MAC 449 – Trabalho de Formatura Supervisionado

IME-USP – BCC, 2001 – Prof. responsável: Carlos Eduardo Ferreira

Texto do Pôster

Aluno: Leo Kazuhiro Ueda
Supervisora: Nami Kobayashi

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

dezembro de 2001

1 Introdução

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.

A quantidade e a diversidade dessas aplicações vem 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.

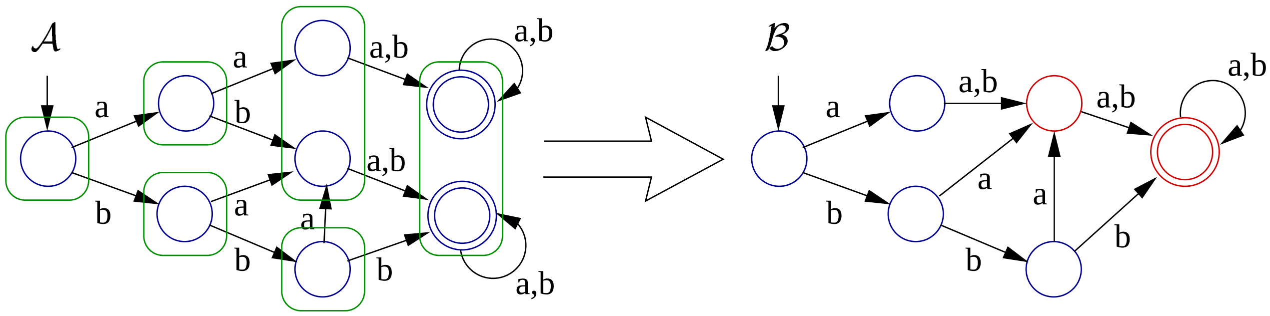
Para compreender e estudar essas aplicações, é importante conhecer os algoritmos e estruturas de dados fundamentais que utilizam autômatos finitos. O objetivo deste projeto foi iniciar um estudo de tais aspectos básicos da Teoria de Autômatos Finitos.

2 Minimização de Autômatos Finitos

Estudamos dois algoritmos para minimização de autômatos finitos determinísticos, a versão de David Gries para o algoritmo de John Hopcroft, com complexidade de tempo $O(|\Sigma|n \log n)$, e o algoritmo de Dominique Revuz para autômatos acíclicos, com complexidade $O(|\Sigma|n)$, onde n é o número de estados do autômato.

O problema que queremos resolver é minimizar o número de estados de uma autômato finito determinístico (*afd*). Isto é, dado um *afd* $\mathcal{A}$, queremos determinar um *afd* $\mathcal{B}$ com o menor número de estados e tal que $L(\mathcal{B}) = L(\mathcal{A})$.

Os métodos utilizados pelos dois algoritmos que estudamos se baseiam numa relação de equivalência sobre o conjunto de estados Q de $\mathcal{A}$. De modo muito simplificado, os algoritmos procuram pelas classes de equivalência em Q.



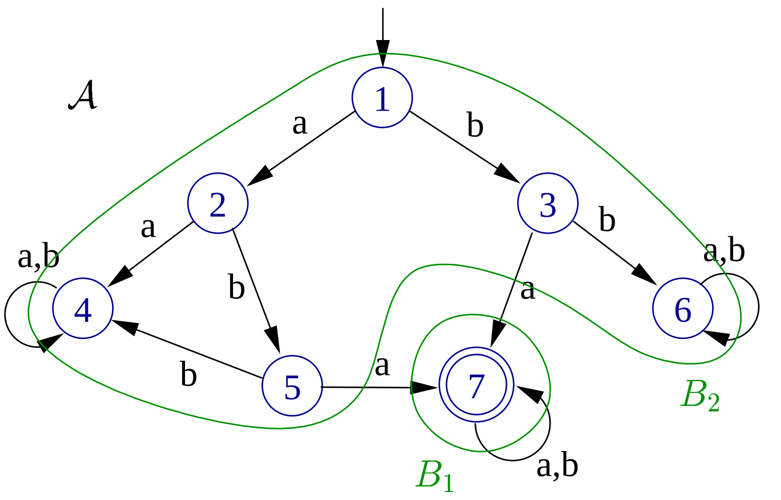
2.1 O algoritmo de Hopcroft

Em [7], Hopcroft apresenta um algoritmo para minimizar o número de estados de um *afd*, e afirma que ele tem complexidade de tempo $O(|\Sigma| n \log n)$.

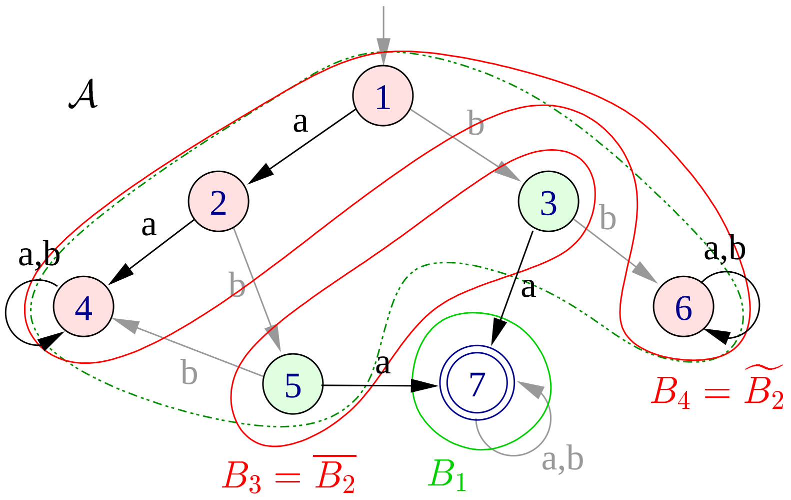
Até aquele momento, os algoritmos conhecidos levavam tempo $O(|\Sigma|n^2)$, ou mais, no pior caso. Porém, a descrição do novo algoritmo e as suas provas de corretude e complexidade de tempo foram consideradas complicadas demais.

Com o objetivo de fornecer uma descrição mais clara, David Gries [6] re-descreve o algoritmo de Hopcroft, mas com uma pequena modificação. O nosso estudo foi baseado nessa descrição.

Para encontrar as classes de equivalência, o algoritmo de Hopcroft começa com uma partição dos estados do autômato $\mathcal{A}$. Essa partição é formada por dois blocos: um contendo os estados finais e outro contendo os não finais.



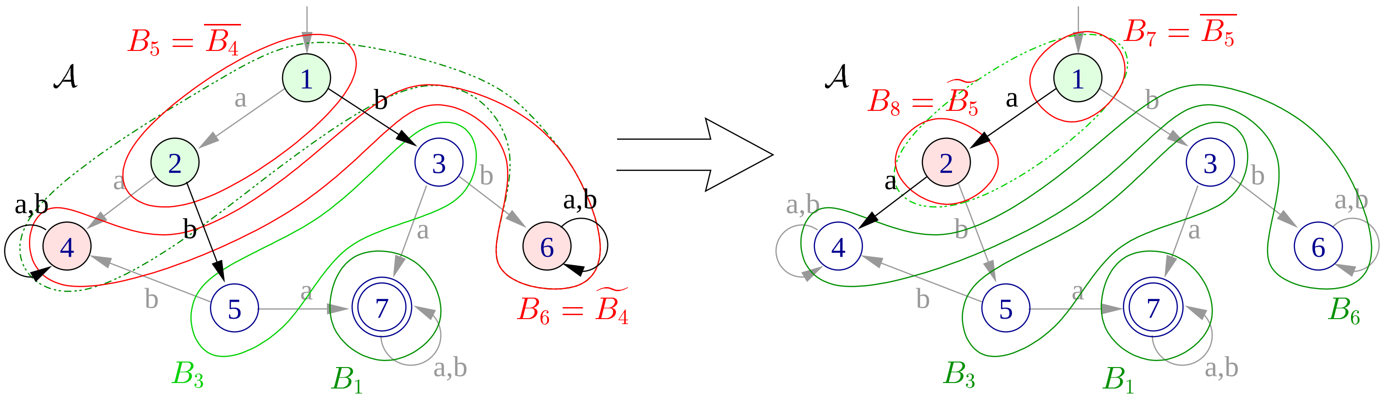
Cada passo do algoritmo é um refinamento dos blocos da partição. Os blocos são divididos de forma que os estados equivalentes permanecem em blocos iguais. A figura seguinte mostra a divisão do bloco B_2 em relação ao bloco B_1 e ao símbolo a .



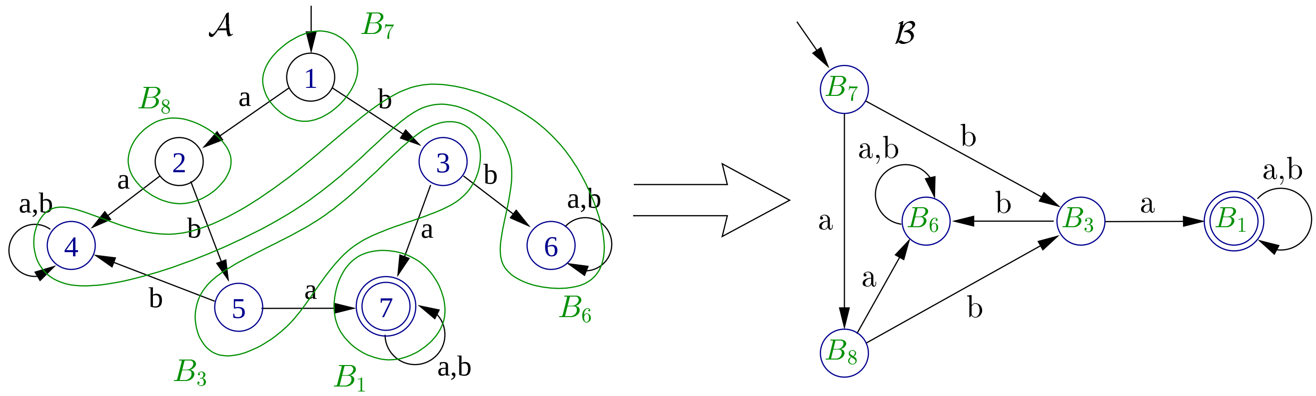
No bloco $\overline{B_2}$ ficaram os estados que possuem uma transição com rótulo a que vai para algum estado do bloco B_1. No bloco $\widetilde{B_2}$ ficaram os estados que não possuem tal transição.

A característica mais importante do algoritmo é a ordem em que são feitas essas divisões, mais precisamente, *como* ela é definida. Veja o resto das divisões.

B_4 em relação a B_3 e b: B_5 em relação a B_5 e a:



Com isso temos as classes de equivalência, e portanto o autômato minimal.

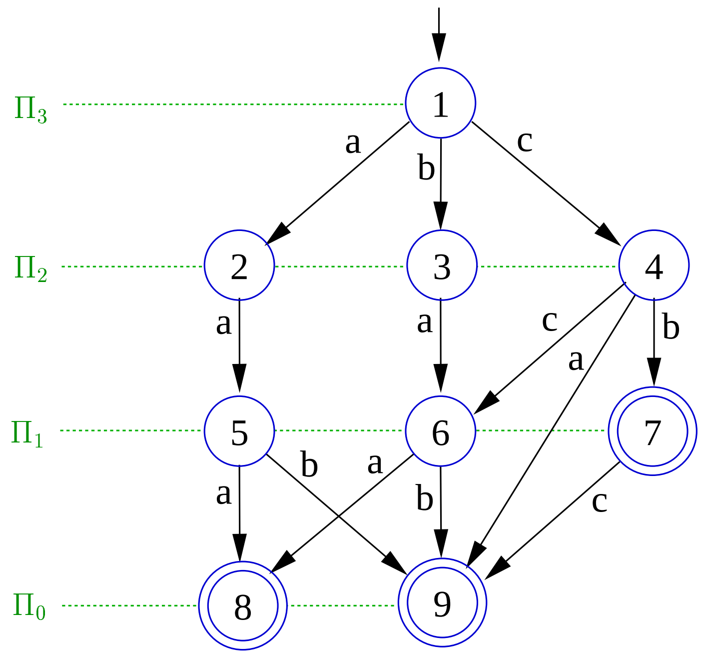


2.2 O algoritmo de Revuz para autômatos acíclicos

Um autômato $\mathcal{A}$ é acíclico se o grafo $G(\mathcal{A})$ é acíclico. Autômatos acíclicos são bastante usados para representar dicionários [2] e manipular funções booleanas [3].

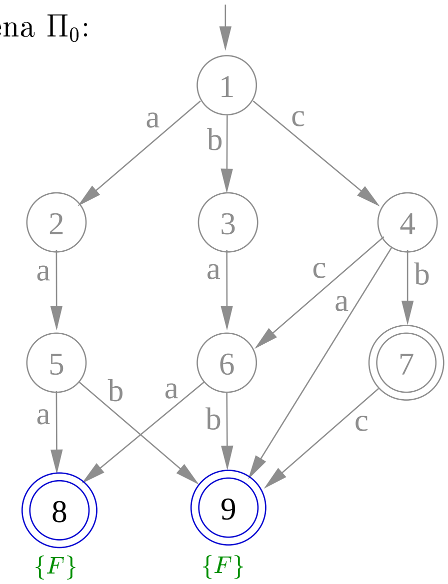
O algoritmo de Revuz tem grande importância para essas aplicações, pois além de diminuir o espaço em memória utilizado e agilizar as buscas com autômatos, ele tem complexidade de tempo $O(|\Sigma|n)$.

O algoritmo começa calculando uma partição $\Pi = \Pi_0, \Pi_1, \cdots$ dos estados de $\mathcal{A}$. Os estados de um bloco Π_i possuem *altura* i. A *altura* de um estado é o tamanho do caminho de maior comprimento que começa no estado e chega a um estado final. Note, portanto, que é possível calcular Π fazendo uma busca em largura no grafo $G(\mathcal{A})$.

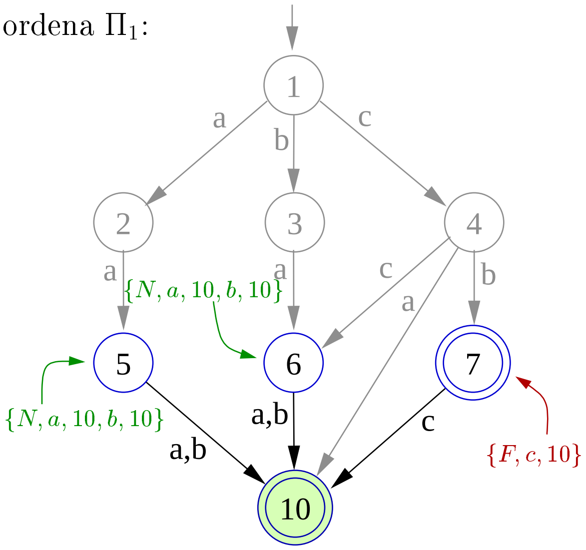


A idéia do algoritmo é nomear cada estado com uma palavra que descreve as suas transições, de forma que dois estados possuem nomes iguais se e somente se eles são equivalentes. Tendo isso, o algoritmo percorre os blocos $\Pi_0, \Pi_1, \Pi_2, \cdots$ nessa ordem, aplicando uma ordenação lexicográfica 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.

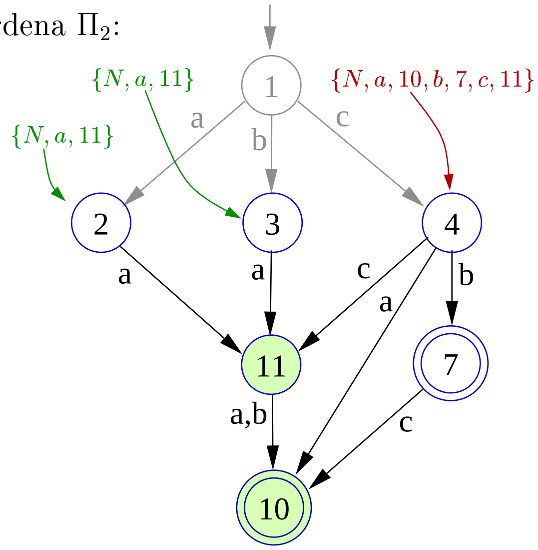
ordena Π_0:



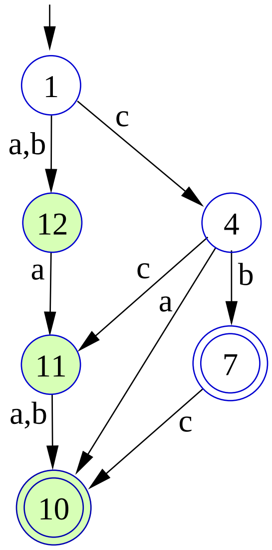
ordena Π_1:



ordena Π_2:



ordena Π_3:



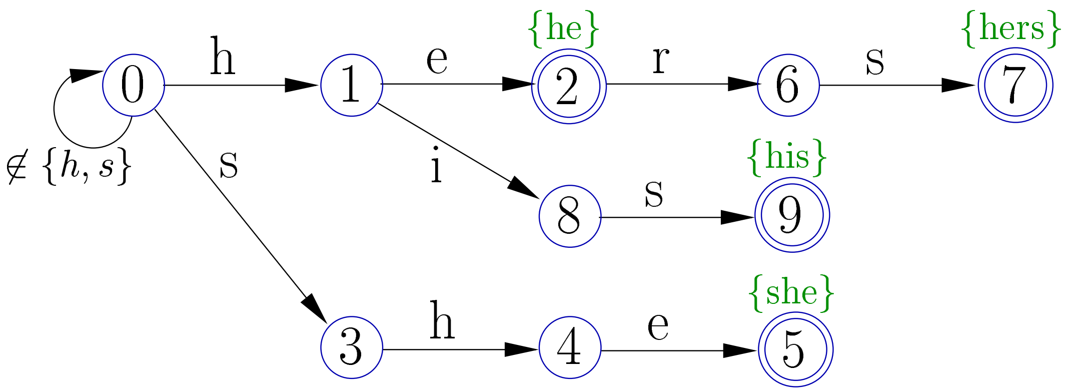
3 Um problema de busca de padrões

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.

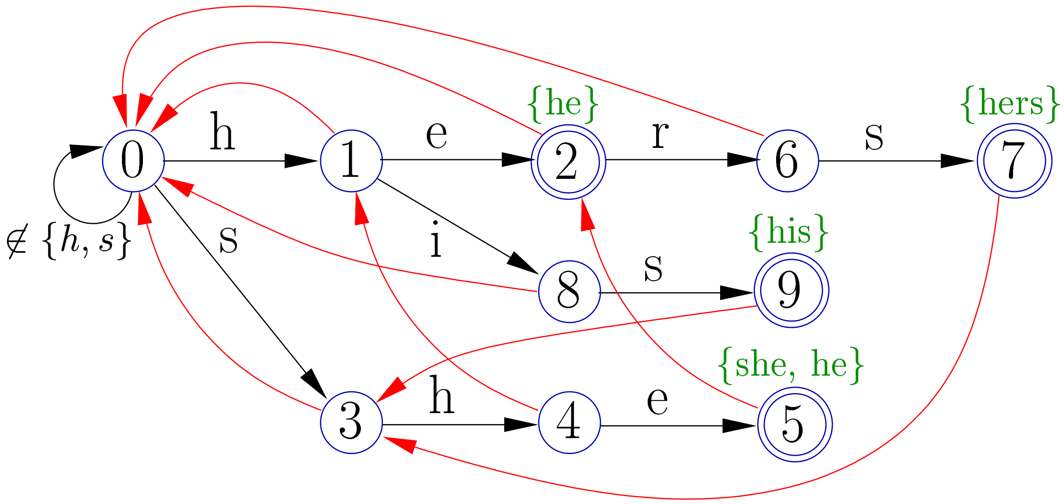
O nosso estudo foi centrado no algoritmo clássico de Alfred Aho e Margaret Corasick. Em [1] eles apresentam o algoritmo, tendo como motivação a otimização de um sistema de consulta a um banco de dados de referências bibliográficas. Os resultados em relação aos algoritmos convencionais da época foram excelentes. Outras descrições do algoritmo podem ser encontradas em [4, 5].

Sejam Σ um alfabeto; $K = \{y_1, y_2, \cdots, y_k\}$ 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*.

O algoritmo inicialmente constrói um autômato que reconhece a linguagem $\Sigma^* K$. Essa construção é feita em etapas, a primeira constrói um autômato que reconhece o dicionário K. Vejamos um exemplo para $K = \{he, she, hers, his\}$.

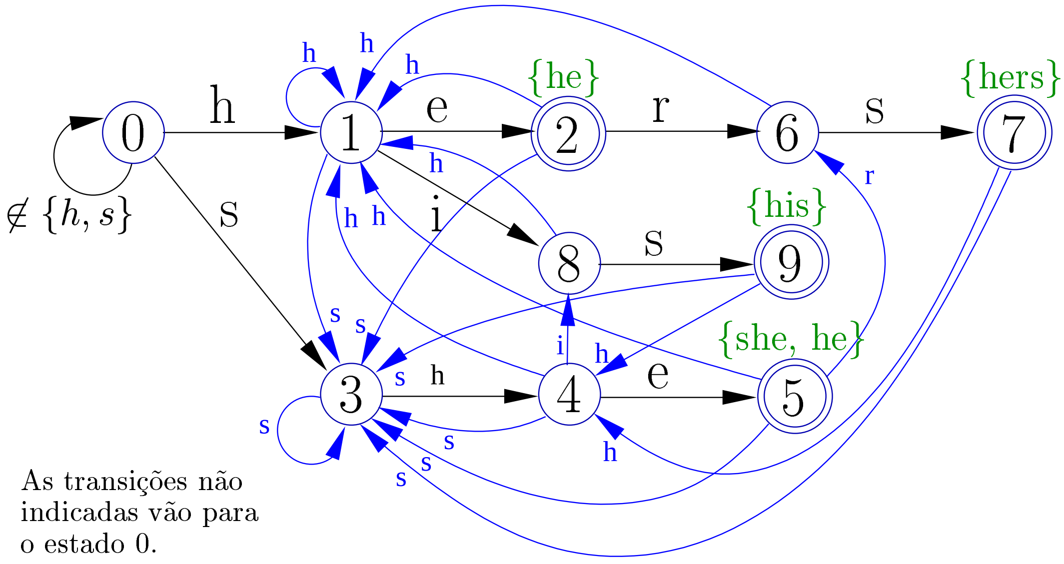


A partir dessa estrutura, o algoritmo constrói uma máquina de estados semelhante a um autômato. A diferença dessa máquina é que ela possui uma segunda função de transição, chamada *falha*. Essa função utiliza as mesmas idéias do algoritmo KMP, de Knuth, Morris e Pratt [8].



Numa busca do *texto* na máquina de estados, a transição de *falha* de um estado é usada quando a transição δ não está definida para o estado. Ela indica o estado para qual devemos voltar e repetir a busca. Por exemplo, numa busca no texto "shine", passaríamos pelos estados nesta ordem: $0, s, 3, h, 4, i, 1, i, 8, n, 0, e, 0$. Note que a máquina já é capaz de reconhecer $\Sigma^* K$.

A última fase é a construção de um *afd* a partir da máquina de estados.



O tempo total consumido por essas três fases pode ser $O(|\Sigma|m)$, dependendo da implementação, onde m é a soma dos comprimentos das palavras em K.

Vejamos novamente a busca das palavras do dicionário no texto "shine": $0, s, 3, h, 4, i, 8, n, 0, e, 0$. Não há ocorrência de nenhuma palavra do dicionário no texto "shine". Podemos notar também que o autômato utiliza um pouco menos transições em comparação com a máquina de estados. Porém, para um *texto* x, tanto o autômato quanto a máquina utilizam $O(|x|)$ transições. Ademais, a quantidade de transições do autômato é bem maior.

4 Implementações

Nós implementamos os três algoritmos na linguagem C, padrão ANSI. Embora essas implementações tivessem como objetivo auxiliar o entendimento das questões de complexidade de tempo e de espaço, elas geraram discussões interessantes sobre outras questões.

Uma delas é utilizar a árvore gerada pelo algoritmo Aho-Corasick para armazenar dicionários. Essa estrutura pode rapidamente responder se uma palavra pertence ao dicionário. Uma possibilidade mais interessante é utilizar um dos algoritmos de minimização para que a representação utilize menos memória. Como essa árvore é um autômato acíclico, foi possível utilizar o algoritmo de Revuz.

Um dicionário com 80000 palavras em português foi dado como entrada para o Aho-Corasick, que devolveu uma árvore com 154452 estados. O autômato minimal obtido possuía 6940 estados. Num teste com um dicionário de 198769 palavras, a árvore correspondente possuía 382803 estados, enquanto que o autômato minimal apresentou 14559 estados.

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, R. Sethi, and J.D. Ullman. *Compilers, Priciples, Techniques and Tools*. Addison Wesley: Reading, MA, 1986.

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

[4] 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.

[5] 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.

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

[7] 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.

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