Incremental DFA minimisation*

Marco Almeida** Nelma Moreira Rogério Reis
{mfa,nam,rvr}@ncc.up.pt

DCC-FC & LIACC, Universidade do Porto
R. do Campo Alegre 1021/1055, 4169-007 Porto, Portugal

Abstract. We present a new incremental algorithm for minimising deterministic finite automata. It runs in quadratic time for any practical application and may be halted at any point, returning a partially minimised automaton. Hence, the algorithm may be applied to a given automaton at the same time as it is processing a string for acceptance. We also include some experimental comparative results.

1 Introduction

We present a new algorithm for incrementally minimise deterministic finite automata. This algorithm may be halted at any point, returning a partially minimised automaton that recognises the same language as the input. Should the minimisation process be interrupted, calling the incremental minimisation algorithm with the output of the halted process would resume the minimisation process. Moreover, the algorithm can be run on some automaton D at the same time as D is being used to process a string for acceptance.

Unlike the usual approach, which computes the equivalence classes of the set of states, this algorithm proceeds by testing the equivalence of pairs of states in the same line of Watson and Daciuk [Wat01,WD03]. The intermediate results are stored for the speedup of ulterior computations in order to assure quadratic running time and memory usage.

This paper is structured as follows. In the next Section some basic concepts and notation are introduced. Section 3 is a small survey of related work. In Section 4 we describe the new algorithm in detail, presenting the proofs of correctness and worst-case running-time complexity. Section 5 follows with experimental comparative results, and Section 6 finishes with some conclusions and future work.

2 Preliminaries

We recall here the basic definitions needed throughout the paper. For further details we refer the reader to the work of Hopcroft et al. [HMU00].

* This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI, project ASA (PTDC/MAT/65481/2006), and project CANTE (PTDC/EIA-CCO/101904/2008).
** Marco Almeida is funded by FCT grant SFRH/BD/27726/2006.

An alphabet Σ is a nonempty set of symbols. A word over an alphabet Σ is a finite sequence of symbols of Σ. The empty word is denoted by ϵ and the length of a word w is denoted by $|w|$. The set $\Sigma^\star$ is the set of words over Σ. A language L is a subset of $\Sigma^\star$.

A *deterministic finite automaton* (DFA) is a tuple $D = (Q, \Sigma, \delta, q_0, F)$ where Q is finite set of states, Σ is the alphabet, $\delta : Q \times \Sigma \to Q$ the transition function, q_0 the initial state, and $F \subseteq Q$ the set of final states. We can extend the transition function to words $w \in \Sigma^\star$ such that $w = au$ by considering $\delta(q, w) = \delta(\delta(q, a), u)$ for $q \in Q$, $a \in \Sigma$, and $u \in \Sigma^\star$. The *language* accepted by the DFA D is $L(D) = \{w \in \Sigma^\star \mid \delta(q_0, w) \in F\}$. Two finite automata A and B are *equivalent*, denoted by $A \sim B$, if they accept the same language. For any DFA $D = (Q, \Sigma, \delta, q_0, F)$, let $\varepsilon(q) = 1$ if $q \in F$ and $\varepsilon(q) = 0$ otherwise, for $p \in Q$. Two states $q_1, q_2 \in Q$ are said to be *equivalent*, denoted by $q_1 \sim q_2$, if for every $w \in \Sigma^\star$, $\varepsilon(\delta(q_1, w)) = \varepsilon(\delta(q_2, w))$. A DFA is *minimal* if there is no equivalent DFA with fewer states. Given an DFA D, the equivalent minimal DFA $D/_\sim$ is called the *quotient automaton* of D by the equivalence relation $\sim$. Minimal DFAs are unique up to isomorphism.

2.1 The UNION-FIND algorithm

The UNION-FIND [Tar75,CLRS03] algorithm takes a collection of n distinct elements grouped into several disjoint sets and performs two operations on it: merges two sets and finds to which set a given element belongs to. The algorithm is composed by the following three functions:

- MAKE(i): creates a new set (singleton) for one element i (the identifier);
- FIND(i): returns the identifier S_i of the set that contains i;
- UNION(i, j, k): combines the sets identified by i and j in a new set $S_k = S_i \cup S_j$; S_i and S_j are destroyed.

An important detail of the UNION operation is that the two combined sets are destroyed in the end. Our implementation of the algorithm (using rooted trees) follows the one by Cormen et al. [CLRS03]. The main claim is that an arbitrary sequence of i MAKE, UNION, and FIND operations, j of which are MAKE, can be performed in $O(i\alpha(j))$, where $\alpha(j)$ is related to a functional inverse of the Ackermann function, and, as such, grows *very* slowly. In fact, for every *practical* values of j (up to 16^{512}), $\alpha(j) \leq 4$.

3 Related work

The problem of writing efficient algorithms to find the minimal equivalent DFA can be traced back to the 1950's with the works of Huffman [Huf55] and Moore [Moo58]. Over the years several alternative algorithms were proposed. In terms of worst-case complexity the best know algorithm (log-linear) is by Hopcroft [Hop71]. Brzozowski [Brz63] presented an elegant but exponential algorithm that may also be applied to non-deterministic finite automata.

The first DFA incremental minimisation algorithm was proposed by Watson [Wat01]. The worst-case running-time complexity of this algorithm is exponential — $O(k^{\max(0,n-2)})$, for a DFA with n states over an alphabet of k symbols. As shown by Watson himself [Wat95], this bound is tight. Later, Watson and Daciuk [WD03] proposed a new version of the algorithm. By using a memoization technique they achieved an almost quadratic run-time. Recently, however, a bug was found on the algorithm and one of the authors is currently trying to fix it.

4 The incremental minimisation algorithm

Given an arbitrary DFA D as input, this algorithm may be halted at any time returning a partially minimised DFA that has no more states than D and recognises the same language. It uses a disjoint-set data structure to represent the DFA's states and the UNION-FIND algorithm to keep and update the equivalence classes. This approach allows us to maintain the transitive closure in a very concise and elegant manner. The pairs of states already marked as distinguishable are stored in an auxiliary data structure in order to avoid repeated computations.

Let $D = (Q, \Sigma, \delta, q, F)$ be a DFA with $n = |Q|$ and $k = |\Sigma|$. We assume that the states are represented by integers, and thus it is possible to order them. This ordering is used to normalise pairs of states, as presented in Listing 1.1.

```
1  def NORMALISE(p, q):
2      if p < q:
3          pair = (p, q)
4      else:
5          pair = (q, p)
6      return pair
```

Listing 1.1. A simple normalisation step.

The normalisation step allows us to improve the behaviour of the minimisation algorithm by ensuring that only $\frac{n^2}{2} - n$ pairs of states are considered.

The quadratic time bound of the minimisation procedure MIN-INCR, presented in Listing 1.2, is achieved by testing each pair of states for equivalence exactly once. We assure this by storing the intermediate results of all calls to the pairwise equivalence-testing function EQUIV-P, defined in Listing 1.3. Some auxiliary data structures, designed specifically to improve the worst-case running time, are presented in Listing 1.4.

```
1  def MIN-INCR(D = (Q, Σ, δ, q0, F)):
2      for q ∈ Q:
3          MAKE(q)
4      NEQ = {NORMALISE(p, q) | p ∈ F, q ∈ Q − F}
5      for p ∈ Q:
6          for q ∈ {x | x ∈ Q, x > p}:
7              if (p, q) ∈ NEQ:
8                  continue
```

```
9                   if FIND(p) = FIND(q):
10                      continue
11                  EQUIV = SET-MAKE(|Q|²)
12                  PATH = SET-MAKE(|Q|²)
13                  if EQUIV-P(p, q):
14                      for (p', q') ∈ SET-ELEMENTS(EQUIV):
15                          UNION(p', q')
16                  else:
17                      for (p', q') ∈ SET-ELEMENTS(PATH):
18                          NEQ = NEQ ∪ {(p', q')}
19      classes = {}
20      for p ∈ Q:
21          lider = FIND(p)
22          classes[lider] = classes[lider] ∪ {p}
23      D' = D
24      joinStates(D', classes)
25      return D'
```

Listing 1.2. Incremental DFA minimisation in quadratic time.

Algorithm MIN-INCR starts by creating the initial equivalence classes (lines 2–3); these are singletons as no states are yet marked as equivalent. The global variable `NEQ`, used to store the distinguishable pairs of states, is also initialised (line 4) with the trivial identifications. Variables `PATH` and `EQUIV`, also global and reset before each call to EQUIV-P, maintain the history of calls to the transition function and the set of potentially equivalent pairs of states, respectively.

The main loop of MIN-INCR (lines 5–18) iterates through all the normalised pairs of states and, for those not yet known to be either distinguishable or equivalent, calls the pairwise equivalence test EQUIV-P. Every call to EQUIV-P is conclusive and the result is stored either by merging the corresponding equivalence classes (lines 13–15), or updating `NEQ` (lines 16–18). Thus, each recursive call to EQUIV-P will avoid one iteration on the main loop of MIN-INCR by skipping (lines 7–10) that pair of states.

Finally, at lines 19–22, the set partition of the corresponding equivalence classes is created. Next, the DFA D is copied to D' and the equivalent states are merged by the call to `joinStates`. The last instruction, at line 25, returns the minimal DFA D', equivalent to D.

```
1   def EQUIV-P(p, q):
2       if (p, q) ∈ NEQ:
3           return False
4       if SET-SEARCH((p, q), PATH) ≠ nil:
5           return True
6       SET-INSERT((p, q), PATH)
7       for a ∈ Σ:
8           (p', q') = NORMALISE(FIND(δ(p, a)), FIND(δ(q, a)))
9           if p' ≠ q' and SET-SEARCH((p', q'), EQUIV) = nil:
10              SET-INSERT((p', q'), EQUIV)
11              if not EQUIV-P(p', q'):
12                  return False
13              else:
14                  SET-REMOVE((p', q'), PATH)
15      SET-INSERT((p, q), EQUIV)
16      return True
```

Listing 1.3. Pairwise equivalence test for MIN-INCR.

Algorithm EQUIV-P, presented in Listing 1.3, is used to test the equivalence of the two states, p and q, passed as arguments.

The global variables `EQUIV` and `PATH` are updated with the pair (p, q) during each nested recursive call. As there is no recursion limit, EQUIV-P will only return when $p \nsim q$ (line 3) or when a cycle is found (line 5). If a call to EQUIV-P returns `False`, then all pairs of states recursively tested are distinguishable and variable `PATH` — used to store the sequence of calls to the transition function — will contain a set of distinguishable pairs of states. If it returns `True`, no pair of distinguishable states was found within the cycle and variable `EQUIV` will contain a set of equivalent states. This is the strategy which assures that each pair of states is tested for equivalence exactly once: every call to EQUIV-P is conclusive and the result stored for future use. It does, however, lead to an increased usage of memory.

The variables `EQUIV` and `PATH` are heavily used in EQUIV-P as several insert, remove, and membership-test operations are executed throughout the algorithm. In order to achieve the desired quadratic upper bound, all these operations must be performed in $O(1)$. Thus, we present in Listing 1.4 some efficient set representation and manipulation procedures.

```
1  def SET-MAKE(size):
2      HashTable = HASH-TABLE(size)
3      List = LIST()
4      return (HashTable, List)
5
6  def SET-INSERT(v, Set):
7      p0 = Set.HashTable[v]
8      LIST-REMOVE(p0, Set.List)
9      p1 = LIST-INSERT(v, Set.List)
10     Set.HashTable[v] = p1
11
12 def SET-REMOVE(v, Set):
13     p0 = Set.HashTable[v]
14     LIST-REMOVE(p0, Set.List)
15     Set.HashTable[v] = nil
16
17 def SET-SEARCH(v, Set):
18     if Set.HashTable[v] ≠ nil:
19         p = Set.HashTable[v]
20         return LIST-ELEMENT(p, Set.List)
21     else:
22         return nil
23
24 def SET-ELEMENTS(Set):
25     return Set.List
```

Listing 1.4. Set representation procedures.

The set-manipulation procedures in Listing 1.4 simply combine a hash-table with a doubly-linked list. This is another space-time trade-off that allows us to assure the desired complexity on all operations. The hash-table maps a given value (state of the DFA) to the address on which it is stored in the linked list. Since we know the size of the hash-table in advance (n^2) searching, inserting, and removing elements is $O(1)$. The linked list assures that, at lines 14–15 and 17–18

of MIN-INCR, the loop is repeated only on the elements that were actually used in the calls to EQUIV-P, instead of iterating through the entire hash-table.

Theorem 1. *Algorithm MIN-INCR, in Listing 1.2, is terminating.*

Proof. It should suffice to notice the following facts:

- all the loops in MIN-INCR are finite;
- the variable `PATH` on EQUIV-P assures that the number of recursive calls is finite.

Lemma 1. *Algorithm EQUIV-P, in Listing 1.3, runs in $O(kn^2)$ time.*

Proof. The number of recursive calls to EQUIV-P is controlled by the local variable `PATH`. This variable keeps the history of calls to the transition function (line 8 in Listing 1.3). In the worst case, all possible pairs of states are used: $\frac{n^2}{2} - n$, due to the normalisation step. Since each call may reach line 7, we need to consider k additional recursive calls for each pair of states, hence $O(kn^2)$.

Lemma 2. *Algorithm EQUIV-P returns `True` if and only if the two states passed as arguments are equivalent.*

Proof. Algorithm EQUIV-P returns `False` only when the two states, p and q, used as arguments are such that $(p, q) \in$ `NEQ` (lines 2–3). This is correct because the global variable `NEQ` contains all the pairs of states already proven to be distinguishable. Conversely, EQUIV-P returns `True` only if $(p, q) \in PATH$ (lines 4–5) or a recursive call returned `True` (line 16). In both cases this means that a cycle with no distinguishable elements was detected, which implies that all the recursively visited pairs of states are equivalent.

Theorem 2. *Given a DFA $D = (Q, \Sigma, \delta, q, F)$, algorithm MIN-INCR computes the minimal DFA D' such that $D \sim D'$.*

Proof. Algorithm MIN-INCR finds pairs of equivalent states by exhaustive enumeration. The loop in lines 5–18 enumerates all possible pairs of states, and, for those not yet proven to be either distinguishable or equivalent, EQUIV-P is called. When line 19 is reached, all pairs of states have been enumerated and the equivalent ones have been found (cf. Lemma 2). The loop in lines 20–22 creates the equivalence classes and the procedure `joinStates`, at line 24, merges the equivalent states, updating the corresponding transitions. Since the new DFA D' does not have any equivalent states, it is minimal.

Lemma 3. *At the top-level call at line 13 in MIN-INCR, when EQUIV-P returns `True`, all the pairs of states stored in the global variable `EQUIV` are equivalent.*

Proof. By Lemma 2, if EQUIV-P returns `True` then the two states, p and q, used as arguments are equivalent. Since there is no depth recursion control, EQUIV-P only returns `True` when a cycle is detected. Thus being, all the pairs of states used as arguments in the recursive calls must also be equivalent. These pairs of states are stored in the global variable `EQUIV` at line 10 of EQUIV-P.

Lemma 4. *At the top-level call at line 13 in MIN-INCR, if EQUIV-P returns* `False`*, all the pairs of states stored in the global variable* `PATH` *are distinguishable.*

Proof. Given a pair of distinguishable states (p, q), clearly all pairs of states (p', q') such that $\delta(p', w) = p$ and $\delta(q', w) = q$ are also distinguishable, for $w \in \Sigma^\star$. By Lemma 2, EQUIV-P returns `False` only when the two states, p and q, used as arguments are distinguishable. Throughout the successive recursive calls to EQUIV-P, the global variable `PATH` is used to store the history of calls to the transition function (line 6) and thus contains only pairs of states with a path to (p, q). All of these pairs of states are therefore distinguishable.

Lemma 5. *Each time that EQUIV-P calls itself recursively, the two states used as arguments will not be considered in the main loop of MIN-INCR.*

Proof. The arguments of every call of EQUIV-P are kept in two global variables: `EQUIV` and `PATH`.

By Lemma 3, whenever EQUIV-P returns `True`, all the pairs of states stored in `EQUIV` are equivalent. Immediately after being called from MIN-INCR (line 13), if EQUIV-P returns `True`, the equivalence classes of all the pairs of states in `EQUIV` are merged (lines 14–15). Future references to any of these pairs will be skipped at lines 9–10.

In the same way, by Lemma 4, if EQUIV-P returns `False`, all the pairs of states stored in `PATH` are distinguishable. Lines 17–18 of MIN-INCR update the global variable `NEQ` with this new information and future references to any of these pairs of states will be skipped at lines 7–8 of MIN-INCR.

Theorem 3. *Algorithm MIN-INCR is incremental.*

Proof. Halting the main loop of MIN-INCR at any point within the lines 5–18 only prevents the finding of *all* the equivalent pairs of states. Merging the known equivalent states on D', a copy of the input DFA D, assures that the size of D' is not greater than that of D and thus, is closer to the minimal equivalent DFA. Calling MIN-INCR with D' as the argument would resume the minimisation process, finding the remaining equivalent states.

Theorem 4 (Main result). *Algorithm MIN-INCR, in Listing 1.2, runs in* $O(kn^2\alpha(n))$ *time.*

Proof. The number of iterations of the main loop in lines 5–18 of MIN-INCR is bounded by $\frac{n^2}{2} - n$, due to the normalisation step. Each iteration may call EQUIV-P, which, by Lemma 1, is $O(kn^2)$. By Lemma 5 every recursive call to EQUIV-P avoids one iteration on the main loop. Therefore, disregarding the UNION-FIND operations and because all operations on variables `NEQ`, `EQUIV`, and `PATH` are $O(1)$, the $O(kn^2)$ bound holds. Since there are $O(kn^2)$ FIND and UNION intermixed calls, and exactly n MAKE calls, the time spent on all the UNION-FIND operations is bounded by $O(kn^2\alpha(n))$ — cf. Subsection 2.1. All things considered, MIN-INCR runs in $O(kn^2 + kn^2\alpha(n)) = O(kn^2\alpha(n))$.

Corollary 1. *Algorithm MIN-INCR runs in $O(kn^2)$ time for all* practical *values of n.*

Proof. Function α is related to an inverse of Ackermann's function. It grows so slowly ($\alpha(16^{512}) \leq 4$) that we may consider it a constant.

5 Experimental results

In this Section we present some comparative experimental results on four DFA minimisation algorithms: Brzozowski, Hopcroft, Watson, and the new proposed incremental one. The results are presented on Table 1, Table 2, and Table 3.

All algorithms are implemented in the `Python` programming language and integrated in the **FAdo** project [**?**]. The tests were executed in the same computer, an Intel® Xeon® 5140 at 2.33 GHz with 4 GB of RAM, running a minimal 64 bit Linux system. We used samples of 20.000 automata, with $n \in \{5, 10, 50, 100\}$ states and alphabets with $k \in \{2, 10, 25, 50\}$ symbols. Since the data sets were obtained with a uniform random generator [AMR07,AAA+09], the size of each sample is more than enough to ensure results statistically significant with a 99% confidence level within a 1% error margin. The sample size is calculated with the formula $N = (\frac{z}{2\epsilon})^2$, where z is obtained from the normal distribution table such that $P(-z < Z < z)) = \gamma$, ϵ is the error margin, and γ is the desired confidence level.

$n = 5$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	1424.50	4	119.71	4	45.24	4	22.66	4
Hopcroft	3442.34	4	980.39	4	469.37	4	238.46	4
Watson	3616.63	4	573.88	4	54.29	4	8.00	4
Incremental	4338.39	4	2762.43	4	1814.88	4	1091.70	4
$n = 10$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	73.45	4	1.89	4	0.50	3248	0.21	1912
Hopcroft	1757.46	4	250.46	4	100.95	4	49.74	4
Watson	691.80	4	0.01	4	0.00	4	0.00	4
Incremental	2358.49	4	1484.78	4	885.73	4	488.75	4

Table 1. Experimental results for ICDFAs with $n \in \{5, 10\}$ states.

Each test was given a time slot of 24 hours. Processes that did not finish within this time limit were killed. Thus, and because we know how many ICDFAs were in fact minimised before each process was killed, the performance of the algorithms is measured in *minimised ICDFAs per second* (column **Perf.**). We also include a column for the memory usage (**Space**), which measures the peak value (worst-case) for the minimisation of the 20.000 ICDFAs in *kilobytes*.

Clearly, the new incremental method always performs better. Both Brzozowski and Watson's algorithm clearly show their exponential character, being

$n = 50$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	0.00	664704	0.00	799992	0.00	1456160	0.00	750312
Hopcroft	39.48	4	6.31	4	2.45	4	1.21	4
Watson	0.00	4	0.00	4	0.00	4	0.00	4
Incremental	117.21	288	94.33	540	73.94	612	53.31	636
$n = 100$								
	$k = 2$		$k = 10$		$k = 25$		$k = 50$	
	Perf.	Space	Perf.	Space	Perf.	Space	Perf.	Space
Brzozowski	0.00	2276980	0.00	1061556	0.00	961144	0.00	2862312
Hopcroft	6.25	4	0.97	4	0.37	4	0.18	4
Watson	0.00	4	0.00	4	0.00	4	0.00	4
Incremental	28.23	1028	24.94	2452	21.58	3444	17.17	3824

Table 2. Experimental results for ICDFAs with $n \in \{50, 100\}$ states.

in fact, the only two algorithms that did not finish several minimisation tests. Hopcroft's algorithm, although presenting a behaviour that appears to be very close to its worst-case ($O(kn \log(n))$, is always slower than the quadratic incremental method.

$n = 1000$						
	$k = 2$		$k = 3$		$k = 5$	
	Perf.	Space	Perf.	Space	Perf.	Space
Hopcroft	0.0121	4	0.0074	4	–	–
Incremental	0.2616	34296	0.2416	34068	0.2411	33968

Table 3. Experimental results for ICDFAs with 1000 states.

Because of the big difference between the measured performance of Hopcroft's algorithm and the new quadratic incremental one, we ran a new set of tests, only for these algorithms, using the largest ICDFA samples we have available. The results are presented on Table 3. These tests were performed on the exact same conditions as the previously described ones but each process was allowed to execute for 96 hours. Surprisingly, while Hopcroft's algorithm did not finish any of the batches within this time limit, it took only a little over 23 hours for the quadratic algorithm to minimise the sample of 20.000 ICDFAs with 1000 states and 5 symbols. The memory usage of the incremental algorithm, however, is far superior. It always required nearly 33 MB while Hopcroft's algorithm did not use more than 4 kB.

6 Conclusions

We presented a new incremental minimisation algorithm. Unlike other non-incremental minimisation algorithms, the intermediate results are usable and reduce the size of the input DFA. This property can be used to minimise a DFA when it is simultaneously processing a string or, for example, to reduce the size

of a DFA when the running-time of the minimisation process must be restricted for some reason.

We believe that this new approach, while presenting a quadratic worst-case running-time, is quite simple and easy to understand and to implement. According to the experimental results, this minimisation algorithm outperforms Hopcroft's $O(kn\log(n))$ approach, at least in the average case, for reasonably sized automata.

References

[AAA+09] A. Almeida, M. Almeida, J. Alves, N. Moreira, and R. Reis. FAdo and GUItar: tools for automata manipulation and visualization. In S. Maneth, editor, *14th CIAA'09*, volume 5642 of *LNCS*, pages 65–74. Springer, 2009.

[AMR07] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theoret. Comput. Sci.*, 387(2):93–102, 2007. Special issue "Selected papers of DCFS 2006".

[Brz63] J. A. Brzozowski. Canonical regular expressions and minimal state graphs for definite events. In J. Fox, editor, *Proc. of the Sym. on Math. Theory of Automata*, volume 12 of *MRI Symposia Series*, pages 529–561, NY, 1963.

[CLRS03] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT, 2003.

[FAd10] Project FAdo. FAdo: tools for formal languages manipulation. `http://www.ncc.up.pt/FAdo`, Access date:1.1.2010.

[HMU00] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2000.

[Hop71] J. Hopcroft. An $n\log n$ algorithm for minimizing states in a finite automaton. In *Proc. Inter. Symp. on the Theory of Machines and Computations*, pages 189–196, Haifa, Israel, 1971. Academic Press.

[Huf55] D. A. Huffman. The synthesis of sequential switching circuits. *The Journal of Symbolic Logic*, 20(1):69–70, 1955.

[Moo58] E. F. Moore. Gedanken-experiments on sequential machines. *The Journal of Symbolic Logic*, 23(1):60, 1958.

[Tar75] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22(2):215 – 225, April 1975.

[Wat95] B. W. Watson. *Taxonomies and toolkit of regular languages algortihms*. PhD thesis, Eindhoven Univ. of Tec., 1995.

[Wat01] B. W. Watson. An incremental DFA minimization algorithm. In *International Workshop on Finite-State Methods in Natural Language Processing*, Helsinki, Finland, August 2001.

[WD03] B. W. Watson and J. Daciuk. An efficient DFA minimization algorithm. *Natural Language Engineering*, pages 49–64, 2003.