O conteúdo do presente relatório é de única responsabilidade do(s) autore(s).
(The contents of this report are the sole responsibility of the author(s).)

Applications of Finite Automata Representing Large Vocabularies

Cláudio L. Lucchesi
and
Tomasz Kowaltowski

Relatório Técnico DCC–01/92

Junho de 1992

Applications of Finite Automata Representing Large Vocabularies

Cláudio L. Lucchesi
and
Tomasz Kowaltowski

Abstract

The construction of minimal acyclic deterministic partial finite automata to represent large natural language vocabularies is described. Applications of such automata include: spelling checkers and advisers, multilanguage dictionaries, thesauri, minimal perfect hashing and text compression.

Part of this research was supported by a grant awarded by the Brazilian National Council for Scientific and Technological Development (CNPq) to the second author.

Authors' Address: Cláudio L. Lucchesi and Tomasz Kowaltowski, Department of Computer Science, University of Campinas, Caixa Postal 6065, 13081 Campinas, SP, Brazil.

E-mail: `lucchesi@dcc.unicamp.br` and `tomasz@dcc.unicamp.br`.

1 Introduction

The use of finite automata (see for instance [5]) to represent sets of words is a well established technique. Perhaps the most traditional application is found in compiler construction where such automata can be used to model and implement efficient lexical analyzers (see [1]). Applications of finite automata to solve some specific problems in natural language processing are exemplified by the works described in [4] and in [7]. However, the idea of compressing a very large vocabulary[1] of words into a minimal acyclic deterministic finite automaton, and its many applications seems to be new. (In 1988, when this idea was being tested, we were not aware of the work described in [2] and published soon after. The existence of a non-disclosure agreement delayed the preparation of this paper even further.)

The initial motivation for this research was the problem of implementing an efficient spelling checker for the Portuguese language.[2] It turned out however that besides providing a very satisfactory solution for this specific problem, the technique is applicable to most languages, including English, and to many other problems which use large vocabularies. For instance, the spelling checker we mentioned can process about 30,000 words per minute on a standard IBM[3]–compatible personal computer, with the automaton for over 200,000 words fitting into about 124 kbytes of memory; on an 80386 model the speed goes up to 300,000 words per minute.

In the following sections we discuss in more detail the reasons for implementing spelling checkers based on automata, describe the algorithms and data structures used, provide some interesting statistics and show some other possible applications: multilanguage dictionaries, thesauri,

[1] Within this text we use the word *vocabulary* to mean a set of words over some finite alphabet.

[2] Portuguese is a member of the family of Romance languages together with French, Spanish, Italian and others. It is particularly close to Spanish and it is the official language of Brazil, with about 200 million speakers throughout the world. All examples in this paper follow Brazilian usage.

[3] IBM is the trademark of International Business Machines Corp.

minimal perfect hashing and text compression.

2 Implementation of spelling checkers

In early 1988 we were approached by a Brazilian software house which was engaged in the development of a spelling checker and adviser for the Portuguese language. The company had collected a fairly complete machine-readable vocabulary of about 206,000 words, but had serious problems in finding a suitable compact representation, so that a fast spelling checker and adviser with its data structures could be fit into the standard 640 kbytes memory of an IBM–compatible personal computer as a memory resident program.

One of the most widely used spelling checkers is the UNIX[4] program `spell` (see [3, 8]). The program starts by stripping from the given word its affixes (prefixes and suffixes); for instance, `re-work-ed` produces `work` and `over-tak-ing` produces `take`. The resulting word is then hashed producing an index into a very large bit table which provides the answer whether the word belongs or not to the vocabulary. By using the affix stripping, the initial vocabulary of about 250,000 words was reduced to about 30,000. The size of the hashing table is computed in such a way that the probability of a non-existing word colliding with an existing one (i.e., a wrong answer) is about 1/4,000 which is perfectly acceptable in practice. Instead of representing the whole table which is obviously very sparse (out of about 134 million bits, only about 30,000 are ones), differences between consecutive indices of non-zero entries are compressed by using the infinite Huffman codes in order to take care of the variable length integers. Search speed is achieved by partitioning the table into 512 segments, with each segment processed sequentially. The final result is a very compact representation of the original vocabulary within 52 kbytes of storage.

An analysis of the method used by UNIX `spell` shows some of its drawbacks. In the first place, affix stripping can lead to acceptance of

[4] UNIX is a trademark of AT&T Bell Laboratories.

`compar`*a*	`compar`*ada*	`compar`*adas*	`compar`*ado*
`compar`*ados*	`compar`*ai*	`compar`*ais*	`compar`*am*
`compar`*amos*	`compar`*ando*	`compar`*ar*	`compar`*ara*
`compar`*ará*	`compar`*aram*	`compar`*áramos*	`compar`*arão*
`compar`*aras*	`compar`*arás*	`compar`*ardes*	`compar`*arei*
`compar`*areis*	`compar`*áreis*	`compar`*arem*	`compar`*aremos*
`compar`*ares*	`compar`*aria*	`compar`*ariam*	`compar`*aríamos*
`compar`*arias*	`compar`*aríeis*	`compar`*as*	`compar`*asse*
`compar`*ásseis*	`compar`*assem*	`compar`*ássemos*	`compar`*asses*
`compar`*aste*	`compar`*astes*	`compar`*ava*	`compar`*avam*
`compar`*ávamos*	`compar`*avas*	`compar`*áveis*	`compar`*e*
`compar`*ei*	`compar`*eis*	`compar`*em*	`compar`*emos*
`compar`*es*	`compar`*o*	`compar`*ou*	

Figure 1: All 51 distinct forms of the Portuguese verb `comparar` (*to compare*).

non-existing words. Most of the practical cases are eliminated by a stop list: `foreswear` will not be accepted instead of `forswear`, even though `fore` is a valid prefix. However, non-words like `soughted`, `printered` or `electrowordlesslikement` will be accepted! It can be argued of course that such nonsense words will hardly ever occur in a real life text. On the other hand, the speller does accept some non-words which might appear as spelling or typographical mistakes: `womans` instead of `woman's`, `tos` instead of `toes` (or maybe `toss`), and `toing` instead of `toeing` (or `towing`). It should be noted also that this technique produces in fact an infinite vocabulary by allowing almost arbitrary combinations of affixes.

The problem becomes more serious in a highly inflected Romance language. A regular verb in Portuguese has 78 forms, of which 51 are distinct (see Figure 1). Actually there are four groups of regular verbs (i.e. conjugations) derived from infinitive forms ending in `-ar` (like `comparar` — *to compare*), `-er` (`comer` — *to eat*), `-ir` (`partir` — *to leave*) and `-or`

compar*e*
compar*es*
compar*ed*
compar*ing*

Figure 2: All four distinct forms of the English verb `to compare`

.

bonit*a*	conselheir*a*
bonit*as*	conselheir*as*
bonit*o*	conselheir*o*
bonit*os*	conselheir*os*

Figure 3: All four distinct forms of the Portuguese adjective `bonito` (*pretty*) and the noun `conselheiro` (*counselor*).

(`compor` — *to compose*). They all have their own forms, but they also share many common suffixes. This should be contrasted with English where a regular verb has only four distinct forms (see Figure 2). A regular adjective in Portuguese has four distinct forms which contrasts with a unique form in English. Nouns can have the same endings as adjectives when both masculine and feminine forms exist (see Figure 3). As a result, verbs, nouns, adjectives and many other words share the same suffixes. Many of these suffixes are endings of other suffixes as well. All this makes it difficult to apply the suffix stripping technique without an elaborate case analysis scheme.

In view of these problems, we decided to try a different approach by building a minimal acyclic deterministic partial finite automaton accepting exactly the about 206,000 words in the available vocabulary, as described in the following section. In this way we could avoid the problems of introducing non-existing words. Besides that, such automata

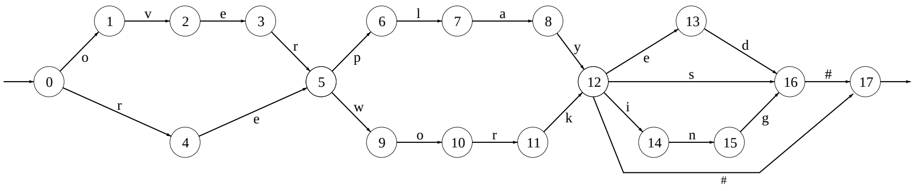

Figure 4: The minimal acyclic finite automaton for all forms of the verbs `rework`, `replay`, `overwork` and `overplay`.

provide a simple and general way of implicitly stripping prefixes and suffixes since each of these will be represented only once. In Figure 4 we show such an automaton for all forms of the English verbs `rework`, `replay`, `overwork` and `overplay`. Notice that in order to include all forms of the verb `work` it suffices to add just one transition labeled by the letter `w` from state 0 to state 9.

An important aspect of this representation is that a word will be found only if it exists explicitly in the vocabulary used to build the automaton. This should be contrasted with the UNIX spelling checker in which it would suffice to insert the verbs `work` and `play` in order to get all those forms and many others, such as `ultrawork` and `pseudoplay`.

On the other hand, as is shown in Section 5, the property of being able to enumerate all words in the vocabulary from its automaton can be very useful in other applications.

3 Implementation of the automaton

The construction of the automaton proceeds according to the basic algorithm:

```
function BuildAutomaton(Vocabulary);
begin
    $\mathcal{A} \leftarrow$ EmptyAutomaton;
    repeat
        while $\mathcal{A}$ not full do
            include the next word of Vocabulary in $\mathcal{A}$;
        $\mathcal{A} \leftarrow minimal(\mathcal{A})$
    until no more words in Vocabulary;
    return $\mathcal{A}$
end
```

After the first execution of the **while** loop the automaton $\mathcal{A}$ is really a digital tree. Figure 5 shows such a tree after the inclusion of all the words used in Figure 4. This tree can grow quite large: if the complete Portuguese vocabulary of 206,000 words were included at once, the tree would have over 600,000 vertices which would be unmanageable on a standard IBM–compatible personal computer running under the MS–DOS[5] system. Therefore the outermost **repeat** loop of the algorithm is necessary. The minimization step takes advantage of the fact that the automaton is acyclic and uses an algorithm which is linear in the size of the automaton. As a matter of fact, this linear algorithm seems to have been discovered independently by others (see for instance [10]).

We use a rather elaborate data structure in order to achieve a very compact memory representation, without sacrificing the access speed, which depends only on the length of the word being searched and not on the size of the automaton or its alphabet. Each state is represented as an array with N entries (N is the size of the alphabet) — most of these entries correspond to non-existing transitions. We take advantage of this fact by shifting and overlapping state arrays in such a way that the existing entries do not collide. This technique is similar to the implementation of *tries* suggested in [6]. To each state we attach one N-bit vector which selects the existing transitions for the state. Array packing is done by a greedy algorithm which in this case gives almost always

[5] MS–DOS is a trademark of MicroSoft Corp.

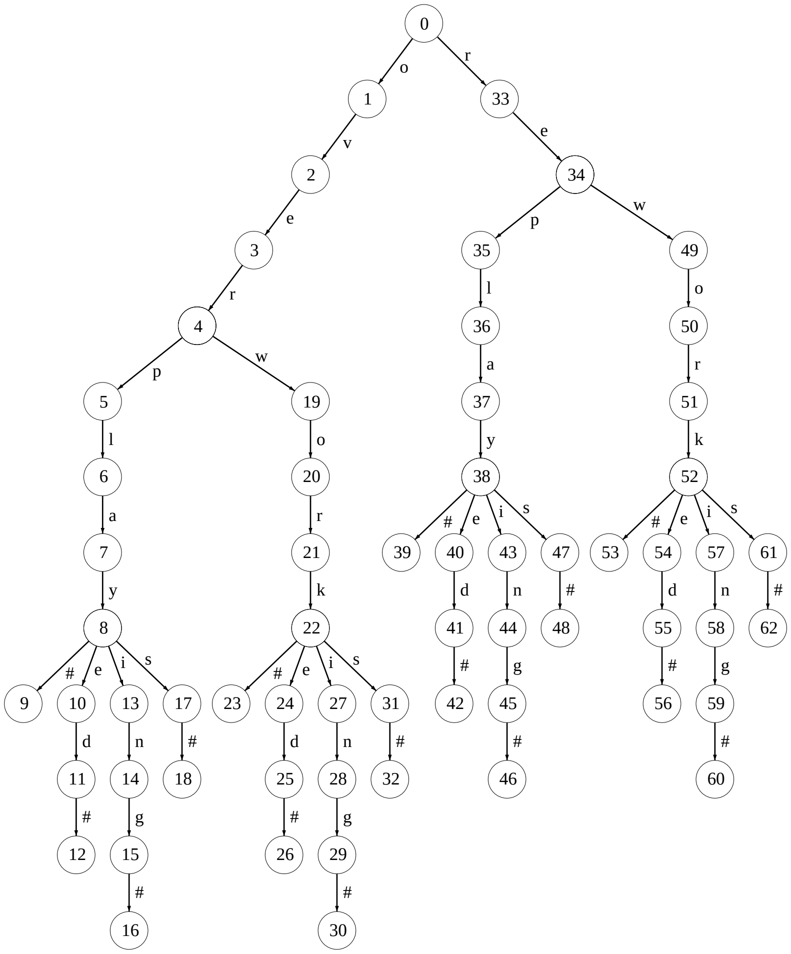

Figure 5: The digital tree for all forms of the verbs `rework`, `replay`, `overwork` and `overplay`.

optimal results due to a very large percentage of states with one, two or three transitions (see Section 4 for some statistics). It also turns out that the number of distinct bit vectors is much lower than the number of states, so that many of them are shared.

The search algorithm for a given word is of course very efficient. Starting from the initial state, it traverses through the automaton by using the consecutive letters of the word to select the transitions, until either a final state is reached or no transition exists (i.e., either the word belongs to the vocabulary or not).

Our final results show that from a practical point of view a simpler data structure could be used, without much increase in storage or access time. For instance, an English vocabulary of about 81,000 words produced an automaton with about 30,000 states and 68,000 transitions. Each state could be represented by a sequence of existing transition pairs: a letter (one byte) and a state index (two bytes); an extra byte for each state would hold the number of existing transitions. Consequently the whole automaton would use about 229 kbytes: a 13% increase over our representation requiring about 203 kbytes. For the Portuguese vocabulary the increase would be of about 22%. The search algorithm for this simpler representation would require a linear pass through the transition sequence for each state, but as we mentioned already most of the states have very few transitions.

Our original representation was chosen mainly because we did not know beforehand either the size of the resulting automaton or its properties, and we tried to minimize the storage requirements. As a matter of fact, besides the packing of the state arrays, we introduced some additional facilities such as short (relative) and long (absolute) state indices, and so on. These may prove useful if we attempt to process much larger vocabularies.

We also included the possibility of representing automata with multiple initial states, each one of them producing a different vocabulary, but minimized together, so that except for the first letters, the common suffixes among words are still properly shared. Strictly speaking such automata are non-deterministic. This is a very restricted form of nondeterminism which can be handled by the searching algorithm through a simple loop without requiring any backtracking. This facility is useful in some applications to be seen in Section 5.

4 Some statistics and measurements

As we mentioned already, our main tests were applied to two cases: an English vocabulary from a popular speller checker with about 81,000 words and a Portuguese vocabulary with about 206,000 words. It should be stressed that the information contents of these two vocabularies are very different. Due to the very high number of derived forms, the Portuguese vocabulary contains a much lower number of "basic" forms than the English one. This fact explains why the English automaton is substantially larger. Figure 6 shows some of the statistics for these vocabularies and automata (the vocabularies are common ASCII files, with one word per line, followed by the *carriage return* and *line feed* characters). We include in this figure the results of compressing the original vocabularies and the automata files with the popular PKZIP and PKPAK[6] utility programs. It should be noticed that the compression rates of the automata files are relatively low, in contrast with the compression of the original vocabulary files. This fact was to be expected, since the automata represent already a compacted form of the vocabulary files (see also Subsection 5.5).

In Figure 7 we show the distribution of the states of the automata according to the number of their transitions per state. In both cases we used the same 26 letter alphabet augmented by the characters ç (*c-cedilla* needed in Portuguese), - (hyphen) and # (word terminator); see the next section for the explanation how Portuguese accented letters are treated. We can see that in both cases about 80% of the states have at most three valid transitions which explains optimal results of our array packing.

Figure 8 shows how these automata grow with the number of words

[6] PKZIP and PKPAK are trademarks of PKWARE, INC.

		English	*Portuguese*
Vocabulary	*Words*	81,142	206,786
	kbytes	858	2,389
	PKPAK	313	683
	PKZIP	253	602
Automaton	*States*	29,317	17,267
	Transitions	67,709	45,838
	kbytes	203	124
	PKPAK	173	105
	PKZIP	183	109

Figure 6: Statistics for the vocabularies and the automata.

included. The words in each vocabulary are distributed first randomly and then alphabetically into ten approximately equal size vocabularies, and then cumulatively added to form the automata. Figure 9 displays the same data graphically.

It is interesting to note that whereas the growth of the automaton is close to linear when the words are included in alphabetical order, a very different behavior is observed with random order inclusion: actually the automaton can decrease in size when more words are included! This behavior is not surprising. With the inclusion in alphabetical order, previously non-existing prefixes and many new word roots keep being included increasing the size of the automaton. When the inclusion follows a random order, most prefixes and roots (and suffixes as well) end up being included in earlier stages. Many of the new words included are simple additions of some derived forms of other words already in the automaton. Such inclusions can make the automaton shrink. For instance, if the word `overplayed` were excluded from the automaton in Figure 4, the resulting automaton would actually grow from 17 states and 22 transitions to 18 states and 25 transitions. As an extreme case, we should remember that, given the 26 letter standard alphabet, a vocabulary of

Transitions	*English*		*Portuguese*	
per state	*States*	%	*States*	%
0	1	0.0	1	0.0
1	14471	49.4	7191	41.6
2	6398	21.8	3639	21.1
3	3369	11.5	2562	14.8
4	1822	6.2	1502	8.7
5	1307	4.5	681	3.9
6	728	2.5	398	2.3
7	375	1.3	315	1.8
8	224	0.8	338	2.0
9	155	0.5	273	1.6
10	114	0.4	153	0.9
11	68	0.2	69	0.4
12	61	0.2	27	0.2
13	48	0.2	22	0.1
14	43	0.1	24	0.1
15	25	0.1	11	0.1
16	17	0.1	17	0.1
17	13	0.0	10	0.1
18	10	0.0	10	0.1
19	16	0.1	10	0.1
20	15	0.1	2	0.0
21	14	0.0	7	0.0
22	9	0.0	2	0.0
23	4	0.0	1	0.0
24	4	0.0	1	0.0
25	2	0.0	0	0.0
26	4	0.0	1	0.0

Figure 7: Distribution of states according to the number of their transitions.

%	*English*					
	Random			*Alphabetical*		
	States	*Transitions*	*kbytes*	*States*	*Transitions*	*kbytes*
10	10,459	17,935	52	4,178	8,540	25
20	16,478	29,904	86	7,109	15,474	44
30	21,201	40,017	114	9,869	21,560	61
40	24,891	48,511	137	12,538	28,219	79
50	27,651	55,434	160	15,351	34,633	97
60	29,781	61,130	179	18,953	42,402	118
70	31,106	65,491	195	21,576	48,608	135
80	31,746	68,626	206	23,705	54,216	153
90	31,418	69,809	210	25,973	60,137	175
100	29,317	67,709	203	29,317	67,709	203

%	*Portuguese*					
	Random			*Alphabetical*		
	States	*Transitions*	*kbytes*	*States*	*Transitions*	*kbytes*
10	17,817	34,751	97	2,126	5,090	15
20	22,713	52,627	153	4,375	11,165	32
30	25,766	65,729	206	5,677	14,602	41
40	27,720	75,370	244	6,853	17,762	49
50	29,007	82,609	275	8,414	22,033	61
60	29,836	88,130	297	10,726	27,465	76
70	30,101	92,081	314	12,845	33,170	90
80	29,333	92,047	312	14,426	37,513	102
90	26,896	84,611	280	15,665	41,166	112
100	17,267	45,838	124	17,267	45,838	124

Figure 8: Growth of the automata.

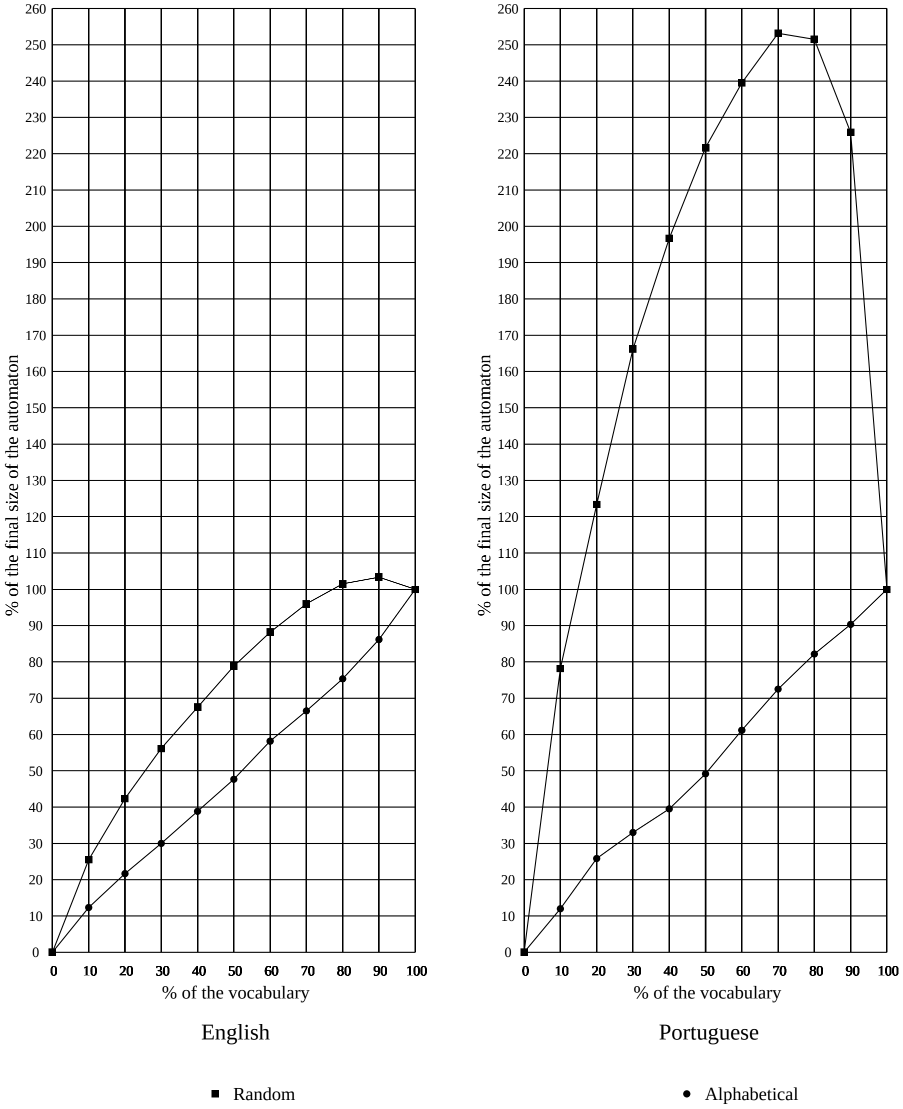

Figure 9: Graph of the growth of the automata.

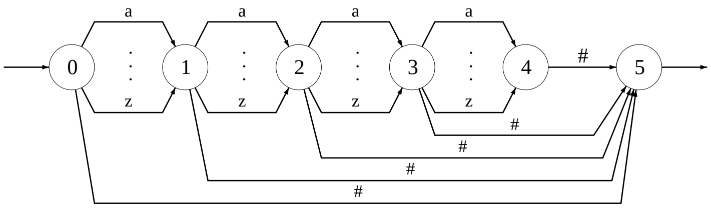

Figure 10: Automaton accepting all words of up to four letters.

all letter sequences of length up to M would have $(26^{M+1}-1)/25$ words; for instance, for $M = 4$ we would have 475,255 words. On the other hand, the automaton for such a vocabulary would have only $27M + 1$ transitions (109 for $M = 4$; see Figure 10).

We also measured the speed with which our automaton can be used. For both languages the results are practically the same, even though the average English word is shorter than the Portuguese one. Our tests were programmed in C and carried out on a standard IBM–compatible personal computer with a 4.77 MHz clock. A simple spelling checker reading a normal text from a hard disk file could process about 30,000 words per minute; on an 80386 model we achieved the speed of 300,000 words per minute.

We would like to mention also that we built the automaton for the UNIX system dictionary `/usr/dict/words` containing about 25,000 commonly used English words (202 kbytes).[7] The automaton has 16,445 states, 38,288 transitions and uses 112 kbytes of memory.

[7] In order to keep a 29 letter alphabet, we used the standard 26 letters plus - (hyphen), ' (quote) and # (word terminator), translated upper case letters into lower case, and eliminated the few words which include digits.

5 Applications

5.1 Spelling checkers and advisers

As we mentioned already, our first motivation was the implementation of a spelling checker and adviser for Portuguese.[8] It should be obvious that the minimal acyclic finite automata we described provide a very convenient basis for the spelling checking part for any language for which such an automaton can be built. An additional problem we had to face is the existence in Portuguese of 12 letters with diacritical marks: à, á, é, í, ó, ú, â, ê, ô, ã, õ and ü. One simple solution would be to increase by 12 the size of the alphabet. We chose however to strip the letters of their marks, and encode them and their positions after the word terminator. This solution contributes to decrease the size of the automaton, since words like compararámos and compararam or órgão and orgânico produce longer common prefixes: compararam- and orga-. Very few words have more than one diacritical mark,[9] so that the distinct suffixes created by the encoding are relatively few.

With regard to the spelling adviser, we relied heavily on the fact that Portuguese uses a fairly phonetic spelling system. Besides that, one of our design decisions was that the program should detect all mistakes (relative to its vocabulary), but would have to give good advice mainly for spelling and not for typing mistakes; the latter ones are easily corrected by the users after they are pointed out.

One of the most common sources of spelling mistakes in Portuguese is the wrong usage of diacritical marks: for instance, necessario instead of necessário (*necessary*) or fôlha instead of folha (*sheet*). Sometimes the adviser will present several alternatives. The three forms sabia (*knew*), sábia (*wise woman*) and sabiá (*a native Brazilian bird*) are correct; however sabía and sâbia do not exist. The encoding of diacritical marks as suffixes makes it particularly easy to find all the existing

[8] A commercial spelling checker and adviser based on the ideas described in this section was implemented by TTI Tecnologia Ltda., São Paulo, SP, Brazil.

[9] The word qüinqüelíngüe (*fluent in five languages*) seems to be an absolute champion with its four marks!

forms of a word which agree except for those marks.

Another source of common spelling mistakes are letter combinations which denote similar sounds: extender instead of estender (*to extend*), pesquiza instead of pesquisa (*research*), essessão instead of exceção (*exception*), humido instead of úmido (*humid*). We take care of this problem by using some phonetic rules. The word whose spelling alternatives we want to find is transformed (by another very simple automaton), after being stripped of its diacritical marks, into a convenient representation: extender would become eS$_1$tender, essessão would become eS$_2$eS$_3$ao, where the symbols S$_1$, S$_2$ and S$_3$ denote the usual sound of the letter s for different letter contexts. Possible substitutions for the symbol S$_1$ are: {s,x}, for the symbol S$_2$: {ss,sc,xc,c,cc}, and for the symbol S$_3$: {ss,ç,cç}. Next, a backtracking algorithm is used to enumerate all the possibilities and check them against the automaton. In the case of the word essessão we would have apparently 15 alternatives like esceçao and excessao. Since the substitutions are generated from left to right, and tested incrementally against the automaton, few alternatives are actually generated, since words starting with esce or eces do not exist. As a final result we get a list of alternatives, in which we include diacritical marks whenever they apply. Most of the time the list is very short and accurate. It should be noted that if the adviser were based on a hashing scheme, incremental testing of the alternatives would not be possible.

We believe that the technique we use for spelling advising could be easily adapted to many languages which use phonetical spelling systems; Spanish and Italian are good candidates. For other languages this approach might be less applicable; we certainly would not advise it for English. It should be noticed, however, that the speed with which the automaton can be traversed would probably make feasible other approaches, such as extensive testing of letter substitutions, transpositions, omissions and insertions.

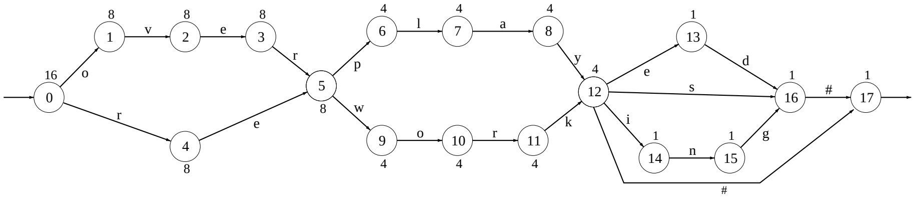

Figure 11: The numbered version of the automaton

5.2 Minimal perfect hashing

Let us assume that the representation of our automaton includes, for each state, an integer which gives the number of words that would be accepted by the automaton starting from that state. We shall refer to such an automaton as a numbered automaton. In Figure 11 we show the numbered version of the same automaton of Figure 4. The numbering can be done by a fairly simple traversal through the automaton, once it was built. The storage requirements for this addition are fairly modest: one integer per state (30 to 35% of storage increase in our examples).

Given such a numbered automaton, we can write two simple functions which implement a one-to-one correspondence between the integers 1 to L (L is the number of words accepted by the automaton) and the words themselves, as shown in Figures 12 and 13.[10]

These functions represent an efficient and compact minimal perfect hashing scheme for the vocabulary which can be used in several applications (some will be mentioned in this section). It should be stressed that this scheme can be used only if the hashing functions do not change very often since the construction of the automaton can be quite time consuming. This should be contrasted with the results described for instance in [9], where a fast method to determine a minimal perfect hashing

[10] The ordering implied by this numbering is the lexicographic ordering of the original vocabulary.

```
function WordToIndex(Word);
    begin
    Index ← 0;
    CurrentState ← InitialState;
    for I ← 1 to Length(Word) do
        if ValidTransition(CurrentState, Word[I]) then
            begin
            for C ← FirstLetter to Predecessor(Word[I]) do
                if ValidTransition(CurrentState, C)
                    then Index ← Index + CurrentState[C].Number;
            CurrentState ← CurrentState[Word[I]];
            if IsFinal(CurrentState)
                then Index ← Index + 1
            end
            else return Undefined;
    if IsFinal(CurrentState)
        then return Index
        else return Undefined
    end
```

Figure 12: Hashing function

function is described. The computation of the resulting hashing requires however that the whole vocabulary be kept as part of the data structure, which is usually much larger than the automaton used in our method.

5.3 Multilanguage dictionaries

Numbered automata can be used to implement multilanguage dictionaries for simple word-to-word translations. Vocabularies for several languages can be represented by one automaton with multiple initial states, one for each language. It is interesting to note that even though differ-

```
function IndexToWord(Index);
    begin
    CurrentState ← InitialState;
    Count ← Index;
    OutputWord ← EmptyWord;
    repeat
        for  C ← FirstLetter to LastLetter do
            if ValidTransition(CurrentState,C) then
                begin
                AuxState ← CurrentState[C];
                if AuxState.Number < Count
                    then Count ← Count − AuxState.Number
                    else
                        begin
                        OutputWord ← OutputWord & C;
                        CurrentState ← AuxState;
                        if IsFinal(CurrentState)
                            then Count ← Count − 1;
                        exit forloop
                        end
                end
    until Count = 0;
    return OutputWord
    end
```

Figure 13: Unhashing function

ent languages are involved, by reversing the words while we build the automaton, the minimization process takes advantage of any existing spelling similarities like for instance common Latin prefixes and roots existing in many European languages. Besides the automaton, for each language we can use an array indexed by the word numbers and map them into lists of indices for other languages. The lists can be part of the arrays themselves as shown in the hypothetical example in Figure 14 for an English–French–Portuguese dictionary.

5.4 Thesauri

Given a word like `work`, a simple thesaurus might produce an output like:

```
work:
noun    avocation,calling,employment,field,job,occupation,
        profession,trade,vocation;
        chore,drudgery,grind,labor,slavery,sweat,tedium,toil,
        travail.
verb    answer,do,fulfill,meet,qualify,satisfy,suffice.
```

Thus for each grammatical category to which the word belongs (noun, verb, etc.) we have a set of lists of related words, with each list corresponding to a different interpretation of the word. Such a thesaurus is usually complete (or closed) in the sense that if we give it any of the words on one of these lists, we get as one of the results the same list (the word given as the key is usually excluded). Thus if we give the thesaurus the word `toil` we might get:

```
toil:
noun    chore,drudgery,grind,labor,slavery,sweat,tedium,
        travail,work.
verb    lumber,persevere,persist,plod,plug.
```

We implemented this kind of thesaurus by using a numbered automaton with multiple initial states: each initial state corresponds to one

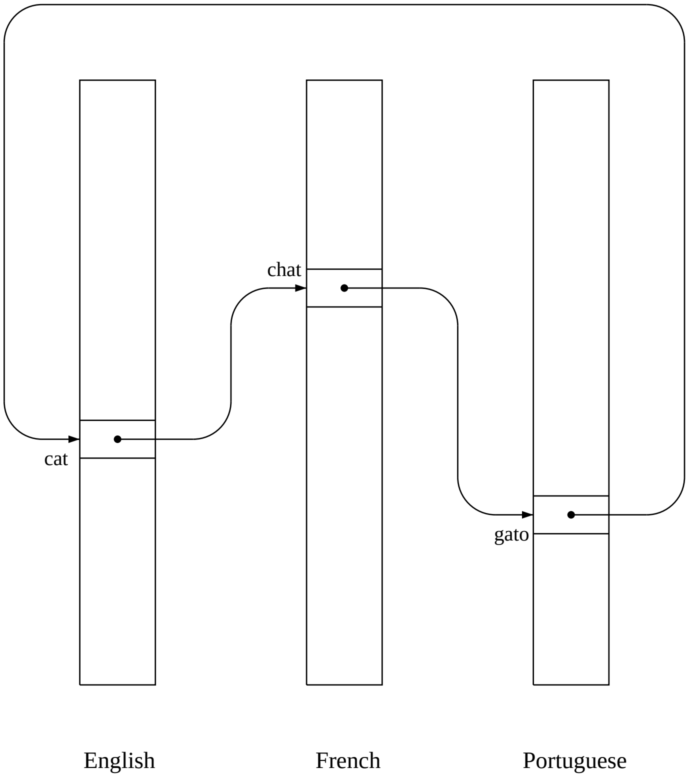

Figure 14: Example of the auxiliary arrays for multilanguage dictionary and the word `cat` (`chat` in French and `gato` in Portuguese).

grammatical category. Besides the automaton, we use some additional data structures to represent the lists of words as sequences of numbers. This implementation was tested for an English thesaurus which is part of a popular commercial product. Its automaton accepts about 9,500 words (a word like `work` is counted twice, since it is a noun and a verb), has less than 9,000 states and 18,000 transitions. The storage requirements are 88 kbytes for the automaton and about 39 kbytes for the additional data structures. The original commercial implementation required about 159 kbytes.

This kind of application is of course very general and does not depend on the language.

5.5 Text compression

A sequence of words belonging to the vocabulary of an automaton can be obviously encoded by the sequence of its numbers which will usually require less space. In practice the problem of text compression is more complicated due to the appearance of words not belonging to the vocabulary, treatment of lower and upper cases and inclusion of non-letter characters. It is possible however to combine the above idea with other compression techniques (see [12]). Our preliminary results show that the performance is sometimes reasonably close to that of the utility programs PKZIP and PKPAK, but not any better. It seems that in this case only applications in some special contexts might prove to be of interest. For instance, if we wish to compress a set of words, regardless of their order, we can use the automaton itself. The size of the automaton can be much smaller than the result of a compression program as shown in Figure 6. It also shows that some additional savings can be achieved by compressing the automaton file itself.

6 Future work

We have shown that finite automata provide a useful tool for many applications where a very compact representation of large vocabularies with direct access is required. One of the directions we would like to follow in the future is to do some experiments on languages other than English and Portuguese, especially on those with different spelling systems like for instance Arabic and Hebrew, or even Japanese and Chinese where a suitable representation for their characters would have to be used. It seems however that machine-readable vocabularies for these languages are not easy to find.

Another interesting direction is to try to build automata for even larger vocabularies, in order to study the statistics they produce and to try to understand what kind of information they provide about the language, or at least about its spelling system.

Finally we would like to study other possible applications for these ideas. One of them might involve vocabularies found in molecular biology.

Acknowledgments

We wish to thank Nilo S. Mismetti and Fernando Mismetti from TTI Tecnologia Ltda. They provided the initial motivation and part of the material support necessary to carry out this research, besides many stimulating discussions and constant challenges. Imre Simon from the University of São Paulo gave us recently some additional hints about applications of automata and provided us with an excellent bibliography (see [11]).

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. *Commun. ACM 31,5* (May 1988), 572–578, 585.

[3] Bentley, J. A spelling checker. *Commun. ACM 28,5* (May 1985), 456–462.

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

[5] Hopcroft, J. E. and Ullman, J. D. *Introduction to Automata Theory, Languages and Computations*, Addison–Wesley, Reading, Ma. 1979.

[6] Knuth, D. E. *The Art of Computer Programming*, vol. 3, *Sorting and Searching*, Addison–Wesley, Reading, Ma. 1973, pp. 481–486, 500.

[7] Liang, F. M. Word Hy-phen-a-tion by Com-pu-ter. Stanford University, Stanford, Ca. 1983 (Ph.D. Dissertation).

[8] McIlroy, M. D. Development of a spelling list. *IEEE Trans. Comm. 30,1* (January 1982), 91–99.

[9] Massalin, H. and Pu, C. Generating Minimal Perfect Hash Functions for Entire Dictionaries. Department of Computer Science, Columbia University, New York, NY 1990 (manuscript).

[10] Revuz, D. Algorithm linéaire de minimisation des automates déterministes acycliques, 1990 (manuscript).

[11] Simon, I. *Palavras, Autômatos e Algoritmos — uma Bibliografia* (*Words, Automata and Algorithms — a Bibliography*), VII Escola de Computação, University of São Paulo, São Paulo 1990.

[12] Storer, J. A. *Data Compression — Methods and Theory*, Computer Science Press, Rockville, Md. 1988.

Relatórios Técnicos

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*

Departamento de Ciência da Computação — IMECC
Caixa Postal 6065
Universidade Estadual de Campinas
13081-970 – Campinas – SP
BRASIL

`reltec@dcc.unicamp.br`