On the performance of automata minimization algorithms

Marco Almeida Nelma Moreira Rogério Reis

Technical Report Series: DCC-2007-03
Version 1.0 June 2007

Departamento de Ciência de Computadores
&
Laboratório de Inteligência Artificial e Ciência de Computadores

Faculdade de Ciências da Universidade do Porto
Rua do Campo Alegre, 1021/1055,
4169-007 PORTO,
PORTUGAL

Tel: 220 402 900 Fax: 220 402 950
`http://www.dcc.fc.up.pt/Pubs/`

On the performance of automata minimization algorithms

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. There are several well known algorithms to minimize deterministic finite automata. Apart from the theoretical worst-case running time analysis, however, not much is known about the average-case analysis or practical performance of each of these algorithms. On this paper we compare three minimization algorithms based on experimental results. The choice of the algorithms was based on the fact that although having different worst-case complexities they are usually considered to be ones that achieve best performance. We used an uniform random generator of (initially-connected) deterministic finite automata for the input data, and thus our results are statistically accurate. Because one of the algorithms allowed to minimize non-deterministic finite automata (NFA), we also developed a non-uniform random generator for NFAs. Nevertheless, although not statistically significant, the results in this case are fairly interesting.

Keywords deterministic finite automata, non-deterministic finite automata, minimal automata, minimization algorithms, random generation

1 Introduction

The problem of writing efficient algorithms to find the minimal deterministic finite automaton equivalent to a given automaton 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. Authors typically present the running time worst-case analysis of their algorithms, but the practical experience is sometimes different. The comparison of algorithms performance is always a difficult problem and, apart from the results presented by Bruce Watson [Wat95], little is known about the practical running time performance of automata minimization algorithms. In particular, there are no studies of average-case analysis of these algorithms, an exception being the work of Nicaud [Nic00], where it is proved that the average-case complexity of Brzozowski's algorithm is exponential for group automata. Lhoták [Lho00] presents a general data structure for DFA minimization algorithms to run in $O(kn \log n)$.

Using the `Python` programming language, we implemented Hopcroft's algorithm, Brzozowski's algorithm, and two versions of an incremental minimization algorithm. This algorithm was first presented by Watson [Wat01] and later improved by Watson and Daciuk [WD03] , by using full memoization. . The main difference is the use of full memoization.

The choice of the algorithms was based on the fact that although having different worst-case complexities they are usually considered to be ones that achieve best performance. We used an uniform random generator of (initially-connected) deterministic finite automata for the input data, and thus our results are statistically accurate. Because one of the algorithms allowed to minimize non-deterministic finite automata (NFA), we also developed a non-uniform random generator for NFAs.

The text is organized as follows. In Section 2 we present some definitions and notation used throughout the paper. In Section 3 we describe each of the compared algorithms, explain how they work. In Section 4 we describe the generation methods of the random automata used as input for each algorithm. In Section 5 we present the experimental results and in Section 6 we expose our final remarks and possible future work.

2 Preliminaries

A *deterministic finite automaton* (DFA) $\mathcal{D}$ is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of *states*, Σ is the *input alphabet* (any non-empty set of symbols), $\delta : Q \times \Sigma \rightarrow Q$ is the *transition function*, q_0 is the *initial state* and $F \subseteq Q$ is the set of *final states*. When the transition function is total, the automaton $\mathcal{D}$ is said to be *complete*. Any finite sequence of alphabet symbols $a \in \Sigma$ is a *word*. Let $\Sigma^\star$ denote the set of all words over the alphabet Σ and ϵ denote the *empty word*. We define the *extended transition function* $\hat{\delta} : Q \times \Sigma^\star \rightarrow Q$ in the following way: $\hat{\delta}(q, \epsilon) = q$; $\hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$. A state $q \in Q$ of a DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ is called *accessible* if $\hat{\delta}(q_0, w) = q$ for some $w \in \Sigma^\star$. If all states in Q are accessible, a complete DFA $\mathcal{D}$ is called (complete) *initially-connected* (ICDFA). The *language* accepted by $\mathcal{D}$, $L(\mathcal{D})$, is the set of all words $w \in \Sigma^\star$ such that $\hat{\delta}(q_0, w) \in F$. Two DFAs $\mathcal{D}$ and $\mathcal{D}'$ are *equivalent* if and only if $L(\mathcal{D}) = L(\mathcal{D}')$. A DFA is called *minimal* if there is no other equivalent DFA with fewer states. Given a DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$, two states $q_1, q_2 \in Q$ are said to be *equivalent*, denoted $q_1 \approx q_2$, if for every $w \in \Sigma^\star$, $\hat{\delta}(q_1, w) \in F \Leftrightarrow \hat{\delta}(q_2, w) \in F$. Two states that are not equivalent are called *distinguishable*. The equivalent minimal automaton $\mathcal{D}/\approx$ is called the *quotient automaton*, and its states correspond to the equivalence classes of $\approx$. It is proved to be unique up to isomorphism.

A *non-deterministic finite automaton* (NFA) is also a tuple $(Q, \Sigma, \Delta, I, F)$, where I is a *set of initial states* and the *transition function* is defined as $\Delta : Q \times \Sigma \rightarrow 2^Q$. Just like with DFAs, we can define the *extended transition function* $\hat{\Delta} : 2^Q \times \Sigma^\star \rightarrow 2^Q$ in the following way: $\hat{\Delta}(S, \epsilon) = S$; $\hat{\Delta}(S, xa) = \bigcup_{q \in \hat{\Delta}(S,x)} \delta(q, a)$. The *language* accepted by $\mathcal{N}$ is the set of all words $w \in \Sigma^\star$ such that $\hat{\Delta}(I, w) \cap F \neq \emptyset$. Every language accepted by some NFA can also be described by a DFA. The *subset construction* method takes a NFA $\mathcal{A}$ as input and computes a DFA $\mathcal{D}$ such that $L(\mathcal{A}) = L(\mathcal{D})$. This process is also referred to as *determinization* and has a worst-case running time complexity of $O\left(2^{|Q|}\right)$.

Following Leslie [Les95], we define the *transition density* of an automaton $\mathcal{A} = (Q, \Sigma, \Delta, I, F)$ as the ratio $\frac{t}{|Q|^2|\Sigma|}$, where t is the number of transitions in $\mathcal{A}$. We also define *deterministic density* as the ratio of the number of transitions t to the number of transitions of a complete DFA with the same number of states and symbols, i.e., $\frac{t}{|Q||\Sigma|}$.

The *reversal* of a word $w = a_0 a_1 \cdots a_n$, written w^R, is $a_n \cdots a_1 a_0$. The reversal of a language $L \subseteq \Sigma^\star$ is $L^R = \{w^R \mid w \in Ł\}$. Further details on regular languages can be found in the usual references (Hopcroft [HMU00b] or Kozen [Koz97], for example).

3 Minimization algorithms

Given an automaton, to obtain the associated minimal DFA we must compute the equivalence relation $\approx$ as defined in Section 2. The way this relation is computed is one of the main differences between the several minimization algorithms. Moore's algorithm and its variations aim to find pairs of distinguishable states. Hopcroft's algorithm, on the other hand, computes the minimal automaton by refining a partition of the states' set.

We have implemented and compared the performance of three different minimization algorithms. Hopcroft's minimization algorithm has the best worst-case running time analysis. Brzozowski's algorithm is simple and elegant. Despite its exponential worst-case complexity, it is supposed to frequently outperform other algorithms (including Hopcroft's). Brzozowski's algorithm also has the particularity of being able to minimize both DFAs and NFAs. Watson presented an incremental DFA minimization algorithm which can be halted at any time, yielding a partially minimized automaton. Later, Watson and Daciuk presented an improved version of the same algorithm, this time including full memoization. Because our main motivation, in the **FAdo** project [pro], was to check if a given automaton was minimal, and not to obtain the equivalent minimal automaton, this algorithm was of particular interest. This choice of algorithms was also motivated by the results presented by Watson [Wat95,WD03], that pointed them as those which achieved better performance, in spite their different worst-case running time analysis.

3.1 Hopcroft's algorithm

Hopcroft's algorithm [Hop71], published in 1971, achieves the best known running time worst-case complexity for minimization algorithms. It runs on $O(kn \log n)$ time for a DFA with n states and an alphabet of size k.

Let $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ be a DFA. Hopcroft's algorithm, unlike Moore's algorithm and several variations [HMU00a,ASU86], does not identify pairs of distinguishable states. Instead, it proceeds by refining the coarsest partition until no more refinements are possible. The initial partition is $P = \{F, Q - F\}$ and, at each step of the algorithm, a block $B \in P$ and a symbol $a \in \Sigma$ are selected to *refine* the partition. This refinement process *splits* each block B' of the partition according to whether the states of B', when consuming the symbol a, go to a state which is in B or not. Formally, we call this procedure `split` and define it by

$$\mathtt{split}(\mathtt{B}', \mathtt{B}, \mathtt{a}) = (\mathtt{B}' \cap \check{\delta}^{-1}(\mathtt{B}, \mathtt{a}), \mathtt{B}' \cap \overline{\check{\delta}^{-1}(\mathtt{B}, \mathtt{a})})$$

where $\check{\delta}(S, a) = \bigcup_{q \in S} \delta(q, a)$.

The algorithm terminates when there are no more blocks to refine. In the end, each block of the partition is a set of equivalent states. Because, for any two blocks $B, B' \in P$, every state $q \in B$ is distinguishable from any state $q' \in B'$, the elements of P represent the states of a new minimal DFA. The complete algorithm is presented in Algorithm 1.1.

```
def hopcroft():
    P = {F, Q−F}
    L = {Q−F}
    while L $\neq \emptyset$:
        S = extract(L)
        for a in $\Sigma$:
            for B in P:
                (B1, B2) = split(B, S, a)
                P = P − {B}
                P = P $\cup$ {B1}
                P = P $\cup$ {B2}
                if |B1| < |B2|:
                    L = L $\cup$ {B1}
                else:
                    L = L $\cup$ {B2}
    return P

def split(B, S, a):
    foo = $\check{\delta}^{-1}$(S, a)
    bar = Q − foo
    return (B ∩ foo, B ∩ bar)
```

Algorithm 1.1. Hopcroft's algorithm.

3.2 Brzozowski's algorithm

Brzozowski's automata minimization algorithm [Brz63] is based on two successive reverse and determinization operations and the entire algorithm is presented (in one single line!) on Algorithm 1.2. This algorithm is the only one able to process both DFAs and NFAs, always yielding a minimal DFA that accepts the same language as the input automaton. Given an automaton $\mathcal{A}$, let `det` be the subset construction method and `rev` to the procedure that computes an NFA that accepts $L(\mathcal{A})^R$. Brzozowski's algorithm can be stated as a simple function composition of the previous procedures: $det \cdot rev \cdot det \cdot rev$. Since the *rev* method can be applied to any finite automaton, perhaps with non-determinism, this algorithm is able to minimize both DFAs and NFAs.

```
def brzozowski(fa):
    return det(rev(det(rev(fa))))
```

Algorithm 1.2. Brzozowski's algorithm.

Having to perform two determinizations, the worst-case running time complexity of Brzozowski's algorithm is exponential. Watson's thesis, however, presents some surprizing results about Brzozowski's algorithm practical performance, usually outperforming Hopcroft's algorithm.

As for the peculiar way that this algorithm computes a minimal DFA, Watson assumed it to be unique and, in his taxonomy, placed it apart all other algorithms. Later, Champarnaud et al. [CKP02] analyzed the way the sequential determinizations perform the minimization and showed that it does compute state equivalences.

3.3 An incremental algorithm

In 2001 Watson presented an incremental DFA minimization algorithm [Wat01]. This algorithm, unlike the other minimization algorithms, can be halted at any time yielding a partially minimized DFA that recognizes the same language as the input DFA. Later, Watson and Daciuk presented an improved version of the same algorithm [WD03] which makes use of full memoization. While the first algorithm has a worst-case exponential running time, the memoized version yields a $O(n^2)$ algorithm (for all *practical* values of n, i.e., $n \leq 2^{2^{16}}$). It was not clear, however, that this algorithm would outperform the Hopcroft's algorithm as the experimental results in [WD03] seemed to point to. Since the use of memoization introduces some considerable overhead in the algorithm, we wanted to discover at what point this extra work begins to pay back.

The incremental algorithm uses an auxiliary function, `equiv`, that tests if two states are equivalent. The third argument, an integer k, is used to control the recursion depth . , and is used only for matters of efficiency. Also for matters of efficiency, a variable S that contains a set of presumably equivalent pairs of states, is made global. The pseudo-code for a non-memoized, specialized for ICDFAs, implementation of `equiv` is presented in Algorithm 1.3. The memoized algorithm is quite extensive and can be found in Watson and Daciuk [WD03].

```
def equiv(p, q, k):
    if k = 0:
        return (p in F and q in F) or (not p in F and not q in F)
    elif (p,q) in S:
        return True
    else:
        eq = (p in F and q in F) or (not p in F and not q in F)
        S = S ∪ {(p,q)}
        for a in $\Sigma$:
            if not eq:
                return False
            eq = eq and equiv($\delta(p,a)$, $\delta(q,a)$, k-1)
        S = S - {(p,q)}
    return eq
```

Algorithm 1.3. Pairwise state equivalence algorithm.

Having a method to verify pairwise state equivalence, we can now implement a test `minimal-p` that calls `equiv` for every pair of states and returns *False* if some pair is found to be equivalent. The complete algorithm for this test is given in Algorithm 1.4.

```
def minimal-p():
    k = max(0, |Q|-2)
    for i in Q:
        for j in Q:
            if i < j:
                S = {}
                if equiv(i, j, k):
                    return False
    return True
```

Algorithm 1.4. Minimal DFA test.

4 Random automata generation

Given the amount of DFAs with n symbols over an alphabet of k symbols [RMA05] for even considerable small values of n and k, in order to compare the practical performance of the minimization algorithms, we must have available an arbitrary quantity of uniformly generated random automata. In a previous work [AMR06] we have already presented a uniform random generator of ICDFAs, which allowed us to obtain all the necessary DFAs.

Because one of the algorithms (Brzozowski's) is able to minimize both DFAs and NFAs, we believe that a fair comparison will have to include the minimization of some NFAs and, in the case of Hopcroft's and Watson's algorithm, account for the time spent in the NFA determinization process. For the NFAs, we implemented a new random generator. This generator combines the van Zijl bit-stream method as presented by Champarnaud et al. [CHPZ04] with one of Leslie's approaches [Les95], which allows us both to generate initially connected NFAs (with one initial state) and to control the transition density. Leslie presents a "generate-and-test" method which may never stop, so we added some minor changes that correct this situation. A brief explanation of the random NFA generator follows. Suppose we want to generate a random NFA with n states over an alphabet of k symbols and a transition density d. Let the states (respectively the symbols) be named by the integers $0, \ldots, n-1$ (respectively $0, \ldots, k-1$). A sequence of n^2k bits describes the transition function in the following way: the occurrence of a non-zero bit at the position $ink + jk + a$ denotes the existence of a transition from state i to state j labeled by the symbol a. Consider the bitstream on Table 1 which represents the NFA of the Figure 1.

$$\underbrace{\overbrace{00}^{q_0} \mid \overbrace{11}^{q_1} \mid \overbrace{10}^{q_2}}_{q_0} \parallel \underbrace{\overbrace{00}^{q_0} \mid \overbrace{00}^{q_1} \mid \overbrace{00}^{q_2}}_{q_1} \parallel \underbrace{\overbrace{01}^{q_0} \mid \overbrace{10}^{q_1} \mid \overbrace{01}^{q_2}}_{q_2}$$

Table 1. Bitstream for the NFA on Figure 1.

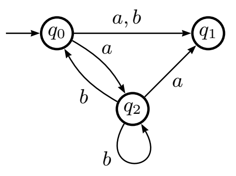

Fig. 1. The NFA built from the bitstream on Table 1.

Starting with a sequence of zero bits, the first step of the algorithm is to create a connected structure and thus ensure that all the states of the final NFA will be accessible. In order to do so, we define the first state as 0, mark it as visited, generate a transition from 0 to any not-visited state i, and mark i as visited. Next, until all states are marked as visited, randomly choose an already visited state q_1, randomly choose a not-visited state q_2, add a transition from q_1 to q_2 (with a random), and mark q_2 as visited. At this point we have an initially connected NFA and proceed by adding random transitions. Until the desired density is achieved, we simply select one of the bitstream's zero bits and set it to one. By maintaining a list of visited states on the first step and keeping record of the zero bits on the second step, we avoid generating either a disconnected NFA or a repeated transition and guarantee that the algorithm always halts. The set of final states can be easily obtained by generating an equiprobable bitstream of size n and considering final all the states that correspond to a non-zero position in the bitstream.

5 Experimental results

To compare algorithms is always a difficult problem. The choice of the programming language, implementation details, and the hardware used may harm the rigor of any benchmark. In order to produce realistic results, the input data should be random so that it represents a typical usage of the algorithm and the test environment should be identical for all benchmarks. We implemented all the algorithms in the `Python 2.4` programming language, using similar data structures whenever possible. All the tests were executed in the same computer, an Intel® Xeon® 5140 at 2.33GHz with 2GB of RAM. For both minimization tests we used samples of automata with $5 \leq n \leq 100$ states and alphabets with $k \in \{2, 5, 10, 20\}$ symbols.

We established a running time limit for both tests. All processes that did not finish within this limit were killed and the correspondent columns will not appear in the graphics. Also, for $n = 5$ no graphics are shown, but now due to the small running times of the algorithms.

5.1 Random ICDFA minimization

On his thesis, Watson used a fairly biased sample. It consisted of 4833 DFAs of which only 7 had 23 states. As Watson himself states, being constructed from regular expressions, the automata "... are usually not very large, they have relatively sparse transition graphs, and the alphabet frequently consists of the entire ASCII character set.". On their paper on the incremental minimization algorithm [WD03], Watson and Daciuk also present some performance comparisons of automata minimization algorithms. They used four different data sets, one from experiments on finite-state approximation of context-free grammars and three that were automatically generated. These are not, however, uniform random samples, and thus, do not represent a typical usage of the algorithms.

The following graphics show the running times for the three algorithms while minimizing a sample of 10.000 random ICDFAs. The running time limit for all algorithms was 24 hours. Because we used a uniform random generator, the size of the sample is sufficient to ensure a 95% confidence level within a 1% error margin. On Table 2 we present a summary of the algorithms' behavior.

	$k = 2$	$k = 5$	$k = 10$	$k = 20$
Hopcroft	✓	✓	✗	✗
Watson	–	–	–	–
Watson+Daciuk	✗	✗	✓	✓
Brzozowski	–	–	–	–

Table 2. Minimization algorithms' performance with samples of 10.000 ICDFAs.

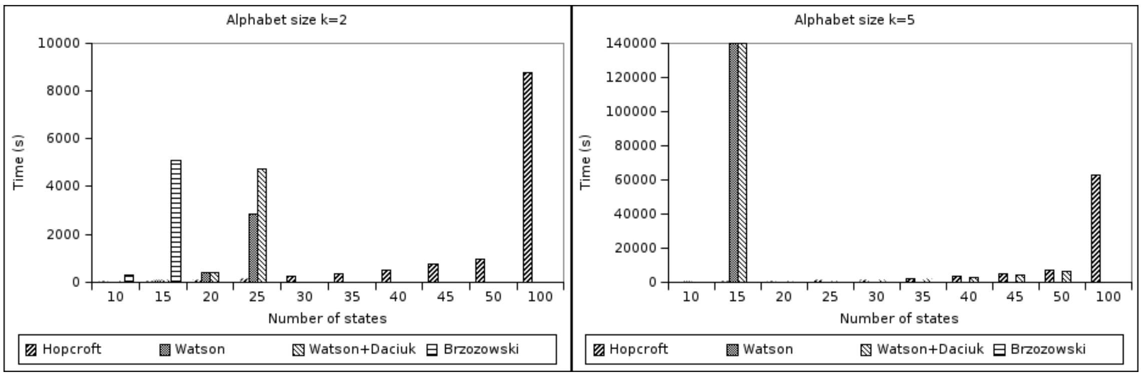

Fig. 2. Running time results for 10.000 ICDFAs with $k = 2$ and $k = 5$.

For small alphabets ($k \leq 5$), Hopcroft's algorithm is always the fastest. When the alphabet size grows ($k \geq 10$), Hopcroft's algorithm is clearly outperformed by the memoized version of Watson and Daciuk's algorithm. The incremental algorithm Watson presents on his thesis

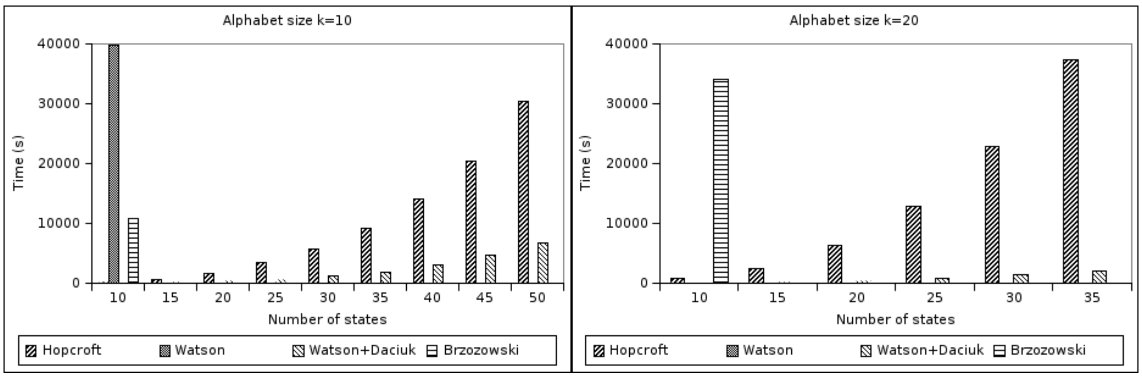

Fig. 3. Running time results for 10.000 ICDFAs with $k = 10$ and $k = 20$.

showed itself quite slow in all tests. The memoized version, however, was over twice as fast as Hopcroft's algorithm when minimizing ICDFAs with an alphabet of size $k \geq 10$. It is important to point out that for $k \geq 5$ all the automata were already minimal and so the speed of the incremental algorithm can not be justified by the possibility of halting whenever two equivalent states are found. The fact that almost all ICDFAs are minimal was observed by several authors, namely Almeida et al. [AMR06]. As Watson himself stated, the incremental algorithm may show exponential performance for some DFAs. This was the case in one of our tests. For the sample of 5 symbols and 15 states the memoized incremental algorithm took an unusual amount of time. Brzozowski's algorithm is never the fastest. In fact, even for small alphabets it was not possible to use it on ICDFAs with more than 15 states.

5.2 Random NFAs minimization

The next set of graphics shows the execution times of the three algorithms when applied to a set of 10.000 random NFAs. The running time limit for all algorithms was 15 hours. It is important to note that the NFA generator we used is not a uniform one, and so we can not prove that each sample is actually a good representative of the universe. Because we are dealing with NFAs, the transition density is an important factor and so each sample was generated with three different transition densities (d): 0.2, 0.5, and 0.8. For both the incremental and Hopcroft's algorithm, which are only able to minimize DFAs, we also accounted for the time spent in the subset construction method.

For alphabets with two symbols there are no significant differences in any of the algorithms' general performance, although Brzozowski's is usually the fastest. Hopcroft's algorithm outperforms Brzozowski's for less than 4% only when $d = 0.5$ and $n \in \{50, 100\}$.

For alphabets with five symbols, Brzozowski's algorithm is always the fastest and, except for occasional cases, Hopcroft's algorithm is slightly faster than the incremental algorithm.

When alphabet size increases, the performance of Brzozowski's algorithm becomes quite remarkable. For an alphabet with size $k \in \{10, 20\}$, Brzozowski's algorithm is definitively the fastest, being the only algorithm to actually finish the minimization process of almost all random samples within the 15 hour limit. As for Hopcroft's and the incremental algorithm, except for two cases, there are no significant performance differences.

For $d = 0.2$ and $n = 10$ all the algorithms showed a particularly bad performance. For $k = 5$ only Brzozowski's algorithm finished the minimization process (taking an unusual

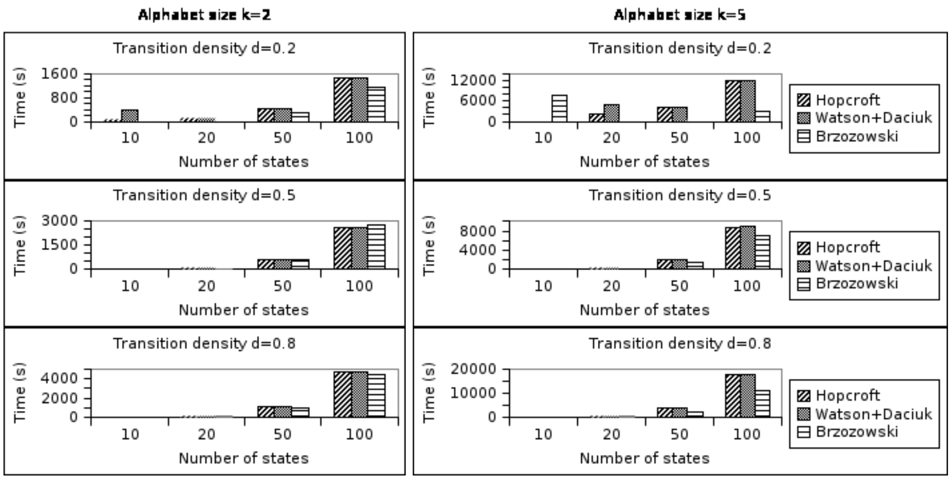

Table 3. Running time results for 10.000 NFAs with $k = 2$ and $k = 5$.

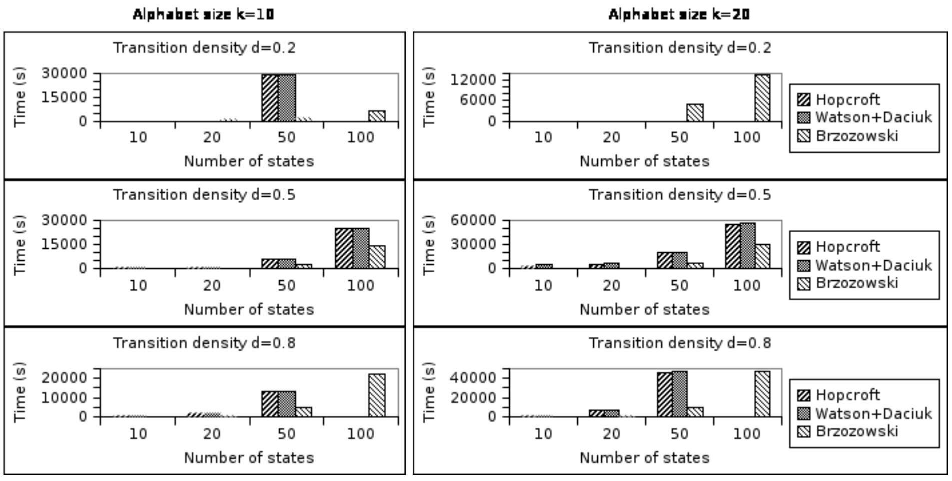

Table 4. Running time results for 10.000 NFAs with $k = 10$ and $k = 20$.

high amount of time) and for $k \in \{10, 20\}$ none of the algorithms completed the minimization process within the 15 hour time limit. This result corroborates Leslie's conjecture [Les95]about the number of states obtained with the subset construction method for a given deterministic density d_d. Leslie's conjecture states that randomly generated automata exhibit the maximum execution time and the maximum number of states at an approximate deterministic density of 2.0. While generating the random NFAs, we considered the transition density $d = \frac{t}{n^2 k}$, which is related to the deterministic density $d_d = \frac{t}{nk}$ by $d_d = nd$. It is easy to see that in our case $d_d = nd = 10 \times 0.2 = 2.0$, which will make the subset construction computationally expensive. In order to achieve the same exponential behavior in the subset method for $d \in \{0.5, 0.8\}$ the number of states would have to be $n \in \{4, 2.5\}$, but for such a small number of states the exponential blowup is not meaningful. This explains why there are no similar results for the test batches with $d \in \{0.5, 0.8\}$. Considering we used a variation of one of Leslie's random NFA generators, this result does not come with any surprise.

6 Conclusion

We compared three automaton minimization algorithms based on experimental results. For input data we used two different types of randomly generated automata (ICDFAs and NFAs) with different number of states and alphabet size.

The tests with ICDFAs were conducted using a uniform random generator and a sample large enough to ensure a 95% confidence level within a 1% error margin. For alphabets with less than 10 symbols, Hopcroft's algorithm is the fastest. As the alphabet size grows, Watson and Daciuk's algorithm performs better than Hopcroft's. We can safely conclude that neither Watson's non-memoized algorithm nor Brzozowski's algorithm perform well regardless of the number of states or size of the alphabet.

As for the tests with NFAs, it is important to note that the random generator we used was not a uniform one, and so, it is possible that the random samples are not a good representative set. Brzozowski's algorithm was definitively the fastest algorithm for NFAs. Hopcroft's and Watson and Daciuk's algorithms, both slower than Brzozowski's algorithm, always showed similar results.

It would be interesting to obtain an average-case running time complexity analysis for the DFA reversal, and thus possibly explain Brzozowski's algorithm behavior with ICDFAs minimization. Also, considering that when minimizing a NFA all algorithms must use the subset construction at least once, the reason that makes Brzozowski's algorithm so efficient is not evident.

References

[AMR06] M. Almeida, N. Moreira, and R. Reis. Aspects of enumeration and generation with a string automata representation. In H. Leung and G.Pighizzini, editors, *Proc. of DCFS'06*, pages 58–69, Las Cruces, New Mexico, 2006. NMSU.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

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

[CHPZ04] J.-M. Champarnaud, G. Hansel, T. Paranthoën, and D. Ziadi. Random generation models for nfas. *J. of Automata, Languages and Combinatorics*, 9(2), 2004.

[CKP02] J.-M. Champarnaud, A. Khorsi, and T. Paranthoën. Split and join for minimizing: Brzozowski's algorithm. In M. Balík and M. Simánek, editors, *Proc. of PSC'02*, Report DC-2002-03, pages 96–104. Czech Technical University of Prague, 2002.

[HMU00a] J. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 2000.

[HMU00b] John E. Hopcroft, Rajeev Motwani, and Jeffrey 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. AP.

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

[Koz97] D. C. Kozen. *Automata and Computability.* Undergrad. texts in computer science. SV, 1997.

[Les95] T. Leslie. Efficient approaches to subset construction. Master's thesis, University of Waterloo, Ontario, Canada, 1995.

[Lho00] O. Lhoták. A general data structure for efficient minimization of deterministic finite automata. Technical report, University of Waterloo, 2000.

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

[Nic00] C. Nicaud. *Étude du comportement en moyenne des automates finis et des langages rationnels.* PhD thesis, Université de Paris 7, 2000.

[pro] FAdo project. FAdo: tools for formal languages manipulation. http://www.ncc.up.pt/fado.

[RMA05] R. Reis, N. Moreira, and M. Almeida. On the representation of finite automata. In C. Mereghetti C. Mereghetti, B. Palano, G. Pighizzini, and D.Wotschke, editors, *Proc. of DCFS'05*, pages 269–276, Como, Italy, 2005.

[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.