Many-MADFAct: Concurrently Constructing MADFAs

Tobias Runge[1], Ina Schaefer[1], Loek Cleophas[2,3], and Bruce W. Watson[2,4]

[1] Software Engineering, TU Braunschweig, Germany
[2] Information Science, Stellenbosch University, South Africa
[3] Software Engineering Technology, Eindhoven University of Technology, The Netherlands
[4] Centre for Artificial Intelligence Research, CSIR, South Africa
{tobias.runge,i.schaefer}@tu-bs.de, {loek,bruce}@fastar.org

Abstract. Minimal acyclic deterministic finite automata (MADFAs) are used to represent dictionaries, i.e., finite sets of finite words, in, e.g., spell checkers and network security applications. Given the size of such dictionaries, which may contain millions of words, their efficient construction is a critical issue. Watson [31] published a classification of such algorithms in an algorithm taxonomy with correctness arguments. We report on a new algorithm which constructs MADFAs in parallel—each for a keyword set from a partition of the original keyword set—and afterwards merges and minimizes the resulting automata into a single MADFA; on our experience implementing the algorithms in a Java-based toolkit; and on empirical performance results obtained.

Keywords: minimal acyclic deterministic finite automata; dictionaries; algorithms; toolkits; benchmarking

1 Introduction

Minimal acyclic deterministic finite automata (MADFAs) are frequently used to represent dictionaries, i.e., finite sets of finite words, in, e.g., spell checkers, network security, packet filtering applications, and other tools. Given the size of such dictionaries, which may run into millions of words, their efficient construction is a critical issue [31]. Acyclic Deterministic Finite Automata (ADFAs) consist of a set of states, one of them being a start state, and one or more of them being final states; and labeled transitions between states. They are deterministic, i.e., each state has at most one out-transition for a specific label; and they are acyclic, i.e. as their name says, no transition cycles occur. An ADFA is a MADFA if no other ADFA with fewer states accepts the same set of words. This makes MADFAs an excellent data structure to store large finite word sets like dictionaries. As a result, quite some research has gone into MADFA construction algorithms (see Section 2). Yet, no coherent implementation covering all these algorithm variants exists.

In this paper, we make the following contributions: we provide a two-dimensional presentation of Watson's implicit taxonomy of sequential MADFA construction algorithms [31], as well as an implementation of the seven sequential MADFA algorithms from that taxonomy in a Java-based toolkit. Furthermore, we develop a new parallel approach to MADFA construction, offering a versatile family of algorithms for MADFA construction in contexts where concurrent processing is available or preferred. Using one of the existing, sequential algorithms, the new algorithm constructs MADFAs in parallel—one for each of a partition of the original keyword set—and afterwards merges and minimizes the resulting automata into a single MADFA. The merger process is newly implemented, but the minimization step is the same as that

Tobias Runge, Ina Schaefer, Loek Cleophas, Bruce W. Watson: Many-MADFAct: Concurrently Constructing MADFAs, pp. 126–142.
Proceedings of PSC 2017, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06193-0 © Czech Technical University in Prague, Czech Republic

in one of the existing, sequential MADFA algorithms. Finally, we provide the results of benchmarking the algorithms to evaluate their performance relative to each other.[1] The work reported here contrasts with related work as follows: typically, there are several implementations of each of the known algorithms (see the next section), but previously only one publicly available comprehensive toolkit (that by Jan Daciuk, available at `www.jandaciuk.pl/fsa.html`); our contribution is another such toolkit, built from a uniformly styled presentation of the algorithms [31]; previously, there have also been few comprehensive benchmarks where the algorithms are implemented in the same style and emperically compared against each other.

2 Related work and a short history

The following history is distilled from [31], which is until now the most comprehensive such collection of derivations of MADFA construction algorithms. Jan Daciuk maintains implementations of many MADFA construction algorithms and has authored what is arguably the most comprehensive work on optimization, minimization and implementation/engineering issues as related to automata [9] in addition to his extensive algorithmic work in this field (detailed below).

Before the 1990s, some MADFA construction algorithms may have been known and used in proprietary (commercial, trade-secret) software. The first efficient (linear time and space) algorithm was published by Dominique Revuz in the early 1990s [21,22]. Revuz's main algorithm uses an ordering of the words to quickly compress the endings of the words within the dictionary. Recent derivations by Johannes Bubenzer and Thomas Hanneforth have yielded efficient new algorithms bearing a resemblance to Revuz's [3]. These algorithm variants are essentially what appears as *Algorithm Trie* in Fig. 1; in that figure, *Algorithm General* is a generalized version of Revuz's algorithm, first presented in [31].

By the mid-1990s, several groups were working independently on *incremental* MADFA construction algorithms. In 1996–1997, Jan Daciuk derived several incremental algorithms as part of his PhD work [8]: one relying on the words being in lexicographic order. In 1996, Richard and Bruce Watson derived a generalized incremental algorithm, which included the possibility of incrementally removing words while maintaining minimality; owing to its commercial value, the algorithm was not published at that time. Collaboration between Daciuk, Watson & Watson led to [11]. More or less concurrently, Stoyan Mihov PhD work derived parts of the same algorithms [18], and further collaboration yielded [10] by Daciuk, Mihov, Watson & Watson. In the domain of pattern matching, Park *et al* derived a similar algorithm [19], while in program verification, Gerard Holzmann and Anuj Puri [14] discovered a restricted form of the algorithm, in which all words accepted by the automaton are the same length. In early 2000, Daciuk unearthed the derivational work of Sgarbas *et al* (an incremental algorithm [24]) and Marcin Ciura and Sebastian Deorowicz (lexicographic order algorithm, including some benchmarking [5]). Also in 2000, Revuz presented essentially the generalized algorithm [23]—though he also sketched word deletion algorithms similar to those previously derived by Watson & Watson. Jorge Graña *et al* subsequently summarized some of the current results and made improvements to several of the algorithms [13]. The generalized algorithm has also been extended by Rafael Carrasco and Mikel Forcada to handle *cyclic* automata [4]. In

[1] We are not focusing on absolute performance or on further tuning of the algorithms.

this paper, the generalized incremental algorithm is *Algorithm Incremental* in Fig. 1, while the sorted-input algorithm is *Algorithm Sorted* in the same figure. An alternative sorted-input algorithm (based on arbitrary sortings of decreasing lengths of the words) was developed in [31, Chapter 10], and appears in Fig. 1 as *Algorithm Depth Layered*.

In 1998, Watson derived a semi-incremental MADFA construction algorithm [29]. Such an algorithm does a form of pseudo (or *near*) minimization incrementally as words are added; after all words are added, a final 'cleanup' phase is required to reach true minimality. This is *Algorithm Semi-Incremental* in Fig. 1. In the same year, Watson used Brzozowski's minimization algorithm to give an elegant MADFA construction algorithm in [27,28] (which maps to *Algorithm Reverse* in Fig. 1). Concurrently, Watson derived a simple recursive algorithm in [30]; that algorithm does not appear separately in this paper's work, as it is a variant of *Algorithm Incremental*. Aside from [31], to which this paper relates, early taxonomies/classifications appeared [26].

3 Taxonomy and Algorithms

Algorithm taxonomies hierarchically structure algorithms stemming from a domain, in order to facilitate comparison and emphasize algorithms' similarities. The root of such a classification is formed by an abstract algorithm, and branches refine a parent algorithm into more concrete child algorithms. As such, by proving or at least considering the correctness of each such branch or refinement, one can prove or convince oneself of the correctness of each and every algorithm in the taxonomy.

Algorithm taxonomies have been in use since at least the 1970s; for example, Darlington [12] and Broy [1] classified different sorting algorithms, while Jonkers [15] classified garbage collection algorithms, and did so with an emphasis on correctness. Building on Jonkers' style, Marcelis considered attribute evaluation algorithms [16], while Watson presented taxonomies of string pattern matching and automata related algorithms [25]. Cleophas [6] similarly treated tree pattern matching and automata. Pieterse, going beyond just taxonomies, recently published a thesis on the use of topic maps for structuring algorithmic knowledge, including a topic map and taxonomy of transitive closure algorithms [20].

Algorithm taxonomies can also form the starting point for the development of implementations, as is done in the TABASCO method [7] where toolkit implementations of the taxonomised family are derived from the taxonomies. The taxonomies' structure guides that of a corresponding toolkit, including ensuring reuse of common algorithm parts and hence common implementation parts. The correctness arguments contained in the taxonomies provide confidence in the correctness of the implementations as well.

3.1 A two-dimensional taxonomy of MADFA construction

The seven known MADFA construction algorithms have been classified hierarchically in a taxonomy, to facilitate comparison, highlight similarities, and reason about correctness [31]. The root represents an abstract model of the algorithms, with methods *add_word* to add an individual word—resulting in a not necessarily minimal ADFA—and *cleanup* to ensure the final result is again a MADFA. The method *add_word* is called for every word in the set of input words, after which a call to *cleanup* minimizes the ADFA to a corresponding MADFA. Some algorithms do not follow the separation between adding words and *cleanup*; these algorithms partially minimize the automaton during *add_word* and need just a single, final call to *cleanup*. In Figure 1, we show a taxonomy graph, conceptually representing the MADFA construction taxonomy that was left somewhat implicit in [31]. The algorithms are depicted as circles. They always have one link to an *add_word* method and one to a *cleanup* method. Both methods are depicted as rectangles. The connectors between the methods show the hierarchy. Algorithm-Skeleton has the two abstract methods *add_word*-Skeleton and *cleanup*-Skeleton; and both are refined by the specific algorithms.

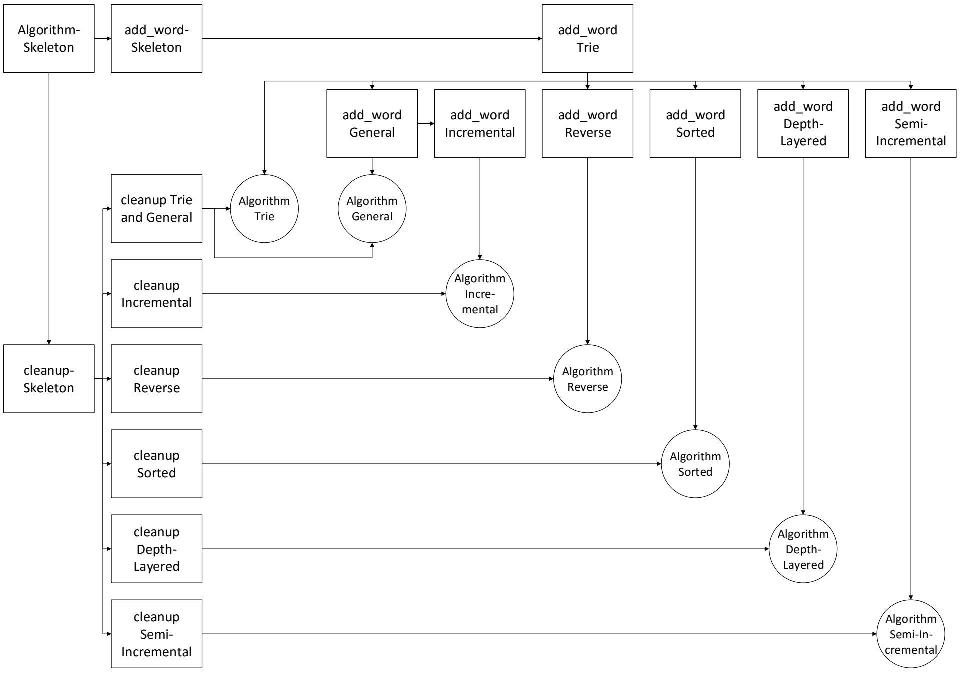


Figure 1. Taxonomy of the sequential MADFA construction algorithms

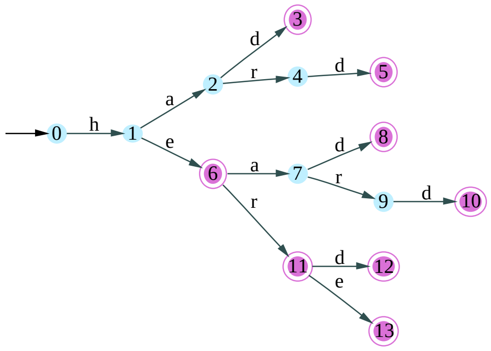
The method *add_word* is classified as follows. *add_word* of Algorithm *Trie* is the base for all the other algorithms. This method adds words by adding a path for every new word. The result is a trie. The Algorithm *General* extends the process such that the method is applicable for arbitrary ADFAs instead of just tries. The other four algorithms directly connected to Algorithm *Trie's add_word* method extend this *add_word* method with a minimization step. Finally, the Algorithm *Incremental* inherits from *General* directly and also adds a minimization step to this method. For the method *cleanup*, we can not find many commonalities between the different variants of the algorithms. All the considered algorithms have different *cleanup* methods, except for Algorithm *Trie* and *General*, which both use the same *cleanup* method. As a result, the method *add_word* is related in each algorithm and can be refined hierarchically, but the method *cleanup* differs between most algorithms.

As we will see in Section 3, a novel, parallel approach was developed which uses parallel threads to construct MADFAs and finally merge them into a single MADFA. This approach forms a whole new family of MADFA construction algorithms, as in each of the parallel MADFA construction threads, any of the algorithms from the above taxonomy can be used. The algorithm family corresponding to this parallel approach is not shown in Figure 1 because the parallelism is orthogonal to the algorithmic solution strategies of the algorithms shown in the figure. In the parallel algorithm, two or more threads call one of the preceding seven algorithms to construct a MADFA in parallel—each for a keyword set such that these sets form a partition of the original keyword set. The chosen algorithm may even be different per thread, since each such algorithm guarantees a MADFA to be constructed. After the construction of the separate MADFAs, a merge method merges these MADFAs into a single ADFA. This ADFA is then minimized with a call to the *cleanup* method of Algorithm Trie, as this *cleanup* method can be used for arbitrary ADFAs.

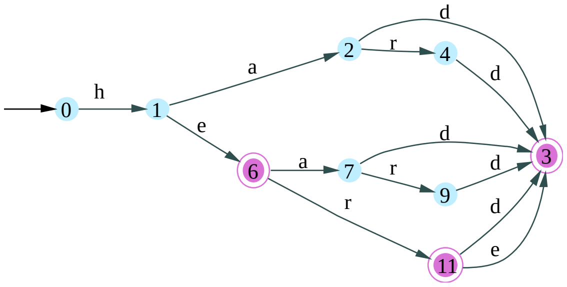
3.2 Algorithms

We briefly discuss four algorithms (Algorithms *Trie*, *Incremental*, *Reverse*, and *Semi-Incremental*) to give some insight into the behavior of MADFA construction algorithms. (More extensive examples of all these algorithms in action can be found throughout [31].) Algorithm *Trie* can be seen as the base algorithm for all the other ones. It realizes the abstract methods *add_word* and *cleanup*. At first, all words are added one by one by calling the method *add_word*. This is done by traversing the automaton according to the word under consideration, until no out-transition is found for a specific letter of the word; then the automaton is extended with new states and transitions so that it accepts the word. The result is a trie, which is then minimized to a MADFA with the help of *cleanup* [31]. This *cleanup* method is a Revuz-like algorithm [22]: it merges equivalent states of the automaton in order of decreasing height level. (A height level is a set of states which have the same length of their longest path to any final state.) The method starts with the leaves of the trie and ends at the root. Two states are equivalent if they have the same right language, i.e., they have the same set of words leading to final states. If that is the case, the states can be merged. During the merge, one state is deleted and its transitions are redirected to another state. If no more states are equivalent, no states can be merged, and therefore the automaton is minimal, i.e., in our context is a MADFA.

In Figure 2 we show an example of how this algorithm works. Firstly, all words are added. In this example the order of the words is lexicographic, i.e. had, hard, he, head, heard, her, herd, here. For every word, the automaton is traversed, and if necessary, new transitions and states are added. To give an example, ‘head’ is added after ‘he’ and before ‘her’ etc., i.e., state 6 is final and has no out-transitions before ‘head’ is added. During *add_word*, the automaton is traversed to state 6, following the letters ‘h’ and ‘e’. Now, we are at a state with no out-transition for the next symbol, ‘a’. We need to add a transition ‘a’ to a new state 7 and from there we add a transition ‘d’ to a new final state 8. Such a process is followed for every word. The result is the ADFA in Figure 2a. To minimize the automaton, Algorithm *Trie* computes height levels and merges equivalent states. In this example the first height level is the set of all states that have no out-transitions, i.e. their longest path to a final state is 0. The states $3, 5, 8, 10, 12, 13$ belong to this set. Every state is equivalent to the other states because every state is final and has no out-transition. That is why all states are merged into state 3 in Figure 2b. The next height level consists of all states with a longest path-length of one to a final state. This set includes $4, 9, 11$. Again, equivalent states are merged (not depicted), i.e. state 4 and 9 are merged, while 11 is not (both because it differs from 4 and 9 in out-transitions, and because it does so in its finality). Afterwards, the height level with a path-length of two is created and equivalent states in it are merged, and so on, until the resulting automaton is a MADFA.



(a) Trie after adding words



(b) First minimization step

Figure 2. Example for Algorithm Trie

The other six MADFA construction algorithms use a similar method to add words but with some specific extensions. Algorithm *Incremental* for example minimizes the automaton directly after each word is added. States on the newly added path are compared with the other states of the automaton and equivalent states are merged. The comparison starts at the end of the path and ends at the start state. *cleanup* only is a skip statement since the automaton is minimized during *add_word* [8,17]. A characteristic of this algorithm is that sometimes states have to be cloned, so that the automaton stays correct.

Algorithm *Reverse* is different from the other ones, in that *add_word* adds words in reverse order compared to the *add_word* method of Algorithm *Trie*. The resulting ADFA is a trie for the reverse of the words; because of that, *cleanup* must reverse and determinize the whole ADFA to obtain a MADFA. (In essence, this is a specialization of Brzozowski's classical result for DFA minimization [2]).

Algorithm *Semi-Incremental* also uses the *add_word* method of Algorithm *Trie*, but it adds words in order of decreasing length; hence the final state added by calling *add_word* is never visited again and all successors of this state can already be considered for merging. These are all the states that are compared with other final states and their successors. This is done during the *add_word* method. The method *cleanup* visits the last non-considered successor states of the start state, which are all states that do not have a predecessor final state [31]. The number of states compared by the *cleanup* method depends on the input word set. In Figure 3 we give an example. We add the word 'herd' after 'heard' because 'herd' is shorter than 'heard'. The new states 6 and 7 are added, the second being final. The result is the upper automaton in Figure 3. *add_word* starts to merge afterwards. The new final state and its successors are compared against all other final states and their successors. In this example, state 5 and 7 are compared. They are equivalent and because of that they are merged into one state. The result is the second automaton in Figure 3. The next step of this algorithm is to add other words and merge final states until the entire input set has been processed.

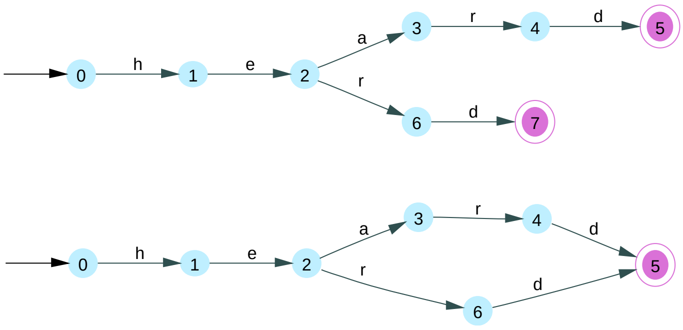


Figure 3. Example Algorithm Semi-Incremental: Adding word 'herd'

4 Parallel MADFA Construction

The seven MADFA construction algorithms [31] included in our taxonomy construct MADFAs sequentially. We present a novel approach to MADFA construction here, based on constructing multiple MADFAs in parallel, such that their keyword sets form a partition of the original keyword set; and then merging these MADFAs and ensuring the result is a single MADFA for the original keyword set. That is, we generate MADFAs in two or more threads and merge them afterwards; as this merger in general may provide an ADFA yet not a MADFA, it must be minimized again to obtain the final single MADFA for the original keyword set.

The new algorithm family forks threads which are then used to create multiple MADFAs, one per thread; and it joins them again once the threads are done. The construction of multiple MADFAs does not require much synchronization. Every call to a method is independent of other calls if both method calls operate on different automata. In our case, we opt to ensure that every state, across the MADFAs created in parallel, gets a unique id, so the access to the id counter is synchronized. The unique id facilitates the merge of automata because every state in the merged automaton can be attached to one input automaton, and the merged automaton does not include states with the same ids. The only point where we need to synchronize threads, therefore, is the creation of states. (This synchronisation has no substantial impact on the total running time, as the observed time for the complete parallel MADFA construction in our experiments was around one twentieth of the time taken for the final merger and minimization.)

We have implemented, two instantiations of the general approach described above. The first version creates two MADFAs in parallel, while the second approach creates four MADFAs. Conceptually, the two algorithms work as follows:

1. Split keyword set into 2 or 4 parts, respectively.
2. Create a thread for each of these parts, and use each such thread to create a MADFA for a particular part, using one of the seven sequential MADFA construction algorithms.

3. Merge the 2 or 4 MADFAs obtained into a single ADFA, using the classical product construction for the union of multiple automata.
4. For minimization, run the *cleanup* method of Algorithm *Trie* on the resulting ADFA, yielding a final MADFA for the original keyword set. We use this particular *cleanup* because it can be used for arbitrary ADFAs (whereas the other MADFA construction algorithms' *cleanup* methods cannot).

In our implementation, steps 3 and 4 are performed for 2 MADFAs at a time, and this process is then repeated once in the case of 4 threads/MADFAs; but in general, the merger could be performed in one go for all the MADFAs. The resulting MADFA is a MADFA for the original keyword set. The details of the above construction can be found in Subsection 5.2, where our Java implementation of the approach is discussed.

The product automaton of step 3 is generated recursively from the start state, following the outgoing transitions. We traverse every state of both automata and generate the combined state. We show an example of this product construction from two MADFAs. We want to merge the automata in Figures 4 and 5. Automaton 1 accepts the words 'he' and 'she'. Automaton 2 accepts 'his' and 'this'. Note that both automata are MADFAs. The product of both automata is shown in Figure 6. We start with the product of both start states, i.e. 0,4. From there, we reach the product state 1,5 with a transition 'h', as 0 has such a transition to 1 and 4 to 5. With a transition 'e', we reach the final state 2 in automaton 1. Automaton 2 has no transition 'e' from state 5 but a transition 'i' to state 6. In the merged automaton we get states 2,null and null,6 or short 2 and 6. state 6 in automaton 2 has a transition to 7, so state 7 is copied to the product automaton. The same goes for state 3 and 1 that are reached from state 0 in automaton 1 and state 8 and 5 that are reached from state 4 in automaton 2. The resulting automaton is an ADFA because state 2 and 7 can be merged to generate the minimal MADFA.

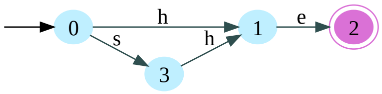


Figure 4. Automaton 1

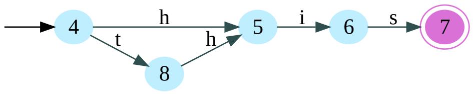


Figure 5. Automaton 2

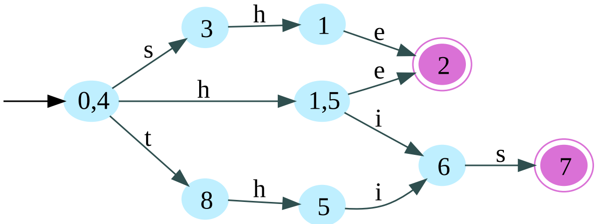


Figure 6. Merged automaton

The second variant of our algorithm generates four MADFAs in parallel. It also generates the product automaton during the merge step and minimize with a call to the *cleanup* method of Algorithm *Trie*. The difference is that this approach has two merge and minimization steps, re-using the merge of two MADFAs as mentioned above. First, two MADFAs are merged at a time and the intermediate ADFAs are minimized. The next step is to merge these two intermediate MADFAs again. The resulting ADFA is minimized to the final MADFA. The first merge and minimization step is also done in parallel. We minimize the intermediate automata because the benchmark shows that this approach is faster than without a minimization step. This approach can be adapted to every number of threads that is a power of two; otherwise the merge scheduling has to be changed. Another possibility is to merge more than two automata at once, but this complicates the merge process.

5 Toolkit

Our MADFA construction toolkit[2], Many-MADFAct, implements a skeleton class which is shared by all the sequential MADFA construction algorithms; specific algorithm implementations directly or indirectly inherit from this base class and override the abstract methods *add_word* and *cleanup* as needed, as shown on the left of Figure 7. Some of the algorithms also need specific helper methods. Helper methods that are used for more than one algorithm are in a component *Util*. For the data representation we use an *automaton* class which includes *states* and *transitions*. The former have in- and out-transitions, and the latter are represented as triples of start state, label, end state, for efficient transition processing. The automaton contains states, of which one is the start state, and zero or more are final i.e. accepting states; and transitions that link states. To distinguish states, every state gets a unique id. The implementation was done in Java. After the data representation was chosen, the pseudo-code from the algorithms in the taxonomy was easily translated to Java.

The helper methods are combined in the component *Util*. This class is divided into three parts. Firstly, we use string manipulation methods, for example for returning the head or tail of a string, and for computing a left derivate or longest common prefix. The second part concerns the analysis of the automaton. It contains methods that creates state subsets of the automaton, like height levels or the state set corresponding to a path. The last part is the check for minimality. We compare states and decide whether they are equivalent or not.

5.1 Sequential Algorithm Implementation

As explained at the beginning of this section, the sequential MADFA construction algorithms are implemented as part of a hierarchy, derived using the TABASCO process mentioned in Section 3. The class diagram is shown on the left side of Figure 7. The root, *AlgorithmSkeleton*, is an abstract class that creates an empty automaton and calls method to generate a MADFA. It also declares the abstract methods *add_word* and *cleanup*. The general approach is to call the method *add_word* for every word and minimize the automaton with *cleanup* afterwards. This general approach is implemented in *createMadfa* using the template method design pattern. The specific algorithms inherit from this class and implement the abstract methods. They also import *Util*. If necessary, the algorithms declare private helper methods. Algorithm *Trie* only inherits from *AlgorithmSkeleton* directly. The other algorithms inherit from *Trie* and extend *add_word*. They call the super class's *add_word* and add specific operations at the end of the method. Method *cleanup* is always overridden, except in the case of Algorithm *General*. It uses the same *cleanup* as *Trie*. Algorithm *Incremental* is an exception: this algorithm inherits from Algorithm *General* because it has nearly

[2] https://github.com/TUBS-ISF/MADFAct

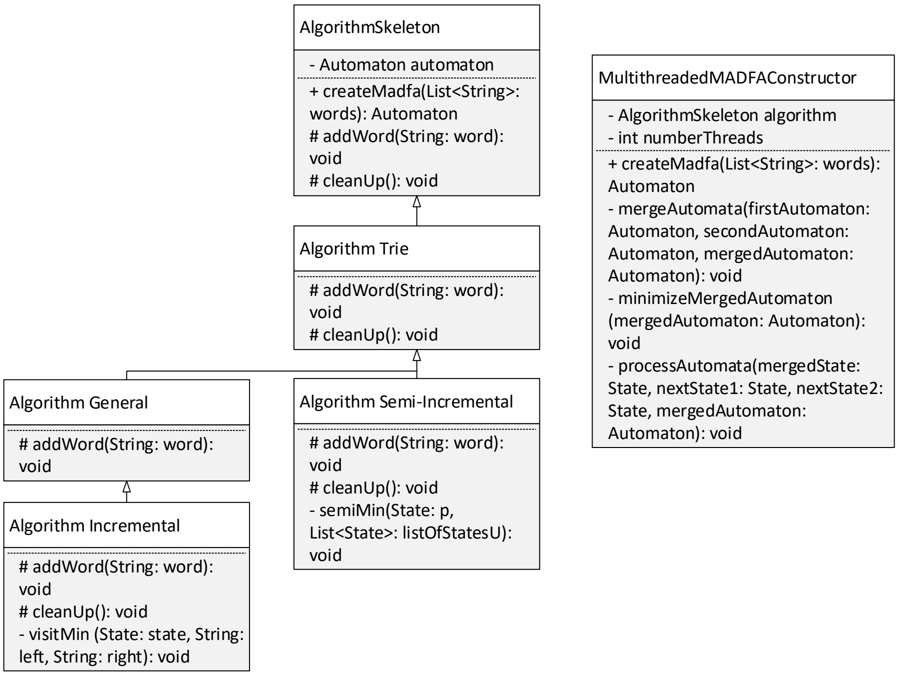

AlgorithmSkeleton
- Automaton automaton
+ createMadfa(List<String>: words): Automaton
# addWord(String: word): void
# cleanUp(): void
Algorithm Trie
# addWord(String: word): void
# cleanUp(): void
Algorithm General
# addWord(String: word): void
Algorithm Incremental
# addWord(String: word): void
# cleanUp(): void
- visitMin (State: state, String: left, String: right): void
Algorithm Semi-Incremental
# addWord(String: word): void
# cleanUp(): void
- semiMin(State: p, List<State>: listOfStatesU): void
MultithreadedMADFAConstructor
- AlgorithmSkeleton algorithm
- int numberThreads
+ createMadfa(List<String>: words): Automaton
- mergeAutomata(firstAutomaton: Automaton, secondAutomaton: Automaton, mergedAutomaton: Automaton): void
- minimizeMergedAutomaton (mergedAutomaton: Automaton): void
- processAutomata(mergedState: State, nextState1: State, nextState2: State, mergedAutomaton: Automaton): void

Figure 7. Class diagram of the toolkit

the same *add_word* method; i.e., *add_word* from *General* is called and extended. We implemented seven different sequential MADFA construction algorithms, of which this diagram shows three to illustrate the design without loosing clarity. The absent algorithms inherit from Algorithm *Trie* directly, just as Algorithm *Semi-Incremental* does.

5.2 Parallel Algorithm Implementation

The class *MultithreadedMAFDAConstructor* on the right of Figure 7 implements methods to create MADFAs in parallel and merge them afterwards. It contains a class variable that determines the construction algorithm used in each of the construction threads, e.g., Algorithm *Incremental*. Method *createMadfa* is the main method of this class. It creates MADFAs in parallel, and it calls the methods *mergeAutomata* and *minimizeMergedAutomaton* respectively to merge the resultant MADFAs into an ADFA, and to finally minimize this ADFA into a MADFA. Method *processAutomata* is a helper method of *mergeAutomata*. It creates the product automaton of multiple MADFAs by traversing the input automata recursively. The creation of MADFAs in parallel is done by forking and joining threads. Java is a multi-threaded programming language, so the implementation is straightforward; for every MADFA that should be constructed in parallel, we create a thread.

The procedure *createMadfa* is shown in Listing 1.1. Firstly, we divide the word list into the specific number of sub-lists (line 4). The next step is to create MADFA-threads and start them with a sub-list as input (line 7-12). They all execute the same algorithm and wait at the end. We implemented an algorithm to merge the intermediate MADFAs and minimize the result. *processAutomata*, the helper method for *mergeAutomata*, is shown in Listing 1.2. It traverses the input automata and merges them. It is a recursive method that gets a merged state and a state from each input automaton as input. We check if the states are not null because it is possible that we process a merged state with one null state. If that is the case, we are at line 34 / 39, and we copy the outgoing transitions and successor states from this state. After that, we call the method for every successor again, i.e. the copied successor state is the new merged state. If both input states are not null, we search for outgoing transitions with the same labels (line 4-6). If we find a pair, we run the code in line 11-23. A new merged state is created if it does not already exist, and this is the merged state for the next call of this method. In the case of transitions that only appear in one input automaton, we do the same as we only have one input state. We copy the new transition and the successor state, cf. line 7-9 and line 26-31.

```
public Aut createMadfa(List<String> words) {
  Aut mergedAut = new Aut();
  List<Aut> intermediateAut = new ArrayList<>();
  List<List<String>> subLists = chop(words, numberT);
  //numberT is the number of threads

  for (int i = 0; i < numberT; i++) {
    List<String> subList = subLists.get(i);
    MadfaThread thread = new MadfaThread(algorithm,
      intermediateAut, subList);
    thread.start();
  }
  Thread.join();

  if (numberT == 2) {
    //merge and minimize two automata
  } else if (numberT == 4) {
    //merge and minimize four automata
  }
  return mergedAut;
}
```

Listing 1.1. Code to start the parallel approach

6 Benchmarking

For benchmarking, we use the Java implementation of our toolkit and created MADFAs for different sets of input words. We use random English words[3] and subsequences from the ecoli genome[4]. We want to compare the runtimes of the seven algorithms. We also want to find out whether and how the lengths of the words impact performance. The results are presented below.

6.1 Setup

As input we decided for random English words to have a set with possibly many common prefixes and suffixes. We wanted to analyze how the algorithms behave if they can merge states during *cleanup*. For a totally different application setting, we also used the ecoli genome as input. Here, we cut substrings from the genome and use these as input. Most of the time such substrings have no common prefixes or suffixes because the probability to get a sequence of equal characters is low—especially for the natural language case. For example, the probability that two words share the same four characters as a prefix is lower than one percent. Therefore, the generated MADFAs consist of parallel state paths that do not have much in common.

[3] http://www-01.sil.org/linguistics/wordlists/english/

[4] http://www.dmi.unict.it/ faro/smart/download.php

```
private static void processAutomata(St mergedSt ,
  St nextSt1 , St nextSt2) {
  if (nextSt1 != null && nextSt2 != null) {
    for (Tran trans1 : nextSt1.getOutgoingTran()) {
      Tran trans2 = getEqualTran(nextSt2 ,
              trans1.getLabel());
      if (trans2 == null) {
        St newMergedSt = copy(mergedSt , trans1);
        processAut(newMergedSt , trans1.EndSt() , null);
      } else {
        String id = trans1.getEndSt().getId() + ";" +
                  trans2.EndSt().getId();
        St newMergedSt = getEqualSt(id, mergedAut);
        if (newMergedSt == null) {
          newMergedSt = new St(id);
        }
        if (mergedSt.getEqualTran(newMergedSt ,
                  trans1.getLabel()) == null) {
          Tran newTran = new Tran(mergedSt ,
                      newMergedSt , trans1.getLabel());
        }
        processAut(newMergedSt , trans1.EndSt() ,
                  trans2.EndSt());
      }
    }
    for (Tran trans2 : nextSt2.getOutgoingTran()) {
      Tran trans1 = getEqualTran(nextSt1 ,
              trans2.getLabel());
      if (trans1 == null) {
        St newMergedSt = copy(mergedSt , trans2);
        processAut(newMergedSt , null , trans2.EndSt());
      }
    }
  } else if (nextSt1 != null) {
    for (Tran trans : nextSt1.getOutgoingTran()) {
      St newMergedSt = copy(mergedSt , trans);
      processAut(newMergedSt , trans.EndSt() , null);
    }
  } else if (nextSt2 != null) {
    for (Tran trans : nextSt2.getOutgoingTran()) {
      St newMergedSt = copy(mergedSt , trans);
      processAut(newMergedSt , null , trans.EndSt());
    }
  }
}
```

Listing 1.2. Code to merge two automata

The setup for our benchmark is as follows. We select random sets of words and run every algorithm five times with each set. To deal with for example caching problems, we take the fastest run among these five as result. The sets form a sequence of increasing size and for every set size we generate 30 different sets, i.e. we add random words until the size of the set is reached. For example we build sets from size one to 2^{16} in the case of random English words. We always double the number of words from one set to the next. In the case of ecoli, we do two different runs. First, we construct sets which consist of strings of the same length. We vary the set size from 1 to 2^{10}, doubling the number in each iteration. For every set size, we insert strings of the same length, ranging from one to 2^6 (64) and again doubling in each iteration. The second benchmark run of ecoli is with substrings of varying length, called varying-length ecoli. We also construct sets from 1 to 2^{10}, but this time, we insert substrings of random length between 1 and 2^6.

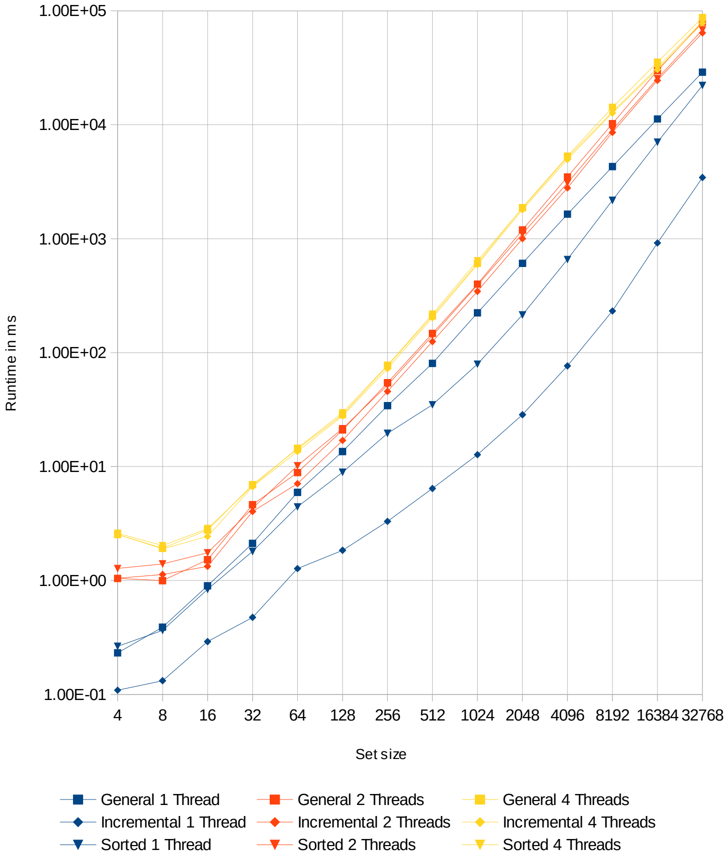


Figure 8. Benchmark result of fast algorithms with English words. The x-axis of the graph shows the set size of input words, the y-axis the runtime in ms.

6.2 Results

The benchmark results diverge between the seven algorithms. For example, with English words as input, Algorithm *Incremental* is the fastest. It needs ca. 15 seconds for 2^{16} words. The next faster algorithms are in this order: Sorted, Trie and General. Trie needs for example ca. 65 seconds for the same number of words. The other three algorithms, *Depth-Layered*, *Reverse* and *Semi-Incremental*, are much slower. They need up to two hours for this word set. We also tested the same sets with the new parallel implementation, using two and four threads. The fast algorithms *Incremental*, *Sorted*, *Trie* and *General* are not getting faster, they are even slower, cf. Figure 8. The scale for both axes is logarithmic. We start at set size four so that, in the case of four threads, every thread gets at least one input word. The graphs show that the runtime for each algorithm increases exponentially. Algorithm *Trie* is not shown in the figure because it behaves like Algorithm *General* if the MADFA is built from an empty automaton. The difference between Algorithm *Trie* and Algorithm *General* is that *General* looks for confluence states, i.e. states which need to be cloned before adding a new transition, and clones them [31]. If the automaton is built from scratch, it is constructed as a trie and no confluence states occur, so both algorithms execute identically. The runtimes for every algorithm for one thread are in every case shorter than the runtime for two or four threads.

The slow algorithms on the other hand get faster. For example we present in Figure 9 the algorithms *Depth-Layered*, *Semi-Incremental* and *Reverse*. For small

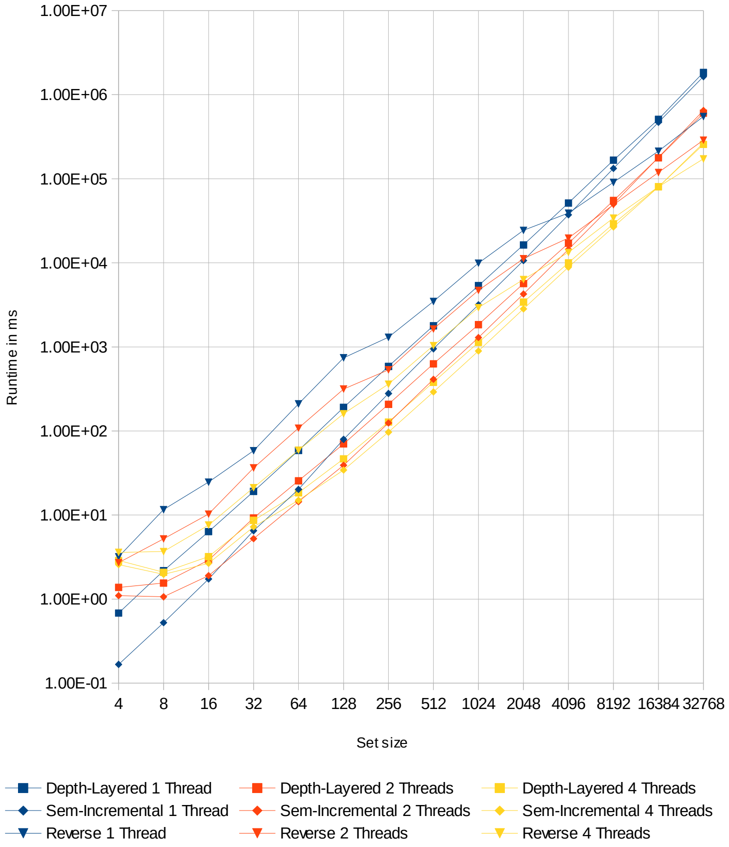


Figure 9. Benchmark result of slow algorithms with English words. The x-axis of the graph shows the set size of input words, the y-axis the runtime in ms.

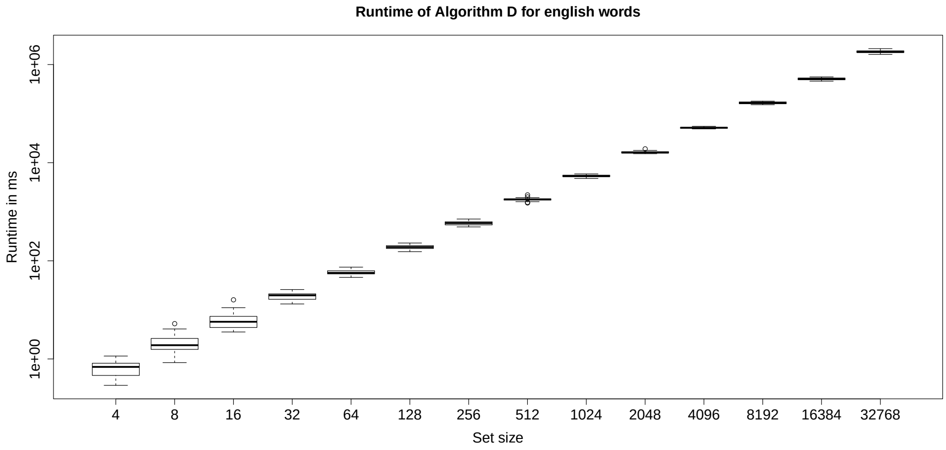


Figure 10. Benchmark result of algorithm Depth-Layered (one thread). The x-axis of the graph shows the set size of input words, the y-axis the runtime in ms.

sets, the execution of *Depth-Layered* and *Semi-Incremental* is the fastest, but the bigger the set is, the better the parallel implementation performs. For big sets, i.e., with 32768 words, the 4 thread implementation is in every case the fastest, followed by the 2 thread one. Here, we can save time by running the algorithm in parallel. A general observation for the three algorithms is, the more threads the faster the runtime. However in the case of small sets, the runtime is slower because the parallel approach has some overhead for creating parallel automata and merging them. The approach does not pay off in such cases, due to the extra merge and minimization steps needed.

Figure 10 only shows the benchmark results for Algorithm *Depth-Layered*. Here, we do not compute the average of the 30 runs for every set size. The graph shows boxplots that include the 30 different runs. As we can see, the runtimes are similar and there are few spikes. The scale of the runtime is logarithmic, causing the boxplots to be very small. The other benchmark results are quite similar. We infer from the few spikes and the small box sizes that the use of mean values is ok, as the spread of values is limited.

The benchmarking using ecoli strings does not uncover new insights. The runtime increases exponentially for every algorithm and the ordering wrt. performance is the same, i.e. Algorithm *Incremental* is the fastest. If we want to compare the benchmark of ecoli with the benchmark of English words, we should not compare sets with the same number of words because the ecoli strings can be much longer. We decided to compare sets with the same summed word length. For example, we compared 256 equal length ecoli strings with fixed string length 64 with an English word set with 2048 words that has ca. the same summed word length. The running times of all seven algorithms are longer for ecoli than for English words. We get the same result, that the runtime is longer for longer string lengths, if we compare two different runs of ecoli with fixed string length. We take for example the results of set size 128 and length 64 and compare it with the results of set size 1024 and length 8. Both have 8192 characters in total and the algorithms are slower in case of the longer strings, i.e., the toolkit performs better for large sets with short words than for small sets with long words.

7 Conclusion and Future Work

In this project, we successfully implemented the algorithms presented by Watson [31]. First, we created a taxonomy graph by identifying the commonalities and differences between the algorithms. The next step was to create a toolkit based on this information, using the TABASCO process to do so. We also implemented a new algorithm family exploiting parallelism. The two algorithm variants from this family that we discussed and implemented create MADFAs in two or four threads and merge and minimize the resulting MADFAs into a single MADFA in the end. We benchmarked the toolkit using English words and ecoli substrings as input. The results show that the parallel approach improves the runtime of the slower algorithms.

For future work an implementation in C++ is planned to compare the implementations with respect to their runtime and their storage space consumption. We also want to improve the current parallel implementation. The current merge process creates an ADFA from two MADFAs. We minimize the ADFA afterwards to create the final MADFA. It should be possible to create a MADFA directly from two or more MADFAs, by adapting the merge process to minimize the product automaton during construction, possibly by reusing and generalizing ideas from Algorithm *Incremental*'s *add_word* method.

References

1. M. Broy: *Program construction by transformations: a family tree of sorting programs*, in Computer Program Synthesis Methodologies, A. W. Biermann and G. Guiho, eds., Reidel, Dordrecht, 1983, pp. 1–49.
2. J. A. Brzozowski: *Canonical regular expressions and minimal state graphs for definite events.* Mathematical theory of Automata, 12(6) 1962, pp. 529–561.
3. J. Bubenzer: *Construction of minimal ADFAs*, diplomarbeit, Universität Potsdam, Germany, 2011.
4. R. C. Carrasco and M. L. Forcada: *Incremental construction and maintenance of minimal finite-state automata.* Computational Linguistics, 28(2) June 2002, pp. 207–216.
5. M. Ciura and S. Deorowicz: *Experimental study of finite automata storing static lexicons*, tech. rep., Silesian Technical University, Poland, Nov. 1999.
6. L. Cleophas: *Tree Algorithms: Two Taxonomies and a Toolkit*, PhD thesis, Technische Universiteit Eindhoven, 2008.
7. L. Cleophas, B. W. Watson, D. G. Kourie, A. Boake, and S. Obiedkov: *TABASCO: using concept-based taxonomies in domain engineering.* South African Computer Journal, 2006, pp. 30–40.
8. J. Daciuk: *Incremental Construction of Finite-State Automata and Transducers, and their Use in the Natural Language Processing*, PhD thesis, Technical University of Gdańsk, Poland, 1998.
9. J. Daciuk: *Optimization of Automata*, Gdańsk University of Technology Publishing House, 2014.
10. J. Daciuk, S. Mihov, B. W. Watson, and R. E. Watson: *Incremental construction of minimal acyclic finite state automata.* Computational Linguistics, 26(1) Apr. 2000, pp. 3–16.
11. J. Daciuk, B. W. Watson, and R. E. Watson: *Incremental construction of minimal acyclic finite state automata and transducers*, in Proceedings of the International Workshop on Finite State Methods in Natural Language Processing, L. Karttunen and K. Oflazer, eds., Ankara, Turkey, June 1998, pp. 48–56.
12. J. Darlington: *A synthesis of several sorting algorithms.* Acta Informatica, 11(1) 1978, pp. 1–30.
13. J. Graña, F. M. Barcala, and M. A. Alonso: *Compilation methods of minimal acyclic finite-state automata for large dictionaries*, in Proceedings of the Sixth Conference on Implementations and Applications of Automata, B. W. Watson and D. Wood, eds., vol. 2494, Pretoria, South Africa, July 2002, Springer Verlag, pp. 116–129.
14. G. J. Holzmann and A. Puri: *A minimized automaton representation of reachable states.* Software Tools for Technology Transfer, 3 (1998)(1) 1998.
15. H. Jonkers: *Abstraction, specification and implementation techniques: with an application to garbage collection*, PhD thesis, Technische Hogeschool Eindhoven, 1982.
16. A. Marcelis: *On the classification of attribute evaluation algorithms.* Science of Computer Programming, 14(1) 1990, pp. 1–24.
17. S. Mihov: *Direct building of minimal automaton for given list.* Annuaire de l'Université de Sofia St. Kl. Ohridski, 91(1) 1998, pp. 212–225.
18. S. Mihov: *Direct building of minimal automaton for given list*, tech. rep., Bulgarian Academy of Science, 1999.
19. K.-H. Park, J.-I. Aoe, K. Morimoto, and M. Shishibori: *An algorithm for dynamic processing of DAWGs.* International Journal of Computational Mathematics, 54 1994, pp. 155–173.
20. V. Pieterse: *Topic Maps for Specifying Algorithm Taxonomies: A Case Study using Transitive Closure Algorithms*, PhD thesis, University of Pretoria, 2017.
21. D. Revuz: *Dictionnaires et lexiques: méthodes et algorithmes.*, PhD thesis, Institut Blaise Pascal, LITP 91.44, Paris, France, 1991.
22. D. Revuz: *Minimisation of acyclic deterministic automata in linear time.* Theoretical Computer Science, 92(1) 1992, pp. 181–189.
23. D. Revuz: *Dynamic acyclic minimal automaton*, in Proceedings of the Fifth Conference on Implementations and Applications of Automata, S. Yu and A. Păun, eds., vol. 2088, London, Canada, July 2000, Springer-Verlag, pp. 226–232.

24. K. Sgarbas, N. Fakotakis, and G. Kokkinakis: *Two algorithms for incremental construction of directed acyclic word graphs*. International Journal of Artificial Intelligence Tools, 4 1995, pp. 369–381.
25. B. W. Watson: *Taxonomies and Toolkits of Regular Language Algorithms*, PhD thesis, Technische Universiteit Eindhoven, 1995.
26. B. W. Watson: *A taxonomy of algorithms for constructing minimal acyclic deterministic finite automata*. South African Computer Journal, (27) 2001, pp. 12–17.
27. B. W. Watson: *Directly constructing minimal DFAs: Combining two algorithms by Brzozowski*. South African Computer Journal, 29 Dec. 2002, pp. 17–23.
28. B. W. Watson: *A fast and simple algorithm for constructing minimal acyclic deterministic finite automata*. Journal of Universal Computer Science, 8(2) 2002, pp. 363–367.
29. B. W. Watson: *A new algorithm for the construction of minimal acyclic DFAs*. Science of Computer Programming, 48(2–3) 2003, pp. 81–97.
30. B. W. Watson: *A new recursive incremental algorithm for building minimal acyclic deterministic finite automata*, in Grammars and Automata for String Processing: From Mathematics and Computer Science to Biology, and Back, C. Martin-Vide and V. Mitrana, eds., Taylor and Francis, 2003, pp. 189–202.
31. B. W. Watson: *Constructing Minimal Acyclic Deterministic Finite Automata*, PhD thesis, University of Pretoria, 2010.