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 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 sketched 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 equivalent if they are either both final or both non-final, and they have the same transitions (the same number, labels, and targets).

The following methods have been investigated:

1. Incremental construction for sorted data ([2]). Input strings are lexicographically sorted. For each string, the longest common prefix with the last string added to the automaton is found. If the last state in the path recognizing the common prefix has outgoing transitions, a path consisting in last transitions of subsequent states is followed, and then, starting from the end of that path up to, but excluding, the last state in the common prefix path, if an equivalent state is found in the register, it replaces the current state, and the current state is deleted. Afterwards, the remaining part of the string is added, creating a chain of states attached to the last state of the path of the common prefix. After the last string has been added, the path of that string is also compared against states in the register and replaced if necessary.
2. Incremental construction for unsorted data ([2]). Input strings come in arbitrary order. For each string, the longest common prefix with any string in the automaton is found. If any state in the common prefix path is reentrant, that state and all states following it in the path are cloned, the first state before any cloned state, or the last state in the common prefix path, is removed from the register. The remaining part of the string is added, creating a chain of states attached to the last state of the path of the common prefix. Afterwards, starting from the end of the chain, if an equivalent state is found, it replaces the current state. The process continues past the appended chain towards the initial state removing states from the register and adding new states to it, until no replacement is made.
3. Semi-incremental construction by Bruce Watson ([6]). Input strings are sorted on decreasing length. For each string, the longest common prefix with any string in the automaton is found. If the string is a prefix of any string already in the automaton, all states reachable from the end of the prefix path (including the last state in that path) and not in the register are put onto stack so that states with smallest height are on top. Then for successive states popped from the stack, if the register contains en equivalent state, it replaces the current state.

After the last word has been added, the remaining states are put onto stack and then replaced if needed.
4. Semi-incremental construction by Dominique Revuz ([4]). Input strings are lexicographically sorted on their *reversal*. A trie-like structure is built, but the longest common suffix between the current string, and the last string added to the automaton is computed. Whenever possible (states have only one outgoing transition, they are not final), the path (or its part) recognizing the common suffix is shared. That phase does not lead to the minimal automaton, so it is followed by true minimization. States are sorted on their height, and then states of the same height are sorted using bucket sort, and redundant states are removed.
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 semi-incremental algorithm by Dominique Revuz (lexicographical sort).

		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

Table 1. Characteristics of data used in experiments

3 Data for Evaluation

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.

Because of huge amounts of memory needed to represent a trie, only sufficiently small parts of data were used for non-incremental methods.

4 Experiments

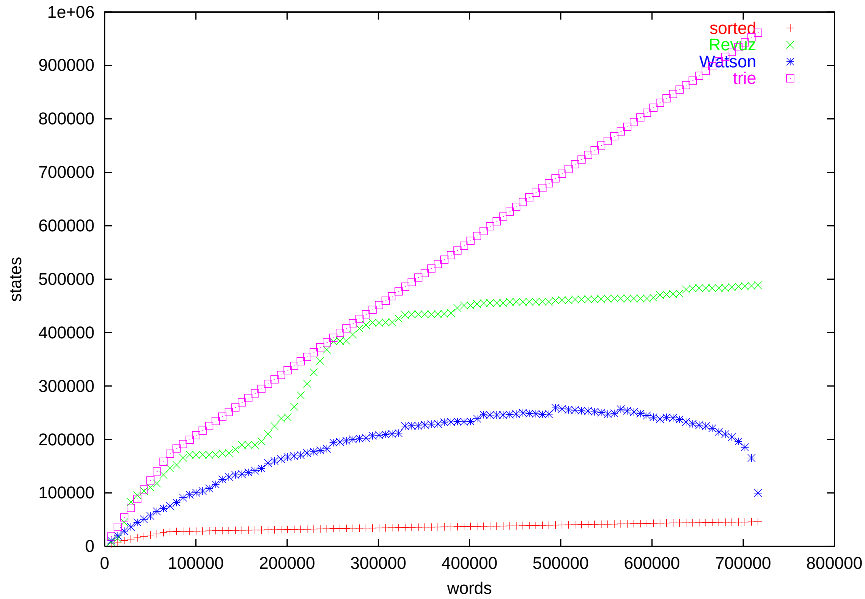

Fig. 1. Memory requirements for German words.

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.

Several characteristics were measured. They included: execution time, number of states during construction: (maximal number of states, number of states as a function of the number of words during construction), use of the register: (number of calls to find an equivalent state in the register, number of calls to remove a state from the register (only for the incremental construction from unsorted data), number of states with the same hash value per call, register fill ratio).

5 Results

5.1 Memory Requirements

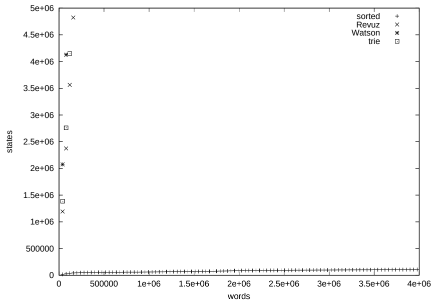

Fig. 2. Memory requirements for German morphology

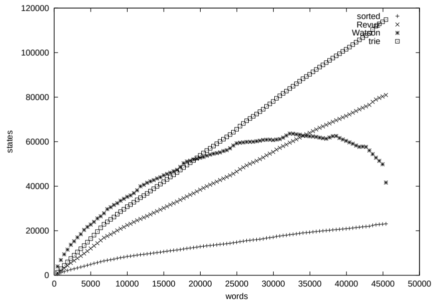

Fig. 3. Memory requirements for English words from `/usr/dict/words`.

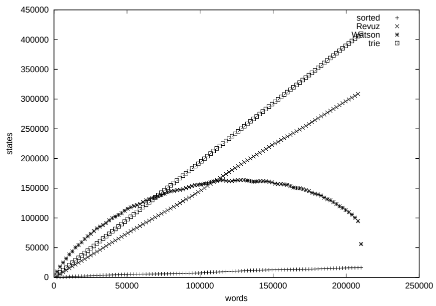

Fig. 4. Memory requirements for French words.

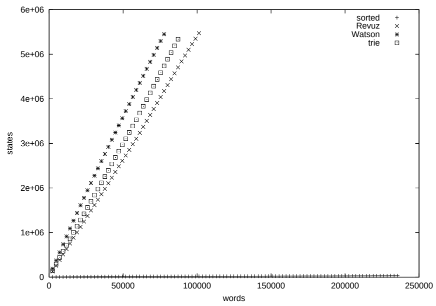

Fig. 5. Memory requirements for French morphology

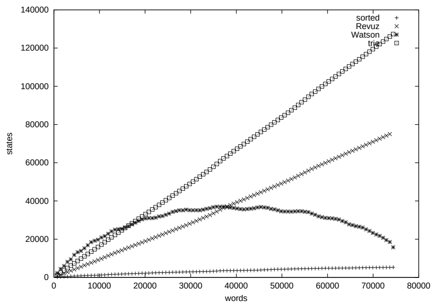

Fig. 6. Memory requirements for Polish words.

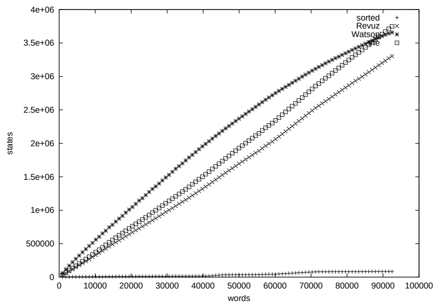

Fig. 7. Memory requirements for Polish morphology.

As it is difficult to trace the exact memory requirements of programs under unix, the number of states of an automaton during construction was measured. The states were represented by the same structure for all algorithms, although certain fields were specific to some algorithms, like i.e. height was specific to Revuz's algorithm. Memory was also used to store transitions. Programs that used register required a constant amount of memory for an empty register, and an amount proportional to the number of states. In addition, Watson's algorithm required memory for a stack of states. Revuz's algorithm did not use the register, but it needed memory for sorting (at least proportional to the number of states). Using Hopcroft's algorithm for minimization of a trie also required memory proportional to the number of states, but much more than a register of states.

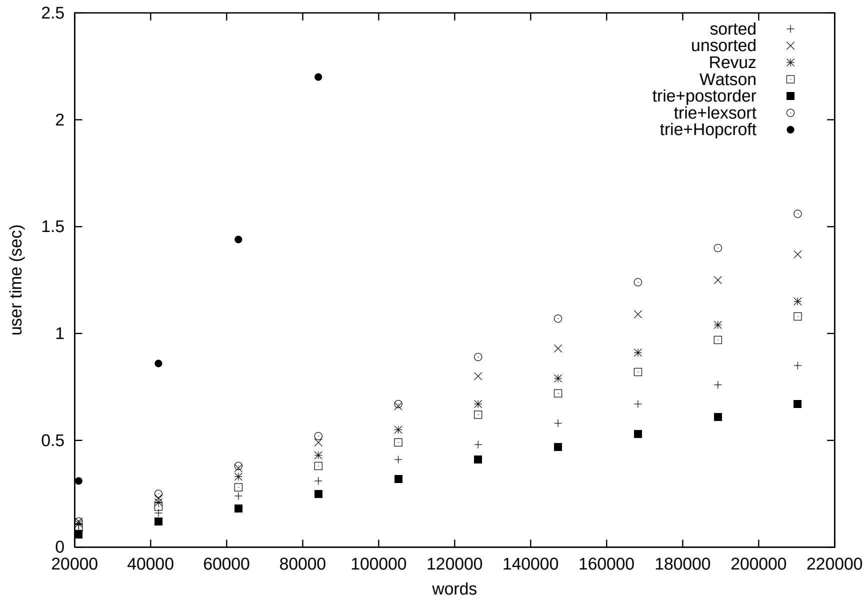

Fig. 8. Execution time for Polish words

Figure 1 is representative for memory requirements for various methods. The points labeled "trie" represent non-incremental methods. 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. 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 word data, as shorter words trigger minimization. That effect is hard to be noticed for morphological dictionaries.

5.2 Execution time

Due to memory memory limitations, and sometimes extrememly long execution time, it was impossible to measure the speed of some algorithms on all data. 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. This happens because in that data very long common suffixes were present. The INTEX program [5] 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.

6 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].

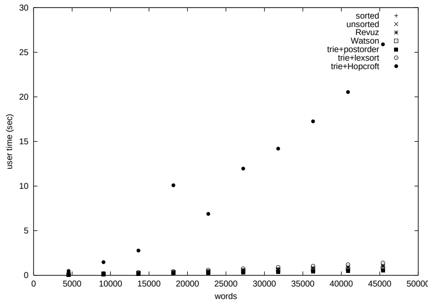

Fig. 9. Execution time for English words.

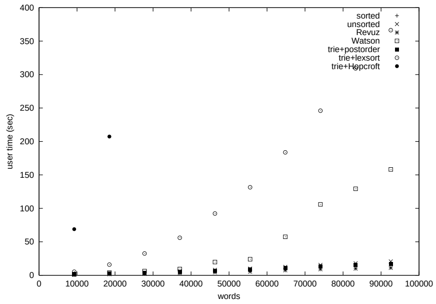

Fig. 10. Execution time for Polish morphology.

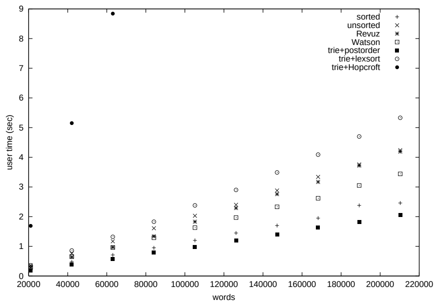

Fig. 11. Execution time for French words.

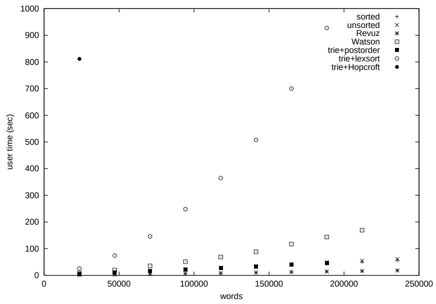

Fig. 12. Execution time for French morphology

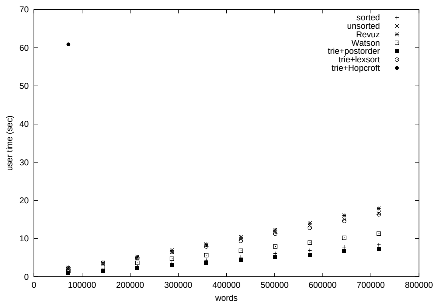

Fig. 13. Execution time for German words.

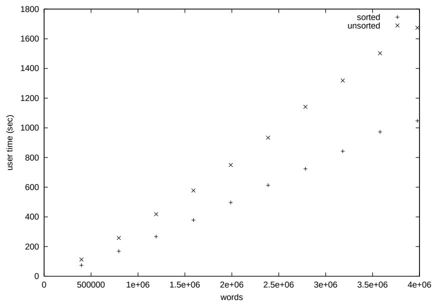

Fig. 14. Execution time for German morphology.

func trie_plus_postorder_minimization;
 start_state := construct_trie;
 Register := $\emptyset$; 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
```

7 Conclusions

- The incremental algorithm for sorted data, and the trie + postorder minimization algorithm are the fastest. The incremental algorithm should always be used for sorted data. For unsorted data, the trie + postorder minimization algorithm is the fastest provided enough memory is available. Revuz's algorithm is the fastest for morphologies.
- 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.
- 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.

- It seems that incremental algorithms do not introduce 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 faster than the original version for words, and 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.
- There is no real reason to use Revuz's algorithm, Watson's algorithm, trie + lexicographical sort (Revuz's algorithm without suffix compression), or trie + Hopcroft minimization algorithm. They are slower, and they have higher memory requirements, than incremental algorithms.

8 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. Max Silberztein. INTEX tutorial notes. In *Workshop on Implementing Automata WIA99 – Pre-Proceedings*, pages XIX–1 – XIX–31. 1999.
6. 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.