Ziv Lempel Compression of Huge Natural Language Data Tries Using Suffix Arrays

Strahil Ristov, Eric Laporte

Ziv Lempel compression of huge natural language data tries using suffix arrays

Strahil Ristov [1] **Eric Laporte** [2]

[1] **Ruđer Bošković Institute**
Laboratory for stohastic signals and processes research
Zagreb, Croatia
ristov@rudjer.irb.hr
[2] **Institut Gaspard Monge**
Centre d'études et de recherches en
informatique linguistique
Université de Marne-la-Vallée, France
laporte@bastille.univ-mlv.fr

Abstract. We present a very efficient, in terms of space and access speed, data structure for storing huge natural language data sets. The structure is described as LZ (Ziv Lempel) compressed linked list trie and is a step further beyond directed acyclic word graph in automata compression. We are using the structure to store DELAF, a huge French lexicon with syntactical, grammatical and lexical information associated with each word. The compressed structure can be produced in O(N) time using suffix trees for finding repetitions in trie, but for large data sets space requirements are more prohibitive than time so suffix arrays are used instead, with compression time complexity O(N log N) for all but for the largest data sets.

1 Introduction

Natural language processing has been existing as a field since the origin of computer science. However, the interest for natural language processing increased recently due to the present extension of Internet communication, and to the fact that nearly all texts produced today are stored on, or transmitted through a computer medium at least once during their lifetime. In this context, the processing of large, unrestricted texts written in various languages usually requires basic knowledge about words of these languages. These basic data are stored into large data sets called lexicons or electronic dictionaries, in such a form that they can be exploited by computer applications like spelling checkers, spelling advisers, typesetters, indexers, compressors, speech synthesizers and others. The use of large-coverage lexicons for natural language processing has decisive advantages: Precision and accuracy: the lexicon contains all the words that were explicitly included and only them, which is not the case with recognizers like spell [5]. Predictability: the behavior of a lexicon-based application can be deduced from the explicit list of words in the lexicon. In this

context, the storage and lookup of large-coverage dictionaries can be costly. Therefore, time and space efficiency is crucial issue.

Trie data structure is a natural choice when it comes to storing and searching over sets of strings or words. In the contemporary usage of the term, a trie for a set of words is a tree in which each transition represents one symbol (or a letter in a word), and nodes represent a word or a part of a word that is spelled by traversal from the root to the given node. The identical prefixes of different words are therefore represented with the same node and space is saved where identical prefixes abound in a set of words - a situation likely to occur with natural language data. The access speed is high, successful look up is performed in time proportional to the length of word since it takes only as many comparisons as there are symbols in the word. The unsuccessful search is stopped as soon as there is no letter in the trie that continues the word at a given point, so it is even faster.

When sets of strings are huge a simple trie can grow to such proportions that its size becomes a restrictive factor in applications. A huge data structure that can't fit into main memory means slower searching on disk, furthermore if the structure is small enough to fit into cache memory the search speed is increased. Numerous researchers did a lot of work on compacting tries, reducing the size and increasing the search speed. As there are many possible uses of a trie, most of the compaction methods are optimized according to specific application requirements. When data must be handled dynamically (databases, compilers) trie has to support insertion and deletion operations as well as a simple lookup; the best results in trie compaction, however, are achieved with static data. Few examples of work on dynamic trie compaction are [3], [7], [8], [15]. Static tries are used successfully in a number of important applications (natural language processing, network routing, data mining) and the efforts in static trie compression are both numerous and justified. Although researchers usually try to establish as good trade-off between speed and size as possible, in most of the work emphasis is on one of the two. Two examples of work where the speed is of main concern are [2] where search speed is increased by reducing the number of levels in a binary trie and [1] where trie data structures are constructed in such manner that they accord well with computer memory architecture. When the size of the structure is of primary concern the work is usually focused on automata compression. With natural language data significant savings in memory space can be obtained if the dictionary is stored in a directed acyclic word graph (DAWG), a form of a minimal deterministic automaton, where common suffixes are shared [4], [12], [13], [17].

Majority of European languages belong to a family of languages where (i) most of the words belong to a set of several morphologically close words (inflectional languages), and (ii) the differences between two such morphologically close words is usually a suffix substitution (suffixal inflection). That accounts for good results with automata minimization, on the average a substantial portion of a word is overlapped with other words' prefixes and suffixes. However, this works well only for simple word lists used mainly in spelling checkers, for most other applications (dictionaries, lexicons, translators) some additional data (lexical tags, index pointers) has to be attached to the word

sharply reducing the overlapping of the suffixes. The additional data can be efficiently incorporated in the trie by more complex implementation [16] or by using the hashing transducers. The hashing transducer of a finite set of words was discovered and described independently in [13] and [17]. This scheme implements a one-to-one correspondence between the set of N words and the set of integers from 1 to N, the words being taken in alphabetical order. The user can obtain the number from the word and the word from the number in linear time in the length of the word, independently of the size of the lexicon therefore producing a perfect hashing. The transducer has the same states and the same transitions as the minimal automaton, but an integer is associated to each transition. The number of a word is the sum of the integers on the path that recognizes the word. Once the number of a word is known, a table is looked up in order to obtain the data associated with the word.

In this paper we investigate a new method of static trie compaction that reduces the size beyond that of minimal finite automaton and allows incorporating the additional data in the trie itself. This involves coding the automaton so that not only common prefixes or suffixes are shared, but also the internal patterns. The procedure is best described as a generic Ziv Lempel compression of a linked list trie. Final compressed structure is formally more complex and has less states than minimal finite automata used in [4] and [13]. Particularly attractive feature is a high repetition rate of structural units in compressed structure that enables space efficient coding of the nodes. The idea has been informally introduced in [18] and [19]. Here we shall describe the method in more detail and demonstrate how it performs when used for storing DELAF, a huge lexicon of French words. We also present some compaction results for various natural language data sets. For the sets on which previous work has been reported in the literature our results are significantly better.

In section 2 we present our method and introduce notation we use throughout the article. Two essentially similar algorithms for compression are described in section 3, the first one is simpler and slower, the second one much faster but requires more space. We also explain some heuristic for simplification of the algorithms and propose a related problem as an open problem in theory of NP completeness. In section 4 we describe experimental data sets, among them a huge French lexicon, and present compression results. Conclusion is in section 5.

2 Overview of the Linked List Trie LZ Compression

A trie T is a finite automaton and is as such defined with the quintuple T = {Q, A, q_0, δ, F}, where Q is a finite set of states, A is an alphabet of input symbols, $q_0 \in Q$ is the initial state, δ is a transition function from Q x A to Q and $F \subseteq Q$ is the set of accepting or final states. When trie T is produced from a set of words W, then W is the language recognized by T.

Natural language data usually produce very sparse tries that lend themselves to various possibilities for space reduction with retained high access speed. Sparseness of a tree is a strong indication for employing the linked list data

structure in representation of the nodes. When linked list is used it is convenient to associate symbols of alphabet with the levels rather than with the transitions in the trie. In this case levels are represented with lists of structural units where four pieces of information (Fig. 1a) are assigned to each unit:

1. **a symbol (letter) $a \in$ A;**
2. **a binary flag *f* indicating whether a word ends at this point (corresponding to a final state);**
3. **a binary flag *c* indicating whether there is a continuation of valid sequence of symbols past the current unit to the next level below;**
4. **a pointer *l* to the next unit at the same level (if null, there are no more elements on the current level); if we use addressing in number of units, the size bound for *l* is the number of units in T.**

A linked list trie is then represented with a sequence or a string of units. Now, the units themselves can be regarded as symbols that make up a new alphabet U and the implemented trie structure can be defined as a string.

DEFINITION: Linked list trie LLT is a string of symbols *u* from alphabet U. If we denote by N the number of structural units in LLT then:

$\mathrm{LLT} = u_0 u_1 u_2 \ldots u_N \mid u_i \in \mathrm{U}, \mathrm{N} = |\mathrm{LLT}|$ where

$u_i = a_i f_i c_i l_i \mid a_i \in \mathrm{A},\ f_i \in \{0, 1\}, c_i \in \{0, 1\},\ \ 0 \leq l_i \leq \mathrm{N}$

To illustrate this, in Fig. 1b units of the trie from Fig. 1a are replaced with a new set of symbols yielding a string representation of LLT. Of course, when each of their parts are identical, two units are identical too and consequently represented with the same symbol.

As on any string, some compression procedure can be attempted now on LLT. Particularly natural approach is to use LZ paradigm of replacing repeated substrings with pointers to their first occurrences in the string [23]. The general condition for compression is that the size of pointer must be less than the size of the replaced substring. We used the constant and equal size units for representation of the elements of U and the pointers so that compression is achieved whenever repeated substring is of size 2 or more elements. In Fig. 1c repeated substrings are replaced with information in parenthesis about the position of the first occurrence of repeated substring and it's size. The first number designates the position in (compressed) string and second the length of replaced substring. Note that the first occurrence of a substring can include a pointer to the previous first occurrence of a shorter substring.

DEFINITION: Let ls_i be the length of i-th substituted substring in LLT and K be the number of substitutions. Then, reduction in space $\mathrm{R} = \Sigma\ (\mathrm{ls}_i - 1)$, for i = 1 - K. Let LLTC denote the compressed linked list trie such as that of Fig. 1c. The size N_C of compressed structure is then $\mathrm{N}_c = |\mathrm{LLTC}| = \mathrm{N} - \mathrm{R}$, and the compression ratio $\mathrm{C} = 1 - \mathrm{N}_c/\mathrm{N}$.

All size values are given in number of structural units. For the example of Fig. 1c, R = 9 and C = 1 - 11/20 = 45%.

The sequence in Fig. 1c is a simplified representation of a compressed trie structure; look up for the input is not performed sequentially as it may seem suggested by the Figs. 1b and 1c, but still by following trie links. Only now when, in reading the structure, at the position P_1 a pointer unit (P_0, ls_1) is encountered, reading procedure jumps to the position P_0, and after ls_1 units read, jumps back to the position P_1 + 1.

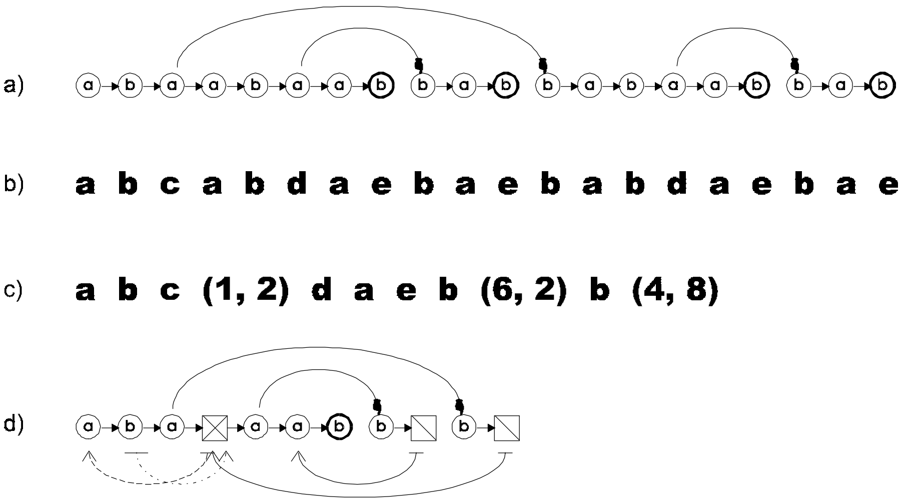
a)
a b a a b a a b b a b b a b a a b b a b
b)
a b c a b d a e b a e b a b d a e b a e
c)
a b c (1, 2) d a e b (6, 2) b (4, 8)
d)
a b a a a b b b

Fig. 1. a) A trie of four words {abaabaab, abaabbab, abbabaab, abbabbab} is presented in a graphical arrangement that points out its sequential features. Final states are indicated by thick circles; horizontal arrows represent *c* flags; inflected arrows represent *l* pointers. Structure is traversed by following the arrows and comparing the current input symbol with one in the trie, if symbols don't match and there is no *l* pointer from the current unit then input is rejected. The input sequence is accepted if it leads to a final state. b) LLT represented with new set of symbols; identical units are replaced with the same symbol. c) Compressed representation of LLT string. The first number in parenthesis is the position of the first occurrence of repeated/replaced substring, the second number is the substring's length. d) Implementation of compressed structure includes two types of pointers: ⧅ signs indicate pointers that replace whole branches and ⊠ sign stands for pointer that replaces only a portion of a branch and carries the information about its length (2 in this case). Inflected arrows below indicate the paths the reading procedure must follow in the structure. Full lines indicate one-way directions, dashed lines indicate directions implied by ⊠ pointer.

The actual implementation of LLT compression is more complex than in straightforward application of a LZ procedure on a string in Fig. 1c where there's no difference in treatment of repeated substrings. The underlying structure of LLT is that of a tree and this divides repeated substrings of LLT into two categories depending on whether the repeated substring represents a complete branch of a tree or just a portion of a branch. Only for this latter case should the pointers carry the information about the number of replaced units;

when the whole branch is replaced, every possible continuation of the current input is contained in the first occurrence of the substring and there is no need for coming back to the original position of a pointer. Second and third pointers of Fig. 1c replace whole branches of the trie and the first one substitutes only a part of a branch. This LLTC sequence with two types of pointers then might look like this: abc(1,2)daeb(6,_)b(4,_) where "_" indicates that there is no possible need for coming back. The Fig. 1d shows how actually the structure of Fig 1a is compressed with two different types of pointers.

DEFINITION: Let's call *one-way* pointers pointers that replace whole branches and *two-way* pointers those that replace only parts of branches. Let's say that a substring s = $u_1 u_2 ... u_{ls}$ of LLT is *closed* if no unit $u_i \in$ s contains a $l_i \in u_i$ pointer that points outside s, and there is no continuation to the next level from the last unit of s. That is:

**s is closed if $\forall\ l_i \in u_i \in$ s | value (l_i) ≤ position (u_{ls}) and $c_{ls} = 0$.
Otherwise let's call s *open*. One-way pointers replace closed repeated substrings, two-way pointers replace open repeated substrings.**

THEOREM: Replacing every closed repeated substring of LLT with one-way pointers produces DAWG for a given set of words W.

Proof: DAWG for a set of words W is the minimal finite automaton recognizing all the words in W. Minimization is obtained by merging all the equivalent states of the automaton.

If two states are reached by sequences s_1 and s_2 they are equivalent if for every sequence z holds that if s_1z is in W then s_2z is also, and if s_1z is not in W neither is s_2z. Since substrings of LLT replaced with one-way pointers are identical it is obvious that they carry identical partial transition function, and since they are closed there exist no other unknown suffixes so the repeated states are indeed equivalent.

The additional compression, above that of automata minimization, is achieved with introduction of two-way pointers capable of replacing open substrings of LLT. It is worth noting that the formal complexity of compressed structure is then higher than that of finite automaton. States replaced by two-way pointers are not equivalent in the finite automata sense and some conditional branching is introduced in the procedure of reading the structure. For example, after reading b in the second position on Fig. 1d further direction depends on whether this is the first time read or the read directed by the pointer at the fourth position. This type of decision is beyond the power of finite automata.

3 Algorithms

We first present a simple quadratic algorithm for producing LLT from W and then replacing repetitions with pointers. Denote with $s_i \in$ LLT a substring of

units starting at the position i in LLT. Let E be a relation of substring prefix equality on LLT such that s_iEs_j means that there are at least two first units of s_i and s_j that are equal. That is: s_iEs_j => $u_i...u_k = u_j...u_k$ and $k \geq 2$. Let R be the relation of substring substitutability where s_iRs_j means that s_j can be replaced with the pointer to s_i. For the algorithmic complexity reasons R covers smaller class of LLT substrings than E; this will be explained a bit latter. The algorithm is then as follows:

ALGORITHM A:

```
sort W
build LLT(W)
for i = 1 to N – 3
    for j = i+2 to N – 1
        if $s_iEs_j$
            if $s_iRs_j$
                check whether substitutable substrings are open or closed
                replace $s_j$ with the appropriate pointer
end
```

Building of LLT(W) is a straightforward procedure of building a trie that can be done in O(N) time and will not be explained here. Initial sorting of W is the simplest way of preventing following situations to occur: Let M be the number of words in W, $w_m \mid m < M$ denote m-th word in W, and LLT_m a linked list trie with m words built into it. If $w_{(m+1)}$ has a prefix w_k that is also a word of W such that $k < m$ and w_m is not a prefix of $w_{(m+1)}$ then there is no place for the suffix of $w_{(m+1)}$ that is the difference between $w_{(m+1)}$ and w_k. This suffix should find its place right after w_k, but since there is already at least one word not prefix of $w_{(m+1)}$ in the structure, this place is occupied. Situation like this would require usage of additional pointers in the construction of the trie and it is more economical instead to arrange the input order in a way to avoid this. The simplest way to do this is to sort W before building LLT(W), then if words exist in W that are prefixes of other words they are all grouped together and any existing prefix of $w_{(m+1)}$ is at the end of LLT_m.

The central part of presented algorithm has clear quadratic time complexity. Double loop of comparing each position in LLT with every other to check whether they are the starting positions of equal substrings takes $N^2/2$ iterations. (The inner loop is only shifted to the right by two – the minimum size of substitutable substrings.) The procedures of checking whether repeated substrings are open or closed and replacing them with pointers are done only once for each replaced substring so they add to the overall complexity only a linear factor proportional to R. The average input sorting procedure is done in O(M log M) time and the total time complexity for producing LLTC from W is then $O(M \log M + N + N^2 + R)$ with $O(N^2)$ being by far the most important bound. In practice this simple procedure is fast enough for smaller data sets such are smaller simple word lists with high prefix repetition rate that produce smaller tries. Unfortunately, for bigger sets of entries that do not share too many

common prefixes, and therefore produce huge tries, the exhaustive quadratic procedure is not feasible.

3.1 Speed up via Suffix Matching

Speed up is possible and in fact a linear time bound can be achieved using suffix tree for finding repetitions in LLT. The idea of assisting LZ compression with suffix tree search has firstly been presented in [21]. A suffix tree of all suffixes in LLT can be built in O(N) time, all the repetitions in LLT are then associated with the nodes in the suffix tree and easily found in linear time [11]. The problem with building suffix trees is that they require to much space when alphabet is large as is the alphabet of all different units of LLT, and for this case a better approach is to use suffix arrays [14]. A suffix array for LLT is an array of starting positions in LLT of sorted suffixes of LLT. Sorting is on the average done in O(N log N) time and then all repeated substrings are grouped together in suffix array.

Now the problem rests of finding the best candidates for replacement with pointers among the substrings grouped together. The simplest way to do this is to delimit groups of suffixes in suffix table that have at least two first elements identical and then to perform quadratic search only on elements in the group. These groups should be sorted according to the suffix starting position in LTT so that search and replace procedure can be done in consecutive order from the beginning of the structure. This is important because it avoids considerable expense of keeping track of all the changes in the structure that can interfere with incoming replacements. Overall, this is much faster way to find possible candidates for the substitution with pointers than the exhaustive quadratic search of Algorithm A. The procedure is then:

ALGORITHM B:

```
1:  sort W
2:  build LLT(W)
3:  build suffix_array(LLT(W))
4:  define partitions of suffix_array(LLT(W)) that comprise two or more
    entries with identical first two units
5:  sort the suffixes in partitions according to their position in LLT(W)
6:  from the first to the last element in partitions compare each element to
    every other from the same group | check whether substitutable
    substrings are open or closed | replace substitutable substrings with the
    appropriate pointers
```

Time complexity of comparing substrings at suffixes' starting positions to possible candidates for replacement within the groups is still quadratic but with much smaller base. If there are G different groups of suffixes with identical beginnings in suffix_array(LLT(W)) and SG_i, $i = 1 – G$ is the number of elements in i-th group, the time complexity of step 6 is $O(\Sigma \, SG_i^2)$. For real data the size of

any group is much smaller than N so this improves strongly on time requirements of Algorithm A. The price is paid in space used for suffix array and tables needed for storing and searching groups. There is also the sorting of the groups procedure of step 5 that requires $O(\Sigma\ SG_i \log SG_i)$ time so the total time complexity of Algorithm B is $O(M \log M + N + N \log N + \Sigma\ SG_i \log SG_i + \Sigma\ SG_i^2 + R)$. When running the experiments it is apparent that steps 3 to 5 consume most of the running time of Algorithm B for values of N up to a million. Only for tries with more units quadratic time complexity of step 6 becomes increasingly important. However, these are the values of N where the difference in complexity of Algorithms A and B matters the most. For the biggest tries that we experimented with (N = 16 million) estimated run time of Algorithm A is 250 times longer.

If some additional structures are used to mark already replaced substrings then $\Sigma\ SG_i^2$ factor can be improved to $\Sigma\ RG_i^2$ where RG_i is the number of substitutions actually performed in i-th group. This has not been justified experimentally since Algorithm B already uses considerably more space than Algorithm A and for large N values the size of additional structures may become a restricting factor.

3.2 Bounds for Compression of LLTC

LLTC produced by Algorithms A or B is not necessarily the smallest possible structure of this sort recognizing W. There exist one obvious structural limitation for compression – a constant size of unit, and some algorithmic limitations that are imposed for the sake of the algorithmic simplicity.

Size of Structural Units. If the size of structural unit is kept constant, which immensely simplifies and speeds up the look up procedure, then the bound for the size of each unit is the size of units holding the largest numerical information. There are two types of structural units in LLTC: the *symbol* units, same as those of LLT that carry the symbol code *a*, *f* and *c* flags and the *l* pointer, and the *pointer* units that are either one- or two–way pointers replacing repeated substrings in LLTC. The size limit for symbol unit in bits is given by $\lceil \log A \rceil + 1 + 1 + \lceil \log N_c \rceil$ and this limit is forced onto pointer units too. Pointer units carry information about the address of the first occurrence of substituted substring, about its length (if two-way) and some information that distinguishes them from symbol units. In symbol units either *f* or *c* flag or both must be 1 (true) because the word can only end with the current symbol or be continued to the next one. Therefore combination of two zeros for *f* and *c* flags is impossible in symbol units and this is used as an indication that the current unit is a pointer. The bound for the size of the address of the first occurrence of replaced substring is $\lceil \log N_c \rceil$ again, so this leaves $\lceil \log A \rceil$ bits in pointer units for storing the length of replaced substring for two-way pointers. This was enough for every data set we have experimented with so far. LLTC normally supports embedded pointers, i.e. a pointer can point to a sequence of units that contains another pointer, and this can have many levels. For reasons of space economy we are storing in two-way

pointers only the number of units that has to be followed on the first level which is usually considerably smaller than the full length of the replaced substring. Apart from this little trick there is another reason why $\lceil \log A \rceil$ bits are enough for two-way pointer information - the longest substituted substrings are usually closed and are therefore replaced with one-way pointers. The problem with constant size units is in that when N_c is big, most of the l pointers are much smaller in value and a considerable amount of space is wasted. If this becomes critical it is always possible to use variable size coding of units or, which should be the best solution for the overall reduction of redundancy in LLTC, to use additional table for minimal size coding of units described latter in section 3.3.

Algorithmic Complexity Constraints on Possible Substring Substitution. There are three algorithmic limitations to compression of LLTC arising from its underlying tree structure and they are defined with the following rules:

Rule 1. If the repeated substrings overlap, then shorten them so that they don't.

Rule 2. If $s_i = u_i...u_k...u_{i+ls}$ is a repeated substring and $l_k \in u_k$ has value(l_k) > i+ls+1 then shorten s_i to $s_{iR} = u_i...u_{k-1}$.

Rule 3. If $s_i = u_i...u_k...u_{i+ls}$ is a repeated substring and there exists $l_h \in u_h \mid h < i$, such that value(l_h) = k | $i+1 \leq k \leq i+ls$, then shorten s_i to $s_{iR} = u_i...u_{k-1}$.

The above three rules account for the aforementioned difference between classes of equal and substitutable substrings of LLT. If these rules are not observed situations would be occurring that would require complicated procedures to solve while at the same time not improving much on the compression. If overlapping of replaced substrings is allowed it would take great pains to avoid never-ending loops and the savings in space would be only one unit per occurrence. (If overlapping is allowed *pattern.pattern.pattern* can be replaced with *pattern.pointer*, and if not, with *pattern.pointer.pointer* with only the cost of one pointer increase in space.) Hence the Rule 1.

Rule 2 prevents the substitution of a substring s_i that contains a l pointer pointing out of s_i by more than one. This is necessary because it is possible that substring of LLT between the end of s_i and the position the l pointer points to can latter be replaced with another pointer unit and then the value of l won't be correct anymore. To account for that a complicated and time costly checking procedure should be employed and the savings would be at most two units per occurrence. (If k = i + ls then only unit u_{i+ls} is not included in the substituted substring, if k = i + ls - 1 then the loss is two units $u_{i+ls-1}u_{i+ls}$, and if k < i + ls - 1 then the part of s_i behind u_k is a new repeated substring and can be replaced with a new pointer so the loss is again only two units.)

Rule 3 for the similar reasons shortens s_i up to the position pointed to by some l pointer positioned before s_i. If s_i is replaced then this l value wouldn't be correct anymore and the necessary checking would be unjustifiably costly. Analogously to Rule 2 the loss in compression is at most two units per occurrence.

It should be noted that situations where Rules 1, 2 and 3 come to effect occur seldom enough in natural language data that we have been experimenting with so far. Apparently, application of these rules worsens the compression by not more than 3%.

Input Ordering Problem. Apart from that, there exists a serious algorithmic impediment in optimization of LLTC compression introduced by the order of input words when building LLT. Fortunately, this has only a theoretical importance and carries a little weight in practice. Let us consider a special case where W can be divided into a set of distinct partitions W_i, $W_i \in W$, such that every word in W_i has the same length L_i and differs from other words in W_i only in the last letter. Let P_i denote a sequence of units in LLT that represents a common prefix of words in W_i, then length(P_i) = L_i - 1. Let u_{Lik} denote the unit representing the last letter in word $w_{ik} \in W_i$ where k = 1 – Ki, Ki = $|W_i|$. Suppose that no word in W is a prefix of another word in W, then when a linked list trie is built each subset W_i produces a LLT branch of type $P_i u_{Li1} u_{Li2} u_{Li3} ... u_{LiKi}$. Units corresponding to the last letters in words are connected with *l* pointers of value one and are identical in every aspect but for the symbol content throughout all the subsets W_i. Ordering of the sequence of u_{Li} units has no bearing on the content of LLT, it is determined by the ordering of input words which can be arbitrary since no word of W is a prefix of another word in W. Now, the problem is how to order sequences of u_{Li} units in such a way as to obtain the highest possible compression achieved by replacing substitutable substrings in LLT with pointers. We haven't been able to find an efficient solution for this problem and we suspect it is NP-hard. We haven't been able to prove that neither so we propose this as an open problem in theory of NP completeness. Reduced for the simplicity it can be stated as:

INSTANCE: Finite set of variables V and a collection T of triples (v_j, v_k, v_l) from V. For each triple holds a statement

$$v_j \angle v_k \quad \text{and} \quad v_j \angle v_l$$

where $\angle$ stands for any transitive, asymmetrical and irreflexive relation such as 'smaller than', 'bigger than', 'has lower/higher rank' etc.

QUESTION: Is there an assignment of values to variables in V such that the number of statements (or triples) that are satisfied is not less than a given integer $I \leq |T|$?

The order of input words may therefore have influence on how well the linked list trie is compressed. With actual natural language data this is not an important factor, the lexicographical sort of input results in highly repetitious LLT structure and this normally solves the problem well enough. When we investigated possible variations between worst and best case orderings on actual data the difference in size of compressed structures could never be above 2%.

3.3 Minimal Size Unit Coding with Table Lookup

An interesting and exploitable feature of LLTC is a high repetition rate of identical units throughout the structure. Apparently, lexicographic sort of input records combined with employed linked list representation produces a high level of structural unit repetitions in both LLT and LLTC. This effect gets more

pronounced with larger data sets. For example, in a compressed trie of over 2 million elements only about 200,000 units are different. A simple and very effective coding of the units is therefore possible for reducing redundancy in the structure. If all the different units are stored separately in a table of size ND × (unit size), where ND is the number of different units, then LLTC can be represented with an array of N pointers of size $\lceil \log \mathrm{ND} \rceil$ bits. On top of this, up to two bits per table unit can be saved by using their position in table instead of flags. In most cases table coding leads to important savings in space and the time needed for table lookup only about halves the search speed, as indicated by our experiments.

The compressed structures produced with Algorithms A or B are very compact and fast to search. Typical access speed for LLTC is measured in tens of thousands of found words per second. This is fast enough for any real time application, even for those that rely on an exhaustive search in space of similar words. In the following section we describe some actual data sets and present results of compaction experiments.

4 Data Sets and Experimental Results

4.1 Natural Language Lexicon

A simple spell-checker needs only to recognize whether a word belongs in the vocabulary of the language or not. In that case, the states of the automaton recognizing a word set are classified as final or non-final. For most other applications, correct words need to be assigned a lexical tag with a grammatical content: part of speech (noun, verb...), inflectional features (plural, 3rd person...), lemma (e.g. the infinitive for a verb). For instance, *woods* should be assigned a tag like *wood*.N:p (i.e. the noun wood in the plural). A minimal automaton can still represent a dictionary that assigns tags to words. Two methods are used to allow for tags in the dictionary. In the first [17], [20], tags are associated to states; the automaton has multiple finalities, i.e. the number of finalities is not necessarily 2 (final/non-final) but the number of tags. In the second method [12], tags are considered as parts of dictionary items. In both cases, minimization is still possible and time efficiency is preserved, but the minimization is less efficient in space, since common suffixes are no longer shared when the words have different tags (e.g. *ds* in the noun *woods* and in the verb *adds*).

When the linguistic information in the tags is limited to basic grammatical information, the number of possible different tags can remain small and these solutions are still optimal. The limit is reached when more elaborate information is included into the tags, namely syntactic information (number of essential complements of verbs, prepositions used with them, distribution of subjects and complements). When this information is provided systematically, the number of

different tags comes close to the number of words, and beyond because this level of description requires more sense distinctions [9]. Consequently, the minimal automaton grows nearly as large as the trie. However, the variety of labels used in tags is more limited and there exists a substantial amount of substring repetition in lexical entries. For this reason LLTC structure seems like a natural choice for storing lexicons.

We used LLTC for compressing a comprehensive dictionary of French, the DELAF [6]. This dictionary lists 600,000 inflected forms of simple words. It is used by the INTEX system of lexical analysis of natural-language texts [22]. Linguistic information are attached to each form: parts of speech (noun, verb...); inflectional features (gender, tense...); lemma (e.g. the infinitive in the case of a verbal form); syntactic information about verbs. In case of ambiguities, appropriate sense distinctions are made. The syntactic information attached to verbal forms is derived from the lexicon-grammar of French, a systematic inventory of formal syntactic constraints: number of essential complements, prepositions used with them, distribution of subjects and complements etc. [10]. The size of DELAF in text format is 21 Mbytes and a typical example of three entries in DELAF is presented in Fig. 2.

```
abandon,.N:ms

abandonna,abandonner.V+t+32CL+32H+36R+38LR+38L1+6+9:IPA3s/abandonner.V+
{s'~}+i+7:IPA3s/abandonner.V+i+31H+35R:IPA3s

abandonnai,abandonner.V+t+32CL+32H+36R+38LR+38L1+6+9:IPA1s/abandonner.V
+{s'~}+i+7:IPA1s/abandonner.V+i+31H+35R:IPA1s
```

Fig. 2. Three entries in DELAF lexicon of French words with attached grammatical, syntactical and lexical data.

Three things are obvious from this example: first, the amount of repeated substrings is high; second, a simple DAWG would be of little use since the endings of entries are highly diversified (i.e. there are not too many equivalent states in finite automaton produced from DELAF); and third, a trie produced from entries such as those on Fig 2 will be huge. The first two facts speak in favor of trying to store DELAF in LLTC, but the third presents a problem. A huge LLT means a huge N and the quadratic part of compression algorithm becomes important. In fact, with Algorithm B the compression time for LLT(DELAF) was 5.5 hours on a 333 MHz PC running Linux. In Table 1 we present all the relevant numbers for experiments with DELAF and other data sets.

The compressed size with table unit coding is 5.5 Mbytes. This is a considerable improvement over currently used format with tags stored separately that is over twice that size. Reduction in size can be important in integrated applications where lexicon is only a part of the system (computer-aided translation, natural language access to databases, information retrieval). The five and half hour compression time is acceptable for this instance because it is unlikely that data sets of this type will be updated on the run. The search speed is high enough for every possible application.

Table 1. Experimental results for various natural language data sets

data set	No. of entries	ASCII size (K)	No. of nodes in trie	No. of nodes in compressed trie	LLTC size with rounded nodes (node size)	table coded minimal LLTC size	compression time (seconds)	search speed (words per second)	
								standard	table coded
DELAF	609,454	21,430	16 M	2.14 M	8.4 M (4)	5.5 M	5.5 hours	30 K	15 K
DELAF word forms	609,454	6,942	1.2 M	130,100	385 K (3)	292 K	164	70 K	35 K
Calgary book1 7-tuples	768,764	6,005	759,200	328,500	1.3 M (4)	785 K	44	45 K	15 K
words	25,486	205	81,700	41,300	120 K (3)	83 K	7	70 K	35 K
linux.words	45,402	399	114,800	47,950	140 K (3)	95 K	12	70 K	35 K
Moby words simple	354,984	3,626	986,000	338,500	1.33 M (4)	780 K	90	80 K	30 K
Moby words compound	256,772	3,134	1.35 M	474,300	1.85 M (4)	1.11 M	189	80 K	30 K
Moby words all	611,756	6,759	2.27 M	780,200	3 M (4)	1.9 M	380	60 K	25 K

4.2 Other Data Sets

In order to demonstrate the potential of our method for compressing static dictionaries we present in Table 1 experimental results for seven additional natural language data sets. Six are publicly available and some compression results have already been published for two of them. Here are the brief descriptions:
- DELAF word forms: all the simple French word forms without any additional information, extracted from DELAF
- Calgary book1 7-tuples: a list of all successive seven-tuples from book1 of Calgary corpus; the compressed size of this set as reported in [7] is about 2.5 M
- words: a list of English words found in /usr/dict/words on Unix systems (older release); the compressed size of this set as reported in [13] is 112 K
- linux.words: a list of English words found in /usr/dict/linux.words on Linux systems
- Moby words simple: a list of simple English words from http://www.dcs.shef.ac.uk/research/ilash/Moby/mwords.html
- Moby words compound: a list of compound English words from http://www.dcs.shef.ac.uk/research/ilash/Moby/mwords.html
- Moby words all: combined simple and compound word lists of above.

The compression times and search speeds were measured on 333 MHz P II PC under Linux OS. The compression times given are for Algorithm B steps 3 to 6, i.e. without initial sorting of input entries and building the trie. Search speed is calculated by measuring the time needed for reading all the input words from disk and looking them up in the compressed structure loaded in the main memory. The first, most densely populated, level of the compressed trie is accessed through the array of starting positions for each letter instead of searching the list. This speeds up the search for up to 20% with the space overhead of only 512 bytes for the array (if long integers are used as pointers to starting positions of different letters in LLTC).

In standard coding of LLTC units node sizes are rounded to a whole byte for optimum speed and simplicity. In some cases this is a considerable waste; for instance, Moby data largest pointer units require 26 bits, leaving 6 bits per 4 byte unit unused. In structures with minimal coding all elements are coded with minimum number of bits. Only a small overhead of few bytes is necessary for denoting table and array element sizes, as well as the distribution of various pointers in the table.

5 Conclusion

Experimental results presented in Table 1 show that our method exhibits considerable potential for storing natural language data, for inflected languages more than for non-inflected - the French word forms set compresses considerably better than the sets of English words. Still, it performs well for every set tested. The only data sets we could find with previously published results (words and 7-tuples) compress better than previously reported. One would expect that increased number of words would always lead to a better overlapping of substrings. It is therefore somewhat surprising that combined sets of Moby simple and compound words do not compress better than when separated. Also, although we are satisfied with the final result, the huge number of different tags in DELAF did not compress as well as we expected. When partitions of DELAF (even as small as 10,000 entries) are compressed separately the compression ratio is roughly the same as for the whole set. Obviously, with LLTC compression, as with any compression method, the degree of success depends on the actual data. Overall, we believe that presented method of LZ linked list trie compression can be successfully used for storing and accessing data in various natural language related applications.

Acknowledgements

We thank the LADL (Laboratoire d'automatique documentaire et linguistique, University Paris 7) for providing DELAF and anonymous referees for helpful comments.

References

1. A. Acharya, H. Zhu and K. Shen, Adaptive Algorithms for Cache-efficient Trie Search, ACM and SIAM Workshop on Algorithm Engineering and Experimentation ALENEX 99, Baltimore, Jan. 1999.
2. A. Andersson and S. Nilsson, Improved Behaviour of Tries by Adaptive Branching. *Information Processing Letters,* Vol. 46, No. 6, 295-300, 1993.
3. J. Aoe, K. Morimoto and T. Sato, An efficient implementation of trie structures, *Software-Practice and Experience,* Vol. 22, No. 9, 695-721, 1992.
4. A.W. Appel and G.J. Jacobson, The world's fastest scrabble program, *Communications of the ACM,* Vol. 31, No. 5, 1988.
5. J. Bentley, A spelling checker, *Communications of the ACM,* Vol. 5, No. 28, 456-462, 1985.
6. B. Courtois, Un systéme de dictionnaires électroniques pour les mots simples du français, in Langue Française 87, Dictionnaires électroniques du français, eds. Blandine Courtois and Max Silberztein, Larousse, Paris, 11-22, 1990.
7. J.J. Darragh, J. G. Cleary and I. H. Witten, Bonsai: A Compact Representation of Trees, *Software-Practice and Experience*, Vol. 23, No. 3, 277-291, 1993.
8. J. A. Dundas, Implementing dynamic minimal-prefix tries, *Software-Practice and Experience*, Vol. 21, No. 10, 1027-1040, 1991.
9. M. Gross, La construction de dictionnaires électroniques, *Ann. Télécommun.* Vol. 44, No. 1-2, 4-19, 1989.

10. M. Gross, Constructing Lexicon-Grammars, in Computational Approaches to the Lexicon, eds. B.T.S. Atkins and A. Zampolli, Oxford University Press, 1994.

11. D. Gusfield, Algorithms on Strings, Trees, and Sequences, Cambridge University Press, 1997.

12. T. Kowaltowski, C. Lucchesi and J. Stolfi, Finite automata and efficient lexicon implementation, Technical report IC-98-2, University of Campinas, Brazil, 1998.

13. C. Lucchesi and T. Kowaltowski, Applications of finite automata representing large vocabularies, *Software-Practice and Experience*, Vol. 23, No. 1, 15-30, 1993.

14. U. Manber and G. Myers, Suffix arrays: a new method for on-line search, *SIAM Journal on Computing*, Vol. 22, No. 5, 935-948, 1993.

15. K. Morimoto, H. Iriguchi and J. Aoe, A method of compressing trie structures, *Software-Practice and Experience*, Vol. 24, No. 3, 265-288, 1994.

16. T. D. M. Purdin, Compressing tries for storing dictionaries, *Proceedings of the 1990 Symposium on Applied Computing*, Fayetteville, Apr. 1990.

17. D. Revuz, Dictionnaires et lexiques: Méthodes et algorithmes, Ph.D. thesis, CERIL, Université Paris 7, 1991.

18. S. Ristov, Space saving with compressed trie format, *Proceedings of the 17th International Conference on Information Technology Interfaces*, eds. D. Kalpić and V. Hljuz Dobrić, 269-274, Pula, Jun 1995.

19. S. Ristov, D. Boras and T. Lauc, LZ compression of static linked list tries, *Journal of Computing and Information Technology*, Vol. 5, No. 3, 199-204, Zagreb, 1997.

20. E. Roche, Analyse syntaxique transformationnelle du français par transducteurs et lexique-grammaire, Ph.D. thesis, CERIL, Université Paris 7, 1993

21. M. Rodeh, V.R. Pratt and S. Even, A linear algorithm for data compression via string matching, *Journal of the ACM*, Vol. 28, No. 1, 16-24, 1981.

22. M. Silberztein, INTEX: a corpus processing system, *Proceedings of COLING-94*, Kyoto, 1994.

23. J. Ziv and A. Lempel, A universal algorithm for sequential data compression, *IEEE Transactions on Information Theory*, Vol. IT-23, No. 3, 337-343, 1977.