Experiments with Automata Compression

Jan Daciuk

Alfa Informatica, Rijksuniversiteit Groningen
Oude Kijk in 't Jatstraat 26, Postbus 716
9700 AS Groningen, the Netherlands
e-mail: *j.daciuk@let.rug.nl*

Abstract. Several compression methods of finite-state automata are presented and evaluated. Most compression methods used here are already described in the literature. However, their impact on the size of automata has not been described yet. We fill that gap, presenting results of experiments carried out on automata representing German, and Dutch morphological dictionaries.

1 Introduction

Finite-state automata are used in various applications. One of the reasons for this is that they provide very compact representations of sets of strings. However, the size of an automaton measured in bytes can vary considerably depending on the storage method in use. Most of them are described in [5], a primary reference for all interested in automata compression. However, [5] does not provide sufficient data on the influence of particular methods on the size of the resulting automaton. We investigate that in this paper. We used only deterministic, acyclic automata in our experiments. However, the methods we used do not depend on that feature. The automata we used were minimal (otherwise the first step in compression should be the minimization).

Our starting point is as follows. An automaton is stored as a sequence of transitions (fig. 1). The states are represented only implicitly. A transition has a label, a pointer to the target state, the number of transitions leaving the target state (*transition counter*), and a final marker. We use the transition counter to determine the boundaries of states, instead of finding them by subtracting addresses of states in a large vector of addresses of states as in [4], because we can get rid of the vector.

We store the final marker as the most significant bit of the transition counter. We use non-standard automata, *automata with final transitions* (see [2]), because they have less states and less transitions than the traditional ones. But since the storage methods are the same, the results are also valid for traditional automata.

2 Compression Techniques

Compression techniques fall into three main categories:

	L	F	#	→
1	s		2	2
2	t		1	4
3	a		1	5
4	a		1	5
5	y	•	0	0

Fig. 1. Starting point storage method. L is a label, F marks final transitions, # is the number of outgoing transitions in the target state, and → is a pointer to the target state. The automaton recognizes words *say* and *stay*.

- coding of input data,
- making some parts of an automaton share the same space,
- reducing the size of some elements of an automaton.

Some techniques may contribute to the compression in two ways, e.g. changing the order of transitions in a state can both make sharing some transitions possible and reduce the size of some pointers. The first category depends on the kind of data that is stored in the automaton. The techniques we use apply to natural language dictionaries.

We define a deterministic finite-state automaton as $A = (\Sigma, Q, i, F, E)$, where Q is a finite set of states, $i \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, and $E \subseteq Q \times \Sigma \times Q$ is a set of transitions. We also define a function *bytes* that returns the number of bytes needed to store its argument: $bytes(x) = \lceil \log_{256} x \rceil$. Total savings (in percents) achieved by using a particular method M on the starting point automaton are $\eta^M(A) = 100\% \cdot \tau^M(A) \cdot \pi^M(A)/sizeof(A)$, where $\tau^M(A)$ is the number of transitions affected by the compression method, $\pi^M(A)$ is the saving in bytes per affected transition, and *sizeof(A)* is the size of the automaton in bytes. The size of an automaton in the starting point representation is $|E|(2 \cdot bytes|\Sigma| + bytes(|E|))$. In all those calculations, we assume that additional one-bit flags they require fit into the space taken by a pointer without the need to enlarge it.

2.1 Coding of Input Data

This section applies to natural language morphological dictionaries. Entries in such dictionaries usually contain 3 pieces of information: the inflected form, the base form, and the categories associated with the inflected form. It is common (e.g. in INTEX [11], [10], and in systems developed at the University of Campinas [6]) to represent that information as one string, with the base form coded. The standard coding consists of one character that says how many characters should be deleted from the end of inflected form so that the rest could match the beginning of the base form, and the string of characters that should be appended to the result of the previous operation to form the base form.

Such solution works very well for languages that do not use prefixes or infixes in their flectional morphology, e.g. French. However, in languages like German and Dutch, prefixes and infixes are present in many flectional forms. So to accommodate for this feature, we need 2 additional codes. The first one says what is the position from the beginning of a prefix or infix, the second code - the length of the prefix or infix. For languages that do not use infixes, but do use prefixes, it is possible to omit the position code.

2.2 Eliminating Transition Counters

There are two basic ways to eliminate transition counters. One uses a very clever sparse matrix representation (see [12], [8], and [9]). Apart from eliminating transition counters, it also gives shorter recognition times, as transitions no longer have to be checked one by one – they are accessed directly. However, that method excludes the use of other compression methods, so we will not discuss it here.

The other method (giving the same compression) is to see a state not as a set of transitions, but as a list of transitions ([5]). We no longer have to specify the transition count provided that each transition has a 1-bit flag indicating that it is the last transition belonging to a particular state (see fig. 2). That bit can be stored in the same space as the pointer to the target state, along with the final marker. We can combine that method with others.

	L	F	S	→
1	s		•	2
2	t			4
3	a		•	5
4	a		•	5
5	y	•	•	0

Fig. 2. States seen as lists of transitions. S is the marker for the last transition in the state.

$$\tau^{sb}(A) = |E|,\ \pi^{sb}(A) = bytes(|\Sigma|)$$

2.3 Transition Sharing

If we look at the figure 1, we can see that we have exactly the same transition twice in the automaton. However, once it is part of a state with 2 different transitions, and another time it is part of a state that has only 1 transition. As the information about state boundaries is not stored in the transitions belonging to the given state, we can share transitions between states (on the left on fig. 3). More precisely, a smaller state (with a smaller number of outgoing transitions) can be stored in a bigger one. It is also possible to place transitions of a state so that part of them falls into one different state, and the rest into another one. This is possible only when we keep the transition counters, so we will not discuss that further.

	L	F	#	→
1	s		2	2
2	t		1	3
3	a		1	4
4	y	•	0	0

	L	F	S	→
1	s		•	2
2	t			3
3	a		•	4
4	y	•	•	0

Fig. 3. Two transitions (number 3 and 4 from figures 1 and 2) occupy the same space. Version with counters on the left, with lists – on the right.

In the version that uses lists of transitions, exactly one of the transitions belonging to a state holds information about one state boundary. The other boundary is defined by the pointer in transitions that lead to the state. If all transitions of a smaller state **A** are present as the last transitions of a bigger state **B**, then we can still store **A** inside **B** (on the right on fig. 3).

2.4 Next Pointers

	L	F	N	#	→
1	s		•	2	
2	t			1	3
3	a		•	1	
4	y	•		0	0

	L	F	S	N	→
1	s		•	•	2
2	t				3
3	a		•	•	4
4	y	•	•		0

Fig. 4. The next flag with sharing of transitions. Version with counters on the left, with lists – on the right. *N* represents the next flag.

Tomasz Kowaltowski *et al.* ([5]) note that most states have only one incoming and one outgoing transition, forming chains of states. It is natural to place such states one after another in the data structure. We call a state placed directly after the current one the *next state*. It has been observed in [5] that if we add a flag that is on when the target state is the next one, and off otherwise, then we do not need the pointer for transitions pointing to the next states. In case of the target being the next state, we still need a place for the flags and markers, but they take much less space (not more than one byte) than a full pointer. In case of the target state not being the next state, we use the full pointer. We need one additional bit in the pointer for the flag. Usually, we can find that space. In our implementation, we used the *next flag* only on the last transition of a state. Therefore, the representation of our example automaton looks like that given on figure 4.

The maximum number of transitions that can use next pointers is equal to the number of states in the automaton minus one, i.e. the initial state. The reason for this is that only one transition leading to a given state may be placed immediately in front of it in the automaton.

The transitions in states having more than one outgoing transition can be arranged in such a way that a transition leading to the next state in the automaton may not be the last one. However, if for a given transition its source state has exactly one outgoing transition, and its target state has exactly one incoming transition, the transition must use the next pointer.

Assuming $p, q, r \in Q$, and $a, b \in \Sigma$, we have:

$$|\{(p,a,q) : ((\forall_{(p,b,r)\in E} b = a, r = q) \wedge (\forall_{(r,b,q)\in E} b = a, r = s))\}| \leq \tau^{np}(A)$$

$$\tau^{np}(A) < |Q|, \quad \pi^{np}(A) = bytes(|E|) - 1$$

2.5 Tails of States

In section 2.3 we assumed that only entire states can share the same space as some larger states. When using the list representation (section 2.2), we can share only parts of states. We can have two or more states that share some but not all of their last transitions.

Let us consider a more complicated example (fig. 5). The transition number 4 holds in fact 3 identical transitions. The states reachable from the start state have both 2 transitions. One of those transitions is common to both states (the one with label a). The second one is different (either labeled with l or t). To avoid confusion, we did not use the *next* flag.

	L	F	S	→	↪
1	p			3	
2	s		•	5	
3	l			4	
4	a		•	6	
5	t			4	4
6	y	•	•	0	

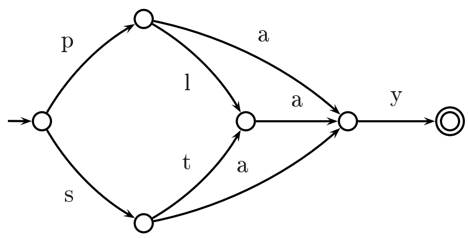

Fig. 5. Automaton recognizing words *pay*, *play*, *say*, and *stay*, and sharing last transitions of states. ↪ is a pointer to the tail of the state.

To implement tail sharing we need two things: a new flag (we call it the *tail flag*, not shown on the figure 5 as its value is implied), and an additional pointer occurring only when the tail flag is set. When the flag is set, then only the first transitions are kept, and the additional pointer points to the first transition of the tail shared with some other state. We need 1 bit for the flag, and we allocate a place for it in the bytes of transition pointers.

2.6 Changing the Order of Transitions

In the examples we showed so far, nothing was told about the order of transitions in a state. Techniques of transition sharing depend on the order of transitions. Automata construction algorithms (e.g. [3]) may impose initial ordering, but it may not be the best one. Kowaltowski *et al.* ([5]) propose sorting the transitions on increasing frequency of their labels. They also propose to change the order in each state individually. The order of transitions also influences the number of states that are to be considered as *next*. To increase the number of *next pointers*, we try to change the order of transitions in states that do not already have that compression. To increase transition sharing, we put every possible set of n transitions (starting from the biggest n) of every state into a register, and then look for states and tails of states that match them.

2.7 Other Techniques

There are other techniques that we have not experimented with. They include local pointers and indirect pointers. Local pointers are only mentioned in [8]. We can only stipulate that what they propose is 1-byte pointers for states that are located close in the automaton, and full-length pointers for other states. A flag is needed to differentiate among the two. Indirect pointers are proposed in US patent 5,551,026 granted August 27, 1996 to Xerox. By putting pointers to frequently referenced locations into a vector of full-length pointers, and replacing those pointers in transitions with (short) indexes in the vector, one can gain more space.

3 Experiments

3.1 Data

Our experiments were carried out on morphological dictionaries for German ([7]) and Dutch ([1]). The German morphological dictionary by Sabine Lehmann contains 3,977,448 entries. The automaton for the version with coded suffixes had 307,799 states (of which 62,790 formed chains), and 436,650 transitions. The version with coded suffixes, prefixes, and infixes had 107,572 states (of which 14,213 formed chains), and 176,421 transitions.

3.2 Results

Table 1 gives the size of automata built with various options. The *sg* automata with *SNTO*, *SNMT*, and *SNMTO* are bigger than expected because there was no space for one more flag in the pointer.

		O	N	NM	NO	NMO
sg	2,178,268	2,117,628	1,747,210	1,733,730	1,706,512	1,694,405
sgi	882,123	822,133	731,713	718,439	692,514	681,007
cd	3,028,893	2,914,528	2,369,485	2,331,985	2,287,664	2,257,565
cdi	2,873,143	2,758,473	2,255,753	2,221,047	2,183,489	2,147,982

	S	SO	SMO	SN	SNM	SNO
sg	1,742,616	1,720,460	1,703,376	1,311,558	1,298,078	1,300,592
sgi	705,700	682,932	668,084	555,290	542,016	544,696
cd	2,423,116	2,382,936	2,350,584	1,763,708	1,726,208	1,737,200
cdi	2,298,516	2,257,728	2,225,540	1,681,126	1,646,420	1,663,904

	SNMO	STO	STMO	SNTO	SNMT	SNMTO
sg	1,282,648	1,703,292	1,690,229	1,507,949	1,511,461	1,495,909
sgi	528,814	666,202	654,613	531,578	542,016	523,299
cd	1,697,742	2,939,005	2,904,985	1,985,335	1,983,531	1,959,747
cdi	1,619,334	2,779,723	2,745,299	1,902,887	1,894,999	1,876,403

Table 1. Size of automata built with various options, Sabine Lehmann's German morphology , and CELEX Dutch morphology, in bytes. In the table, *sg* means Sabine Lehmann's German morphology with coded suffixes, *cd* – CELEX Dutch, *i* – coded prefixes and infixes, *O* – shared transitions, *S* - stop bit (lists of transitions), *N* – next pointers, *T* – tails of states, and *M* – changing the order of transitions.

3.3 Conclusions

In case of German morphology, we managed to compress the initial automaton more than fourfold. With coded infixes and prefixes, we compressed the input data more than 696 times. Gzip compressed the input data (with coded infixes and prefixes) to 16,342,908 bytes. All automata for given input data could be made smaller by using compression by over 40% (43.9% for Dutch). The smallest automaton we obtained could still be compressed with gzip by 27.77%. The best compression method for German turned out to be a good preparation of the input data. It gave savings from 57.66% to 65.02%. For Dutch, those savings were only 4.15-5.50%, as words with prefixes and infixes constituted 3.68% of data, and not 22.93% as in case of German. As predicted, elimination of transition counters gave 20% on average. The figure was higher (up to 25.13% for German, 25.98% for Dutch) when next pointers were also used, as counters took proportionally more space in transitions. The figure was lower (16.93%) when only transition sharing was in use, because distributing a state over two other states was no longer possible. For German, next pointers gave savings from 15.77% to 24.70%, i.e. within the predicted range (3.22% –28.26%). For Dutch: 20.84% – 34.51%. The savings were bigger when the stop bit option was used. Surprisingly, transition sharing is less effective (0.84% – 3.01% on sg, and 1.91% – 7.24% on sgi, 0.98% – 4.45% on Dutch), and works better on sgi because sg contains many chains of states. Compression of tails of states adds only 0.71% to 2.45% for German, and does not work for Dutch data because the additional bit for a flag crosses the byte boundary. Changing the order of transitions gives small results (up to 2.92%). We managed to speed it up so that it takes a fraction of the construction time.

4 Acknowledgments

We express our gratitude to Sabine Lehmann for making her dictionary available to us.

References

[1] R. H. Baayen, R. Piepenbrock, and H. *The CELEX Lexical Database (CD-ROM)*. Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA, 1993, 1993.

[2] Jan Daciuk. *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing*. PhD thesis, Technical University of Gdańsk, 1998.

[3] Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, April 2000.

[4] George A. Kiraz. Compressed storage of sparse finite-state transducers. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, pages XIX–1 – XIX–31. 1997.

[5] Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. Minimization of binary automata. In *First South American String Processing Workshop*, Belo Horizonte, Brasil, 1993.

[6] Tomasz Kowaltowski, Cláudio L. Lucchesi, and Jorge Stolfi. Finite automata and efficient lexicon implementation. Technical Report IC-98-02, January 1998.

[7] Sabine Lehmann. Multext German morphology. ISSCO, Geneva, September 12 1996.

[8] Claudio Lucchiesi and Tomasz Kowaltowski. Applications of finite automata representing large vocabularies. *Software Practice and Experience*, 23(1):15–30, Jan. 1993.

[9] Dominique Revuz. *Dictionnaires et lexiques: méthodes et algorithmes*. PhD thesis, Institut Blaise Pascal, Paris, France, 1991. LITP 91.44.

[10] Max Silberztein. *Finite-State Language Processing*, chapter The Lexical Analysis of Natural Languages, pages 175–203. MIT Press, 1997.

[11] Max Silberztein. INTEX tutorial notes. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, pages XIX–1 – XIX–31. 1999.

[12] Robert Endre Tarjan and Andrew Chi-Chih Yao. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, November 1979.