo resolve o problema da busca de padrões para o caso em que o padrão é uma palavra.

3.3.1 Descrição do Problema

Seja $K = \{y_1, y_2, \cdots, y_k\}$ um conjunto finito de palavras em Σ^*, as quais chamaremos de *palavras-chave*, e x, também em Σ^*, uma palavra qualquer que chamaremos de *texto*. O problema que queremos resolver é localizar e identificar todos os fatores de x que são também *palavras-chave*.

Para isso, utilizaremos um autômato que reconhece a linguagem Σ^*K. O autômato recebe como entrada o *texto* x e gera uma saída contendo as posições em x onde alguma *palavra-chave* aparece como fator. Essa fase é a *busca* propriamente dita, e sua complexidade de tempo é $O(|x|)$, dependendo da implementação da função de transição. Note que esse tempo não depende do número de *palavras-chave*.

Há também a fase de construção do autômato. Ela é feita em duas etapas, em tempo $O(|\Sigma'|m)$ no total, onde m é a soma dos comprimentos das palavras em K e $\Sigma' \subseteq \Sigma$ é o conjunto dos símbolos que ocorrem em K. A primeira etapa consiste em construir, a partir do conjunto K, uma máquina de estados muito semelhante a um autômato finito determinístico. Essa construção utiliza as idéias do algoritmo KMP, e pode ser feita em tempo $O(m)$, dependendo da implementação. Em seguida, a partir da máquina de estados, obtém-se em tempo $O(|\Sigma'|m)$ o autômato finito determinístico, que reconhece a linguagem Σ^*K.

Portanto, o tempo de construir e aplicar a máquina de estados é $O(|x| + m)$. Note que aplicando o algoritmo KMP k vezes com entrada x, uma vez

Algoritmo 13

```
Calcule h para todos os estados e crie a partição Π;
Una todos os estados em Π_0; // eles são equivalentes
para j ← 1 até h(q_0) − 1 faça
   coloque Π_j como uma lista em FILA2; i ← 0;
   repita
      mova FILA2 para FILA1;
      i++;
      enquanto FILA1 não vazia faça
        seja L a primeira lista em FILA1; remova L de FILA1;
        enquanto L não vazia faça
          seja S o primeiro estado em L; remova S de L;
          mova S para o bucket B[a_i];
        fim;
        para cada a ≠ $ tal que
                        B[a] tem mais de um elemento faça
          insira os elementos de B[a] como uma lista em FILA2;
        fim;
        una todos os estados em B[$];
        reinicialize os buckets;
      fim;
   até FILA2 vazia;
fim.
```

(13)

para cada palavra em K, a complexidade total de pior caso seria $O(k|x|+m)$.

3.3.2 O algoritmo de Aho e Corasick

Como já mencionamos, o algoritmo funciona em duas fases. A primeira constrói um autômato finito determinístico que reconhece a linguagem Σ^*K. A segunda executa a *busca* fornecendo o texto x como entrada para o autômato.

A máquina de estados inicial

Começaremos a construção do autômato com a construção de uma máquina de estados que possui um conjunto finito de estados e três funções. Essa máquina funcionará da seguinte forma: ela recebe uma palavra x e lê cada símbolo de x em seqüência. Para cada símbolo, ela executa algumas transições de estado e possivelmente gera uma saída. De fato, ela é muito semelhante a um autômato finito determinístico, a diferença é que há duas funções de transição, a usual, que chamaremos de g, e a de falha, que chamaremos de f. A função f é usada para "voltar" algumas transições no caso em que a função g indica $falha$; ela tem o mesmo significado da função de

falha do algortimo KMP. Há também a função *saída*, que associa uma saída a cada estado.

Para facilitar o nosso estudo, vamos definir a máquina de estados a partir de um autômato finito determinístico.

A máquina de estados é uma tripla $(\mathcal{B}, f, saída)$, onde

- $\mathcal{B} = (Q \cup \{falha\}, \Sigma, g, q_0, F)$ é um autômato finito determinístico onde

  - Q é um conjunto finito de estados;
  - Σ é o alfabeto de entrada;
  - $g : Q \times \Sigma \to Q \cup \{falha\}$ é a função de transição;
  - q_0 é o estado inicial;
  - F é o conjunto de estados finais.

  Note que adicionamos um estado com nome $falha$. Para todo q em Q e todo σ em Σ, chamaremos as transições tais que $g(q, \sigma) = falha$ de transições *não definidas*. Convencionamos também que $g(q_0, \sigma) \neq falha$, para todo $\sigma \in \Sigma$.

- $f : Q \to Q$ é a função de transição de falha;
- $saída : Q \to 2^K$ é a função de saída;

Cada *ciclo* da máquina é definido da forma a seguir. Seja q o estado atual e σ o símbolo corrente da entrada x.

1. Se $g(q, \sigma) = q'$, $q' \in Q$, a máquina faz uma transição usual, correspondente à transição do autômato $\mathcal{B}$. Ou seja, muda para o estado q' e avança a cabeça de leitura. Se $saída(q') \neq \emptyset$, então o máquina emite $saída(q')$ e a posição do último símbolo lido como parte da saída. A máquina então começará outro ciclo.

2. Se $g(q, \sigma) = falha$, que é uma transição *não definida* em $\mathcal{B}$, a máquina faz uma *transição de falha*: digamos que $f(q) = q'$, então a máquina reinicia o ciclo com $q = q'$ e σ continuando como símbolo corrente. Note que a máquina está fazendo uma nova tentativa de encontrar uma palavra-chave, já que o autômato $\mathcal{B}$ apontou uma falha.

O Algoritmo 14 descreve mais precisamente o comportamento da máquina.

Algoritmo 14

Entrada: Uma palavra $x = \sigma_1, \sigma_2, \cdots, \sigma_n$, onde cada σ_i é um símbolo de entrada, e a máquina de estados descrita acima.
Saída: As posições das ocorrências das palavras-chave em x.

```
estado ← 0;
para i ← 1 até n faça
   enquanto g(estado, σ_i) = falha faça
     estado ← f(estado);
   fim;
   estado ← g(estado, σ_i);
   se saída(estado) ≠ ∅
     então
        imprima i;
        imprima saída(estado);
fim.
```

(14)

Para que essa máquina seja capaz de resolver o problema, ela deve satisfazer os requisitos que discutiremos informalmente a seguir.

1. Considerando o grafo $G(\mathcal{B})$ sem as transições *não definidas* e sem os possíveis laços do estado q_0, teremos que ter uma árvore onde:

   - Cada nó (estado) representa um prefixo de alguma palavra-chave.
   - A raiz é o estado q_0 e representa a palavra vazia.
   - Sejam q e p em Q e σ em Σ. Sejam também x o prefixo representado por q e y o prefixo representado por p. Então, a transição $g(q, \sigma) = p$ significa que $x\sigma = y$.

   Essa árvore é conhecida como árvore de busca digital que contém as palavras em K. Vamos chamar essa árvore de T.

   Além disso, por definição, faremos com que $g(q_0, \sigma) = q_0$, para todo σ tal que $g(q_0, \sigma)$ não foi definido acima. Para as outras transições $g(q, \sigma)$ não definidas, onde $q \in Q$ e $\sigma \in \Sigma$, faremos com que $g(q, \sigma) = falha$.

2. Sejam q e p em Q, u o prefixo representado por q e v o prefixo representado por p. A função f deve ser tal que $f(q) = p$ se e somente se v é o sufixo de maior comprimento de u que é também prefixo de alguma palavra em K. A função $falha$ definida desse modo tem grande importância para a complexidade final do algoritmo. Ela permite que na busca nunca seja necessário voltar na entrada x. Lembrando novamente, essa é a mesma idéia usada no algoritmo KMP.

3. Sejam q em Q e u o prefixo representado por q. A função *saída* deve ser tal que

$$saída(q) = \{v : v \in K \text{ e } v \text{ é sufixo de } u\} \quad .$$

Construção da máquina de estados

Construiremos primeiro o autômato $\mathcal{B}$, e nessa construção já é possível definir parte da função *saída*. Então, a partir de $\mathcal{B}$, construiremos a função f. A construção definitiva de *saída* também será feita em conjunto com a contrução de f.

A construção de $\mathcal{B}$ será feita a partir da árvore T descrita anteriormente. Inicialmente, a árvore T só possui o nó raiz, que, lembrando, representa a palavra vazia. Para cada palavra y em K, insira y em T da seguinte forma.

1. Percorra T até o estado p de forma que p represente o maior prefixo de y que esteja em T.

2. A partir de p, insira um novo caminho em T de tal forma que o último estado do caminho represente a palavra y. Seja q esse último estado.

3. Defina $saída(q) = \{y\}$.

Execute então a finalização a seguir.

1. Para todo $\sigma \in \Sigma$ tal que $g(q_0, \sigma)$ ainda não foi definido, considere $g(q_0, \sigma) = q_0$.

2. Para todo $\sigma \in \Sigma$ e $q \in Q$ tais que $g(q, \sigma)$ ainda não foi definido, considere $g(q, \sigma) = falha$.

3. O conjunto de estados finais de $\mathcal{B}$ é $F = \{q \in Q : saída(q) \neq \emptyset\}$.

O Algoritmo 15 mostra esse procedimento em linguagem mais precisa.

Para a construção de f e o restante de *saída*, novamente nos guiaremos pelos requisitos discutidos anteriormente.

Usaremos o Algoritmo 16, que percorre a árvore T como em uma busca em largura. Portanto, fica claro que a construção de f é feita a partir da função g. Vamos introduzir então a noção de *profundidade*. A *profundidade* de um estado q em T é o tamanho do caminho de menor comprimento que começa em q_0 e termina em q.

O Algoritmo 16 percorre a árvore por nível de profundidade, começando da profundidade 1. A função falha de um estado é definida a partir dos estados das profundidades menores do que a dele. Podemos já definir inicialmente $f(q) = q_0$ para todo q que tenha profundidade 1. Suponha agora que f já tenha sido definida para todos os estados de nível menor do que d. Sejam $q \in Q$ um estado de nível $d-1$ e $p \in Q$ um estado tal que $g(q, \sigma) = p$ para algum $\sigma \in \Sigma$ (p é de nível d), queremos definir $f(p)$. Seja u a palavra representada por q em T, e $v = u\sigma$ a palavra representada por p. Temos que $f(q) = r \in Q$ representa o prefixo de maior comprimento z de alguma palavra-chave que é também um sufixo de u. O algoritmo então procura pelo estado $g(r, \sigma) = s$ que representa o prefixo $z\sigma$, tal que $v = tz\sigma$, para algum $t \in \Sigma^*$. Se $s \neq falha$, então podemos definir $f(p) = s$. Caso contrário, consideramos $f(r)$, que também representa um sufixo de u que é

Algoritmo 15

Entrada: O conjunto de palavras-chave $K = \{y_1, y_2, \cdots, y_k\}$.
Saída: A função g e a função *saída* parcialmente definida.
Observação: Assumiremos que, para todo novo estado q criado, $g(q, \sigma) = falha$ para todo $\sigma \in \Sigma$.

```
novoestado ← 0;
q_0 ← novoestado;
saída(q_0) = ∅;

para i ← 1 até k faça
    seja y_i = σ_1 σ_2 ··· σ_{m_i};
    estado ← q_0;   j ← 1;
    enquanto g(estado, σ_j) ≠ falha faça
        estado ← g(estado, σ_j);
        j++;
    fim;
    para p ← j até m_i faça
        novoestado++;
        g(estado, σ_p) ← novoestado;
        saída(novoestado) = ∅;
        estado ← novoestado;
    fim;
    saída(estado) ← {y_i};
fim;

para todo σ ∈ Σ tal que g(q_0, σ) = falha faça
    g(q_0, σ) ← q_0;
fim.
```

(15)

prefixo de alguma palavra-chave. Logo, podemos aplicar a mesma idéia até encontrarmos $f(p)$. Num caso extremo, $f(p) = q_0$, pois $g(q_0, \sigma) \neq falha$.

Ao definirmos $f(p)$, temos que a palavra w representada por $f(p)$ é um sufixo da palavra v representada por p. Logo, podemos dizer que $saída(f(p))$ deve estar contido em $saída(p)$.

Veja então o Algoritmo 16.

Comentários sobre as complexidades

- *Busca*

  Podemos ver que cada passo do laço mais externo do Algoritmo 14 processa um símbolo de x. Logo, o número de iterações é $|x|$. É visível também que a cada iteração, o algoritmo executa uma transição g, então o número de transições usuais também é $|x|$. Temos ainda que o número total de transições de falha não pode ultrapassar o número

Algoritmo 16

Entrada: As funções g e *saída* devolvidas pelo Algoritmo 15.
Saída: Funções f e *saída*.

```
fila ← fila vazia;
para cada σ tal que g(q0, σ) ≠ q0 faça
   s = g(q0, σ);
   insira s na fila;
   f(s) ← q0;
fim;
enquanto fila não vazia faça
   seja r o próximo estado da fila;
   remova r da fila;
   para cada σ tal que g(r, σ) ≠ falha faça
      s = g(r, σ);
      insira s na fila;
      estado ← f(r);
      enquanto g(estado, σ) = falha faça
         estado ← f(estado);
      fim;
      f(s) ← g(estado, σ);
      saída(s) ← saída(s) ∪ saída(f(s));
   fim;
fim.
```

(16)

de transições usuais em nenhum momento da execução do algoritmo. Logo, o número de transições de falha é, no pior caso, $|x|$. Disso segue que o tempo gasto pelo Algoritmo 14 é $O(|x|)$.

- *Construção*

  Não há muitas dificuldades para observarmos que o Algoritmo 15 tem complexidade de tempo $O(m)$.

  O Algoritmo 16 também possui complexidade de tempo $O(m)$, mas é preciso usar um argumento semelhante ao da justificativa da *busca*.

Portanto, o algoritmo todo, que compreende a *construção* e a *busca*, tem complexidade de tempo $O(|x| + m)$.

Obtenção do autômato finito determinístico final

Podemos obter um autômato finito determinístico a partir da máquina de estados descrita até agora.

Para isso definiremos a função de transição δ de forma que ela faça o papel das funções g e f. Note que desse modo a busca fica mais simples,

assim como o cálculo da complexidade de tempo, pois é feita exatamente uma transição por símbolo da entrada x.

A idéia é esboçada a seguir. Sejam $q \in Q$ e $\sigma \in \Sigma$. Se $g(q, \sigma) = falha$, então podemos dizer que $\delta(q, \sigma) = \delta(f(q), \sigma)$. Caso contrário, temos simplesmente que $\delta(q, \sigma) = g(q, \sigma)$.

A descrição completa pode ser vista no Algoritmo 17.

Algoritmo 17

Entrada: A função g devolvida pelo Algoritmo 15 e a função f devolvida pelo Algoritmo 16.
Saída: A função δ para o autômato final.

```
fila ← fila vazia;
para cada σ ∈ Σ faça
   δ(q0, σ) = g(q0, σ);
   se g(q0, σ) ≠ q0 então insira g(q0, σ) na fila;
fim;
enquanto fila não vazia faça
   seja r o próximo estado da fila;
   remova r da fila;
   para cada σ ∈ Σ faça
     se g(r, σ) ≠ falha
        então
           s = g(r, σ);
           insira s na fila;
           δ(r, σ) ← s;
        senão δ(r, σ) ← δ(f(r), σ);
   fim;
fim.
```

(17)

A complexidade de tempo dessa construção é $O(|\Sigma|m)$. O número de operações da busca usando a máquina de estados pode ser reduzido em até metade se usarmos o autômato, já que o autômato não faz transições de falha. Porém, não há uma forma confiável de estimar essa redução. Ademais, a complexidade de tempo da busca é a mesma.

4 Observações e Conclusões

Não foi possível cumprir o plano integralmente. Porém, a renovação da bolsa foi aprovada, e o novo plano prevê o estudo dos tópicos que não puderam ser pesquisados e uma continuação. Em resumo, o novo plano é o seguinte:

- Implementar e testar os algoritmos de Revuz e Aho-Corasick.
- Estudar o Autômato dos Sufixos.

Independentemente das metas não cumpridas, o projeto tem preparado o aluno para futuros projetos que envolvam estudo de aplicações mais complexas da Teoria de Autômatos Finitos. Ademais, as atividades realizadas até o momento fizeram com que o aluno chegasse mais próximo ao ambiente de pesquisa e da pós-graduação.

Referências

[1] A.V. Aho and M.J. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[2] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms*. Addison Wesley: Reading, MA, 1974.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Priciples, Techniques and Tools*. Addison Wesley: Reading, MA, 1986.

[4] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(5):677–691.

[5] M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2, Linear Modeling: Background and Application, chapter 9, pages 399–462. Springer-Verlag, 1997.

[6] M. Crochemore and C. Hancart. Pattern matching in strings. In M.J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 11, pages 11.1–11.28. CRC Press, 1998.

[7] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[8] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[9] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[10] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in string. *SIAM Journal of Computing*, 6:323–350, 1977.

[11] H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation*. Prentice Hall, 1997.

[12] D. Revuz. Minimization of acyclic deterministic automata in linear time. *Theoretical Computer Science.*, 92:181–189, 1992.

São Paulo, 10 de dezembro de 2001.

Aluno: Leo Kazuhiro Ueda

Orientadora: Nami Kobayashi