Hopcroft's Minimization Technique: Queues or Stacks?

Andrei Păun[1,2,3], Mihaela Păun[4], and Alfonso Rodríguez-Patón[3]

[1] Bioinformatics Department, National Institute of Research and Development for Biological Sciences, Splaiul Independenţei, Nr. 296, Sector 6, Bucharest, Romania
[2] Department of Computer Science, Louisiana Tech University, Ruston PO Box 10348, Louisiana, LA-71272 USA
apaun@latech.edu
[3] Departamento de Inteligencia Artificial, Facultad de Informática, Universidad Politécnica de Madrid, Campus de Montegancedo s/n, Boadilla del Monte 28660 Madrid, Spain
arpaton@fi.upm.es
[4] Department of Mathematics and Statistics, Louisiana Tech University, Ruston PO Box 10348, Louisiana, LA-71272 USA
mpaun@latech.edu

Abstract. We consider the absolute worst case time complexity for Hopcroft's minimization algorithm applied to unary languages (or a modification of this algorithm for cover automata minimization). We show that in this setting the worst case is reached only for deterministic automata or cover automata following the structure of the de Bruijn words. We refine a previous result by showing that the Berstel/Carton example reported before is actually the absolute worst case time complexity in the case of unary languages for deterministic automata. We show that the same result is valid also when considering the setting of cover automata and an algorithm based on the Hopcroft's method used for minimization of cover automata. We also show that a LIFO implementation for the splitting list is desirable for the case of unary languages in the setting of deterministic finite automata.

1 Introduction

This work is in part a continuation of the result reported by Berstel and Carton in [2] regarding the number of steps required for minimizing a unary language through Hopcroft's minimization technique. The second part of the paper considers the same problem in the setting of Cover Automata. This new type of automata was introduced by Prof. Dr. Sheng Yu and Drs. Sântean and Câmpeanu in [6] and since then was investigated by several authors such as in [4], [8], [14], [16], [19], etc.

In the first part of the paper we will analyze and extend the result by Bestel and Carton from [2]. There it was shown that Hopcroft's algorithm for minimizing unary languages requires O(n log n) steps when considering the example of automata following the structure induced by de Bruijn words (see [3]) as input and when making several "bad" implementation decisions. The setting of the paper [2] is for languages over an unary alphabet, considering the automata associated to the language(s) having the number of states a power of 2 and choosing "in a specific way" which set to become a splitting set in the case of ties. In this context, the previous paper showed that the algorithm needs $O(n\ log\ n)$ steps for the algorithm to complete, which is reaching the theoretical asymptotic worst case time complexity for the algorithm as reported in [9,10,11,13] etc.

We were initially interested in investigating further the complexity of an algorithm described by Hopcroft, specifically considering the setting of unary languages, but for a stack implementation in the algorithm. Our effort has led to the observation that when considering the worst case for the number of steps of the algorithm (which in this case translates to the largest number of states appearing in the splitting sets), a LIFO implementation indeed outperforms a FIFO strategy as suggested by experimental results on random automata as reported in [1].

One major **observation/clarification** that is needed is the following: we do not consider the asymptotic complexity of the run-time, but the actual number of steps. For the setting of the current paper when comparing $n\ log\ n$ steps and $n\ log(n-1)$ or $\frac{n}{2}\ log\ n$ steps we will say that $n\ log\ n$ is worse than both $n\ log(n-1)$ and $\frac{n}{2}\ log\ n$, even though when considering them in the framework of the asymptotic complexity (big-O) they have the same complexity, i.e. $n\ log(n-1) \in \Theta(n\ log\ n)$ and $\frac{n}{2}\ log\ n \in \Theta(n\ log\ n)$.

In Section 2 we give some definitions, notations and previous results, then in Section 3 we give a brief description of the algorithm discussed and its features. Section 4 describes the properties for the automaton that reaches worst possible case in terms of steps required for the algorithm (as a function of the initial number of states of the automaton). We then briefly consider the case of cover automata minimization with a modified version of the Hopcroft's algorithm in Section 6 and conclude by giving some final remarks in the Section 7.

2 Preliminaries

We assume the reader is familiar with the basic notations of formal languages and finite automata, see for example the excellent work by Yu [20]. In the following we will be denoting the cardinality of a finite set T by $|T|$, the set of words over a finite alphabet Σ is denoted Σ^*, and the empty word is λ. The length of a word $w \in \Sigma^*$ is denoted with $|w|$. For $l \geq 0$ we define the following sets of words:

$$\Sigma^l = \{w \in \Sigma^* \mid |w| = l\},\ \Sigma^{\leq l} = \bigcup_{i=0}^{l} \Sigma^i, \text{ and for } l > 0 \text{ we define } \Sigma^{<l} = \bigcup_{i=0}^{l-1} \Sigma^i.$$

A deterministic finite automaton (DFA) is a quintuple $A = (\Sigma, Q, \delta, q_0, F)$ where Σ is a finite set of symbols, Q is a finite set of states, $\delta : Q \times \Sigma \longrightarrow Q$ is the transition function, q_0 is the start state, and F is the set of final states.

We can extend δ from $Q \times \Sigma$ to $Q \times \Sigma^*$ by $\overline{\delta}(s, \lambda) = s$, $\overline{\delta}(s, aw) = \overline{\delta}(\delta(s, a), w)$. We will usually denote the extension $\overline{\delta}$ of δ by δ when there is no danger of confusion.

The language recognized by the automaton A is $L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$. In what follows we assume that δ is a total function, i.e., the deterministic automaton is also complete.

For a DFA $A = (\Sigma, Q, \delta, q_0, F)$, we can always assume, without loss of generality, that $Q = \{0, 1, \ldots, |Q| - 1\}$ and $q_0 = 0$. Throughout this paper we will assume that the states are labeled with numbers from 0 to $|Q| - 1$. If L is finite, $L = L(A)$ and A is complete, there is at least one state, called the sink state or dead state, for which $\delta(sink, w) \notin F$, for any $w \in \Sigma^*$. If L is a finite language, we denote by l the maximum among the lengths of all words in L.

For the following definitions we assume that L is a finite language over the alphabet Σ and l is the length of the longest word(s) in L.

Definition 1. *Cover Language. A language L' over Σ is called a cover language for the finite language L if $L' \cap \Sigma^{\leq l} = L$. A deterministic finite cover automaton (DFCA) for L is a deterministic finite automaton (DFA) A, such that the language accepted by A is a cover language of L.*

Definition 2. *State equivalence. Let $A = (\Sigma, Q, \delta, 0, F)$ be a DFA. We say that $p \equiv_A q$ (state p is equivalent to q in A) if for every $w \in \Sigma^*$, $\delta(p, w) \in F$ iff $\delta(q, w) \in F$.*

Definition 3. *Level. Let $A = (\Sigma, Q, \delta, 0, F)$ be a DFA (or a DFCA for L). We define, for each state $q \in Q$, $level(q) = \min\{|w| \mid \delta(0, w) = q\}$.*

The right language of a state $p \in Q$ and for a DFCA $A = (Q, \Sigma, \delta, q_0, F)$ for L is $R_p = \{w \mid \delta(p, w) \in F, |w| \leq l - level_A(p)\}$.

Definition 4. *Word similarity. Let $x, y \in \Sigma^*$. We define the following similarity relation in the following way: $x \sim_L y$ if for all $z \in \Sigma^*$ such that $xz, yz \in \Sigma^{\leq l}$, $xz \in L$ iff $yz \in L$, and we write $x \not\sim_L y$ if $x \sim_L y$ does not hold.*

Definition 5. *State similarity. Let $A = (\Sigma, Q, \delta, 0, F)$ be a DFCA for L. We consider two states $p,\ q \in Q$ and $m = \max\{level(p), level(q)\}$. We say that p is similar with q in A, denoted by $p \sim_A q$, if for every $w \in \Sigma^{\leq l-m}$, $\delta(p, w) \in F$ iff $\delta(q, w) \in F$. We say that two states are dissimilar if they are not similar (the above does not hold).*

If the automaton is understood, we may omit in the following the subscript A when writing the similarity of two states in a DFCA A.

Lemma 1. *Let $A = (\Sigma, Q, \delta, 0, F)$ be a DFCA for a finite language L. Let $p, q \in Q$, with $level(p) = i$, $level(q) = j$, and $m = \max\{i, j\}$. If $p \sim_A q$, then $R_p \cap \Sigma^{\leq l-m} = R_q \cap \Sigma^{\leq l-m}$.*

Proof. See the proof given in [6].

Definition 6. *A DFCA A for a finite language is a minimal DFCA if and only if any two distinct states of A are dissimilar.*

Once two states have been detected as similar, one can merge the higher level one into the lower level one by redirecting transitions. We refer the interested reader to [6] for the merging theorem and other properties of cover automata.

3 Hopcroft's State Minimization Algorithm

In [11], an elegant algorithm for state minimization of DFAs was described. This algorithm was proven to be of the order $O(n\ log\ n)$ in the worst case (asymptotic evaluation). We will study further the complexity of the algorithm by considering the various implementation choices of the algorithm. We will show that by implementing the list of the splitting sets as a queue, one will be able to reach the absolute worst possible case with respect to the number of steps required for minimizing an automaton. We will also show that by changing the implementation strategy from a queue to a stack, we will never be able to reach that absolute worst case in the number of steps for minimizing automata, thus, at least from this perspective, the programmers should implement the list S from the following algorithm as a stack (LIFO).

The algorithm uses a special data structure that makes the set operations of the algorithm fast. We will give in the following the description of the minimization algorithm working on the automaton $(\Sigma, Q, \delta, q_0, F)$ that has an arbitrary alphabet Σ and later we will restrict the discussion to the case of unary languages.

```
1: $P = \{F,\ Q - F\}$
2: for all $a \in \Sigma$ do
3:       Add((min($F,\ Q - F$), $a$), $S$)    (min w.r.t. the number of states)
4: while $S \neq \emptyset$ do
5:       get $(C, a)$ from $S$   (we extract $(C, a)$ according to the
         strategy associated with the list $S$: FIFO/LIFO/...)
6:       for each $B \in P$ that is split by $(C, a)$ do
7:           $B'$, $B''$ are the sets resulting from splitting of $B$ w.r.t. $(C, a)$
8:           Replace $B$ in $P$ with both $B'$ and $B''$
9:           for all $b \in \Sigma$ do
10:              if $(B, b) \in S$ then
11:                  Replace $(B, b)$ by $(B', b)$ and $(B'', b)$ in $S$
12:              else
13:                  Add((min($B', B''$), $b$), $S$)
```

where the splitting of a set B by the pair (C, a) (the line 6) means that $\delta(B, a) \cap C \neq \emptyset$ and $\delta(B, a) \cap (Q - C) \neq \emptyset$. We have denoted above by $\delta(B, a)$ the set $\{q \mid q = \delta(p, a),\ p \in B\}$. The B' and B'' from line 7 are defined as the following two subsets of B: $B' = \{b \in B \mid \delta(b, a) \in C\}$ and $B'' = B - B'$.

It is useful to explain briefly the working of the algorithm: we start with the partition $P = \{F, Q - F\}$ and one of these two sets is then added to the splitting sequence S. The algorithm proceeds by breaking the partition into smaller sets according to the current splitting set retrieved from S. With each splitting of a set in P the number of sets stored in S grows (either through instruction 11 or instruction 13). When all the splitting sets from S are processed, and S becomes empty, then the partition P shows the state equivalences in the input automaton: all the states contained in a same set B in P are equivalent. Knowing all equivalences, one can easily minimize the automaton by merging all the states found in the same set in the final partition P at the end of the algorithm.

We note that there are three levels of "nondeterminism" in the implementation of the algorithm. All these three choices influence the algorithm by changing the number of steps performed for a specific input automaton. We describe first the three implementation choices, and later we show the worst case scenario for each of them.

The "most visible" implementation choice is the choice of the strategy for processing the list stored in S: as a queue, as a stack, etc. The second and third such choices in implementation of the algorithm appear when a set B is split into B' and B''. If B is not present in S, then the algorithm is choosing which set B' or B'' to be added to S, choice that is based on the minimal number of states in these two sets (line 13). In the case when both B' and B'' have the same number of states, then we have the second implementation choice (the choosing of the set that will be added to S). The third such choice appears when the split set (B, a) is in the list S; then the algorithm mentions the replacement of (B, a) by (B', a) and (B'', a) (line 11). This is actually implemented in the following way: (B'', a) is replacing (B, a) and (B', a) is added to the list S (or vice-versa). Since we saw that the processing strategy of S matters, then also the choice of which B' or B'' is added to S and which one replaces the previous location of (B, a) matters in the actual run-time of the algorithm.

In the original paper [11] and later in [9], and [13], when describing the complexity of the minimization method, the authors showed that the algorithm is influenced by the number of states that appear in the sets processed in S. Intuitively, that is why the smaller of the (add) B' and B'' is inserted in S in line 13; and this is what makes the algorithm sub-quadratic. In the following we will focus on exactly this issue of the number of states appearing in sets processed in S.

4 Worst Case Scenario for Unary Languages

Let us start the discussion by making several observations and preliminary clarifications: we are discussing about languages over an unary alphabet. To make the proof easier, we restrict our discussion to the automata having the number of states a power of 2. The three levels of implementation choices are clarified/set in the following way: we assume that the processing of S is based on a FIFO approach, we also assume that there is a strategy of choosing between two sets that have been just splitted. These two sets have the same number of elements in such a way that the one that is added to the queue S makes the third implementation nondeterminism irrelevant. In other words, no splitting of a set already in S will take place (line 11 will not be executed).

Let us assume that such an automaton with 2^n states is given as input for the minimization algorithm described in the previous section. We note that since we have only one letter in the alphabet, the states (C, a) from the list S can be written without any problems as C, thus the list S (for the particular case of unary languages) becomes a list of sets of states. So let us assume that the automaton $A = (\{a\}, Q, \delta, q_0, F)$ is given as the input of the algorithm, where $|Q| = 2^n$. The algorithm proceeds by choosing the first splitter set to be added to S. The first such set will be chosen between F and $Q - F$ based on their number of states. Since we are interested in the worst case scenario for the algorithm, and the algorithm run-time is influenced by the total number of states that will appear in the list S throughout the running of the algorithm (as shown in [11], [9], [13] and mentioned in [2]), it is clear that we want to maximize the sizes of the sets that are added to S. It is time to give a Lemma that will be useful in the following.

Lemma 2. *For deterministic automata over unary languages, if a set R with $|R| = m$ is the current splitter set, then R cannot add to the list S sets containing all together more than m states (line 13).*

Proof. We can rephrase the Lemma as: for all the sets B_i from the current partition P such that $\delta(B_i, a) \cap R \neq \emptyset$ and $\delta(B_i, a) \cap (Q - R) \neq \emptyset$. Then $\sum_{\forall\ i} |B'_i| \leq m$, where B'_i is the smaller of the two sets that result from the splitting of the set $B_i \in P$ with respect to R.

We have only one letter in the alphabet, thus the number of states q such that $\delta(q, a) \in R$ is at most m. Each B'_i is chosen as the set with the smaller number of states when splitting B_i thus $|B'_i| \leq |\delta(B_i, a) \cap R|$ which implies that $\sum_{\forall\ i} |B'_i| \leq \sum_{\forall\ i} |\delta(B_i, a) \cap R| = |(\bigcup_{\forall\ i} \delta(B_i, a)) \cap R| \leq |R|$ (because all B_i are disjoint).

Thus we proved that if we start splitting according to a set R, then the new sets added to S contain at most $|R|$ states. □

Coming back to our previous setting, we will start with the automaton $A = (\{a\}, Q, \delta, q_0, F)$ (where $|Q| = 2^n$) given as input to the algorithm and we have to find the smaller set between F and $Q - F$. In the worst case (according to Lemma 2) we have that $|F| = |Q - F|$, as otherwise, fewer than 2^{n-1} states will be contained in the set added to S and thus less states will be contained in the sets added to S in the second stage of the algorithm, and so on. So in the worst case we have that the number of final states and the number of non-final states is the same. To simplify the discussion we will give some notations. We denote by $S_w,\ w \in \{0, 1\}^*$ the set of states $p \in Q$ such that $\delta(p, a^{i-1}) \in F$ iff $w_i = 1$ for $i = 1..|w|$, where $\delta(p, a^0)$ denotes p. As an example, $S_1 = F$, S_{110} contains all the final states that are followed by a final state and then by a non-final state and S_{00000} denotes the states that are non-final and are followed in the automaton by four more non-final states.

With these notations we have that at this initial step of the algorithm, either $F = S_1$ or $Q - F = S_0$ can be added to S as they have the same number of states. Either one that is added to the queue S will split the partition P in the worst case scenario in the following four possible sets $S_{00}, S_{01}, S_{10}, S_{11}$, each with 2^{n-2} states. This is true as by splitting the sets F and $Q - F$ in sets with sizes other than 2^{n-2}, then according to Lemma 2 we will not reach the worst possible number of states in the queue S and also splitting only F or only $Q - F$ will add to S only one set of 2^{n-2} states not two of them.

All this means that half of the non-final states go to a final state ($|S_{01}| = 2^{n-2}$) and the other half go to a non final state (S_{00}). Similarly, for the final states we have that 2^{n-2} of them go to a final state (S_{11}) and the other half go to a non-final state. The current partition at this step 1 of the algorithm is $P = \{S_{00}, S_{01}, S_{10}, S_{11}\}$ and the splitting sets are one of the S_{00}, S_{01} and one of the S_{10}, S_{11}. Let us assume that it is possible to chose the splitting sets to be added to the queue S in such a way so that no splitting of another set in S will happen, (chose in this case for example S_{10} and S_{00}). We want to avoid splitting of other sets in S since if that happens, then smaller sets will be added to the queue S by the splitted set in S (see such a choice of splitters described also in [2]).

We have arrived at step 2 of the algorithm. Since the first two sets from S are now processed, in the worst case they will be able to add to the queue S at most 2^{n-2} states each by each splitting two of the four current sets in the partition P. Of course, to reach the worst case, we need them to split different sets, thus we obtain eight sets in the partition P corresponding to all the possibilities of words of length 3 on a binary alphabet: P=$\{S_{000}, S_{001}, S_{010}, S_{011}, S_{100}, S_{101}, S_{110}, S_{111}\}$ having 2^{n-3} states each. Thus four of these sets will be added to the queue S. And we could continue our reasoning up until the i-th step of the algorithm:

We now have 2^{i-1} sets in the queue S, each having 2^{n-i} states, and the partition P contains 2^i sets S_w corresponding to all the words w of the length i. Each of the sets in the splitting queue is of the form $S_{x_1x_2\ldots x_i}$, then a set $S_{x_1x_2x_3\ldots x_i}$ can only split at most two other sets $S_{0x_1x_2x_3\ldots x_{i-1}}$ and $S_{1x_1x_2x_3\ldots x_{i-1}}$ from the partition P. To reach the worst case for the algorithm, no set (of level i) from the splitting queue should be splitting a set already in the queue, and also, it should split 2 distinct sets in the partition P, making the partition at step $i+1$ the set $P = \{S_w \mid |w| = i+1\}$. Furthermore, each such S_w should have exactly 2^{n-i-1} states. In this way the process continues until we arrive at the n-th step. If the process would terminate before the step n, of course we would not reach the worst possible number of states passing through S.

We will now describe the properties/restrictions of an automaton that would obey a processing through the Hopcroft's algorithm as described above (for the worst case scenario). We started with 2^n states, out of which we have 2^{n-1} final and also 2^{n-1} non-final. Out of the final states, we have 2^{n-2} that precede another final state (S_{11}), and also 2^{n-2} non-final states that precede other non-final states for S_{00}, etc. The strongest restrictions are found in the final partition sets S_w, with $|w| = n$, each have exactly one element, which means that all the words of length n over the binary alphabet can be found in this automaton by following the transitions between states and having 1 for a final state and 0 for a non-final state. It is clear that the automaton needs to be circular and following the pattern of de Bruijn words, [3]. Such an automaton for $n = 3$ was depicted in [2] as in the following Figure 1.

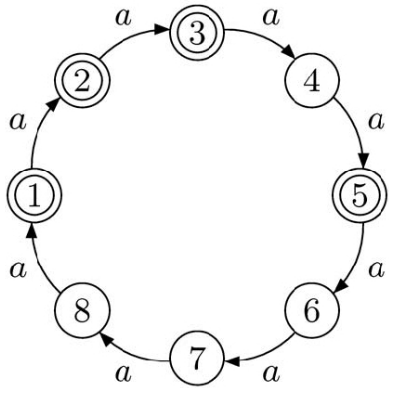

Fig. 1. A cyclic automaton of size 8 for the de Bruijn word 11101000, containing all words of size 3 over the binary alphabet $\{0,\ 1\}$

It is easy to see now that a stack implementation for the list S will not be able to reach the maximum as smaller sets will be processed before considering larger sets. This fact will lead to splitting of sets already in the list S. Once this happens for a set with j states, then the number of states that will appear in S is decreased by at least j because the splitted sets will not be able to add as many states as a FIFO implementation was able to do. We conjecture that in such a setting the LIFO strategy could prove the algorithm linear with respect to the size of the input. We reference [15] for previous work in this direction. If the aforementioned third level of implementation choice is set to add the smaller set of B', B'' to the stack and B to be replaced by the larger one. We proved the following result:

Theorem 1. *The absolute worst case run-time complexity for the Hopcroft's minimization algorithm for unary languages is reached when the splitter list S in the algorithm is following a FIFO strategy and only for automata having a structure induced by de Bruijn words of size n. In that setting the algorithm will pass through the queue S exactly $n2^{n-1}$ states for the input automaton of size 2^n. Thus for m states of the input automaton we have exactly $\frac{m}{2}log_2m$ states passing through S.*

Proof. Due to the previous discussion we now know that the absolute maximum for the complexity of the Hopcroft's algorithm is reached in the case of the FIFO strategy for the splitter list S. The maximum is reached when the input automaton is following the structure of the *de Bruijn* words for a binary alphabet.

What remains to be proven is the actual number of states that pass through the queue S: in the first stage exactly half of all states are added to S through one of the sets S_0 or S_1, in the second stage half of the states are again added to S through two of the four sets $S_{00}, S_{01}, S_{10}, S_{11}$. At the third stage half of states are added to S because four of the following eight sets $S_{000}, S_{001}, S_{010}, S_{011}, S_{100}, S_{101}, S_{110}, S_{111}$ are added to S, each having exactly 2^{n-3} states. We continue this process until the last stage of the algorithm, stage n: when still 2^{n-1} states are added to S through the fact that exactly 2^{n-1} sets, each containing exactly one state, are added to the splitting queue. Of course, at this stage we have the partitioning into $\{S_w \mid |w| = n\}$ and half of these sets will be added to S through the instruction at line 13. It should be now clear that we have exactly n stages in this execution of the algorithm, each with 2^{n-1} states added to S, hence the result.

5 Stacks

In this section we will consider the case of the LIFO implementation for the splitting sequence S used in the algorithm. Let us assume that we start with a minimal automaton having 2^n states.

Following the steps of the procedure and lemma 2 one can easily see that in the list S we can have in the worst case scenario the following number of states: $2^{n-1} \mid 2^{n-2}, 2^{n-2} \mid 2^{n-3}, 2^{n-3}, 2^{n-2} \mid 2^{n-4}, 2^{n-4}, 2^{n-3}, 2^{n-2} \mid \ldots$ and finally $2, 2, 2^2, 2^3, 2^4, \ldots, 2^{n-4}, 2^{n-3}, 2^{n-2}$

In this way one reaches in n steps a set in S that contains exactly one state. At this moment this single state will be partitioning all the states that lead to this single state in the automaton, thus if the state is on the loop of the automaton in exactly another n steps all the states will be in their own partitions and the algorithm finishes. The worst case scenario is when this first partitioned state is actually the start state and this state does not have any incoming transitions, thus it does not split any other state. This case is still not reaching the $O(n2^n)$ states in S because at the next step another single state will be in its own partition set in P. To be in the worst case, this state has to be very near of the start state of the automaton (otherwise all the states that precede it will be split and added to the partition P). We will have an increasing number of states that are partitioned by much smaller states which is bringing down the number of the states that pass through S. Another observation is the following: the first $n - 1$ sets that appear in S are actually coming from specific splittings; if one considers that the first single state in a set in S is $S_{x_1x_2x_3\ldots x_n}$ then it comes from the splitting of $S_{x_1x_2\ldots x_{n-1}}$ through $S_{x_2x_3\ldots x_{n-1}Y}$. Through a careful analysis we notice that the restrictions on words do not allow an "explosion" of the number of states that can be added to the list S, so in linear time we obtain the most refined partition P.

When considering the start automaton (still unary) non-minimal, the procedure is changed a little as the actual discussion is delayed by at most $log_2 a$ where a is the period of the loop. The main observation is that even though the procedure will choose the set by the smallest number of states in the set, that set could actually contain more "minimal states" than the set that was not added to S, thus the "lag" in the algorithm amounts to $log_2 a$ with a being the maximal number of states that are in an equivalence class in the automaton. The discussion proceeds in a similar fashion, still in at most $n + log_2 a$ steps we reach the first state in the minimal automaton, and after that stage the stack implementation makes the order of consideration of the splitting of partition in such a way that it is linear.

Following this discussion we showed in a sketched way that the Hopcroft's algorithm with stack implementation applied for unary automata has a linear time requirement for completion.

6 Cover Automata

In this section we discuss briefly about an extension to Hopcroft's algorithm to cover automata. Körner reported at CIAA'02 and also in [14] a modification of the Hopcroft's algorithm so that the resulting sets in the partition P will give the similarities between states with respect to the input finite language L.

To achieve this, the algorithm is modified as follows: each state will have its level computed at the start of the algorithm; each element added to the list S will have three components: the set of states, the alphabet letter and the current length considered. We start with $(F, a, 0)$ for example. Also the splitting of a set B by (C, a, l_1) is defined as before with the extra condition that we ignore during the splitting the states that have their level$+l_1$ greater than l (l being the longest word in the finite language L). Formally we can define the sets $X = \{p \mid \delta(p,a) \in C,\ level(p) + l_1 \leq l\}$ and $Y = \{p \mid \delta(p,a) \notin C,\ level(p) + l_1 \leq l\}$. Then a set B will be split only if $B \cap X \neq \emptyset$ and $B \cap Y \neq \emptyset$.

The actual splitting of B ignores the states that have levels higher than or equal to $l - l_1$. This also adds a degree of implementation nondeterminism to the algorithm when such states appear because the programmer can choose to add these sets in either of the two splitted sets obtained from B. The worst implementation choice would be to put the states with level higher than $l - l_1$ in such a way that they balance the number of states in both B' and B'' (where $B' = X \cup Z'$ and $B'' = Y \cup Z''$ and $Z' \cap Z'' = \emptyset$, and $Z' \cup Z'' = B - (X \cup Y)$ are all the states of level higher than or equal to $l - l_1$). We note that this is an obviously "bad" implementation choice, thus we assume that the programmer would avoid it. We will make in this case the choice to have the states split as in the case of DFA, according to whether $\delta(p, a) \in C$, then $p \in X$, otherwise, $p \in Y$. This choice will make the Lemma 2 valid also for the Cover automata case, with the modifications to the algorithm mentioned above.

The algorithm proceeds as before to add the smaller of the newly splitted sets to the list S together with the value $l_1 + 1$.

Let us now consider the same problem as in [2], but in this case for the case of DFCA minimization through the algorithm described in [14]. We will consider the same example as before, the automata based on de Bruijn words as the input to the algorithm (we note that the modified algorithm can start directly with a DFCA for a specific language, thus we can have as input even cyclic automata). We need to specify the actual length of the finite language that is considered and also the starting state of the de Bruijn automaton (since the algorithm needs to compute the levels of the states). We can choose the length of the longest word in L as $l = 2^n$ and the start state as $S_{111...1}$. For example, the automaton in figure 1 would be a cover automaton for the language $L = \{0, 1, 2, 4, 8\}$ with $l = 8$ and the start state $q_0 = 1$. Following the same reasoning as in [2] but for the case of the new algorithm with respect to the modifications, we can show that also for the case of DFCA a queue implementation (as specifically given in [14]) is a worse choice than a stack implementation for S. We note that the discussion is not a straight-forward extension of the work reported by Berstel in [2] as the new dimension added to the sets in S, the length and also the levels of states need to be discussed in detail. We will give the details of the construction and the step-by-step discussion of this fact in the following:

We start as before with an automaton with 2^n states working on a unary language given as: $A = (\{a\}, Q, \delta, q_0, F)$ where $|Q| = 2^n$. Let us take a look at the possible levels of the states in deterministic automata over unary languages: Such an automaton is formed by a line followed by a loop. The line or the loop can be possibly empty: if the loop is empty (or containing only non-final states), then the automaton accepts only a finite set of numbers, if the loop contains at least one state that is final, it accepts an infinite set. In either case the levels of the states is $0,\ 1,\ 2,\ 3,\ ...,\ n-2,\ n-1$. One can see that the highest level in such a unary DFA is at most $n-1$.

Following the variant of Lemma 2 for DFCA it is clear that the worst possible case is when $|F| = |Q - F|$. Let us consider that S starts with the pair $(F, 0)$ or $(Q - F, 0)$, in either case at the second stage of the algorithm the partition P will be split in the following four possible sets (similarly as in the case of DFA): $S_{00}, S_{01}, S_{10}, S_{11}$. To continue with the worst possible case, each of these sets need to contain exactly 2^{n-2} states (otherwise, according to Lemma 2 for the DFCA case, a set with less states is added to S and also in the next steps less states will be added to S). Also in this case it is necessary to make a "bad" choice of the sets that will be added next to S (one from the S_{00}, S_{01} and one from S_{10}, S_{11}). We will use the same choosing strategy as before. The difference is that these sets will be added to S and with the length 1: for example, at the next step S will contain $(S_{00}, 1)$ and $(S_{10}, 1)$. At the next stage of the algorithm we will observe a difference from the DFA case: one of the states at the next stage will not be splitted from the set because of its high level. Considering that we have a state of level $l-1$, at this step this high-level state will not be considered for splitting, thus can be added to either one or the other of the halves of the state containing it. For the final automaton, considering that the state $S_{11..1}$ is the start state, the high level state is $S_{011..1}$.

We continue the process in a similar fashion until the i-th stage of the algorithm by carefully choosing the splitting sets, and by having at each stage yet another state that would not be considered in the splitting due to its high level. But because the forth implementation choice, the number of states in each set remains the same. At this moment we will have 2^{i-1} pairs in the queue S, each formed between a set containing 2^{n-i} states and the value $i-1$. Thus we will compute the splitter sets X and Y as given before in the case of DFA with the extra condition that the sets p satisfying the condition also satisfy the fact that $level(p) + i - 1 \leq l$.

At this moment the partition P has exactly 2^i components that are in the worst case exactly the sets S_w for all $w \in \{0,1\}^i$. In the worst case all the level i states from the splitting queue S will not break a set already in the queue S, but at the same time will split two other sets in the partition P. This is achieved by the careful choosing of the order in which these sets arrive in the queue (one of these "worst" additions to S strategies was described in [2]). In this way, at the end of the stage i in the algorithm we will have the partition P containing all the sets S_w with $w \in \{0,1\}^{i+1}$ and each of them having 2^{n-i-1} states. In the queue S there will be 2^i pairs of states with the number i. These splitting pairs will be used in the next stage of the algorithm.

This process will continue until the $n-1$-th stage as before (otherwise we will not be in the worst possible case) and at the n-th stage exactly $n-2$ sets will not be added to the queue S (as opposed to the DFA case), thus only $2^{n-1} - n + 2$ singleton sets will be added.

This makes the absolute worst case for the run-time of the minimization of DFCA based on Hopcroft's method have exactly $n2^{n-1} - n + 2$ states pass through S. The input automaton still follows the structure induced by de Bruijn words; and when considering the start state as $S_{11\ldots1}$, the states that will be similar with other states are the $n-2$ states of highest levels: $S_{011\ldots1}, S_{001..1}, \ldots, S_{00\ldots011}$. In fact we will have several similarities between these high level states and other states in the automaton, more precisely, for an automaton with 2^n states (following the structure of de Bruijn words containing all the subwords of size n) we have the following pattern of similarities: the state $S_{011\ldots1}$ will have exactly $2^{n-2} - 1$ similarities with other states in the automaton (because the level of this state is $2^n - 1$, thus only the pattern 01 is making the difference to other states), for $S_{001\ldots1}$ we will have $2^{n-3} - 1$ similarities (as for its level $2^n - 2$ the pattern making the difference is 001), and so on, until $S_{000\ldots01}$ will actually have $2^{n-(n-1)} - 1 = 2 - 1 = 1$ similarities (since its level is $2^n - n + 2$ and the pattern making the difference is 000...01). These values are obtained from considering the fact that the structure of the automaton will have all the sub-words of size n, thus we can compute how many times a particular pattern appears in the automaton.

This shows that a result similar to Theorem 1 holds also for the case of DFCA with the only difference in the counting of states passing through S: $n2^{n-1} - n + 2$ rather than $n2^{n-1}$. It should be clear now that a stack implementation for the list

S is more efficient, at least for the case of unary languages and when considering the absolute worst possible run-time of the algorithm.

7 Final Remarks

We showed that at least in the case of unary languages, a stack implementation is more desirable than a queue for keeping track of the splitting sets in Hopcroft's algorithm. This is the first instance when it was shown that the stack is outperforming the queue. Thus at least for the special case of unary languages we know that it is better to have the implementation of S in the algorithm as a stack rather than the intuitive implementation as a queue.

It remains open whether these results can be extended to languages containing more than one letter in the alphabet.

For the case of cover automata one should settle the extra implementation choice (the forth implementation choice as mentioned in the text) as follows: rather than balancing the number of states in the two splitted sets, actually try to un-balance them by adding all the high level states to the bigger set. These remarks should achieve a reasonable speed-up for the algorithm.

References

1. Baclet, M., Pagetti, C.: Around Hopcroft's Algorithm. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 114–125. Springer, Heidelberg (2006)
2. Berstel, J., Carton, O.: On the complexity of Hopcrofts state minimization algorithm. In: Domaratzki, M., Okhotin, A., Salomaa, K., Yu, S. (eds.) CIAA 2004. LNCS, vol. 3317, pp. 35–44. Springer, Heidelberg (2005)
3. de Bruijn, N.G.: A Combinatorial Problem. Koninklijke Nederlandse Akademie v. Wetenschappen 49, 758–764 (1946)
4. Câmpeanu, C., Păun, A., Yu, S.: An Efficient Algorithm for Constructing Minimal Cover Automata for Finite Languages. International Journal of Foundations of Computer Science 13(1), 83–97 (2002)
5. Câmpeanu, C., Salomaa, K., Yu, S.: Tight Lower Bound for the State Complexity of Shuffle of Regular Languages. Journal of Automata, Languages and Combinatorics 7(3), 303–310 (2002)
6. Câmpeanu, C., Sântean, N., Yu, S.: Minimal Cover-Automata for Finite Languages. In: Champarnaud, J.-M., Maurel, D., Ziadi, D. (eds.) WIA 1998. LNCS, vol. 1660, pp. 32–42. Springer, Heidelberg (1999); Theoretical Computer Science 267, 3–16 (2001)
7. Champarnaud, J.M., Maurel, D.: Automata Implementation. In: Champarnaud, J.-M., Maurel, D., Ziadi, D. (eds.) WIA 1998. LNCS, vol. 1660. Springer, Heidelberg (1999)
8. Domaratzki, M., Shallit, J., Yu, S.: Minimal Covers of Formal Languages. In: Kuich, W., Rozenberg, G., Salomaa, A. (eds.) DLT 2001. LNCS, vol. 2295, pp. 319–329. Springer, Heidelberg (2002)
9. Gries, D.: Describing an algorithm by Hopcroft. Acta Informatica 2, 97–109 (1973)
10. Hopcroft, J.E., Ullman, J.D., Motwani, R.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading (2001)

11. Hopcroft, J.E.: An *n log n* algorithm for minimizing states in a finite automaton. In: Kohavi, Z., Paz, A. (eds.) Theory of Machines and Computations, pp. 189–196. Academic Press, London (1971)
12. Ilie, L., Yu, S.: Follow automata. Inf. Comput. 186(1), 140–162 (2003)
13. Knuutila, T.: Re-describing an algorithm by Hopcroft. Theoretical Computer Science 250(1-2), 333–363 (2001)
14. Körner, H.: A Time and Space Efficient Algorithm for Minimizing Cover Automata for Finite Languages. International Journal of Foundations of Computer Science 14(6), 1071–1086 (2003)
15. Paige, R., Tarjan, R.E., Bonic, R.: A Linear Time Solution to the Single Function Coarsest Partition Problem. Theoretical Computer Science 40, 67–84 (1985)
16. Păun, A., Santean, N., Yu, S.: An $O(n^2)$ Algorithm for Constructing Minimal Cover Automata for Finite Languages. In: Yu, S., Păun, A. (eds.) CIAA 2000. LNCS, vol. 2088, pp. 243–251. Springer, Heidelberg (2001)
17. Salomaa, A.: Formal Languages. Academic Press, London (1973)
18. Salomaa, K., Wu, X., Yu, S.: Efficient Implementation of Regular Languages Using Reversed Alternating Finite Automata. Theor. Comput. Sci. 231(1), 103–111 (2000)
19. Sântean, N.: Towards a Minimal Representation for Finite Languages: Theory and Practice. MSc Thesis, Department of Computer Science, The University of Western Ontario (2000)
20. Yu, S.: Regular Languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, pp. 41–110. Springer, Heidelberg (1998)
21. Yu, S.: State Complexity of Finite and Infinite Regular Languages. Bulletin of the EATCS 76, 142–152 (2002)
22. Yu, S.: State Complexity: Recent Results and Open Problems. Fundam. Inform. 64(1-4), 471–480 (2005)
23. Yu, S.: On the State Complexity of Combined Operations. In: Ibarra, O.H., Yen, H.-C. (eds.) CIAA 2006. LNCS, vol. 4094, pp. 11–22. Springer, Heidelberg (2006)
24. Wood, D., Yu, S.: Automata Implementation. In: Proceedings of Second International Workshop on Implementing Automata. LNCS, vol. 1436. Springer, Heidelberg (1998)
25. The Grail + Project. A symbolic computation environment for finite state machines, regular expressions, and finite languages, `http://www.csd.uwo.ca/research/grail/`