Comparison of Construction Algorithms for Minimal, Acyclic, Deterministic, Finite-State Automata from Sets of Strings

Jan Daciuk

Alfa-Informatica, Rijksuniversiteit Groningen
j.daciuk@let.rug.nl

Abstract. This paper compares various methods for constructing minimal, deterministic, acyclic, finite-state automata (recognizers) from sets of words. Incremental, semi-incremental, and non-incremental methods have been implemented and evaluated.

1 Motivation

During last 12 years, one could see emergence of construction methods specialized for minimal, acyclic, deterministic, finite-state automata. However, there are various opinions about their performance, and how they compare to more general methods. Only partial comparisons are available. What has been compared so far was complete programs, which performed not only construction, but computation of a certain representation (e.g. space matrix representation, or various forms of compression).

The aim of this paper is to give answers to the following questions:

- What is the fastest construction method?
- What is the most memory-efficient method?
- What is the fastest method for practical applications?
- Do incremental methods introduce performance overhead?

The third question is not the same as the first one, because one has to take into account the size of data and available main memory. Even the fastest algorithm may become painfully slow during swapping.

2 Construction Methods

Due to lack of space, the construction methods under investigation have only been enumerated here. The reader is referred to the bibliography for proper descriptions. Some of the methods use a structure called the *register* of states. In those algorithms, states in an automaton are divided into those that have been minimized, i.e. they are unique in that part, and other states, i.e. those that are to be minimized. The register is a hash table, and two states are considered

J.-M. Champarnaud and D. Maurel (Eds.): CIAA 2002, LNCS 2608, pp. 255–261, 2003.

Table 1. Characteristics of data used in experiments

		strings			automaton	
		words	characters	av. len.	states	trans.
German	words	716 273	10 221 410	14.27	45 959	97 239
	morph.	3 977 448	364 681 813	91.69	107 198	435 650
French	words	210 284	2 254 846	10.72	16 665	43 507
	morph.	235 566	17 111 863	72.64	32 078	66 986
Polish	words	74 434	856 176	11.50	5 289	12 963
	morph.	92 568	5 174 631	55.90	84 382	106 850
`/usr/dict/words`		45 407	409 093	9.01	23109	47346

equivalent if they are either both final or both non-final, and they have the same transitions (the same number, labels, and targets). Methods 1, 3, and 4 require sorted data; the strings must be sorted lexicographically, lexicographically on reversals of strings, and on decreasing lengths respectively. The following methods have been investigated:

1. Incremental construction for sorted data ([2]).
2. Incremental construction for unsorted data ([2]).
3. Semi-incremental construction by Bruce Watson ([7]).
4. Semi-incremental construction by Dominique Revuz ([4]).
5. Building a trie and minimizing it using the Hopcroft algorithm ([3], [1]).
6. Building a trie and minimizing it using the minimization phase from the incremental construction algorithms (postorder minimization).
7. Building a trie and minimizing it using the minimization phase from the algorithm by Dominique Revuz (lexicographical sort [5]).

3 Experiments

Data sets for evaluation were taken from the domain of Natural language Processing (NLP). Acyclic automata are widely used as dictionaries. Both word lists, and morphological dictionaries, for German, French, and Polish, as well as a word list for English were used. Word lists and morphological dictionaries have different characteristics. Strings in word lists are usually short, sharing short suffixes. Strings in morphological dictionaries are much longer, with long suffixes shared between entries. The data is summarized in Table 1.

All methods were implemented in a single program, with data structures and most functions shared among different algorithms. Unfortunately, there is not enough space in a short paper to describe the implementation in detail. In the experiments, the hash function had 10001 possible values. The register was implemented with an overflow area for each hash value being a set class from the C++ standard template library - a tree-like structure.

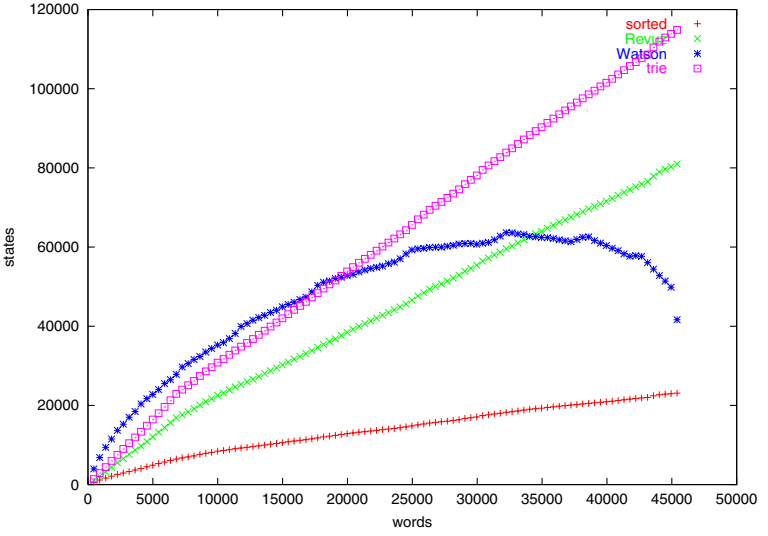

Fig. 1. Memory requirements for English words.

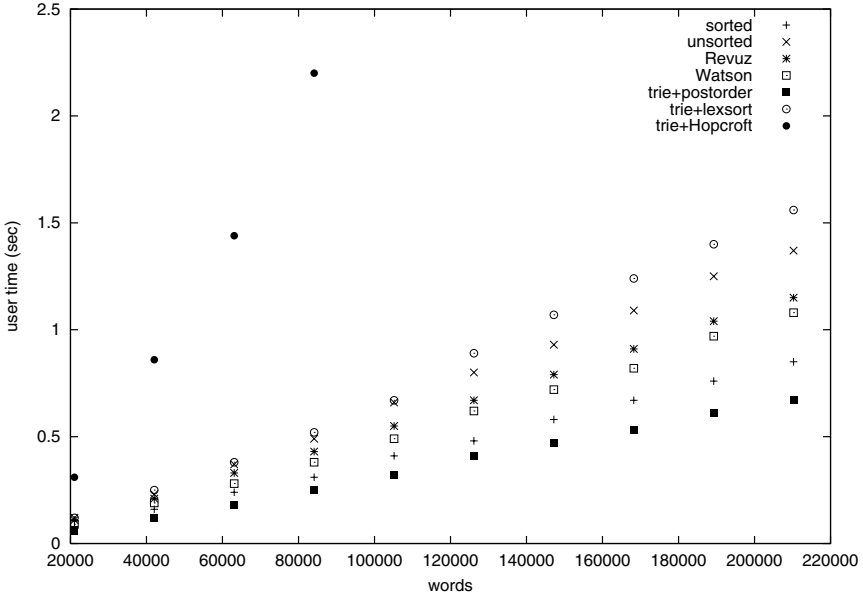

Fig. 2. Execution time for Polish words

Figure 1 shows how the number of states grows during construction of an automaton for a representative data set using various algorithms. The points labeled "trie" represent non-incremental methods. Memory requirements for construction algorithms are proportional to the number of states of the largest au-

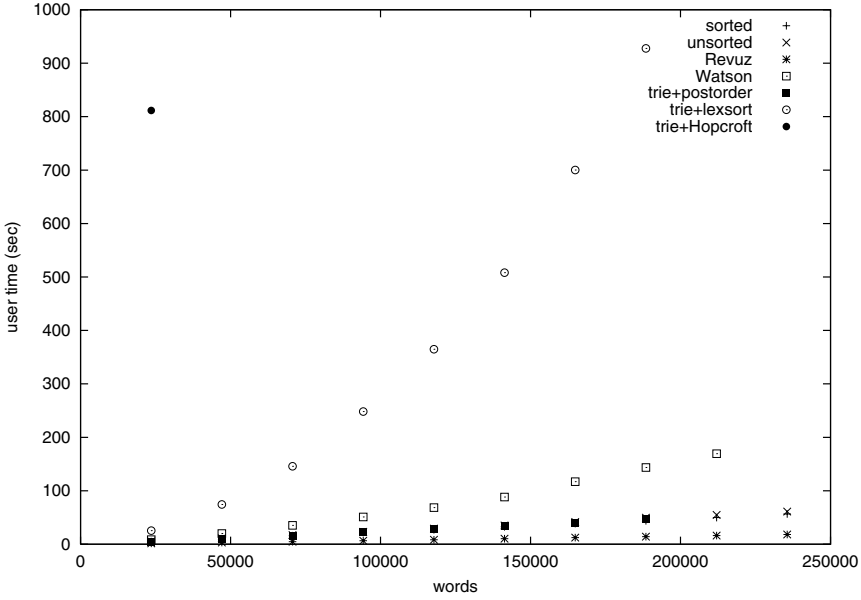

Fig. 3. Execution time for French morphology

tomaton during construction. The diagram shows that only incremental methods keep the automaton minimal throughout the process – other methods require memory for additional redundant states before they arrive at the minimal automaton. The intermediate automaton for non-incremental methods can be much larger than minimal. Memory requirements for the incremental method for unsorted data are identical to those for the sorted method on the same data, and only slightly higher for data sorted for other methods. The Watson's algorithm displays an unusual behavior. The largest number of states is achieved approximately half way during the construction process. This phenomenom is caused by two factors: sorting of input data (from longest strings to the shortest ones), and minimization scheme (prefixes cause minimization of larger words). Initial memory requirements for Watson's algorithm are higher than for a trie made from data sorted lexicographically, as longer words come first. They become lower towards the end of data, as shorter words trigger minimization.

To test the relation between the speed, and the size of the data, each algorithm was tested on 0.1, 0.2, ... 0.9, and on the whole data. In case of Revuz's algorithm, and Watson's algorithm, those parts were sorted accordingly, instead of taking the same number of words from the beginning of the whole file sorted according to the requirements. This is different than measurements of memory requirements, because they were all taken during a single run on the whole appropriate file. Also, due to multiuser, multitask unix environment, only processor times were measured, not the elapsed "real" time. It means that the effects of swapping do not show up on diagrams. Only initial values for trie + Hopcroft

minimization algorithm are shown to underline differences between other algorithms. For most data, the trie + postorder minimization method was the fastest. It was slightly faster than the algorithm for sorted data, and in some cases their values are not distinguishable on diagrams. For morphological dictionaries, Revuz's algorithm was faster (Fig 3). This happens because in that data very long common suffixes were present. The INTEX program [6] uses Revuz's algorithm without pseudo-minimization phase to save both time and disk space, but annotations are kept short, and their expansions are kept elsewhere.

4 The Fastest Algorithm

Surprisingly, the fastest construction algorithm is not yet described in literature. This is probably due to its simplicity. We define a deterministic finite state automaton as $M = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the alphabet, $\delta \in Q \times \Sigma \longrightarrow Q \cup \{\perp\}$ is the transition function, q_0 is the initial state, and $F \subseteq Q$ is the set of final states. A somewhat formally awkward notation of assignment to the delta function in the algorithm below means creating or modifying a transition. This algorithm has exactly the same complexity as both incremental algorithms from [2].

```
func trie_plus_postorder_minimization;
    start_state := construct_trie; Register := ∅;
    postorder_minimize(start_state);
    return start_state;
cnuf

func construct_trie();
    start := new state;
    while file not empty
        word := next word form file; i := 0; s := start;
        while i < length(word)
            if δ(s, word_i) ≠ ⊥ → δ(s, word_i) := new state; fi
            s := δ(s, word_i); i := i + 1;
        elihw
        F := F ∪ {s};
    elihw
cnuf

proc postorder_minimize(s);
    foreach a ∈ Σ : δ(s, a) ≠ ⊥
        postorder_minimize(δ(s, a));
        if ∃_{q∈Register} δ(s, a) ≡ q → δ(s, a) = q;
        else Register := Register ∪{s}; fi
    hcaerof
corp
```

5 Conclusions

- For unsorted data, the trie + postorder register-based minimization algorithm is the fastest, provided that we have enough memory to use it. The difference between the minimal automaton and the corresponding trie can be huge.
- All but incremental methods produce intermediate automata much larger than the minimal ones. All alternatives to the incremental algorithm for unsorted data build a trie first – the worst possible case from the point of view of memory efficiency. Therefore, the incremental algorithm for unsorted data is the fastest algorithm for unsorted data in practical applications, and in fact the only algorithm for that purpose.
- For typical sorted data, the trie + postorder register-based minimization algorithm can be used, but as it builds a trie, it requires huge amounts of memory. The incremental algorithm for sorted data can be used instead with almost no performance penalty. For sorted data where strings share long suffixes, like in certain morphological data, Revuz's algorithm is the fastest. However, such data can easily be transformed so that the long suffixes are stored separately (as it is done e.g. in INTEX). For the transformed data, Revuz's algorithm is no longer the fastest one. It also requires non-standard sorting that cannot be performed efficiently using ready-made programs. Moreover, tools for constructing natural language morphologies have additional data that can be used for faster construction algorithms. The author has implemented such an algorithm.
- Both fully incremental algorithms are the most economical in their use of memory. They are orders of magnitude better than other, even semi-incremental methods. Even for a very small data set, like the one presented on Figure 1, the intermediate automata in semi-incremental methods are 3-4 times larger than the minimal ones.
- Both incremental algorithms are the fastest in practical applications, i.e. for large data sets. The algorithm for sorted data is the fastest, but if data is not sorted, sorting it (and storing it in memory) may be more costly than using the algorithm for unsorted data. However, when the same data is used repeatedly, sorting is always beneficial.
- It seems that incremental algorithms do not introduce much overhead when compared to non-incremental methods. The differences between the incremental sorted data algorithm and its non-incremental counterpart are minimal. The non-incremental version of the semi-incremental Revuz's algorithm (trie + lexical sort) is sometimes faster than the original version for words, and always slower for morphologies.
- Trie + Hopcroft minimization is the slowest algorithm. While all other algorithms are linear, this one has an additional $\mathcal{O}(\log(n))$ overhead, and it is quite complicated compared to register-based algorithms.

Acknowledgements. The outline of experiments was discussed with Bruce Watson. This research was carried out within the framework of the PIONIER

Project *Algorithms for Linguistic Processing*, funded by NWO (Dutch Organization for Scientific Research) and the University of Groningen. The program used in the experiments is available from
`http://www.eti.pg.gda.pl/~jandac/adfa.html`.

References

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Publishing Company, 1974.
2. 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.
3. John E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *The Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.
4. Dominique Revuz. *Dictionnaires et lexiques: méthodes et algorithmes*. PhD thesis, Institut Blaise Pascal, Paris, France, 1991. LITP 91.44.
5. Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theoretical Computer Science*, 92(1):181–189, 1992.
6. Max Silberztein. INTEX tutorial notes. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, pages XIX–1 – XIX–31. 1999.
7. Bruce Watson. A fast new (semi-incremental) algorithm for the construction of minimal acyclic DFAs. In *Third Workshop on Implementing Automata*, pages 91–98, Rouen, France, September 1998. Lecture Notes in Computer Science, Springer.