Smaller Representation of Finite State Automata

Jan Daciuk[a], Dawid Weiss[b]

[a] *Department of Intelligent Interactive Systems, Gdańsk University of Technology, Poland*
[b] *Institute of Computing Science, Poznan University of Technology, Poland*

Abstract

This paper is a follow-up to Jan Daciuk's experiments on space-efficient finite state automata representation that can be used directly for traversals in main memory [4]. We investigate several techniques of reducing the memory footprint of minimal automata, mainly exploiting the fact that transition labels and transition pointer offset values are not evenly distributed and so are suitable for compression. We achieve a size gain of around 20–30% compared to the original representation given in [4]. This result is comparable to the state-of-the-art dictionary compression techniques like the LZ-trie [12] method, but remains memory and CPU efficient during construction.

1. Introduction

Minimal, deterministic, finite-state automata are a good data structure for representing natural language dictionaries [7]. They are fast while maintaining small memory footprint. Small memory requirements are achieved through careful design of the contents of dictionaries, through minimization of automata, and through their compression.

The Wirth's law states that software gets slower faster than hardware gets faster. A similar law should hold for memory capacity. Memory for computers becomes exponentially bigger and cheaper with time. However, data stored in dictionaries grows even faster. This creates a need for more effective storage structures for dictionaries.

The remaining part of the paper is organized as follows. Section 2 describes natural language dictionaries. Section 3 introduces formal definitions. Section 4 serves as a survey of compression techniques that are used for finite automata. Our motivation and goals are given in Section 5. Section 6 introduces the data sets used in evaluating various methods described later in the paper. Section 7 describes our attempts to reduce the size of automata representation in memory. Section 8 provides an overview of computational experiments and their results, comparing them to the known state of the art. Section 9 concludes the paper.

2. Natural Language Dictionaries

Natural language dictionaries fall into three categories:

1. word lists,
2. dictionaries where words are associated with information that depends on suffixes or prefixes of words,

3. dictionaries where words are associated with information that does not depend on suffixes or prefixes of words.

Simple words lists are useful in, for example, spelling correction. They require no additional coding of information. Morphological dictionaries of inflectional languages represent the second category. Inflected forms of words can be associated with information such as their syntactic categories and their canonical forms (lemmas). That information can be appended to inflected forms separated with an out-of-language separator symbol. A fragment of such a layout is shown below (Polish morphological dictionary, the first entry is the inflected form, the second contains the lemma, the third carries grammatical annotations; separator character is set to tabulation):

```
jajka   jajko   subst:pl:acc.nom.voc:n+subst:sg:gen:n
jajkach jajko   subst:pl:loc:n
jajkami jajko   subst:pl:inst:n
```

Since syntactic categories imply a set of possible suffixes and that set is usually limited for a given language, suffixes form just a few different strings, so the dictionary automaton after minimization has a small number of states. The canonical forms depend also on the stem of inflected words. However, as the stems of the inflected form and the canonical form are usually identical, or they differ slightly, it is possible to code how many characters to delete from the end of the inflected form to get the common part (or entire) stem, and then add the suffix of the canonical form. Both the code and the suffix of the canonical form do not depend on the stem, so minimization brings an even smaller automaton.[1] The above example in such coding could look as shown below (encoding after [3]):

```
jajka+Bo+subst:pl:acc.nom.voc:n+subst:sg:gen:n
jajkach+Do+subst:pl:loc:n
jajkami+Do+subst:pl:inst:n
```

The third category contains semantic dictionaries, translation dictionaries, frequency lists, etc. The information they associate with words is arbitrary – it does not depend on morphology of those words. Therefore, appending it to words would create unique suffixes. Minimization of an automaton that contains strings with mostly unique suffixes would not reduce its size significantly. A solution can be achieved by placing that additional information outside the automaton. The role of the automaton is then limited to providing a function that maps all words recognized by the automaton to that additional information and this mapping can be done by uniquely numbering all words in the automaton – a technique called *minimal perfect hashing*.

When the number of pieces of additional information we want to associate with the words is relatively small, i.e. if the number of possible indexes is relatively small, a *pseudo-minimal* automaton may be the best solution, as each word recognized by it has at least one unique transition that is not shared with any other word, provided that words ends with a special end-of-word symbol. That *proper* transition can be used to hold arbitrary information. There may

[1] Alternative implementations use transducers.

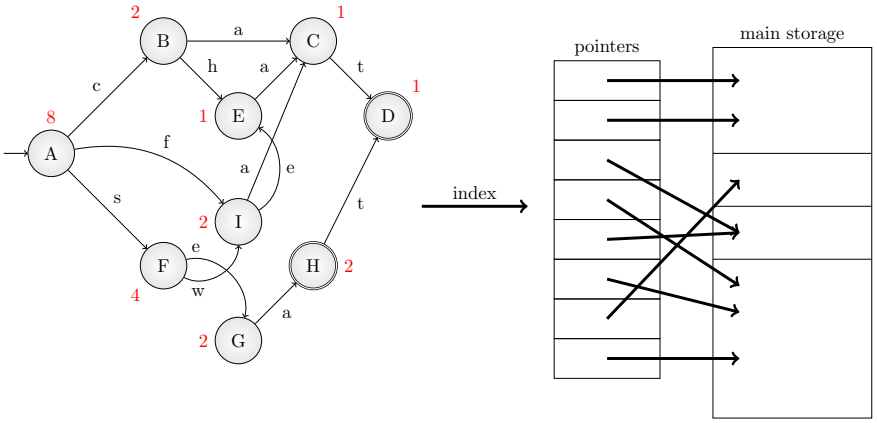

Figure 1: Minimal perfect hashing used for indexing additional information associated with words. The dictionary contains words *cat, chat, fat, feat, sea, seat, swat*, and *sweat*.

be more than one transition for a word. In such cases, additional information is stored on the first one, as later transitions may then share space with other transitions. Pseudo-minimal automata are also good devices for indexing information of variable size, as there is no need to maintain a vector of pointers. Construction of pseudo-minimal automata can be done using only slightly modified incremental construction algorithm for sorted data – with the equivalence relation replaced by pseudo-equivalence (details in [5]).

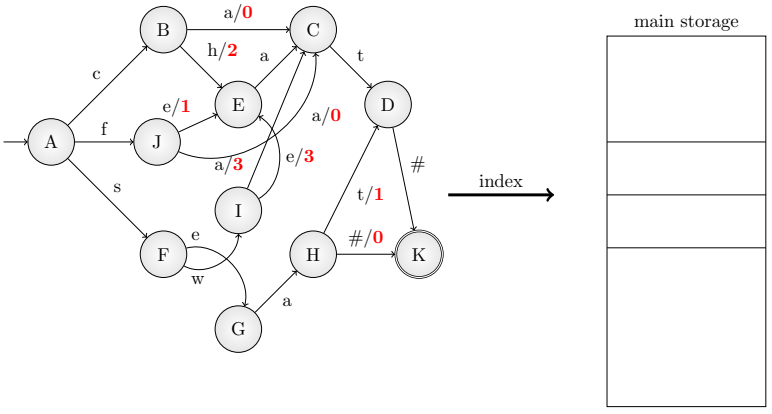

Figure 2: A pseudo-minimal automaton implementing the same mapping as the automaton and a vector of pointers in Figure 1.

3. Formal Definitions

A *deterministic finite-state automaton* (a *DFA*) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of symbols called the *alphabet*, $\delta : Q \times \Sigma \to Q$ is a *transition function*, $q_0 \in Q$ is the *initial* state (also called the *start* state), and $F \subseteq Q$ is the set of *final* (*accepting*) states. The transition

function δ can be extended so that its second argument is a string of symbols (a word), i.e. $\delta : Q \times \Sigma^* \to Q$ and for $q \in Q$, $a \in \Sigma, w \in \Sigma^*$:

$$\begin{array}{lcl} \delta(q, \varepsilon) & = & q \\ \delta(q, aw) & = & \delta(\delta(q, a), w) \end{array} \tag{1}$$

The *language* recognized by a DFA is the set of all words that lead from the initial state to any of the final states:

$$\mathcal{L}(A) = \{w \in \Sigma^* : \delta(q_0, w) \in F\} \tag{2}$$

Among all automata that recognize the same language there is one (up to isomorphism) that has the minimal number of states. It is called the *minimal* automaton:

$$(\forall A : \mathcal{L}(A) = \mathcal{L}(A_{min}))\, |A| \geq |A_{min}| \tag{3}$$

A *right language* of a state q is the set of words that lead from the state to any of the final states:

$$\overrightarrow{\mathcal{L}}\,(q) = \{w \in \Sigma^* : \delta(q, w) \in F\} \tag{4}$$

It is easy to note that the language of an automaton is the right language of its initial state:

$$\mathcal{L}(A) = \overrightarrow{\mathcal{L}}\,(q_0) \tag{5}$$

Two states p and q are *equivalent* if and only if they have the same right language:

$$(p \equiv q) \Leftrightarrow \overrightarrow{\mathcal{L}}\,(p) = \overrightarrow{\mathcal{L}}\,(q) \tag{6}$$

A minimal automaton has no pair of states that are equivalent. The right language can be defined recursively:

$$\overrightarrow{\mathcal{L}}\,(q) = \bigcup_{\sigma \in \Sigma : \delta(q,\sigma) \in Q} \sigma\, \overrightarrow{\mathcal{L}}\,(\delta(q, \sigma)) \cup \begin{cases} \{\varepsilon\} & \text{if } q \in F \\ \emptyset & \text{otherwise} \end{cases} \tag{7}$$

Therefore, cardinality of the right language of a state can be computed as:

$$|\overrightarrow{\mathcal{L}}\,(q)| = \sum_{\sigma \in \Sigma : \delta(q,\sigma) \in Q} |\overrightarrow{\mathcal{L}}\,(\delta(q, \sigma))| + \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases} \tag{8}$$

Let us define a function *fin* as:

$$\text{fin}(q) = \begin{cases} 1 & \text{if } q \in F \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

If $w = w_1 \dots w_{|w|} \in \mathcal{L}(A)$, then we can compute the ordinal number of w among words recognized by the automaton as:

$$\begin{array}{rl} h(w) = & \sum_{i=1}^{|w|} \left(\sum_{\sigma : \delta(q_0, w_{1\dots i-1}\sigma) \in Q \wedge \sigma < w_i} |\overrightarrow{\mathcal{L}}\,(\delta(q_0, w_{1\dots i-1}\sigma))| \right) + \\ & + \sum_{i=0}^{|w|-1} \text{fin}(\delta(q_0, w_{1\dots i})) \end{array} \tag{10}$$

Two states p and q are *pseudo-equivalent* if they are equivalent and the cardinality of their right languages is 1:

$$(p \cong q) \Leftrightarrow \left(\overrightarrow{\mathcal{L}}(p) = \overrightarrow{\mathcal{L}}(q) \wedge |\overrightarrow{\mathcal{L}}(p)| = 1\right) \tag{11}$$

In a *pseudo-minimal* automaton, $p \cong q$ means $p = q$.

4. Compression of Automata

Automata are not only fast in construction and traversals, but also take little space. Small memory footprint stems from minimality, but it is possible to reduce it even further using various compression and bit-packing schemes. It is also possible to change the definition of an automaton so that transitions, and not the states, can become final [3]. Ciura and Deorowicz [2] call such an automaton a *Mealy's acceptor* to underline the parallel with Moore's transducers and Mealy's transducers. Moore's transducers store their output in their states, Mealy's transducers – in their transitions.

One compression technique is universal in all implementations. Fields like a transition's label, target state's address (a pointer), or various flags can be packed into a minimal bit field required for their representation. Packing the fields so that they occupy as few bytes or bits as possible greatly reduces memory requirements. Decoding bit-aligned representation on modern hardware does not impose a large overhead on processing time and compact memory representation contributes nicely to reuse of CPU cache lines.

Packing fields bit- and byte-wise may be done in two manners: the fields can maintain fixed length, or their length may become variable, that is different instances of the field may have different lengths. The latter can be implemented using Huffman coding or other variable-bit representation schemes. While it can lead to greater savings, such compression requires additional memory lookups for decoding and can even lead to increased overall size if addresses need to be bit- or byte-aligned instead of being multiplications of a node's fixed size.

In most efficient implementations, an automaton is a vector of transitions and states are represented implicitly. There are two major methods of representing states. In the first one, a state is a list of outgoing transitions. In the other one, a state is a vector of possible outgoing transitions with an allocated place for a transition for every transition label from the alphabet. The vector for each state is put into a larger vector of transitions so that states overlap whenever possible without conflicts between transitions. The latter method is called *superimposed coding* [9]. It is faster for recognition, as each time we traverse a transition, we go to it directly without looking at any other transition going out from the same state (a transition is indexed directly by its label), and it is slower for exploration, as we need to check which transitions exist. That method allows for fewer compression techniques, so we will focus on the state-as-a-list representation.

In the state-as-a-list representation, it is possible to link subsequent transitions with pointers, but using a vector is more economical. The next transition in the vector is the next transition on the list. The problem of knowing what the last transition is can be solved by either storing an outgoing transition counter in an incoming transition, or by using a flag [8] (we call it `L` for `LAST`) to mark the last transition on the list. The latter approach saves more space.

A transition connects two states. Since we group transitions going out from a state, we need to specify the target of a transition, that is the address of the target state, which in state-as-a-list representation is the address of its first outgoing transition. There are several methods of reducing the size of that field. When the target state is placed directly after the current transition, it is possible to omit the field altogether at the cost of adding a new flag that we call `N` for `NEXT`. When this flag is set, the transition has no target address field, and the target state begins right after the current transition. It is also possible to vary the size of the address field so that there are local (short) and global (long) pointers as in [10] at the cost of an additional flag. In a US patent 5 551 027 granted on August 7th, 1996 to Xerox, frequently used addresses are put into a vector of full length pointers, and the addresses are replaced with shorter indexes to the pointers in the vector.

Since a state is stored as a list of outgoing transitions, it is possible to share transitions between states. When all transitions of one state are also present as transitions of another state (that has more transitions), then the "smaller" state can be stored inside the "bigger" one. When we use the `L` flag, the transitions of the smaller state have to be the last transitions of the bigger state. If it is not the case, the transitions need to be rearranged to conform to this condition. There may be many combinations of smaller states fitting into some larger ones, so heuristics have to be used. Note that once a state is stored inside another one, there is no speed or memory penalty for using this type of compression, it just reuses the same memory regions.

Another technique of reusing states' transitions is based on the fact that two states may share a subset of their transitions, but are not subsets of each other (each of the states has transitions that the other one does not have). In such case, one state is stored intact, the unique transitions of the second state are stored as usual, but the last transition has a flag we call `T` for `TAIL`, followed by the address of the common set of transition stored in the first state. Reordering of transitions inside individual states may lead to greater savings.

A generalization of transition sharing is presented in the LZ-trie method by Strahil Ristov [12]. The LZ-trie method treats an automaton as a sequence of transitions and applies compression to this sequence. A suffix tree (or array) is used for finding all subsequences of transitions, storing them once, and replacing redundant instances with pointers to their previous occurrences. This gives state-of-the-art compression ratios [1]. Note that combining LZ-trie with other methods described above gives much poorer results [6].

Some research has been devoted to finding substructures in an automaton – subautomata [14, 13]. Although conceptually different from the LZ-trie method, these methods can be seen as a variant of the LZ-trie method with some restrictions that limit compression efficiency. On the other hand, subautomata can have applications other than mere reduction of representation size.

5. Motivation and Goals

Many of the compression techniques described in the introduction are implemented in Jan Daciuk's fsa package [4]: transition-based representation, accepting transitions (Mealy's recognizers), optimizations of pointers in the form of the `N` bit or bit-packing of the target address with the rest of the flags. These tricks allow for direct, incremental construction in the compressed format, suitable for

immediate serialization to disk or storage in memory, and implementation of traversals over the packed format with very little overhead. The goals of this work were to investigate the following open problems:

1. Is it possible to construct a more space-efficient automaton representation that would retain the features present in the fsa package?
2. There is a trade-off between compressing representation and traversal efficiency. Is there a representation that would balance small size with an efficient (read: simple) automaton traversals?

6. Test Data

The research presented in this paper was mostly trial-and-error driven, where the baseline was acquired by comparing the output to the equivalent automata compiled using the fsa_build command from the fsa package. The choice of test data was thus important. The test files, their size and number of terms, are given in Table 1. The first five files on that list were collected by the authors of this work and the remaining files come from [2]. The `pl` data set is a morphological dictionary of inflected forms and their encoded lexemes and morphological annotations. It has highly repeatable suffixes (a limited set of inflection frames and morphological tags). The two data sets named `wikipedia` and `wikipedia2` contain terms from an inverted index of English Wikipedia (`wikipedia` is a sample, `wikipedia2` is an index of full content). Data sets called `streets` and `streets2` carry street and city names covering the area of Poland and have been acquired from a proprietary industrial application. The first five files in Table 1 contained UTF-8 encoded text. We did not alter the original character encoding used to represent the remaining data sets – they all used single-byte encodings of their respective languages (ISO8859-2 for Polish, for example). Our automata implementation was byte-based, so input character encoding was simply preserved in the automaton structure.

Table 1: File size (bytes), number of terms (lines) and an average number of bits per term for all the files used in experiments.

Name	Size (bytes)	Terms	BPT
`pl`	165 767 147	3 672 200	361
`streets`	706 187	59 174	95
`streets2`	203 590	17 144	95
`wikipedia`	105 316 228	9 803 311	86
`wikipedia2`	504 322 111	38 092 045	106
	—		
`deutsch`	2 945 114	219 862	107
`dimacs`	7 303 884	309 360	189
`enable`	1 749 989	173 528	81
`english`	778 340	74 317	84
`eo`	12 432 197	957 965	104
continued			
`esp`	8 001 052	642 014	100
`files`	212 761 171	2 744 641	620
`fr`	2 697 825	221 376	97
`ifiles`	212 761 171	2 744 641	620
`polish`	18 412 441	1 365 467	108
`random`	1 151 303	100 000	92
`russian`	9 933 320	808 310	98
`scrable`	1 916 186	172 823	89
`unix`	235 236	25 481	74
`unix_m`	191 786	20 497	75
`webster`	985 786	92 342	85

7. Size Reduction Techniques

Figure 3 shows a binary data layout of fields in a single transition in Jan Daciuk's fsa package. Recall this was the baseline representation we started from. A single transition is composed of the initial label, then a byte with three

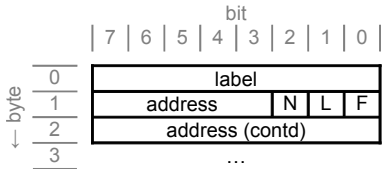

Figure 3: Binary layout of data fields in a single transition. fsa package compiled with `N` and `L` options. `N`, `F` and `L` are bit flags, address field's length is as large, as the largest state offset in the automaton (but constant for every transition).

flags – (`F` for `FINAL`, acceptor transition), `N` (no address, the target state follows this state's last transition) and `L` (this is the last transition of the current state). If the `N` bit is not set, partial address is bit-packed into the remaining five bits of the flags byte and as many bytes, as are needed to encode the largest integer offset in this automaton.

Starting with the baseline above, we tried numerous variations to decrease the representation size of each transition (and the transition graph as a whole). We describe these that yielded maximum gains in the paragraphs below.

V-coding of target addresses and of numbers implementing hashing. The fsa package uses fixed-length address encoding integrated with the flags byte. This has an effect of abrupt increases of automaton size once 1, 2, 3 or more bytes are needed to encode the largest state's offset. The same holds for numbers that are used in implementing hashing. In case of perfect hashing, they represent the cardinality of the right language stored at the beginning of a state, in case of pseudo-minimal automata – numbers stored in proper transitions. We used a simple form of variable length encoding for non-negative integers (*v-coding*), where the most significant bit of each byte is an indicator whether this is the last byte of the encoded integer and the remaining bits carry the integer's data in 7-bit chunks, lowest bits first. For example, 0 is encoded as (binary representation) `0000 0000`, 127 as `0111 1111`, 128 using two bytes: `1000 0000` and `0000 0001`, and so on. Encoding and decoding of v-coded integers can be implemented efficiently if we have them in consecutive bytes, so we moved the transition's target address to separate bytes, which left us with 5 unused bits in the flags byte. Table 2 shows the minimum number of bytes required to encode a given integer.

Table 2: Integer ranges encoded in a given number of bytes in *v-coding*. In general, the largest integer encoded in n bytes is $2^{7n} - 1$.

value range (hexadecimal)	largest integer (decimal)	bytes
0 – 0x7f	127	1 byte
0x80 – 0x3fff	16 383	2 bytes
0x4000 – 0x1fffff	2 097 151	3 bytes
0x200000 – 0xffffff	268 435 455	4 bytes

Transitions with index-coded label. We assumed each transition's label is a single byte. Each transition's label can be therefore an integer between 0 and 255 and this is a challenge for multi-byte or variable-byte character encoding

schemes, such as UTF-8. We opted for the automaton to store raw binary text representation in whatever input encoding is given on input. When performing traversals or lookups, the automaton's encoding must be respected – the input text must be converted to the automaton's code page, for example. A side effect of this is that certain transitions can lead to incomplete character codes, but we never had a problem with this in real applications (even with multi-byte Unicode).

In reality, for automata created on non-degenerate input, and in particular on text, the distribution of label values is often heavily skewed. Figure 4 illustrates the distribution of labels in the `pl` data set, for example – there are many transitions with a small subset of the label range and a few transitions outside this range.

The observation that labels have uneven distribution leads to an optimization that has a profound effect on automaton size: we can integrate the 31 most frequent labels ($2^5 - 1$) into the flags byte as an index to a static lookup table. Zeros on all these bits would indicate the label is not indexed and is stored separately. Note that we tried to avoid any complex form of encoding (like Huffman trees); a fixed-length table with 31 most frequent labels is a balanced tradeoff between auxiliary lookup structures and label decoding overhead at runtime.

Combining v-coding of the target address and table lookup for the most frequent labels yields two alternative transition formats, as shown in Figure 5. With such encoding most transitions take $1 + \textit{length(address)}$ bytes. In an extreme case when the `N` bit is also set (target follows the current state's last transition), the entire transition is encoded in a single byte.

Rearranging states to minimize the total length of address fields. By default states (actually a list of transitions of each state) in an automaton are serialized in a depth-first order to maximize the number of occurrences of the `N` flag and hence the gain from not having to emit the target address for such transitions. For these transitions where `N` is not set, the target address must be emitted and the amount of space taken by such an address depends on its absolute offset value (recall Table 2 on page 8). If we move certain states (these to which there are a lot of incoming transitions) to the beginning of the automaton, the global amount of space for address encoding should be smaller

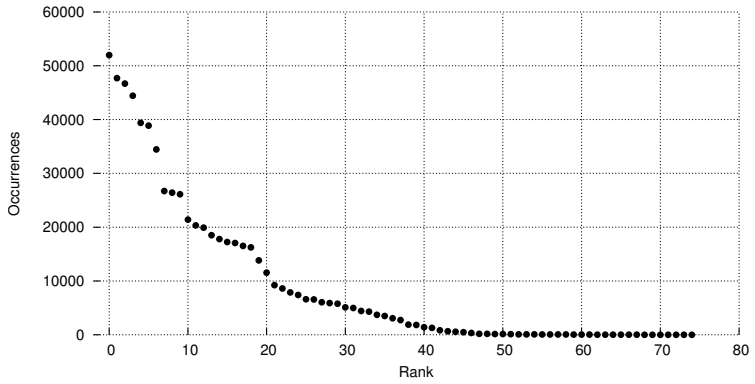

Figure 4: Number of occurrences of 75 most frequent labels in the `pl` data set.

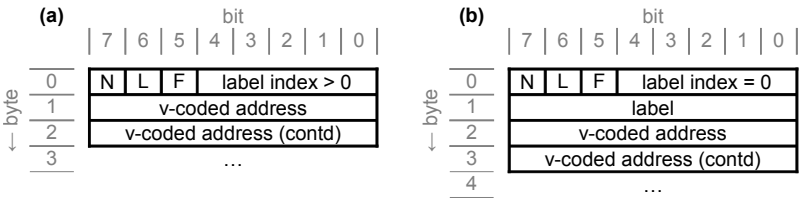

Figure 5: Binary layout of data fields in a single transition with v-coding of target address and indexed labels. Two variants of each transition are possible: (a) with the index to the label, (b) with the label directly embedded in the transition structure.

than if we leave these states somewhere farther in the serialized automaton structure. The question is which states we should move and in what order they should appear in the automaton structure.

The problem of rearranging states to minimize the global sum of bytes required for encoding target addresses is complicated. There are several things to consider:

- States located at offsets 0–127 require only one byte for target address code, states located at offsets 128–16 383 two bytes, and so on. But then, a single state may have many transitions, so it occupies a variable number of bytes. We can move to the front a single large state with many incoming transitions or, alternatively, many smaller states with fewer incoming transitions but all fitting in the "one-byte" offset range.
- By moving a state from its original location we also shift the offset of other states, possibly rearranging the fields across the entire automaton.
- We may lose the gain from applying the `N` flag optimization if we move a state (or its predecessor) to which the `N` flag applies.

The question if there is an "optimal" arrangement of states to minimize the global serialized automaton length remains open. The problem itself seems to be equivalent to bin-packing (in terms of computational complexity) and thus not have a solution working in a reasonable (polynomial) time.

Our first attempt to solve this issue was a simple heuristic: in the first step, we determine the serialization order for all states as to maximize the number of `N` bits (depth-first traversal). Then, we create a priority queue of states in the decreasing number of their incoming transitions and keep moving states from the top of the queue to the start of the automaton as long as the serialized automaton is smaller than before.

This heuristic has a serious flaw because serialized automaton size does not decrease monotonically with the number of moved states. For example, Figure 6 depicts automaton size in relation to the number of moved states for the `pl` data set. The minimum size is reached at around 2 900 reordered states with the largest number of incoming transitions, but a closer look at the beginning of this chart shows that the function is not monotonic – see the zoomed rectangle inside Figure 6. At around 29 reordered states the size goes up from 1 911 758 to 1 912 027, only to drop further down after more states are reordered.

The second take at the state reordering heuristic was a simulated-annealing like process that worked similar to the first approach (initial states order to

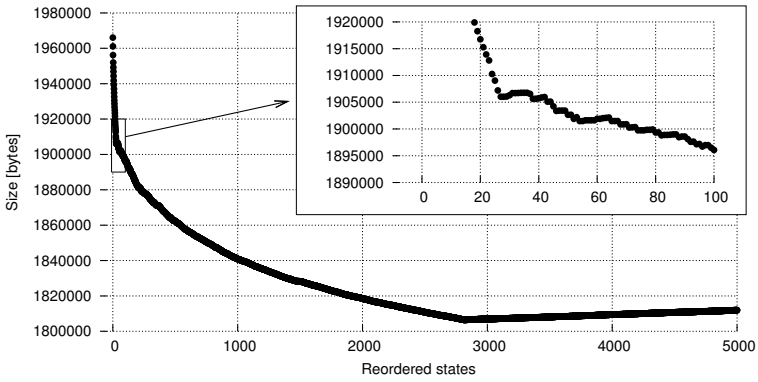

Figure 6: Automaton size in relation to the number of moved states, `pl` data set (first 5 000 state reorderings shown). The zoomed-in section of the chart shows the relation is not monotonic, even at the very beginning.

maximize the use of the `N` bit, then a queue of states with most inlinks), and then probed at various subsets of the states' queue, decaying over time and focusing on ranges that promised the smallest output.

To compute the resulting automaton size after each state reordering, both heuristics performed its full serialization. This was the key factor slowing down automaton compression and took vastly more time than automaton construction itself. Nonetheless, if size is of major importance, even these simple heuristics provide significant gain: as shown in Figure 6, for the `pl` data set the serialized size decreases from 1 966 038 bytes achieved by depth-first order traversal of states to 1 806 621 bytes (8% gain) with around 2 900 states moved to the front.

8. Experiments and Results

8.1. Compression ratios

Table 3 shows the results of compressing the test data sets into finite state automata using fsa_build (version 0.50, with patches), LZ-trie and a binary format utilizing optimizations presented in this paper, called cfsa2, implemented as part of the Morfologik project. The data sets and software used to compress them are available at: https://github.com/dweiss/paper-fsa-compression.

Automata packed using cfsa2 were on average 29% smaller compared to the result of fsa_build, regardless of the nature of the input file ($\sigma = 3.31\%$). LZ-trie produced files smaller by 13.7% on average (compared to cfsa2), but here the standard deviation is $\sigma = 11.86$ and there is a notable exception of the `pl` data set, smaller by 27% when packed using cfsa2. We strongly suspect that this difference is caused by the fact that the `pl` data set has a huge number of repetitive suffixes (morphological tags); it is very likely that the transitions to these repetitive suffixes ended up being moved to the front of the automaton and thus resulted in small sizes of target address pointers of many arcs, whereas in LZ-trie each such pointer is represented as a constant-size link node (with the link node's size determined by the global number of all nodes in the trie).

Yet, smaller files produced by LZ-trie come at a much longer compression time – for example, building LZ-trie for `wikipedia2` took 16 hours, while the (Java-based) cfsa2 compressed it in 25 minutes (of which 42 seconds were spent

Table 3: The size of on-disk automaton representation and bits per byte and term ratios for the input files compressed with fsa_build (fsa), cfsa2 and LZ-trie (LZ). The %[1] column shows size drop from fsa to cfsa2, %[2] from cfsa2 to LZ-trie. The smallest compressed size of each data set is marked with a ∎ symbol.

Name	Output size (KB)			%		Bits per byte			Bits per term		
	fsa	cfsa2	LZ	%[1]	%[2]	fsa	cfsa2	LZ	fsa	cfsa2	LZ
pl	2 655	1 764 ∎	2 245	34	−27	0.13	0.09	0.11	5.9	3.9	5.0
streets	334	244	217 ∎	27	11	3.87	2.83	2.52	46.2	33.8	30.1
streets2	128	93	86 ∎	28	7	5.16	3.73	3.46	61.3	44.3	41.1
wikipedia	—	40 362	36 413 ∎	—	10	—	3.14	2.83		33.7	30.4
wikipedia2	—	168 683	157 126 ∎	—	7	—	2.74	2.55		36.3	33.8
deutsch	285	215	188 ∎	24	13	0.79	0.60	0.52	10.6	8.0	7.0
dimacs	2 436	1 487	1 299 ∎	39	13	2.73	1.67	1.46	64.5	39.4	34.4
enable	401	290	264 ∎	28	9	1.88	1.36	1.23	18.9	13.7	12.4
english	243	173	145 ∎	29	16	2.56	1.82	1.53	26.8	19.1	16.0
eo	211	147	109 ∎	31	26	0.14	0.10	0.07	1.8	1.3	0.9
esp	385	268	187 ∎	30	30	0.39	0.27	0.19	4.9	3.4	2.4
files	12 425	9 205	7 120 ∎	26	23	0.48	0.35	0.27	37.1	27.5	21.3
fr	220	153	120 ∎	30	22	0.67	0.47	0.36	8.2	5.7	4.4
ifiles	12 770	9 748	8 147 ∎	24	16	0.49	0.38	0.31	38.1	29.1	24.3
polish	676	477	352 ∎	29	26	0.30	0.21	0.16	4.1	2.9	2.1
random	1 162	832	798 ∎	28	4	8.27	5.92	5.68	95.2	68.2	65.4
russian	505	354	262 ∎	30	26	0.42	0.29	0.22	5.1	3.6	2.7
scrable	435	310	263 ∎	29	15	1.86	1.33	1.12	20.6	14.7	12.4
unix	132	95	83 ∎	28	13	4.61	3.30	2.88	42.6	30.5	26.6
unix_m	104	72	63 ∎	30	12	4.42	3.09	2.70	41.4	28.9	25.3
webster	417	298	248 ∎	28	17	3.46	2.48	2.06	37.0	26.4	22.0

Table 4: Automata compression times (in seconds). Experiments were performed on the following hardware: cfsa2 and fsa5 – Intel Core i7 CPU 860 @ 2.80GHz, 8GB RAM, Ubuntu Linux; LZ-trie – Intel Xeon W3550 @ 3.07 Ghz, 12GB RAM, CentOS. Ratios are shown only for compression times greater than a few seconds (cfsa2 is written in Java and the timings include HotSpot warm-up time, so times for really short input data are not directly comparable).

Name	Compression time (s)			Ratio (%)	
	fsa	cfsa2	LZ	cfsa2/fsa	cfsa2/LZ
pl	40.01	20.15	6 000.00	50	0.34
streets	0.16	1.15	0.84		
streets2	0.13	3.57	0.21		
wikipedia		226.90	1 860.00		12
wikipedia2		1 556.84	57 600.00		3
deutsch	0.22	1.10	1.00		
dimacs	1.66	8.38	100.00		
enable	0.21	1.20	0.89		
english	0.12	0.90	0.50		
eo	0.84	1.10	5.00		
esp	0.57	1.14	3.00		
files	645.17	99.77	7 200.00	15	1
fr	0.21	0.85	1.00		
ifiles	453.53	102.10	25 200.00	23	0.41
polish	1.33	2.46	9.00		
random	0.61	4.12	3.00		
russian	0.66	1.50	4.00		
scrable	0.25	1.16	0.84		
unix	0.06	0.50	0.11		
unix_m	0.04	0.50	0.72		
webster	0.20	1.21	0.68		

Table 5: Ratios of integrated and separate labels and lengths of v-coded target state addresses. V-code zero is equivalent to the presence of the `N` flag.

Name	Labels (%)		V-code length (%)				
	int.	sep.	0	1	2	3	4
pl	94	6	30	9	32	29	
streets	98	2	39	17	23	20	
streets2	98	2	46	13	20	21	
wikipedia	79	21	38	18	14	13	16
wikipedia2	85	15	45	13	11	14	17
deutsch	100	0	31	15	35	19	
dimacs	98	2	48	12	19	21	
enable	100	0	25	27	30	19	
english	100	0	26	29	27	17	
eo	100	0	19	34	33	14	
esp	100	0	17	34	35	14	
files	86	14	64	3	6	12	15
fr	99	1	25	33	26	16	
ifiles	88	12	65	1	4	11	18
polish	99	1	20	33	29	18	
random	100	0	60	5	13	23	
russian	100	0	20	33	30	16	
scrable	100	0	23	30	29	17	
unix	99	1	26	31	21	22	
unix_m	100	0	26	33	23	18	
webster	100	0	24	30	28	18	
$\mu =$	96	4	34	22	23	18	3

in constructing the FSA and the rest seeking for the optimum number of states to reorder, which yet again proves the point of improving this heuristic somehow). Table 4 shows a complete list of compression times for the three methods used. Note that these comparisons and times should be considered anecdotal evidence only because LZ-trie compression was performed on a different hardware (in both cases the computational power of machines used for experimenting was nearly identical though). Judging from how LZ-trie is built we believe the order-of-magnitude difference in compression time should hold for other implementations and architectures as well.

The largest size reduction is achieved by integrating transition labels with the flags byte (see Table 5) – most data sets did not even use transitions with separate label byte. Note that even a truly random byte sequence would still benefit from integrated labels at around 12% (even if label distribution is uniform, 31 labels would still be integrated in the flags field). Table 5 also shows the benefit of using the `N` bit (34% of transitions on average) and v-coding of transition pointers (an average of 45% of transitions used one or two bytes for the address).

8.2. Automata for perfect hashing

Automata for perfect hashing provide unique numbering for all accepted sequences. Like we mentioned in the introduction, this allows one to move some of the auxiliary information about a given entry to external storage and store only an integer identifier to that information.

Perfect hashing can be implemented in various ways (in automata and also in transducers). In automata, and in particular in the fsa package, perfect hashing is implemented by adding numbers on each state that store cardinality of their

Table 6: Output automaton size for cfsa2 without numbers (state reordering phase not exhaustive so results slightly bigger than in Table 3), cfsa2 with numbers stored using fsa5 scheme (fixed byte count), and cfsa2 with v-coded numbers. The $\%^1$ column shows cfsa2/fixed to cfsa2 size ratio, $\%^2$ column shows cfsa2/vint to cfsa2 size ratio, and $\%^3$ column shows cfsa2/vint to cfsa2/fixed size ratio.

Name	Output size (KB)			%		
	cfsa2	cfsa2/fixed	cfsa2/vint	$\%^1$	$\%^2$	$\%^3$
pl	1 845	2 783	2 179	151	118	78
streets	246	376	312	153	127	83
streets2	93	151	122	162	131	81
wikipedia	44 290	68 462	52 325	155	118	76
wikipedia2	184 700	340 569	223 436	184	121	66
deutsch	221	361	269	163	122	75
dimacs	1 556	2 905	1 999	187	128	69
enable	296	464	353	157	119	76
english	175	278	210	159	120	76
eo	147	213	176	145	119	83
esp	273	379	311	139	114	82
files	9 451	18 927	12 691	200	134	67
fr	154	240	185	156	120	77
ifiles	9 977	20 026	13 401	201	134	67
polish	492	699	568	142	115	81
random	844	1 820	1 171	216	139	64
russian	362	523	421	144	116	80
scrable	316	485	374	153	118	77
unix	95	133	114	140	120	86
unix_m	72	102	87	141	121	86
webster	304	473	361	156	119	76
			$\mu =$	162	123	76

right language – see Equation (8). While traversing the automaton a counter of skipped final nodes is maintained and entire skipped subtrees are added simply by taking a number stored in their root state. In fsa5 format, the above numbers are stored in the minimal number of bytes required to hold the largest integer representing a state's right language cardinality (in effect equal to the number of input sequences accepted by the automaton on the root state). We performed an experiment in which cfsa2 algorithm was extended with numbers on each state. Table 6 shows the sizes of output automata for cases when numbers were not included at all, were encoded using fsa5 scheme and finally were v-coded.

The results show that v-coding of numbers on states yields an average 23% growth in size compared to an automaton without encoded numbers. This result is quite good compared to fixed-size encoding scheme from fsa5 (62% larger than the baseline automaton on average, with steadily higher savings for larger automata).

8.3. Applying cfsa2 optimizations to LZ-trie

We have spent a significant amount of time trying to port the optimizations described in this paper to LZ-trie. Unfortunately the code for that algorithm is not publicly available (at the time of writing LZ-trie authors had been working on improvements promising prompt code release). We implemented LZ-trie independently based on its description from the paper and we present the conclusions here.

The first conclusion is that LZ-trie's public description allows for many interpretations of how double-linked references work (and whether outlinks from

inside are allowed or not). Depending on how one implements these references and depending on the *order of recursive compaction* of trie nodes the output size (the number of unique nodes that need to be kept) varies even up to 30% compared to the best case found in multiple runs. It is also not at all clear to us if there exists an "optimal" layout of references that results in minimal number of nodes. As far as we could tell from our experiments the problem is too complex to be tackled with accurate methods and requires greedy heuristics. Our implemented variation of the algorithm differed from Strahil Ristov's version because we could not achieve his reported results (even though we know they were correct and differed from ours only in how and which nodes were compacted first).

We then tried to apply the optimizations described in this paper to our version of LZ-trie. LZ-trie heavily relies on the assumption that node representation (blocks) are of fixed length. This assumption is used to:

- simplify encoding of relative block offsets (forward jumps),
- to make the representation more repetitive when there are many common "suffix subtrees", and finally
- to make everything smaller when in the last step all distinct blocks are uniquely numbered and the trie is stored as a bit-packed sequence of such identifiers along with a lookup table of single copies of expanded blocks.

Any size reduction gained from these techniques vanishes once we allow variable size blocks – both relative and absolute offsets quickly diversify individual blocks and their distinct number explodes. Note that nearly everything covered in this paper (v-coding, index-coded labels) *requires* that each individual state's representation is of variable length. It does not come as a surprise then that our implementation of "variable block length" LZ-trie was never even close in size to the "fixed block length" version. We omit the details here because they are of no practical relevance in the light of what has been said.

To summarize, contrary to our initial feeling, we now think the applicability of methods introduced in this paper is very limited (if any at all) in the context of LZ-tries. There is definitely a lot of room for improvement there in how repeated subsequence selection progresses but this is another topic.

9. Conclusions

We have shown that three basic techniques:

- table-lookup encoded labels, exploiting their uneven distribution,
- variable-length coding of transition target addresses as well as perfect hashing indexes, and
- state ordering to minimize the global size of encoded target addresses

make it possible to compress (already compact) dictionaries considerably, in some cases even better than the LZ-trie method, whose results were so far considered the best in the field. Not of less importance is the fact that the representation presented in this paper retains simple automaton structure and

allows very efficient, non-recursive traversals. There is a considerable space for further research in how to efficiently determine an optimal or nearly-optimal arrangement of states to minimize their global representation length, but even the presented naïve heuristic implemented in Java turns out to be much faster than fsa_build or LZ-trie, especially on large data sets.

Comparing our method to the LZ-trie method, the main difference is that we do not search for repeatable substructures. By finding subautomata, we might possibly boost the compression ratio at the cost of slightly increased traversal time and moderately increased construction time. LZ-trie takes a different angle at minimizing representation size and the techniques used there seem incompatible with what we have presented in this paper.

Another interesting aspect that requires attention is automaton traversal speeds. All methods exercised in this paper represent a state's transitions in a form that requires a linear lookup scan to find a matching label. This is highly ineffective when traversing highly fanning-out states, which unfortunately usually happen to be close to (and including) the automaton root. We created a simple benchmark where the same traversal routine was executing a simple hit/miss test using a mix of random and matching sequences. The traversal speed (same hardware as in Table 4) on an automaton in fsa5 format averaged around 1.9 million checks per second, on cfsa2 – around 800 thousand checks per second (variable transition length requires partial decoding hence the slowdown). These figures compare favorably to the speed achieved by LZ-trie, which, as reported by the author, achieves around 1 million checks per second.

A few simple improvements can be made to make the traversal much, much faster at a slight size penalty. The most obvious improvement is to expand states with a larger fan-out into a form allowing direct table-lookup (or binary search) of a given label. This has been implemented in Apache Lucene recently and yields nearly 4 million terms/ second check speed. Another optimization hint is related to utilizing CPU caches better – we can clump together the representation of states reachable from the root state so that they fit in as few cache lines as possible. This can be easily done by breadth-first traversal to a given depth and even combined with state reordering mentioned earlier. We plan to tackle these ideas in our future work on the subject.

Pseudo-minimal automata are larger than the minimal ones. In Dominique Revuz'es [11] experiments, they had up to eight times more states. However, when used to implement hashing, they need to store numbers only in their proper transitions, and not in every state, as it is the case with minimal automata implementing perfect hashing. Additionally, they can implement arbitrary hashing, so in case of variable size information, or when the number of pieces of (long) information associated with words is smaller than the number of words, there is no longer need for a vector of pointers; the numbers stored in proper transitions can serve as those pointers directly. In such cases, a comparison with minimal perfect hashing automata should take into account also the size of the vector of pointers. We did not manage to implement pseudo-minimal automata at the time of submission of this paper. We hope to have the implementation and the results of experiments ready in time for the final version.

As a concluding remark, let us note that morphological dictionaries compressed very well in our experiments, achieving incredible compression ratios (1.3 bits per entry for `eo` or 3.9 bits per entry for the `pl` data set). Knowing that finite state automata can be used for calculating perfect hashes (or

with minor modifications as transducers) it is somewhat surprising to learn that quite a few tools for natural language processing still opt for using traditional databases to store and search for linguistic data.

Acknowledgments. Strahil Ristov and Damir Korencic kindly responded to our request to run LZ-trie on the provided data sets and passed back valuable comments and suggestions. Michael McCandless provided a list of terms extracted from Wikipedia and was always ready for long (and fruitful) discussions on the subject of automata compression and traversal speeds. Finally, we really appreciate all the corrections, comments and remarks given by anonymous reviewers of this work. Thank you.

References

[1] Budiscak, I., Piskorski, J., Ristov, S., 2009. Compressing Gazetteers Revisited. 8th International Workshop on Finite-State Methods and Natural Language Processing (FSMNLP 2009), Pretoria, South Africa.

[2] Ciura, M., Deorowicz, S., 2001. How to squeeze a lexicon. Software – Practice and Experience 31 (11), 1077–1090.

[3] Daciuk, J., 1998. Incremental construction of finite-state automata and transducers, and their use in the natural language processing. Ph.D. thesis, Technical University of Gdańsk.

[4] Daciuk, J., July 2000. Experiments with automata compression. In: Conference on Implementation and Application of Automata CIAA'2000. University of Western Ontario, London, Ontario, Canada, pp. 113–119.

[5] Daciuk, J., Maurel, D., Savary, A., 2006. Incremental and semi-incremental construction of pseudo-minimal automata. In: Farre, J., Litovsky, I., Schmitz, S. (Eds.), Implementation and Application of Automata: 10th International Conference, CIAA 2005. Vol. 3845 of LNCS. Springer, pp. 341–342.

[6] Daciuk, J., Piskorski, J., 2006. Gazetteer Compression Technique Based on Substructure Recognition. In: Proceedings of the International Conference on Intelligent Information Systems, Ustroń, Poland. pp. 87–95.

[7] Daciuk, J., Piskorski, J., Ristov, S., 2010. Scientific Applications of Language Methods. Mathematics, Computing, Language, and Life: Frontiers in Mathematical Linguistics and Language Theory. World Scientific Publishing, Ch. Natural Language Dictionaries Implemented as Finite Automata, pp. 133–204.

[8] Kowaltowski, T., Lucchesi, C. L., Stolfi, J., 1993. Minimization of binary automata. In: Proceedings of the First South American String Processing Workshop. Belo Horizonte, Brasil, pp. 105–116.

[9] Liang, F. M., 1983. Word hy-phen-a-tion by com-put-er. Ph.D. thesis, Stanford University.

[10] Lucchiesi, C., Kowaltowski, T., 1993. Applications of finite automata representing large vocabularies. Software Practice and Experience 23 (1), 15–30.

[11] Revuz, D., 1991. Dictionnaires et lexiques: méthodes et algorithmes. Ph.D. thesis, Institut Blaise Pascal, Paris, France, lITP 91.44.

[12] Ristov, S., Laporte, E., 1999. Ziv Lempel Compression of Huge Natural Language Data Tries Using Suffix Arrays. In: Proceedings of Combinatorial Pattern Matching (CPM 1999), LNCS 1645. pp. 196–211.

[13] Tounsi, L., 2008. Sous-automates à nombre fini d'états. application à la compression de dictionnaires électroniques. Ph.D. thesis, Université François Rabelais Tours.

[14] Tounsi, L., Bouchou, B., Maurel, D., 2008. A Compression Method for Natural Language Automata. In: Proceedings of the 7th International Workshop on Finite-State Methods and Natural Language Processing, Ispra, Italy. pp. 146–157.