Minimization of Binary Automata

Tomasz Kowaltowski
Claudio L. Lucchesi
Jorge Stolji

Relatório Técnico DCC-22/93

Setembro de 1993
Minimization of Binary Automata

Tomasz Kowaltowski
Cláudio L. Lucchesi
Jorge Stolfi

Abstract

Finite automata used to represent large vocabularies of natural languages are quite sparse in the following sense: for the vast majority of states, almost all transitions lead to the rejecting state. This suggests a representation of the automaton in which each state is a small list of transitions entering non-rejecting states. It is then possible to factor parts of those lists, thereby saving further space. Depending on the order in which the transitions appear on the lists, the degree of saving varies. This leads to the problem of minimizing such representations. This problem is interesting from a theoretical point of view and quite useful from a practical point of view, as the experimental results presented herein indicate.

1 Applications Involving Finite Automata

Many applications use finite automata to represent sets of words. One of the first uses was perhaps in compiler construction, where such automata were used to model and implement efficient lexical analysers [1].

*Departamento de Ciência da Computação, Universidade Estadual de Campinas, 13081-970 Campinas, SP. E-mail: {tomasz, lucchesi, stolfi}@dcc.unicamp.br. (Authors are listed in alphabetical order.) This research was partially supported by grants from the Brazilian National Council for Scientific and Technological Development (CNPq).

More recently, Lucchesi and Kowaltowski [8] used finite automata to represent large natural language vocabularies used in applications such as spelling checkers and advisers, multilanguage dictionaries, thesauri and minimal perfect hashing. In particular, an efficient spelling checker and adviser for the Portuguese language was implemented, in which the vocabulary consisted of approximately 200,000 words; the finite automaton used 124Kbytes of memory; the spelling checker can process about 30,000 words per minute on a standard IBM-compatible personal computer. Unlike the technique used by the Unix program spell [3, 9], finite automata can be used to implement spelling checkers for most languages. In a companion paper we show an application for debugging natural language vocabularies [6].

In this paper we deal with the question of minimizing a certain type of automata, called binary automata, which we use to represent finite automata. It must be said that we do not know any algorithm for solving the problem other than by using brute force. In Section 2 we formalize the notion of binary automaton. In Section 3 we present the algorithm for binary automaton reduction that was most successful in our experiments. Those experiments involved 13 different vocabularies of 11 languages. In Section 4 we give a brief description of our programming environment and tools and present the experimental results.

2 Binary Automata

A vocabulary is any nonnull finite set of words over some finite alphabet \( \Sigma \). Let \( A := (\Sigma, Q, F, q_0, \delta) \) be the minimum deterministic finite automaton that accepts a vocabulary \( L \). As usual, \( Q \) denotes the set of states of \( A \), \( F \subseteq Q \) is its set of final states, \( q_0 \in Q \) is its initial state and \( \delta: Q \times \Sigma \rightarrow Q \) is its transition function.

In order to avoid some technicalities we assume without loss of generality that

\[
L \subseteq (\Sigma - \#)^*\#
\]
where $\# \in \Sigma$. Under this assumption, $F$ has just one state, the accepting state of $A$, denoted $q_a$.

Observe that the finiteness of $L$ implies that $A$ has just one rejecting state, denoted $q_r$. That is, for every state $q$ in $Q \setminus q_r$ there exists a word $w$ in $\Sigma^*$ such that $\delta(q, w) = q_a$; moreover, $q_r \neq q_a$ and $\delta(q, a) = q_r$ for every letter $a$ in $\Sigma$ and every $q$ in $\{q_a, q_r\}$.

Figure 1 depicts the minimum finite automaton that accepts vocabulary

$$\{a, ai, a\alpha, as, ei, em, a, oi, os\}\#.$$

![Figure 1: A minimum finite automaton (rejecting state omitted) that accepts the language \{a, ai, a\alpha, as, ei, em, a, oi, os\}\#.

A naïve representation of $\delta$ could be a matrix $D$ having $|Q|$ lines and $|\Sigma|$ columns, such that

$$D_{q,a} := \delta(q, a), \forall q \in Q, \forall a \in \Sigma.$$
states and uses an alphabet of roughly 30 letters. Experience shows that the automaton for a natural language vocabulary is very sparse, in the following sense: for the vast majority of states in $Q$ and for almost all letters $a$ in $\Sigma$, $\delta(q,a) = q_r$. In other words, the vast majority of the entries in matrix $D$ are equal to $q_r$. Entries not equal to $q_r$ are called *useful*. In the case of the Portuguese language, for example, almost half of the lines of $D$ have just one useful entry and roughly 70% of the lines have at most two such entries [8].

The sparseness of $D$ leads naturally to another representation of $A$, in which each state $q$, whose corresponding line in $D$ has, say, $r$ useful transitions, is represented by a list of $r$ pairs, each of which describes a useful entry of line $q$ in $D$. Figure 2 shows such a representation for the automaton of Figure 1.

This way of reasoning leads naturally to considering each element of a list a “state” of an automaton-like device, in which the pointer to the next element of the list is a “failure transition”. We call these device *binary automata*. At this point it is important to recall that our finite automata have just one accepting state, $q_a$, from which all transitions enter the rejecting state $q_r$. Thus no list is needed for $q_a$. A binary automaton may then be represented by the data structure given in Figure 3, where function $\delta$ evaluates $\delta : Q \times \Sigma \rightarrow \Sigma$.

One way, but not the only one, to ensure that $\delta$ is well-defined is to require that there exists a partial order $\preceq$ on $Q$ satisfying the following property:

$$\forall q \in Q \setminus \{q_a, q_r\}, q \not\preceq q.$$  \hfill (1)

The vocabulary accepted by $B$ is then defined as usual:

$$\{w \in \Sigma^* : \delta(q_0, w) = q_a\},$$

where $q_0$ denotes the initial state of $B$.

Binary automata appear quite often in the literature. Perhaps its most famous occurrence is in the well known pattern matching algorithm of Knuth, Morris and Pratt [5].

At this point it is important to observe that the finiteness of $L$ and the optimality of $A$ together imply that $A$ is *acyclic* in the following
Figure 2: A representation using lists, of the automaton shown in Figure 1.
type
State = | StateRecord;
StateRecord = record
  case final: boolean of
  true   : ();
  false  :
    (    
      l : Letter;
      s : State;  {success transition}
      f : State;  {failure transition}
    )
end;
BinaryAutomaton = record
  InitialState : State;
end;

function delta(q: State; a: Letter): State;
begin
  if q = nil then delta := nil {rejecting state}
  else if q|final then delta := nil
  else if q|.l = a then delta := q|.s
  else delta := delta(q|.f,a)
end;

Figure 3: A data structure to represent binary automata and the corresponding implementation of transition function $\delta$. 
Minimization of Binary Automata

sense:

\[ \forall q \in Q, \forall w \in \Sigma^+, \left[ \delta(q, w) = q \Leftrightarrow q = q_r \right]. \]

That is, there is no directed circuit in the underlying directed graph of \( A \), except for the loops entering its rejecting state \( q_r \). Or, equivalently, using the concept of topological sorting, there is a partial order \( \preceq \) on the state set \( Q \) of \( A \) such that

\[ \forall q \in Q, \forall a \in \Sigma, \delta(q, a) \preceq q, \]

with equality if, and only if, \( q = q_r \).

It then follows that the corresponding binary automaton not only satisfies (1), but its set of states \( Q \) also has a partial order \( \preceq \) such that

\[ \forall q \in Q \setminus \{ q_0, q_r \}, q \prec q \text{ and } q \prec f \prec q. \]

This property is quite convenient, since we may extend \( \preceq \) to a total order and represent binary automata by vectors, in which \( Q = \{ 0, 1, \ldots, q_0 \} \), 0 is the rejecting state, 1 is the accepting state, and each state \( q \geq 2 \) is represented by an entry in the vector; that entry is a triple \((a_q, s_q, f_q)\) in \( \Sigma \times Q \times Q \) and such that both \( s_q \) and \( f_q \) are strictly smaller than \( q \). Figure 4 shows the vector representation of the binary automaton of Figure 2.

Binary automata may yield memory savings greater than one expects at first. In Figure 2, states 3 and 9 are identical. This fact leads to a smaller equivalent binary automaton, shown in Figure 5.

Observe also in Figure 5 that we may permute states 4 and 5 in the list of state 6, since there is just one pointer entering state 4. Such interchange then yields an even smaller automaton, depicted in Figure 6.

Finally, Figure 7 illustrates another binary automaton equivalent to that of Figure 2 having the minimum number of states. Figure 8 shows yet another optimum binary automaton for the same language.

The problem of minimizing a binary automaton is of both theoretical and practical interest. We present in Section 4 some practical results that indicate significant savings in the size of binary automata representing several natural languages. These results were obtained using an algorithm called Fold, described in Section 3.
Figure 4: A vectorial representation of the binary automaton shown in Figure 2.
Figure 5: A smaller binary automaton equivalent to that of Figure 2.
Figure 6: Yet a smaller binary automaton equivalent to that of Figure 2.
Figure 7: A minimum binary automaton equivalent to that of Figure 2.
Figure 8: Yet another optimum binary automaton equivalent to that of Figure 2.
We suspect that the problem of deciding whether or not a given binary automaton is minimum is NP-complete. However, we do not have a proof of its NP-completeness.

3 Algorithm Fold

We tried several algorithms to decrease the size of binary automata. We present here Fold, the most successful one. In its implementation, Fold represents binary automata by vectors, as explained in Section 2. However it is more convenient to describe it using the data structure presented in Figure 3.

Fold works iteratively. Each iteration, given a binary automaton \( B \), produces a new binary automaton \( B' \) equivalent to \( B \). In the first iteration, \( B \) is the automaton which we would like to optimize. If the automaton \( B' \) obtained in an iteration has fewer states than \( B \), the algorithm enters a new iteration, with \( B' \) playing the role of \( B \). If \( B' \) has as many states as \( B \), the algorithm stops.

To describe an iteration, we need to introduce some concepts and notation. A state \( q \in Q \setminus \{q_a, q_r\} \) of \( B \) is proper if it satisfies at least one of the following conditions:

1. \( q \) is the initial state of \( B \),
2. for some state \( r \not\in \{q_a, q_r\}, q = r \downarrow a \downarrow s \),
3. \( \exists q', q'' \in Q \setminus \{q_a, q_r\}, q' \neq q'' \) and \( q' \downarrow f = q'' \downarrow f \).

For example, in Figure 5, the proper states are states 2, 3, 6, 10 and 13.

For any state \( q \) of \( B \), the pair set of \( q \), denoted \( P(q) \), is defined as follows:

\[
P(q) := \begin{cases} \emptyset & \text{if } q \in \{q_a, q_r\} \\ \{(q \downarrow a, q \downarrow s)\} \cup P(q \downarrow f) & \text{if } q \not\in \{q_a, q_r\} \end{cases}
\]

In each iteration, Fold visits every proper state and tries to permute the elements of its list of transitions in order to decrease the number of states. Proper states are visited in increasing order, according to partial
order \leq. For example, in Figure 6, Fold will visit proper states 2, 3, 6, 10 and 13 in this order.

Let us now describe what constitutes a visit to a proper state \( q \) (see Figure 9). The first step is to determine set

\[
R := \{ r \in Q : r \text{ visited, } r < q, P(r) \subseteq P(q) \} \cup \{q_r\}.
\]

One element \( r \in R \) is then chosen, such that \( P(r) \) is maximum. The list of transitions emerging from state \( q \) is then ordered in such a way as to leave \( P(r) \) at the end, in the same order that the transitions of state \( r \) are listed, ensuring that the list of \( r \) will be shared by the list of \( q \).

It now remains to order the remaining transitions, namely \( P(q) \setminus P(r) \). Those transitions will be selected one at a time. The selection criterium is the frequency of transitions of \( P(q) \setminus P(r) \) in the pair sets of those states in

\[
T := \{ t \in Q : t \text{ is proper, } t \text{ not visited, } P(r) \subseteq P(t) \}.
\]

One of the pairs \((a, e)\) having the highest frequency is then selected. A new state is then created, in which \( a \) is the letter, \( e \) the success transition and \( r \) the failure transition. Fold then updates \( r \), assigning to it the new state, just created. The selection process continues, until \( P(r) = P(q) \).

After that, \( r \) is the "new version" of \( q \).

Intuitively, the reason for initially choosing \( r \) to maximize \( P(r) \) is to find the largest "factor" of \( P(q) \) already in the "new" automaton. The reason for selecting pair \((a, e)\) in \( P(q) \setminus P(r) \) having the highest frequency in the pair sets of states in \( T \) is to increase the probability of finding new "factors" at later iterations.

### 3.1 Automaton Compression

As mentioned earlier, the automaton is represented by a vector that associates with each state \( q \geq 2 \) a triple \((a_q, s_q, f_q)\) \( \in \Sigma \times Q \times Q \). In order to save disk space, the automaton is stored in a compressed format, by coding each triple with a variable number of bits, as described below.
function Visit(q: State): State;
begin
  R := \{ r \in Q : r \text{ is proper, visited, } P(r) \subseteq P(q) \} \cup \{ q_r \};
  mark q visited;
  choose r \in R \text{ such that } P(r) \text{ is maximum};
  while P(r) \neq P(q) do
    begin
      T := \{ t \in Q : t \text{ is proper, not visited, } P(r) \subseteq P(t) \};
      for each \((a, e)\) \in P(q) \setminus P(r) do
        c(a, e) := |\{ t \in T : (a, e) \in P(t) \}|;
      choose \((a, e)\) such that \(c(a, e)\) is maximum
      t := New(State); t \| l, t \| s, t \| f := a, e, r;
      r := t
    end;
  Visit := r
end;

Figure 9: A visit to proper state q.
Triple \((a_2, s_2, f_2)\) is always \((\#1, 0)\), thus it is not coded. Each triple is coded sequentially, starting with the triple for state 3 and proceeding towards the maximum state.

The coding of \(a_q\) is done using \(\lceil \log_2(n + 1) \rceil\) bits, where \(n\) is the number of characters actually occurring as \(a_q\) in the dictionary. The special symbol \(\#\) is always coded as zero.

For any state \(q \geq 2, 1 \leq s_q \leq q - 1\); moreover, \(s_q = 1\) if, and only if, \(a_q = \#\). In a large percentage of the states (roughly half of them), \(s_q \in \{1, q - 1\}\). When \(s_q = 1\) then \(a_q = \#\) and there is no need to code \(s_q\). For the remaining cases, we distinguish between \(s_q = q - 1\) and \(s_q \neq q - 1\) by means of an extra bit. In the latter case, \(s_q - 1\) is coded, where \(0 < s_q - 1 \leq q - 2\).

Likewise, for any state \(q \geq 2, 0 \leq f_q \leq q - 1\). In a large percentage of the states (roughly 75%), \(f_q \in \{0, q - 1\}\). We use one or two extra bits to distinguish between those two possibilities and the case in which \(f_q \not\in \{0, q - 1\}\). In the latter case, \(f_q - 1\) is coded, with \(0 \leq f_q - 1 \leq q - 2\).

Whenever it is necessary to code either or both of \(s_q\) and \(f_q\), this is done using precisely \(k\) bits, where \(1 + 2^{k-1} < q \leq 1 + 2^k, q \geq 3\).

4 Experimental Results

4.1 Environment Programming and Tools

We chose the language Modula-3 [10] to implement our environment. The main reasons are the simplicity of the language, its object-oriented programming paradigm and its availability on most Unix workstations. The environment consists of a set of modules which provide the basic abstractions. Some of the more important ones are described in a companion paper [6].

Program Fold contains about 1000 lines and is rather slow when the automaton is large, although it has complexity quadratic on the size of the automaton: for example, it ran for more than a week on a SPARC station 2 on an automaton representing a huge English vocabulary of about 300,000 words.
4.2 Experimental Results

We tried Fold on several automata, each of which representing a large vocabulary (Figure 10).

The resulting savings varied from vocabulary to vocabulary, from a minimum of 6% in the case of the Swedish vocabulary to a maximum of 24% in the case of the Portuguese vocabulary. We also represented the automata in compressed form. In this case the savings obtained with folding varied from 1% in the case of the German vocabulary to 18% in the case of the Portuguese vocabulary (Figure 11).

It is interesting to observe that there is some relation between the compression achieved by the folding process and the ratio of letters per transition. For higher ratios, as in the Portuguese, French and Italian...
## Vocabulary size (%)

<table>
<thead>
<tr>
<th>Language</th>
<th>Unfolded</th>
<th>Folded</th>
<th>Compressed Unfolded</th>
<th>Compressed Folded</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Portuguese</td>
<td>630,667</td>
<td>478,421</td>
<td>88,310</td>
<td>72,442</td>
<td>82</td>
</tr>
<tr>
<td>French</td>
<td>771,162</td>
<td>607,444</td>
<td>105,298</td>
<td>91,400</td>
<td>87</td>
</tr>
<tr>
<td>Italian</td>
<td>392,144</td>
<td>312,902</td>
<td>52,398</td>
<td>43,882</td>
<td>84</td>
</tr>
<tr>
<td>English3</td>
<td>1,038,056</td>
<td>895,163</td>
<td>138,250</td>
<td>127,750</td>
<td>92</td>
</tr>
<tr>
<td>Hebrew</td>
<td>512,016</td>
<td>443,890</td>
<td>76,458</td>
<td>67,462</td>
<td>88</td>
</tr>
<tr>
<td>English2</td>
<td>1,649,166</td>
<td>1,436,506</td>
<td>218,038</td>
<td>201,894</td>
<td>93</td>
</tr>
<tr>
<td>English1</td>
<td>5,619,002</td>
<td>4,914,568</td>
<td>764,802</td>
<td>726,310</td>
<td>95</td>
</tr>
<tr>
<td>Russian</td>
<td>1,040,145</td>
<td>919,580</td>
<td>138,810</td>
<td>131,818</td>
<td>95</td>
</tr>
<tr>
<td>Dutch</td>
<td>2,721,289</td>
<td>2,412,142</td>
<td>353,866</td>
<td>338,162</td>
<td>96</td>
</tr>
<tr>
<td>Esperanto</td>
<td>290,445</td>
<td>258,440</td>
<td>41,510</td>
<td>39,894</td>
<td>96</td>
</tr>
<tr>
<td>Norwegian</td>
<td>1,142,192</td>
<td>1,044,673</td>
<td>145,246</td>
<td>140,150</td>
<td>96</td>
</tr>
<tr>
<td>German</td>
<td>3,580,355</td>
<td>3,342,096</td>
<td>428,746</td>
<td>425,778</td>
<td>99</td>
</tr>
<tr>
<td>Swedish</td>
<td>385,514</td>
<td>361,019</td>
<td>49,182</td>
<td>46,930</td>
<td>95</td>
</tr>
</tbody>
</table>

**Note:** The last column is the ratio between the sizes of the folded compressed automaton over the sizes of the unfolded uncompressed automata.

Figure 11: Experimental results with Fold on automata (sizes are in bytes).
vocabularies, one finds compression rates in the range of 24 to 20%, whereas in vocabularies with lower ratios, like the Russian, Norwegian and Swedish vocabularies, one finds compression rates in the range of 12 to 6%.

We do not know how to explain why the Dutch and German vocabularies do not follow this rule, since both have medium range ratios but low range compression rates.

It is also worth mentioning the relatively constant compression rate which is obtained by compounding Fold and the compression technique described in Section 3.

Finally, we would like to emphasize that the vocabularies differ significantly in completeness. For some languages no inflected forms were included, whereas for others the vocabulary is quite complete. Thus, differences between the numbers above could be justified by the quality of the vocabularies, not by intrinsic differences in the languages themselves.

References


Relatórios Técnicos – 1992

01/92 Applications of Finite Automata Representing Large Vocabularies, C. L. Lucchesi, T. Kowaltowski

02/92 Point Set Pattern Matching in $d$-Dimensions, P. J. de Rezende, D. T. Lee

03/92 On the Irrelevance of Edge Orientations on the Acyclic Directed Two Disjoint Paths Problem, C. L. Lucchesi, M. C. M. T. Giglio

04/92 A Note on Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi Diagrams, W. Jacometti

05/92 An $(l, u)$-Transversal Theorem for Bipartite Graphs, C. L. Lucchesi, D. H. Younger

06/92 Implementing Integrity Control in Active Databases, C. B. Medeiros, M. J. Andrade

07/92 New Experimental Results For Bipartite Matching, J. C. Setubal

08/92 Maintaining Integrity Constraints across Versions in a Database, C. B. Medeiros, G. Jomier, W. Cellary

09/92 On Clique-Complete Graphs, C. L. Lucchesi, C. P. Mello, J. L. Szwarejter

10/92 Examples of Informal but Rigorous Correctness Proofs for Tree Traversing Algorithms, T. Kowaltowski

11/92 Debugging Aids for Statechart-Based Systems, V. G. S. Elias, H. Liesenb erg

12/92 Browsing and Querying in Object-Oriented Databases, J. L. de Oliveira, R. de O. Anido
Relatórios Técnicos – 1993

01/93 Transforming Statecharts into Reactive Systems, Antonio G. Figueiredo Filho, Hans K. E. Liesenberg

02/93 The Hierarchical Ring Protocol: An Efficient Scheme for Reading Replicated Data, Nabor das C. Mendonça, Ricardo de O. Anido

03/93 Matching Algorithms for Bipartite Graphs, Herbert A. Baier Saip, Cláudio L. Lucchesi

04/93 A lexBFS Algorithm for Proper Interval Graph Recognition, Celina M. H. de Figueiredo, João Meidanis, Célia P. de Mello

05/93 Sistema Gerenciador de Processamento Cooperativo, Ivonne. M. Carrazana, Nelson. C. Machado, Célio. C. Guimarães

06/93 Implementação de um Banco de Dados Relacional Dotado de uma Interface Cooperativa, Naseif A. Abousalh Neto, Ariadne M. B. R. Carvalho

07/93 Estadogramas no Desenvolvimento de Interfaces, Fábio N. de Lucena, Hans K. E. Liesenberg

08/93 Introspection and Projection in Reasoning about Other Agents, Jacques Wainer

09/93 Codificação de Seqüências de Imagens com Quantização Vetorial, Carlos Antonio Reinaldo Costa, Paulo Lício de Geus

10/93 Minimização do Consumo de Energia em um Sistema para Aquisição de Dados Controlado por Microcomputador, Paulo César Centoducatte, Nelson Castro Machado
11/93 An Implementation Structure for RM-OSI/ISO Transaction Processing Application Contexts, Flávio Morais de Assis Silva, Edmundo Roberto Mauro Madeira

12/93 Boole’s conditions of possible experience and reasoning under uncertainty, Pierre Hansen, Brigitte Jaumard, Marcus Poggi de Araújo

13/93 Modelling Geographic Information Systems using an Object Oriented Framework, Fatima Pires, Claudia Bauzer Medeiros, Ardemiris Barros Silva

14/93 Managing Time in Object-Oriented Databases, Lincoln M. Oliveira, Claudia Bauzer Medeiros

15/93 Using Extended Hierarchical Quorum Consensus to Control Replicated Data: from Traditional Voting to Logical Structures, Nabor das Chagas Mendonça, Ricardo de Oliveira Anido


17/93 Metodologias para Conversão de Esquemas em Sistemas de Bancos de Dados Heterogêneos, Ronaldo Lopes de Oliveira, Geovane Cayres Magalhães

18/93 Rule Application in GIS – a Case Study, Claudia Bauzer Medeiros, Geovane Cayres Magalhães

19/93 Modelamento, Simulação e Síntese com VHDL, Carlos Geraldo Krüger e Mário Lúcio Côrtes

20/93 Reflections on Using Statecharts to Capture Human-Computer Interface Behaviour, Fábio Nogueira de Lucena e Hans Liesenbørg
Applications of Finite Automata in Debugging Natural Language Vocabularies, Tomasz Kowaltowski, Cláudio Leonardo Lucchesi e Jorge Stolfi

Minimization of Binary Automata, Tomasz Kowaltowski, Cláudio Leonardo Lucchesi e Jorge Stolfi