Minimization of Binary Automata

Tomasz Kowaltowski, Cláudio L. Lucchesi and Jorge Stolfi*
Department of Computer Science
University of Campinas
13081-970 Campinas, SP, Brazil
{tomasz,lucchesi,stolfi}@dcc.unicamp.br.

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. By reordering those lists and merging common sublists, it is possible to reduce the number of transitions in the automaton.

The problem of reordering the lists so as to minimize the size of the automaton is quite interesting and practically relevant. Unfortunately, it also appears to be quite difficult. We describe here a heuristic algorithm that can be be applied to large binary automata (with hundreds of thousands of transitions) and achieves practically significant reductions.

A simple encoding for binary automata is also described; this special compressor outperforms the best general-purpose file compression programs. Together with the minimization heuristic above, this encoding makes it possible to store a 2.1 megabyte Portuguese wordlist in slightly less than 68 kilobytes, with no loss of information.

keywords: finite automata, natural language vocabularies, spelling checking, data compression

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].

Gross and Perrin [4], Liang [7], Appel and Jacobson[2] are examples of the use of finite automata in Computational Linguistics.

*(Authors are listed in alphabetical order.)

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 the size of a popular representation of finite automata, which we call *binary automaton*.

This problem is quite interesting from a theoretical point of view, and has obvious practical relevance. Unfortunately, it also appears to be quite difficult. We do not know how to find the optimal solution, other than by using a brute-force search whose cost is exponential on the number of states.

As a partial solution to this problem, we present here a heuristic minimization algorithm, Fold, that can handle binary automata with hundreds of thousands of transitions. We have tested the Fold heuristic on about a dozen such automata, derived from large word lists in several languages, and achieved reductions ranging from 6% up to 24%.

We also present here a simple but very compact encoding scheme for binary automata, that is about twice as effective as the best general-purpose file compression tools. Combined with the Fold heuristic, it allows us to store a 2.1 megabyte Portuguese wordlist in slighlty less than 68 kilobytes, with no loss of information.

In Section 2 we formalize the notion of binary automaton and describe the basic data structures and algorithms. In Section 3 we define the minimization problem for binary automata. Sections 4 and 5 describe the Fold heuristic and the compact encoding of binary automata. Section 6 summarizes our experiments.

2 Binary Automata

2.1 Finite Automata

A *vocabulary* is any nonnull finite set of words over some finite alphabet Σ. Let $\mathcal{A} := (\Sigma, Q, F, q_0, \delta)$ be the minimum deterministic finite automaton that accepts a vocabulary L. As usual, Q denotes the set of *states* of $\mathcal{A}$, $F \subseteq Q$ is its set of *final states*, $q_0 \in Q$ is its *initial state* and $\delta : Q \times \Sigma \to Q$ 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 $\mathcal{A}$, denoted q_a.

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

Figure 1 depicts the minimum finite automaton that accepts vocabulary

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

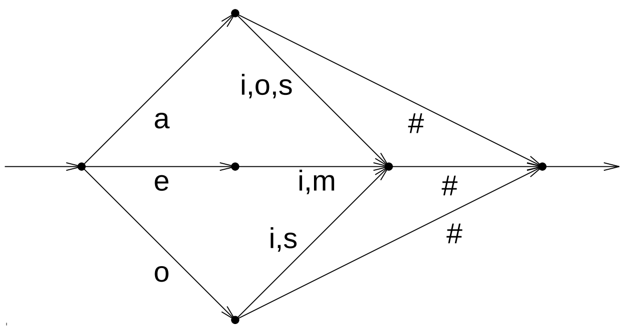

Figure 1: The minimum finite automaton that accepts the language $\{a, ai, ao, as, ei, em, o, oi, os\}\#$. The rejecting state q_r and the transitions leading to it have been omitted for clarity.

2.2 Exploiting Sparsity

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

$$D_{q,a} := \delta(q, a), \forall q \in Q, \forall a \in \Sigma.$$

Since we are considering L to be a rather large vocabulary of a natural language, this approach is not practical. For example, a typical automaton for a Portuguese language vocabulary contains more than 14,000 states and uses an alphabet of roughly 30 letters—implying about 420,000 transitions. 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 Σ, $\delta(q, a) = q_r$. Consequently, 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 $\mathcal{A}$, 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.

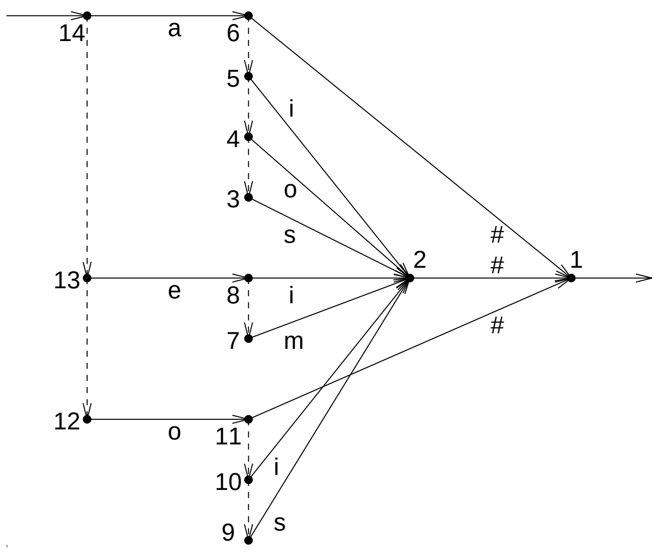

Figure 2: A list-based representation of the automaton shown in Figure 1.

2.3 Binary Automata

With the representation indicated in the previous subsection, it is natural to view the individual list nodes as "states" of an automaton-like device, where each state has just one "success" transition (labeled with a letter) and one "failure" transition (represented by the next-node pointer). We call this device a *binary automaton*.

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. The same conclusion holds for rejecting state q_r. A binary automaton may then be represented by the following data structure:

```
type
   State = ↑StateRecord;
   StateRecord = record
      case special: boolean of
         true  : { accepting or rejecting state }
            (
              accepting: boolean;
            );
         false :
            (
            l : Letter;
            s : State;      { success transition }
            f : State;      { failure transition }
            )
      end;
   BinaryAutomaton = record
      InitialState : State;
      end;
```

Given this data structure, the finite automaton's transition function $\delta : Q \times \Sigma \to \Sigma$ can be computed as follows:

```
function delta(q: State; a: Letter): State;
begin
   if q↑.special then delta := q_r
   else if q↑.l = a then delta := q↑.s
   else delta := delta(q↑.f,a)
end;
```

We assume that the lists defined by the f links are not circular, so that the function *delta* always terminates after a finite number of steps, whatever the input letter.

The vocabulary accepted by $\mathcal{B}$ is then defined as usual:

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

where q_0 denotes the initial state of $\mathcal{B}$.

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].

2.4 Vector representation

In some situations, it may be desirable to replace the linked data structure by a vector-based representation. It suffices to number the states of the binary automaton with consecutive integers, and store all *StateRecords* in a vector indexed by state number.

In this scheme, it is convenient to assign index 0 to the rejecting state q_r and index 1 to the accepting state q_a. The corresponding entries of the vector can be omitted. Furthermore, all other entries have the *special* bit equal to **false**; so that bit can be omitted, too. Thus, if the ith state is q, the ith vector element contains the triple (l, s, f); where the fields s and f are now state numbers, and not pointers.

Figure 3 shows a vector representation of the binary automaton of Figure 2, using the same state numbering. The inital state is 14.

q	a_q	s_q	f_q
2	#	1	0
3	s	2	0
4	o	2	3
5	i	2	4
6	#	1	5
7	m	2	0
8	i	2	7
9	s	2	0
10	i	2	9
11	#	1	10
12	o	11	0
13	e	8	12
14	a	6	13

Figure 3: A vectorial representation of the binary automaton shown in Figure 2.

At this point it is important to observe that the finiteness of L and the optimality of $\mathcal{A}$ together imply that $\mathcal{A}$ is *acyclic* in the following sense:

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

That is, there is no directed circuit in the underlying directed graph of $\mathcal{A}$, except for the loops entering its rejecting state q_r. It follows trivially that any binary automaton $\mathcal{B}$ equivalent to $\mathcal{A}$ is also acyclic. Therefore, there is a partial order $\preceq$ on the state set Q of $\mathcal{A}$ such that

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

with equality if, and only if, $q = q_r$.

Furthermore, using the concept of topological sorting, we can always extend the partial order $\preceq$ to a total order with this same property. In other words, when building a vector representation for $\mathcal{B}$, we can always number the states as $Q = \{0, 1, \cdots, n\}$ in such a way that 0 is the rejecting state, 1 is the accepting state, n is the initial state, and the s and f fields of entry number i are both strictly smaller than i.

3 Minimization of Binary Automata

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 4.

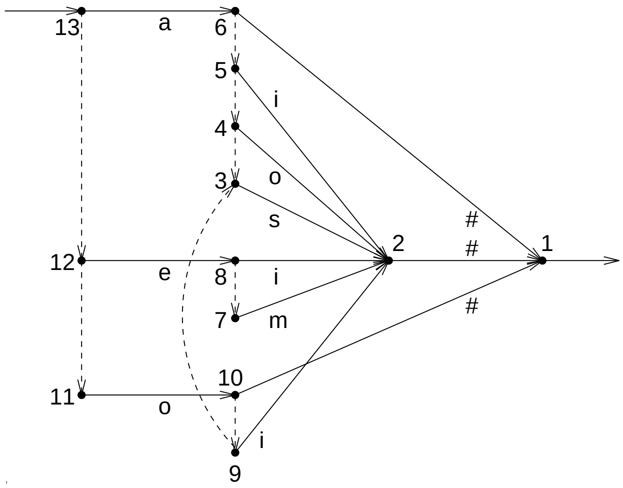

Figure 4: A smaller binary automaton equivalent to that of Figure 2.

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

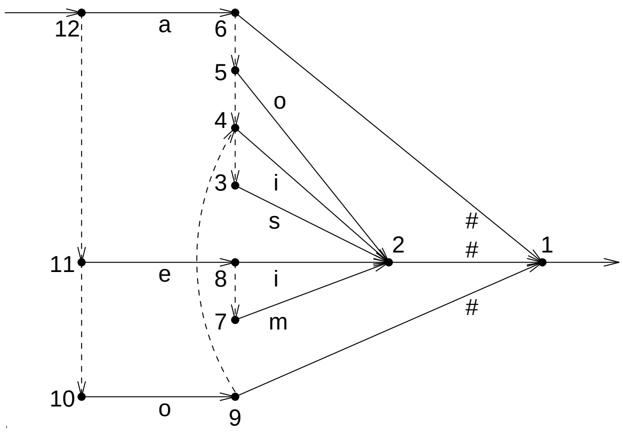

Figure 5: Yet a smaller binary automaton equivalent to that of Figure 2.

Finally, Figure 6 illustrates another binary automaton equivalent to that of Figure 2 having the minimum number of states. Figure 7 shows yet another optimum binary automaton for the same language.

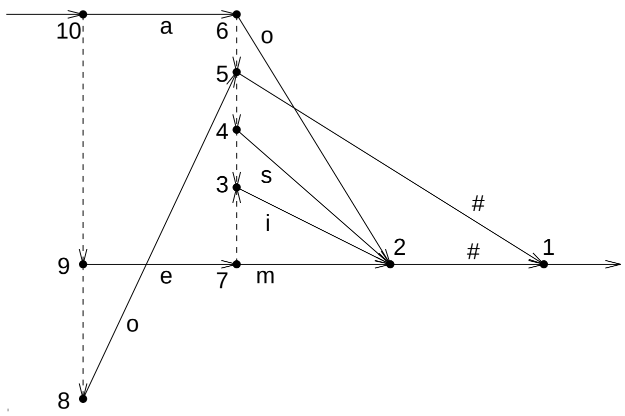

Figure 6: A minimum binary automaton equivalent to that of Figure 2.

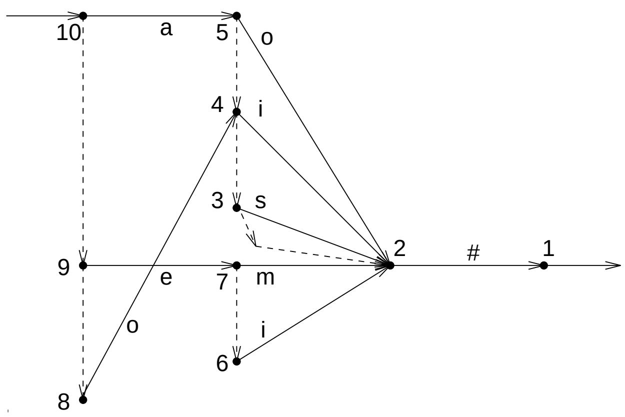

Figure 7: Yet another optimum binary automaton equivalent to that of Figure 2.

The point of these examples is that the extent of sharing between the state lists depends strongly on the order in which transitions appear in each list. An interesting question is thus to find the permutations of the lists that minimize the number of distinct states of the binary automaton.

Apart from its practical importance, this problem is very interesting from the theoretical point of view. It seems to be a very hard problem. We don't know how to find the true optimum, except by a brute-force search that grows exponentially in the number of states. We don't even know how to recognize the minimum binary automaton (or rather *a* minimum, since there may be more than one); or even to compute a useful estimate of its size. In fact, we suspect that finding the true minimum is an NP-complete problem.

4 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.

Fold works iteratively. Each iteration, given a binary automaton $\mathcal{B}$, produces a new binary automaton $\mathcal{B}'$ equivalent to $\mathcal{B}$. In the first iteration, $\mathcal{B}$ is the automaton which we would like to optimize. If the automaton $\mathcal{B}'$ obtained in an iteration has fewer states than $\mathcal{B}$, the algorithm enters a new iteration, with $\mathcal{B}'$ playing the role of $\mathcal{B}$. If $\mathcal{B}'$ has as many states as $\mathcal{B}$, 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 $\mathcal{B}$ is *proper* if it satisfies at least one of the following conditions

either	(i)	q is the initial state of $\mathcal{B}$,
or	(ii)	for some state $r \notin \{q_a, q_r\}$, $q = r{\uparrow}.s$,
or	(iii)	$\exists q', q'' \in Q \setminus \{q_a, q_r\}, q' \neq q''$ and $q'{\uparrow}.f = q = q''{\uparrow}.f$.

For example, in Figure 4, the proper states are states 2, 3, 6, 8, 10 and 13.

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

$$P(q) := \begin{cases} \emptyset & \text{if } q \in \{q_a, q_r\} \\ \{(q{\uparrow}.a,\ q{\uparrow}.s)\} \cup P(q{\uparrow}.f) & \text{if } q \notin \{q_a, q_r\}. \end{cases}$$

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 $\preceq$. For example, in Figure 4, Fold will visit proper states 2, 3, 6, 8, 10 and 13 in this order.

Let us now describe what constitutes a visit to a proper state q (see Figure 8). The first step is to determine set

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

One element $r \in R(q)$ 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, allowing the list of r to 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

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

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.

```
function Visit(q: State): State;
var e, r, t: State; a: Letter;
      R, T: set of State;
begin
   R :=
{q_r} ∪ {r ∈ Q : r is proper, visited, P(r) ⊆ P(q)};
   mark q visited;
   choose r ∈ R such that P(r) is maximum;
   while P(r) ≠ P(q) do
   begin
      T :=
{t ∈ Q : t is proper, not visited, P(r) ⊆ P(t)};
      choose (a, e) ∈ P(q) \ P(r)
        that maximizes
          |{t ∈ T : (a, e) ∈ P(t)}|;
      t := New(State); t↑ := (l: a, s: e, f: r);
      r := t
   end;
   Visit := r
end;
```

Figure 8: A visit to proper state 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 $T(r)$ is to increase the probability of finding new "factors" at later iterations.

5 Compact Encoding of Binary Automata

If our goal is to minimize the amount of storage used by a binary automaton, then, besides minimizing the number of states, it is important that we also try to code the state information as economically as possible.

A straightforward vector representation, where all *StateRecords* have the same bit layout with fixed-length fields, requires at least

$$\lceil \log_2 m \rceil + 2\lceil \log_2 n \rceil$$

bits of storage per state, where m is the number of letters in Σ, and n is the highest state number. For example, the binary automaton for our Portuguese wordlist (32 letters, 24,810 states after Fold) would require 108,500 bytes of storage in this format.

In our experiments with natural language vocabularies, we found it advantageous to encode each state record with a variable number of bits, as follows:

- The entry for state number 2 is always $(\#, 1, 0)$, and hence is not encoded. The remaining entries, from 3 up, are coded sequentially.

- The letter field l is Huffman-compressed, the code being chosen based on the relative frequency of occurrence of each letter in the whole state vector.

- The pointer fields s and f are encoded independently. Note that $1 \leq s_q \leq q-1$ for any state $q \geq 2$. In fact, in a large percentage of the states (roughly half of them), s_q is either 1 or $q-1$. Since $s_q = 1$ if and only if $l_q = \#$, we do not need to code s_q if $l_q = \#$. Otherwise, we use one bit to distinguish between the cases $s_q = q-1$ and $1 < s_q < s_q - 1$. In the latter case, we encode the value s_q, using exactly $\lceil \log_2 q \rceil$ bits.

  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 is either 0 or $q-1$. We use one or two extra bits to distinguish between those two possibilities and the case in which $0 < f_q < q-1$. In the latter case, f_q is coded, using $\lceil \log_2 q \rceil$ bits.

Consecutive fields are packed together without any "fillers" or padding, ignoring byte boundaries; and the same applies to consecutive entries of the vector.

With this encoding, the binary automaton for Portuguese, after Fold, was reduced to only 67,898 bytes—about 63% of the fixed-length representation above.

6 Experimental Results

We chose the language MODULA-3[10] to implement our environment. Our software represents the binary automata by a vector, as described in Section 2.4. All binary automata produced by our software are minimal in the following sense: no two states have the same triple (l, s, f); or, equivalently, common sublists are merged.

We tried Fold heuristic on several automata, each representing a large word list. A SPARCstation 2 was used to run most of the tests. Minimization of the largest automaton (English1) took more than a week.

Figure 9 shows the results. Figure 9(a) gives the sizes of the vocabularies involved. Figure 9(b) gives the number of states of two equivalent binary automata: the standard automaton and the automaton obtained by applying Fold to it. By *standard* we mean the binary automaton obtained from the minimum automaton by representing the list of transitions of each state in alphabetical order of its letters (common sublists merged).

	(a)		(b)			(c)		
	vocabulary		number of states of binary automaton			size in bytes of compressed binary automaton file		
Vocabulary	words	letters	standard	Fold	%	Fold + fbe + gzip	Fold + Code	%
Dutch	189,249	1,948,308	131,855	117,369	89	522,531	328,386	63
English1	297,011	2,856,356	261,525	227,827	87	1,070,666	706,314	66
English2	104,216	951,565	83,159	71,947	87	316,169	195,418	62
English3	81,142	715,513	53,399	45,786	86	193,655	123,734	64
Esperanto	12,256	86,863	15,567	13,843	89	54,806	37,082	68
French	133,092	1,274,078	39,627	31,226	79	127,421	81,134	64
German	174,573	2,043,212	169,136	157,556	93	696,279	395,046	57
Hebrew	38,231	193,009	26,600	22,412	84	93,093	65,810	71
Italian	61,183	509,388	20,858	16,743	80	67,568	42,162	62
Norwegian	61,839	527,371	58,555	53,314	91	222,152	135,554	61
Portuguese	219,953	2,168,541	32,608	24,810	76	102,658	67,898	66
Russian	50,292	416,988	52,372	45,992	88	197,107	123,774	63
Swedish	14,944	106,690	20,317	18,967	93	75,335	43,298	57

Figure 9: Experimental results. (fbe = fixed-length binary enconding)

The resulting savings varied from vocabulary to vocabulary, from a minimum of 7% in the case of the German and Swedish vocabularies to a maximum of 24% in the case of the Portuguese vocabulary.

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.

It is interesting to observe that the reductions achieved for Romance languages, such as Portuguese, Italian and French, are highest, from 24% to 20%. The reduction obtained for the three English vocabularies are around 13%.

We also experimented with two compressors. In both experiments, we started with the binary automaton obtained by applying Fold to the standard binary automaton. In one experiment, we applied the well-known gzip, with the –best option, that gives the highest compression, to a straightforward fixed-length binary encoding of the automaton; in the other experiment we applied Code, described in Section 5. Figure 9(c) gives the sizes in bytes of the files obtained. The resulting savings vary from 29% to 43%.

It is important to emphasize that Fold and Code have rather distinct objectives: Fold decreases the number of states of the automaton, thereby saving RAM space; Code compresses the file representing the automaton, thereby saving disk space. It could be argued that Code is enough, there is no need for Fold. We would like to recall that direct access to a given state of the binary automaton is essential for the performance of almost all applications: this feature is lost when the file is compressed.

Finally, it would be nice to have some measure of the effectiveness of Fold, or of any other heuristics. So far we could not find any reasonable such measure.

Acknowledgements The work done by the three authors was supported in part by grants from CNPq, from Brazil.

We would also like to thank the referees for very useful criticisms and suggestions. In particular, one of the referees suggested the use of Huffman coding for compressing the letters of each state. That suggestion improved the compression rate significantly.

References

[1] Aho, A. V., Sethi, R. and Ullman, J. D. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1985.

[2] Appel, A. W. and Jacobson, G. J. The world's fastest Scrabble program. *Communications of the ACM*, 31:572–578, 5, 585, 1988.

[3] Bentley, J. A spelling checker. *Communications of the ACM*, 28:456–462, 5, 1985.

[4] Gross, M. and Perrin, D., editors. *Electronic Dictionaries and Automata in Computational Linguistics*, volume 377 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1989.

[5] Knuth, D. E., Morris, J. H. and Pratt, V. R. Fast pattern matching in strings. *SIAM Journal on Computing*, 6:323–50, 1977.

[6] T. Kowaltowski, C. L. Lucchesi, and J. Stolfi. Application of finite automata in debugging natural language vocabularies. This issue.

[7] Liang, F. M. *Word Hy-phen-a-tion by Com-pu-ter.* PhD thesis, Stanford University, Stanford, CA, 1983.

[8] Lucchesi, C. L. and Kowaltowski, T. Applications of finite automata representing large vocabularies. *Software – Practice & Experience*, 23:15–30, 1993.

[9] McIlroy, M. D. Development of a spelling list. *IEEE Trans. Comm.*, 30:91–99, 1, 1982.

[10] Nelson, G., editor. *System Programming with Modula-3*. Prentice Hall, 1991.