How to squeeze a lexicon

Marcin G. Ciura, Sebastian Deorowicz

July 14, 2001

This is a preprint of an article published in
Software—Practice and Experience, 2001; 31(11):1077–1090

Abstract

Minimal acyclic deterministic finite automata (ADFAs) can be used as a compact representation of finite string sets with fast access time. Creating them with traditional algorithms of DFA minimization is a resource hog when a large collection of strings is involved. This paper aims to popularize an efficient but little known algorithm for creating minimal ADFAs recognizing a finite language, invented independently by several authors. The algorithm is presented for three variants of ADFAs, its minor improvements are discussed, and minimal ADFAs are compared to competitive data structures.

KEY WORDS: static lexicon; static dictionary; trie compaction; directed acyclic graph; acyclic finite automaton

INTRODUCTION

Many applications involve accessing a database, whose keys are variable-length finite sequences of characters from a fixed alphabet (*strings*). Such databases are known as *symbol tables* or *dictionaries*. Sometimes no data are associated with the keys, we need only to know whether a string belongs to a given set. Following Revuz [23], we call a set of bare strings a *lexicon* to distinguish it from a complete dictionary.

In spelling checkers and other software that deals with natural language the lexicons often contain hundreds of thousands words, yet they need no frequent updates. Knowing all the keys in advance allows to prepare a static data structure that outperforms dynamic ones in size or time of searching. At least several ways to construct such a data structure are conceivable.

Hashing

Perfect hashing, a classical solution to the static dictionary problem [5], requires storage for all the strings; at most they can be compressed with a static method. When they are just words, affix stripping [13, 17] reduces the space requirements. It can be applied to many languages, but for languages more inflective and irregular than English the accurate morphological rules are complex, and grammatical classification of corpus-based word lists demands either human labour or sophisticated software.

A sparse hash table allows discarding the strings entirely, at the cost of occasional false matches, and can be encoded compactly yielding a Bloom filter [17]. Sometimes, though, absolute certainty is desired. Moreover, once the strings are dropped, it is impossible to reconstruct them.

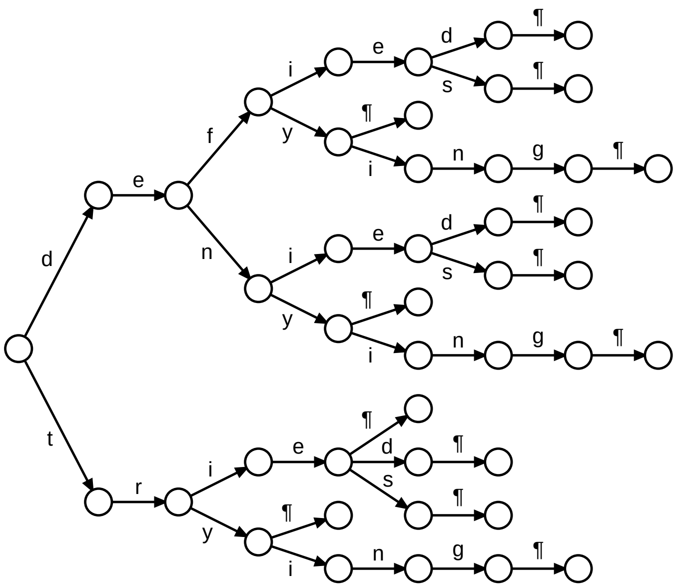

Figure 1: A full trie for a contrived set of strings.

Tries

A character tree, also known as a *trie* [11, Chapter 6.3], is an oriented tree, in which every path from the root to a leaf corresponds to a key and branching is based on successive characters. Tries are sometimes preferable to hashing, because they detect unsuccessful searches faster and answer partial match or nearest neighbour queries effectively.

Tries are found in two varieties: *abbreviated* and *full*. The former comprise only the shortest prefixes necessary to distinguish the strings; finding a string in them must be confirmed by a comparison to a suffix stored in a trie leaf. The latter comprise entire strings, character by character, up to a string delimiter, as shown in Figure 1.

In a plain trie the set of arcs coming from each node is represented as an array of size equal to the alphabet size, whose elements are either pointers to child nodes or null pointers. This technique needs excessive amounts of storage for large collections of strings. Memory usage decreases at the cost of retrieval time by eliminating null pointers and arranging the remaining pointers into a list or a binary search tree. Furthermore, static lists and static complete binary trees occupying subsequent memory locations can get along with one bit instead of an explicit link to a successor, as shown in Figure 2. An alternative approach, which does not compromise the search time, is to overlap the nodes, so that all the non-null pointers retain their positions within a node and occupy empty locations of the other nodes [2].

Path compression generalizes the concept of abbreviated tries: a *path-compressed trie* replaces single paths with single arcs, using strings for branching. In our example trie in Figure 1, the paths d-e and t-r, as well as i-e, i-n-g-¶, d-¶, and s-¶ (thrice) could be compressed. Still other methods of reducing the trie size, such as C-trie [16] and Bonsai [8], lower the space occupied by a node, ingeniously using only one bit per pointer or one pointer to all the offspring nodes.

Finite automata

We concentrate on yet another technique that consists in merging all equivalent subtries of a full trie and can coexist with any method of node packing that preserves distinct pointers to child nodes. Comparing the trie in Figure 1 with the directed acyclic graph in Figure 3 clears this idea. Intuitively, it is especially attractive when the keys come from a natural language, since they are likely to share many suffixes, and indeed we shall see that such lexicons can be substantially compacted this way. The obtained structure, which can be viewed as a minimal deterministic finite automa-

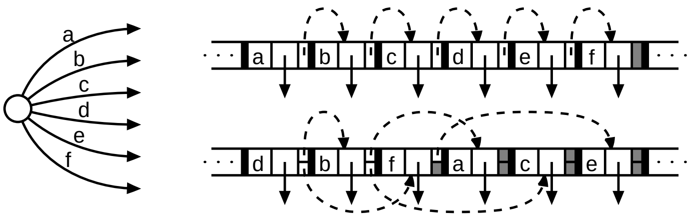

Figure 2: A sample node and its representations as a static list and a static complete binary tree. Dashed arrows indicate implicit links.

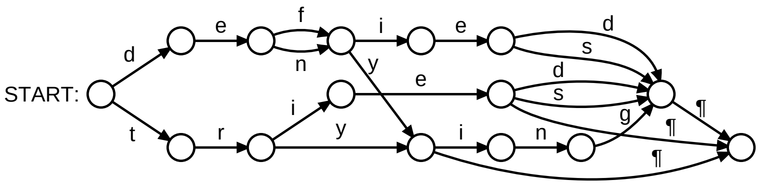

Figure 3: Transition diagram of a simple recognizer.

ton that recognizes a given set of strings, used to be named directed acyclic word graph (DAWG) in some publications, but DAWG is actually another concept [4]. From now on we call it a *recognizer* for short [1, Section 3.6], and use the terms *states* and *transitions* from automata theory rather than nodes and arcs from graph theory.

Recognizers have been successfully employed to store routing tables [28], hyphenation rules in TEX [12, Part 43], and large lexicons for speech recognition [14], word game programs [3, 9], and spelling checkers [15]. Recognizers inherit the advantages of tries, but cannot associate distinct information with the keys, which seems to limit their applications to lexicons. However, augmenting their states with numeric data provides a minimal perfect hash function of the given strings [15], making recognizers a proper dictionary structure.

This paper discusses static recognizers; dynamic recognizers, which can be updated without rebuilding them from scratch, are dealt by Park *et al.* [21] and Sgarbas, Fakotakis & Kokkinakis [24].

THE RECOGNIZERS AND THEIR CONSTRUCTION

The straightforward method of constructing a static recognizer: minimizing an incrementally built dynamic trie, either with generic DFA minimization algorithms [25], or algorithms specialized for acyclic automata [12], can consume a large amount of memory even if the output is relatively small. Revuz [22] and Watson [27] have developed incremental algorithms that produce a partially minimized automaton instead of a full trie. A direct, both space-efficient and fast algorithm to construct a minimal acyclic automaton has been independently invented several times, with secondary differences, by Jan Daciuk [6], Stoyan Mihov [18, 19], Bruce W. Watson & Richard E. Watson (see Reference [7]), and the first author.

Simple recognizers

To ease the explanation, we begin with what we tentatively call a *simple recognizer* (see Figure 3). Let us treat transitions as pairs ⟨*attribute character*, *destination state*⟩ and states as sets of their *out-transitions*. Two transitions are equal if both of their elements are equal. Informally, two states are *equivalent* if joining them, i.e. pointing all their *in-transitions* to one of them and removing the other state, does not change the set of strings recognized by the automaton. In particular, states with equal sets of out-transitions are equivalent. As the recognizer is a *minimal* finite automaton, it contains no equivalent states.

01 $s_0[0] \leftarrow$ '¶'; $i \leftarrow 0$; $\mathit{larval\_state}[0] \leftarrow \emptyset$;
02 **while not** *eof* **do**
03   read next string into $s_1[0\,..\,q-1]$, and set q to its length;
04     $s_1[q] \leftarrow$ '¶'; $p \leftarrow 0$;
05   **while** $s_0[p] = s_1[p]$ **do** $p \leftarrow p+1$; **end while**; $\{p \le q\}$
06   **while** $i > p$ **do** $\mathit{new\_state} \leftarrow \mathit{make\_state}(\mathit{larval\_state}[i])$;
07         $i \leftarrow i-1$; $\mathit{larval\_state}[i] \leftarrow \mathit{larval\_state}[i] \cup \{\langle s_0[i], \mathit{new\_state}\rangle\}$;
08   **end while**; $\{i \le p \le q\}$
09   **while** $i \le q$ **do** $s_0[i] \leftarrow s_1[i]$;
10         $i \leftarrow i+1$; $\mathit{larval\_state}[i] \leftarrow \emptyset$;
11   **end while**; $\{i = q+1\}$
12 **end while**;
13 **while** $i > 0$ **do** $\mathit{new\_state} \leftarrow \mathit{make\_state}(\mathit{larval\_state}[i])$;
14     $i \leftarrow i-1$; $\mathit{larval\_state}[i] \leftarrow \mathit{larval\_state}[i] \cup \{\langle s_0[i], \mathit{new\_state}\rangle\}$;
15 **end while**;
16 $\mathit{start\_state} \leftarrow \mathit{make\_state}(\mathit{larval\_state}[0])$;

Figure 4: The algorithm for creating a simple recognizer.

There is a distinguished *start state*, without in-transitions, equivalent to the root of a trie. As in a full trie, the alphabet of a simple recognizer includes a string delimiter to distinguish the strings that are prefixes of other strings. All the transitions tagged with ¶ go to a *terminal state*, which has an empty set of out-transitions. Due to the minimality of the automaton, it contains exactly one terminal state.

Figure 4 shows the algorithm for constructing a simple recognizer. It uses a temporary variable *new_state*, three integer variables i, p, and q, two character arrays s_0 and s_1—both of range $[0\ ..\ l_{\max}]$, and an array of sets of transitions $\mathit{larval\_state}[0\ ..\ l_{\max}+1]$, where $l_{\max}$ stands for the maximal possible length of input string (without delimiter). The point of the algorithm, first noticed by Mihov [19], lies in such an order of adding states to the recognizer that once a state is formed, it need not be updated. Therefore, any external representation of the recognizer is supported.

The algorithm is sequentially fed with strings. For each prefix of strings in the lexicon, the strings beginning with it shall appear consecutively; otherwise a non-deterministic automaton is constructed instead of a deterministic one. The most natural way to meet this condition is to sort the strings lexicographically. The presented version fails if the same string occurs more than once, but a trivial modification of line *05*, which finds the length of a common prefix of two strings, would immunize it against this case. In the set union operation in lines *07* and *14*, it is unnecessary to check if the transition already belongs to $\mathit{larval\_state}[i]$, because it can never happen. For this reason, we can implement the elements of *larval_state* not as generic sets, but simply as arrays of as many elements as the alphabet has (including the sentinel character ¶) coupled with length-of-array counters. Also, lines *13–15* are superfluous if we ensure that the empty string is read after all the keys.

As soon as it is known that no more transitions will be added to a larval state, it is passed to the function *make_state*. Given a larval state, this function searches the portion of the recognizer built so far for a state equal to it or else adds such a state to the recognizer. It returns a descriptor of the found or added state, which is an index of its first transition or a pointer to it in a most natural representation of a recognizer. It turns out that joining equal states at this moment yields a minimized recognizer without any equivalent states.

The algorithm maintains an auxiliary data structure, not mentioned explicitly in Figure 4. It is updated and searched by the function *make_state* and stores information about already created states. Of course, it is not needed for using the recognizer when its construction finishes. The number of states in the recognizer is bounded by $N+2$, where N denotes the total length of strings in the lexicon (usually, it is at least an order of magnitude less than N). Therefore, a hash table

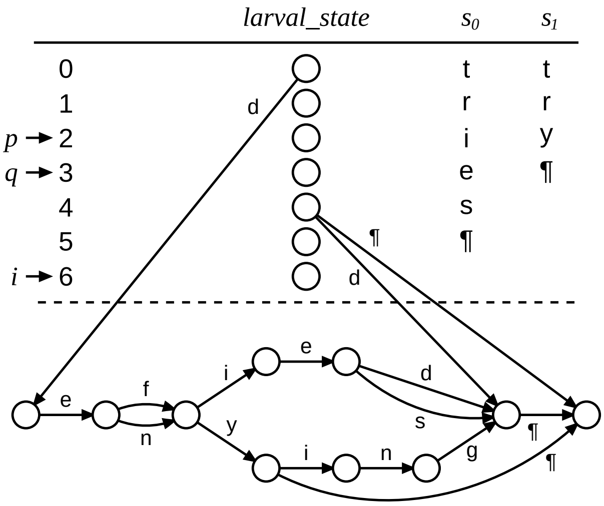

Figure 5: A snapshot of the algorithm at work.

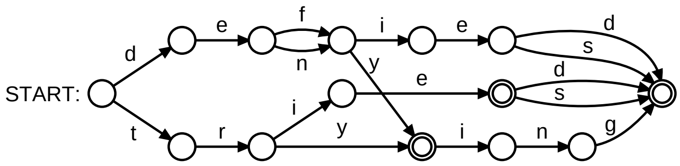

Figure 6: Transition diagram of a Moore recognizer. Double circles indicate terminal states.

with N/r buckets has $O(r)$ expected time of retrieval and insertion when used as this auxiliary structure. When the size of the hash table is chosen so that r is small, the expected overall time complexity of the algorithm is linear in the number of characters in the lexicon. A naive implementation missing the hash table can linearly search the recognizer, achieving quadratic running time.

Figure 5 shows the values of the variables in the course of an example construction, between the lines *05* and *06* of the algorithm. The reader is advised to execute the loop *06–08* by hand to digest its method of operation. To this end, we can imagine *larval_state*$[i]$ dragged below the dashed line, even though the function *make_state* need not destroy its contents. Whereas this function finds in the recognizer a state equal to the given one using its hash table, we have to find it below the dotted line with our eyes. In the initial two iterations, it is there; in the remaining two iterations, it is not.

Other types of recognizers

This section treats recognizers with fewer states and transitions than their simple counterparts.

Finite automata with output encoded in their states are called *Moore machines* [10, Chapter 2.7]. Changing our definition of a state to a pair $\langle$*set of transitions*, *terminality bit*$\rangle$ yields a specimen of acyclic Moore machine with one-bit output. Such a Moore recognizer for the sample set of strings is presented in Figure 6. It has one state fewer than an equivalent simple recognizer and misses all the transitions labelled with ¶.

There are also *Mealy machines* that have their output encoded in the transitions. Defining a transition as a triple $\langle$*attribute character*, *destination state*, *terminality bit*$\rangle$, equality of transitions as equality of triples, and a state again as a set of transitions, we get a Mealy recognizer (see Figure 7). The only distinction in using them is rather of theoretical interest: Mealy recognizers cannot recognize the empty string, Moore (and simple) recognizers can.

Direct algorithms creating Moore and Mealy recognizers are presented in Figure 8. They work

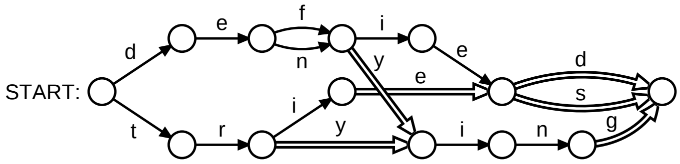

Figure 7: Transition diagram of a Mealy recognizer. Thick arrows indicate terminal transitions.

```
01        macro generic_make_state(var i : integer);
02-Moore      new_state ← make_state(larval_state[i], is_terminal[i]);
03-Moore      i ← i − 1; larval_state[i] ← larval_state[i] ∪ {⟨s0[i], new_state⟩};
02-Mealy      new_state ← make_state(larval_state[i]);
03-Mealy      i ← i − 1; larval_state[i] ← larval_state[i] ∪ {⟨s0[i], new_state, is_terminal[i + 1]⟩};
04        end macro;
05        s0[0] ← '¶'; i ← 0; larval_state[0] ← ∅; is_terminal[0] ← false;
06        while not eof do
07            read next string into s1[0 .. q − 1], and set q to its length;
08            s1[q] ← '¶'; p ← 0;
09            while s0[p] = s1[p] do p ← p + 1; end while;   {p ≤ q}
10            while i > p do generic_make_state(i); end while;   {i ≤ p ≤ q}
11            while i < q do s0[i] ← s1[i];
12                i ← i + 1; larval_state[i] ← ∅; is_terminal[i] ← false;
13            end while;   {i = q}
14            s0[q] ← '¶'; is_terminal[q] ← true;
15        end while;
16        while i > 0 do generic_make_state(i); end while;
17-Moore  start_state ← make_state(larval_state[0], is_terminal[0]);
17-Mealy  start_state ← make_state(larval_state[0]);
```

Figure 8: The algorithms for creating Moore and Mealy recognizers.

under the same assumptions about the input as the algorithm for creating a simple recognizer in Figure 4 and use analogical data structures (except *larval_state* that needs only the elements $[0\,..\,l_{\max}]$), plus a Boolean array $is\_terminal[0\,..\,l_{\max}]$.

One can imagine also dual automata with attribute characters encoded in states instead of transitions, and writing algorithms creating minimal acyclic automata of this kind is an easy exercise. However, using such automata requires twice as many memory reads as using conventional ones, and, moreover, they have in general more states and transitions.

We experimented also with automata that have distinct start states for strings of different length and thus need no explicit terminal states. Curiously, neither has a minimal recognizer of this type to be unique and acyclic, nor does the presented algorithm guarantee a minimal construction. The recognizer for the six-character string 'banana' that could have only three transitions instead of six illustrates this phenomenon. Anyhow, the acyclic instances obtained with the direct algorithm have many more states and transitions than ordinary automata.

As a final remark for this section, we note that the presented algorithms can be easily modified to handle several types of string delimiters or several bits of terminality. For example, consider a spelling checker for any language written with an alphabet that distinguishes the case of letters. The lexicon contains common, all-minuscule words, capitalized Proper Names, ACRONYMS, and a few irregularities like 'PhD' or 'PostScript'. Although all the words can be stored in a recognizer as they are usually written, it is wiser to convert them to one case and let the terminal state or transition determine the canonical spelling of a word (the irregularities being a class on their own with a mask of capitals attached). This way the recognizer needs to be consulted exactly once for each word checked, regardless of its nontypical capitalization, e.g. starting a sentence.

Table 1: Parameters of test lexicons.

Lexicon	Raw size [KB]	Number of strings	Number of base forms	Average string length	Median string length	Alphabet size
English	688	74 317	35 063	8.47	8.25	53
Esperanto	11 205	957 965	17 588	10.98	10.90	62
French	2 418	221 376	47 618	10.19	10.09	69
German	2 661	219 862	40 174	11.40	11.26	58
Polish	16 647	1 365 467	92 368	11.48	11.31	64
Russian	8 911	808 310	80 241	10.29	10.12	60
Spanish	7 187	642 014	52 870	10.46	10.38	56
ENABLE	1 709	173 528	·	9.08	8.74	26
DIMACS	6 831	309 360	·	21.61	20.44	40
Pathnames	202 414	2 744 641	·	75.52	73.82	111
Random	1 027	100 000	·	9.51	9.54	26

Also, as a rule it contains slightly fewer states and transitions than a recognizer with a potpourri of lowercase and uppercase characters.

Another application of recognizers for more than one class of strings is discussed in the next section.

Comparison of the representations

The three types of automata presented above differ in size. To compare them in practice, we used several lexicons:

- English, Esperanto, French, German, Polish, Russian and Spanish lexicons for the spelling checker Ispell [13].
- ENABLE (Enhanced North American Benchmark Lexicon), a list of English words used as a reference in word games (see http://personal.riverusers.com/~thegrendel/enable2k.zip).
- DIMACS—Harvard Library call numbers from the DIMACS Implementation Challenge (see http://theory.stanford.edu/~csilvers/libdata/). Each string contains up to 30 characters like LJA_ _ _ _84_C_63_A_57_X_1982 or WSOC_ _ _ _ _ _5305_ _ _370_ _ _ _5.
- Pathnames that represent the whole contents of ftp://ftp.funet.fi/pub/. This lexicon comprises the leaves of the directory tree, i.e. besides files, also symbolic links and empty directories. In a search engine, reversed pathnames would be more useful, so we include also results for them.
- Random strings. Their length is uniformly distributed between 4 and 15 characters; each character is independently chosen from a 26-character alphabet.

Table 1 contains some relevant parameters of the lexicons listed above, and Table 2 shows the number of states and transitions in the recognizers constructed for them. As was expected, the Mealy recognizers are superior to the others in this view.

On the other hand, the virtue of simple recognizers is their clean structure. They can prove useful when a few dozen types of strings exist, i.e. each string has a few bits of data connected with it. Their sample application are lexicons that comprise word bases for the morphological analysis of text. Classes of affixes allowed for a base are encoded in terminal states or transitions of an automaton. Words with the same base can take different affixes, as shown by Polish puszcz-[a,y,...] 'a forest' and puszcz-[am,asz,...] 'to release', Latin secur-[us,a,um,...] 'secure' and secur-[is,is,...] 'an axe', or Spanish dur-[o,as,...] 'to last' and dur-[o,os,...] 'hard'. The number of terminal

Table 2: Number of states and transitions in the three types of automata.

Lexicon	States			Transitions		
	Simple	Moore (terminal)	Mealy	Simple	Moore	Mealy (terminal)
English	33 671	33 670 (5 192)	33 141	74 934	69 742	69 056 (14 693)
Esperanto	21 508	21 507 (3 325)	21 259	61 960	58 635	58 184 (21 406)
French	27 492	27 491 (3 610)	27 204	66 905	63 295	62 771 (14 095)
German	46 436	46 435 (4 027)	46 050	90 737	86 710	86 266 (9 960)
Polish	67 486	67 485 (10 615)	66 268	199 397	188 782	185 952 (53 197)
Russian	50 846	50 845 (8 742)	49 761	148 936	140 194	137 716 (47 625)
Spanish	34 778	34 777 (5 303)	34 321	111 345	106 042	105 170 (33 895)
ENABLE	54 336	54 335 (9 064)	53 767	132 515	123 451	122 645 (28 439)
DIMACS	442 986	442 985 (2 418)	441 856	706 781	704 363	703 207 (25 449)
Pathnames	3163 283	3163 282 (41 492)	3155 636	4192 514	4151 022	4143 076 (56 988)
Rev. pathnames	3356 173	3356 172 (9)	3356 167	4361 963	4361 954	4361 949 (9)
Random	328 915	328 914 (1 488)	328 625	428 766	427 278	426 989 (9 840)

state/transition types needed in a Moore or Mealy recognizer equals the number of possible *sets of classes* that can be exponential in the number of different terminal states needed in a simple recognizer. In these circumstances maintaining a simple recognizer is less cumbersome.

The reader, however, should not become convinced that acyclic automata, which are the subject of this paper, are a panacea for natural language processing. For an account of using automata with cycles to store the potentially infinite lexicons of agglutinative languages, see e.g. Reference [20].

IMPLEMENTATION DETAILS

Below we describe our implementation of Mealy recognizers. The implementation of the other varieties can be easily deduced from it. A C program can be downloaded from http://www-zo.iinf.polsl.gliwice.pl/∼sdeor/pub.htm. Another implementation by Jan Daciuk, featuring also construction from unsorted data, is available at http://www.eti.pg.gda.pl/∼jandac/fsa.html.

The recognizer is stored in an array, whose consecutive subarrays represent the out-transitions of recognizer states. Each transition occupies four bytes of memory divided into the following fields:

- 8 bits for the attribute character;
- 22 bits for the index of a destination state;
- 1 bit to distinguish terminal transitions from non-terminal ones;
- 1 bit to mark the last out-transition of the given state.

Such a representation can store recognizers that have up to 2^{22} transitions, which suffices for almost all lexicons used by us. For storing the reversed pathnames, the index field is 30 bits long, making the transitions occupy five bytes each. The element at index zero represents the state that has no out-transitions, but must occupy space, lest the next state should be mistaken for it. This otherwise unused location is a convenient place to store the index of the start state.

Our implementation of the construction algorithm remembers information on the states in 2^{18} hash buckets, implemented as linear lists. The elements of the lists are records with three fields:

- the index of the first transition of a state;
- the number of transitions in this state (dispensable at the cost of running time);

- a link to the next element.

Memory for these records is allocated in large chunks to decrease the time and space overhead of library allocation routines. On a typical 32-bit machine, the hash table occupies one megabyte for the list heads plus a megabyte for each roughly 87 thousand states.

On a UltraSPARC II clocked at 250 MHz, our program processes between 3 and 6 megabytes of strings in a second, and its speed is mostly I/O bound. The average speed of successful lookups, measured by searching all the strings in a lexicon, is between 2 and 4 megabytes per second when the transitions of each state are alphabetically ordered. Storing the states in static complete binary trees based on the alphabetical order decreases the searching time by at most 1/3, protracting the construction by about 1/5 for our data sets.

Another way to accelerate searching is to sort the transitions of each state with decreasing frequency of use [26]. It can be done with a training collection of strings, assuming that it is typical for future searches. Recognizing the training strings and counting how many times each transition is taken allows to reorganize the order of transitions in each state.

SQUEEZING A RECOGNIZER STILL MORE

Besides shortening the fields allocated for the attribute character or destination index, which would make the transitions occupy fractional parts of bytes, there are also other ad hoc ways to decrease somewhat the space occupied by a recognizer.

A binary recognizer that branches on bits of the ASCII code or a static Huffman code of the strings is several times bigger than a recognizer with a 256-character alphabet. Conversely, the way to reduce the size of recognizers, at least for some lexicons, is to substitute single symbols for frequent groups of characters. It is however obscure which groups should be replaced. Among several heuristics tried by us, no method to determine feasible substitutions produces consistent results for different lexicons. The following simple one decreases the size of recognizers for English and German by 10–12%: every pair of successive characters of each string, except the last pair, scores $+1$; the last pairs score -1; replace 100-odd pairs with the highest score. For other lexicons, though, it proves better to refrain from the replacement or to do it for only a few character pairs.

A more general way to save a little memory is to include some states in larger ones if the former are their subsets. It is practicable when the states are represented as lists, and needs only modifying the *make_state* function. Our first-fit algorithm for doing the inclusion slows down the construction at most twice in our tests, which seems an acceptable tradeoff for shrinking the recognizers by 4–5%.

To this end, we distinguish *fresh states*, whose transitions are alphabetically ordered, and *frozen states* that have been reordered to contain a smaller state at the end of the transition list.

While searching a hash bucket for a state identical to a given larval state, states of matching size are considered: fresh states are compared to it transition by transition; frozen states are compared to it by a simple algorithm, whose time complexity is quadratic in the number of transitions. For our data, frozen states are rare and rather short, so implementing more elaborate methods is not worth the effort.

If no state equal to the given one is found, we prepare a short list of fresh states that could embody it, using an additional hash table indexed by all the transitions created so far—a directory of their source states. All the transitions of the larval state are examined to find one that hashes to a bucket with the fewest entries. The fresh states from this bucket that are larger than the larval state are scanned in order of increasing size for the larval transitions (all the buckets are not kept sorted all the time, since only a fraction of them is ever searched). The scan is pretty fast, as the transitions of both fresh and larval states are alphabetically ordered. If some state contains all the needed transitions, they are moved to its end and considered a fresh state, and the larger state is frozen. Otherwise, a fresh state is created the usual way.

Table 3: Size in kilobytes of various data structures storing the sample lexicons.

Lexicon	Raw	Mealy recognizer	Path-compressed trie	Bonsai	C-trie	Ispell
English	688	270	467	645	1 070	729
Esperanto	11 205	227	3 841	4 001	11 462	837
French	2 418	245	1 245	1 571	3 397	1 248
German	2 661	337	1 028	1 398	3 327	1 048
Polish	16 647	726	8 467*	8 280	18 893	3 627
Russian	8 911	538	3 893	4 432	10 438	2 269
Spanish	7 187	411	3 190	4 085	7 910	2 399
ENABLE	1 709	479	981	1 369	1 779	·
DIMACS	6 831	2 747	3 507	6 853	5 371	·
Pathnames	202 414	16 184	42 272*	109 745	115 725	·
Rev. pathnames	202 414	21 299*	141 515*	623 741	596 557	·
Random	1 027	1 668	1 096	2 317	895	·

* with 30-bit index fields

COMPARING RECOGNIZERS WITH OTHER DATA STRUCTURES

Table 3 compares the size of automata and other data structures that store the sample lexicons. The columns contain size in kilobytes of the following data structures:

- Raw lexicon, i.e. strings separated with a single character.
- Mealy recognizer using four bytes per transition, without any additional squeezing.
- Path-compressed trie. Unlike in the trie from Figure 1, no string delimiters are used: the alphabet size is restricted to 2^7, and each character has a one-bit marker to determine if a string ends at it. This way multicharacter arcs need not be split at string ends. Arcs labelled with one character occupy four bytes; longer labels consist of a 7-bit length followed by 7-bit characters. Such arcs labelled with $n > 1$ characters occupy $4 + n$ bytes. For some lexicons, it was necessary to expand the index field of each arc by 8 bits, so that the arcs occupy $5 + [n > 0]n$ bytes.
- Bonsai [8], a hashing technique for storing tries that uses three bytes *per node* (we assume 80% occupancy of the hash table). String ends are marked in nodes, too. Bonsai is a dynamic data structure, only by 10% smaller than a static (non-path-compressed) trie that uses four bytes per arc and encodes string ends in a Moore fashion.
- C-trie, a trie compaction scheme proposed by Maly [16].
- Ispell lexicon file. This file, tailored to spell checking, contains a hash table, a list of base words and allowed affixes, and a few kilobytes of other information.

From these structures, the recognizers occupy the least memory for almost all data sets. The exception of random data, where common prefixes and suffixes are rare, shows that they work best for lexicons that exhibit similarities, such as natural languages. The lookup speed in all the trie-derived data structures is comparable. In Ispell hash tables it is slower and depends more on the properties of a language.

CONCLUSIONS

This paper presents an algorithm to construct minimal acyclic deterministic finite automata recognizing a given set of strings, whose particular variants were described formerly by Daciuk, Mihov, Watson, and Watson. Speed of operation and frugal use of memory distinguish it from universal minimization algorithms applied to acyclic automata.

The algorithm is shown in three variations for automata that mark string ends in different ways. The automata using terminal transitions occupy the least memory, the automata using string delimiters suit some applications better. Particular techniques to decrease the size of automata and accelerate searching them are also discussed.

A comparison for real-world data sets shows that the automata, while retaining attractive properties of tries, usually occupy several times less memory than alternative data structures.

Acknowledgements

We are grateful to Zbigniew J. Czech, Bruce W. Watson, and the anonymous referees for their critical comments on this paper. Free lexicons that we used for benchmarks are maintained by Geoff Kuenning, Sergei B. Pokrovsky, Martin Boyer, Heinz Knutzen, Piotr Gackiewicz, Włodzimierz Macewicz, Mirosław Prywata, Alexander Lebedev, Santiago Rodríguez, Jesús Carretero, and Mendel Cooper. The work was partially financed by a State Committee for Scientific Research grant no. 8T11C 007 17.

References

[1] Aho A, Sethi R, Ullman JD. *Compilers: Principles, Techniques and Tools.* Addison–Wesley, 1985.

[2] Al-Suwaiyel M, Horowitz E. Algorithms for trie compaction. *ACM Transactions on Database Systems* 1984; **9**(2):243–263.

[3] Appel AW, Jacobson GJ. The world's fastest Scrabble program. *Communications of the ACM* 1988; **31**(5):572–578, 585.

[4] Blumer A, Blumer J, Haussler D, McConnell R, Ehrenfeucht A. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM* 1987; **34**(3):578–595.

[5] Czech ZJ, Havas G, Majewski BS. Perfect hashing. *Theoretical Computer Science* 1997; **182**(1–2):1–143.

[6] Daciuk J. *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing.* PhD thesis, Politechnika Gdańska, 1998. http://www.eti.pg.gda.pl/~jandac/thesis.ps.gz

[7] Daciuk J, Mihov S, Watson BW, Watson RE. Incremental construction of minimal acyclic finite-state automata. *Computational Linguistics* 2000; **26**(1):3–16.

[8] Darragh JJ, Cleary JG, Witten IH. Bonsai: a compact representation of trees. *Software—Practice and Experience* 1993; **23**(3):277–291.

[9] Gordon SA. A faster Scrabble move generation algorithm. *Software—Practice and Experience* 1994; **24**(2):219–232.

[10] Hopcroft JE, Ullman JD. *Introduction to Automata Theory, Languages, and Computation.* Addison–Wesley, 1979.

[11] Knuth DE. *The Art of Computer Programming*, vol. 3: *Sorting and Searching.* Addison–Wesley, 1998.

[12] Knuth DE. *Computers & Typesetting*, vol. C: *TEX: The Program*. Addison–Wesley, 1986.

[13] Kuenning GH. *International Ispell*. http://ficus-www.cs.ucla.edu/ficus-members/geoff/ispell.html

[14] Lacouture R, De Mori R. Lexical tree compression. In *Proceedings of the Second European Conference on Speech Communication and Technology*. Genoa, 1991; 581–584.

[15] Lucchesi C, Kowaltowski T. Applications of finite automata representing large vocabularies. *Software—Practice and Experience* 1993; **23**(1):15–30.

[16] Maly K. Compressed tries. *Communications of the ACM* 1976; **19**(7):409–415.

[17] McIlroy MD. Development of a spelling list. *IEEE Transactions on Communications* 1982; **COM-30**(1):91–99.

[18] Mihov S. Direct building of minimal automaton for given list. *Annuaire de l'Université de Sofia* 1997; **91**(1):33–40. http://www.lml.bas.bg/~stoyan/tagie.ps.gz

[19] Mihov S. Direct construction of minimal acyclic finite states automata. *Annuaire de l'Université de Sofia* 1998; **92**. http://www.lml.bas.bg/~stoyan/anu2.ps

[20] Oflazer K. Error-tolerant finite state recognition with applications to morphological analysis and spelling correction. *Computational Linguistics* 1996; **22**(1):73–89.

[21] Park K-H, Aoe J-I, Morimoto K, Shishibori M. An algorithm for dynamic processing of DAWG's. *International Journal of Computational Mathematics* 1994; **54**(3–4):155–173.

[22] Revuz D. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science* 1992; **92**(1):181–189.

[23] Revuz D. *Dictionnaires et lexiques—méthodes et algorithmes*. PhD thesis, Université Paris VII, 1991.
http://www-igm.univ-mlv.fr/~dr/thdr/

[24] Sgarbas KN, Fakotakis ND, Kokkinakis GK. Two algorithms for incremental construction of directed acyclic word graphs. *International Journal on Artificial Intelligence Tools* 1995; **4**(3):369–381.

[25] Watson BW. *Taxonomies and Toolkits of Regular Language Algorithms*. PhD thesis, Technische Universiteit Eindhoven, 1995.

[26] Watson BW. Practical optimizations for automata. *Lecture Notes in Computer Science* 1997; **1436**:232–240.

[27] Watson BW. A fast new semi-incremental algorithm for construction of minimal acyclic DFAs. *Lecture Notes in Computer Science* 1998; **1660**:91-98.

[28] Watson BW. Private communication, November 2000.