RAIRO-Theor. Inf. Appl. 45 (2011) 59–75
DOI: 10.1051/ita/2011011

Available online at:
www.rairo-ita.org

HOPCROFT'S ALGORITHM AND TREE-LIKE AUTOMATA *

G. Castiglione[1], A. Restivo[1] and M. Sciortino[1]

Abstract. Minimizing a deterministic finite automata (DFA) is a very important problem in theory of automata and formal languages. Hopcroft's algorithm represents the fastest known solution to the such a problem. In this paper we analyze the behavior of this algorithm on a family binary automata, called tree-like automata, associated to binary labeled trees constructed by words. We prove that all the executions of the algorithm on tree-like automata associated to trees, constructed by standard words, have running time with the same asymptotic growth rate. In particular, we provide a lower and upper bound for the running time of the algorithm expressed in terms of combinatorial properties of the trees. We consider also tree-like automata associated to trees constructed by de Brujin words, and we prove that a queue implementation of the waiting set gives a $\Theta(n \log n)$ execution while a stack implementation produces a linear execution. Such a result confirms the conjecture given in [A. Paun, M. Paun and A. Rodríguez-Patón. *Theoret. Comput. Sci.* **410** (2009) 2424–2430.] formulated for a family of unary automata and, in addition, gives a positive answer also for the binary case.

Mathematics Subject Classification. 68Q45, 68Q25.

Introduction

Minimization of deterministic finite automata (DFA) is a classical and widely studied problem in Theory of Automata and Formal Languages. It consists in finding the unique (up to isomorphism) finite automaton with the minimal number of states, recognizing the same regular language of a given DFA.

Keywords and phrases. Automata minimization, Hopcroft's algorithm, word trees.

* *Supported by the MIUR Project PRIN 2007 Mathematical aspects and emerging applications of automata and formal languages*

[1] Università di Palermo, Dipartimento di Matematica e Applicazioni, via Archirafi, 34, 90123 Palermo, Italy; giusi@math.unipa.it

Article published by EDP Sciences

Describing a regular language by its minimal automaton is important in many applications, such as, for instance, text searching, lexical analysis or coding systems, where space considerations are prominent.

Several methods have been developed to minimize a deterministic finite automaton. Some of them operate by successive refinements of a partition of the states leading to the coarsest congruence of the automaton. For instance, we recall the well known algorithm proposed by Moore in 1956 (*cf.* [11]) with time complexity $O(kn^2)$, where n is the number of states of the DFA and k is the cardinality of the alphabet. More efficient is the algorithm provided by Hopcroft in 1971 (*cf.* [8]). It works by successive refinements and compute the the minimal automaton in $O(kn \log n)$. Besides, such an algorithm is the fastest known solution to the automata minimization problem.

The Hopcroft's algorithm has some degrees of freedom due to the fact that at each iteration of the main loop a free choice is allowed. One of the most relevant degrees of freedom is related to the fact that in the algorithm an auxiliary data structure, called *waiting set*, is used. Such a structure contains some subsets of the set of all states that are processed in order to produce successive refinements of the partition of states. Hence, different implementations of the waiting set involve different executions that could lead to the same sequence of refinements or not. Each of these executions can involve different running time. In [1] the authors defined an infinite family of unary cyclic automata associated to the de Brujin words. They described an execution with running time $\Omega(n \log n)$ for such automata and stated that also a linear execution exists. In [13,14] the authors proved that the absolute worst case for unary automata is reached only for this kind of automata. Furthermore, they prove that such a worst execution is obtained when the waiting set is implemented as a queue and they conjectured that the stack implementation leads to a linear time execution. In [5] we provided an infinite family of unary cyclic automata for which the execution is unique independently from the implementation of the waiting set. Such automata are associated to a very well-known class of words, called *standard words* [10]. In [5] we proved that for unary cyclic automata associated to finite Fibonacci words there is a unique execution with running time $\Theta(n \log n)$. Such a result is extended in [3].

Although there exist unary automata for which Hopcroft's algorithm runs effectively in $\Theta(n \log n)$, we want to underline that automata minimization can be achieved in linear time when the alphabet has only one letter (*cf.* [12]). But, the solution does not seem to extend to larger alphabet. Motivated by this fact, in [6,7] we analyzed the tightness of the algorithm when the alphabet contains more than one letter together with the uniqueness of the refinement process of the set of states. We defined an infinite family of binary automata associated to particular finite labeled binary trees, called *standard tree*, for which the refinement process during Hopcroft's algorithm is unique even if there could be different executions. Also in the binary case we gave an infinite family of automata associated to particular standard trees, constructed by using finite Fibonacci words, that are minimized in $\Theta(n \log n)$ by the algorithm.

In this paper we study the binary case in detail. We consider automata associated to binary trees that we call *word trees* because constructed by words. We provide a lower and upper bound for the running time of the algorithm when it is applied on automata associated to word trees constructed by standard words. Such bounds are expressed in terms of combinatorial properties of the trees. We prove that, for this case, such bounds have the same asymptotic growth rate. We conclude that all the possible executions of Hoprcroft's algorithm on automata associated to word trees constructed by standard words produce the same sequence of refinements and have running time of the same order.

Although in the case of standard word trees the choice of the implementation of the waiting set does not imply substantial differences in the running time, there exist binary automata associated to word trees for which some executions have running times of different order. Indeed, we prove that, in case of binary automata associated to word trees constructed by de Brujin words, a queue implementation of the waiting set gives a $\Theta(n \log n)$ execution while a stack implementation produces a linear execution. Such a result confirms the conjecture given in [14] in addition to giving a positive answer also for the binary case.

1. Hopcroft's algorithm

In 1971 Hopcroft proposed an algorithm for minimizing a deterministic finite state automaton with n states, over an alphabet Σ, in $O(|\Sigma| n \log n)$ time (*cf.* [8]). This algorithm has been widely studied and described by many authors (see for example [9,15]) cause of the difficult to give its theoretical justification, to prove correctness and to compute running time.

In Section 1.1 we give a general description of the algorithm by focusing our attention on its degrees of freedom. In Section 1.2 we describe the behavior of the algorithm on unary automata by focusing our attention on families of automata that represent the worst case of the algorithm and for which the algorithm has a unique execution.

1.1. The algorithm

In Figure 1 we give a brief description of the algorithm's running.

Given an automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, it computes the coarsest congruence that saturates F (*i.e.* F is the union of some of the equivalence classes). Let us observe that the partition $\{F, Q \setminus F\}$, trivially, saturates F. Given a partition $\Pi = \{Q_1, Q_2, \ldots, Q_m\}$ of Q, we say that the pair (Q_i, a), with $a \in \Sigma$, *splits* the class Q_j if $\delta_a^{-1}(Q_i) \cap Q_j \neq \emptyset$ and $Q_j \not\subseteq \delta_a^{-1}(Q_i)$. In this case, the class Q_j is split into $Q'_j = \delta_a^{-1}(Q_i) \cap Q_j$ and $Q''_j = Q_j \setminus \delta_a^{-1}(Q_i)$. Furthermore, we have that a partition Π is a congruence if and only if for any $1 \leq i, j \leq m$ and any $a \in \Sigma$, the pair (Q_i, a) does not split Q_j.

Hopcroft's algorithm produces a sequence $\Pi_1, \Pi_2, \ldots, \Pi_l$ of successive refinements of a partition of the states and it is based on the so-called "smaller half" strategy. Actually, it starts from the partition $\Pi_1 = \{F, Q \setminus F\}$ and refines it by

```
HOPCROFT MINIMIZATION (A = (Q, Σ, δ, q0, F))
 1. Π ← {F, Q \ F}
 2. for all a ∈ Σ do
 3.       W ← {(min(F, Q \ F), a)}
 4. while W ≠ ∅ do
 5.       choose and delete any (C, a) from W
 6.       for all B ∈ Π do
 7.            if B is split from (C, a) then
 8.                 B' ← δ_a^{-1}(C) ∩ B
 9.                 B'' ← B \ δ_a^{-1}(C)
10.                 Π ← Π \ {B} ∪ {B', B''}
11.                 for all b ∈ Σ do
12.                      if (B, b) ∈ W then
13.                           W ← W \ {(B, b)} ∪ {(B', b), (B'', b)}
14.                      else
15.                           W ← W ∪ {(min(B', B''), b)}
```

FIGURE 1. Hopcroft's algorithm.

means of splitting operations until it obtains a congruence, *i.e.* until no split is possible. To do that it maintains the current partition Π_i and a set $\mathcal{W} \subseteq \Pi_i \times \Sigma$, called *waiting set*, that contains the pairs for which it has to check whether they split some classes of the current partition. The main loop of the algorithm takes and deletes one pair (C, a) from $\mathcal{W}$ and, for each class B of Π_i, checks if it is split by (C, a). If it is the case, the class B in Π_i is replaced by the two sets B' and B'' obtained from the split. For each $b \in \Sigma$, if $(B, b) \in \mathcal{W}$, it is replaced by (B', b) and (B'', b), otherwise the pair $(\min(B', B''), b)$ is added to $\mathcal{W}$ (with the notation $\min(B', B'')$ we mean the set with minimum cardinality between B' and B''). Let us observe that a class is split by (B', b) if and only if it is split by (B'', b), hence, the pair $(\min(B', B''), b)$ is chosen for convenience.

We call *refinement process* the sequence of successive refinements *i.e.* the different partitions produced during the execution of the algorithm.

We point out that the algorithm has a degree of freedom because the pair (C, a) to be processed at each iteration is freely chosen. Another free choice is allowed when a set B is split into B' and B'' with the same size and it is not present in $\mathcal{W}$. In this case, the algorithm can, indifferently, add to $\mathcal{W}$ either B' or B''.

Such considerations imply that there can be several refinement processes and different executions depending on the data structure used to implement the waiting set. Hence, for a given automaton, there could be different executions with different time complexity.

In the following example we describe two different executions of the Hopcroft's algorithm on the same automaton.

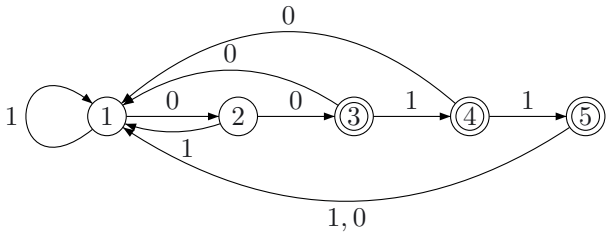

FIGURE 2. An automaton for which the Hopcroft's execution is not uniquely determined. Initial state is labeled by 1.

Example 1.1. Let us consider the automaton in Figure 2. Two possible different refinement processes are

$$\Pi_1 = \{[1,2],[3,4,5]\}, \Pi_2 = \{[1],[2],[3,4,5]\}, \Pi_3 = \{[1],[2],[3,4],[5]\},$$

$$\Pi_4 = \{[1],[2],[3],[4],[5]\},$$

and

$$\Pi_1 = \{[1,2],[3,4,5]\}, \Pi_2 = \{[1,2],[3,4],[5]\}, \Pi_3 = \{[1],[2],[3,4],[5]\}$$

$$\Pi_4 = \{[1],[2],[3],[4],[5]\}.$$

Indeed at the first step the waiting set contains the pairs $([1,2],0)$ and $([1,2],1)$. If we choose the first pair, the class $[1,2]$ is split in $[1]$ and $[2]$, if we choose the second one, the class $[3,4,5]$ is split in $[3,4]$ and $[5]$.

As regards the running time of the algorithm we can observe that the splitting of classes of the partition, with respect to the pair (C,a) extracted from the waiting set, takes a time proportional to the cardinality of the set C. Hence, the running time of the algorithm is proportional to the sum of the cardinality of all sets processed. Since the sequence of the extractions from the waiting set depends on its implementation we have that different implementations could produce different running time. Hopcroft proved that the running time is bounded by $O(|\Sigma||Q|\log|Q|)$.

1.2. Unary worst case

In [1] the authors proved that the bound of the Hopcroft's algorithm is tight. They provided a family of unary automata for which there exist a sequence of refinements such that the time complexity of the algorithm is $\Theta(|\Sigma||Q|\log|Q|)$. Such a family is composed of cyclic automata defined as follows.

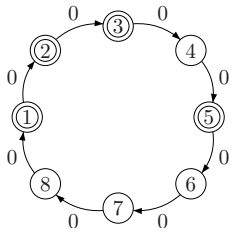

FIGURE 3. Unary cyclic automaton $\mathcal{A}_w$ for de Brujin word $w = bbbabaaa$.

Definition 1.2. Let w be a binary word, where $w = a_1 a_2 \dots a_n$ is a word of length n over the binary alphabet $A = \{a, b\}$. The *cyclic automaton associated to* w, denoted by $\mathcal{A}_w$, is the automaton (Q, Σ, δ, F) such that:

- $Q = \{1, 2, \dots, n\}$
- $\Sigma = \{0\}$
- $\delta(i, 0) = (i + 1)$, $\forall\ i \in Q \setminus \{n\}$ and $\delta(n, 0) = 1$
- $F = \{i \in Q |\ a_i = b\}$.

In Figure 3 the automaton associated to de Brujin word *bbbabaaa* containing all words of length 3 is depicted. Such automata associated to de Brujin words are the automata given in [1].

However, for the same automata there exist other sequences of refinements produced by executions that run in linear time. In [14] the authors considered the running time of the algorithm for these automata and compute the actual number of steps. In particular, they proved that the absolute worst case running time complexity for the Hopcroft's algorithm for unary automata is reached when the waiting set is implemented by a FIFO strategy and only for cyclic automata associated to de Brujin words. Moreover, they proved that an implementation by a stack is better and they conjectured that such a strategy gives a linear running time.

In [5] we presented a family of cyclic unary automata for which there is a unique sequence of refinements and unique execution whatever data structure is chosen to implement the waiting set.

Furthermore, we defined a subclass of such a family of automata for which the running time is $\Theta(|\Sigma||Q| \log |Q|)$. This is the family of cyclic unary automata associated to Fibonacci finite words. In Figure 4 the unary cyclic automaton $\mathcal{A}_w$ associated to the Fibonacci word $w = abaababa$ is depicted.

Such a result is stated in the following theorem proved in [5].

Theorem 1.3. *Hopcroft's algorithm on cyclic automata associated to finite Fibonacci words has a unique execution that runs in time* $\Theta(|Q|log|Q|)$.

Such a subclass of unary automata was extended in [2].

Actually, unary automata represent a very special case for the automata minimization problem. In fact, the minimization can be achieved also in linear time

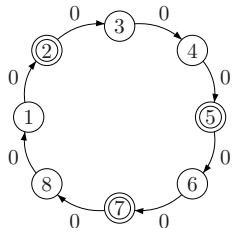

FIGURE 4. Unary cyclic automaton $\mathcal{A}_w$ for $w = abaababa$.

when the alphabet has only one letter (*cf.* [12]). So, in the next section, we study the time complexity of the algorithm in the binary case.

2. Word trees and tree-like automata

In this section we present a family of binary automata for which we proved (*cf.* [7]) that the sequence of refinements of the set of states is unique. Differently from the unary case, although for these automata the refinement process is unique, there may be different executions that produce the same sequence of partitions of the states. Obviously, such executions may have different running time. In the main result of this section we compute the running time of the algorithm on these automata both in the best and in the worst execution.

The definition of these automata is based on the notion of binary tree below described.

Let $\Sigma = \{0, 1\}$ and $A = \{a, b\}$ be two binary alphabets. A *binary labeled tree* over A is a map $\tau : \Sigma^* \to A$ whose domain $dom(\tau)$ is a prefix-closed subset of Σ^*. The elements of $dom(\tau)$ are called *nodes*, if $dom(\tau)$ has a finite (resp. infinite) number of elements we say that τ is *finite* (resp. *infinite*). The *height* of a finite tree τ, denoted by $h(\tau)$, is defined as $\max\{|u| + 1, u \in dom(\tau)\}$. We say that a tree $\bar{\tau}$ is a *prefix* of a tree τ if $dom(\bar{\tau}) \subseteq dom(\tau)$ and $\bar{\tau}$ is the restriction of τ to $dom(\bar{\tau})$. A *complete infinite tree* is a tree whose domain is Σ^*. Besides, a *complete finite tree* of height n is a tree whose domain is Σ^{n-1}. The *empty tree* is the tree whose domain is the empty set.

If $x, y \in dom(\tau)$ are nodes of τ such that $x = yi$ for some $i \in \Sigma$, we say that y is the *father* of x and in particular, if $i = 0$ (resp. $i = 1$) x is the *left son* (resp. *right son*) of y. A node without sons is called *leaf* and the node ϵ is called *the root* of the tree. Given a tree τ, the *outer frontier* of τ is the set $Fr(\tau) = \{xi | x \in dom(\tau), i \in \Sigma, xi \notin dom(\tau)\}$.

Example 2.1. In Figure 5 an example of an infinite tree τ is depicted. We have, for instance, that $0111, 1011 \in dom(\tau)$ and $0110, 1001, 1000 \in Fr(\tau)$. The nodes belonging to the outer frontier are represented by a box.

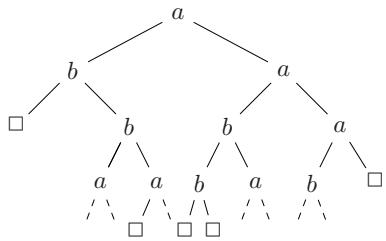

FIGURE 5. Binary infinite labeled tree.

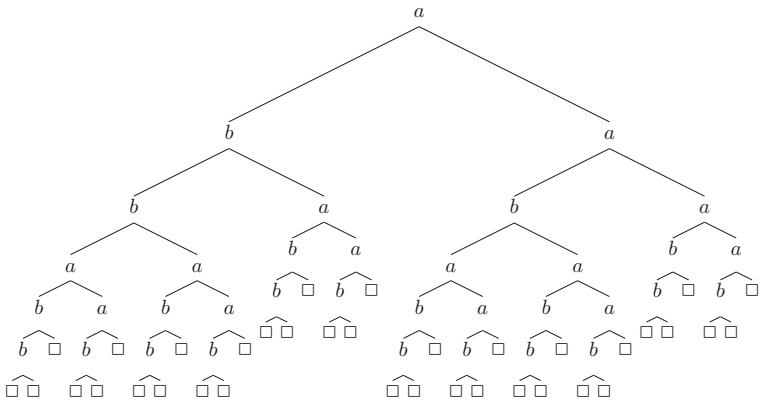

FIGURE 6. The tree $\tau \circ \tau$.

Let τ and τ' be two binary labeled trees. We have that τ is a *subtree* of τ' if there exists a node $v \in dom(\tau')$ such that:

(i) $v \cdot dom(\tau) = \{vu | u \in dom(\tau)\} \subseteq dom(\tau')$
(ii) $\tau(u) = \tau'(vu)$ for all $u \in dom(\tau)$.

In this case we say that τ is a subtree of τ' that *occurs at* node v.

We use the operation of *simultaneous concatenation* defined in [6]. The tree $\tau_1 \circ \tau_2$ is the *simultaneous concatenation* of τ_2 to all the nodes of $Fr(\tau_1)$, *i.e.* the root of τ_2 is attached to all the nodes of the outer frontier of τ_1. More formally, it is defined as follows:

(i) $dom(\tau_1 \circ \tau_2) = dom(\tau_1) \cup Fr(\tau_1)dom(\tau_2)$;
(ii) $\forall x \in dom(\tau_1 \circ \tau_2),\ \tau_1 \circ \tau_2(x) = \begin{cases} \tau_1(x) & \text{if } x \in dom(\tau_1) \\ \tau_2(y) & \text{if } x = zy,\ z \in Fr(\tau_1),\ y \in dom(\tau_2). \end{cases}$

Let τ be a tree, with τ^ω the infinite simultaneous concatenation $\tau \circ \tau \circ \tau \circ \ldots$ is denoted. Notice that, by infinitely applying the simultaneous concatenation, a complete infinite tree is obtained. In Figure 6, $\tau \circ \tau$ is depicted, where τ is defined in Figure 7a.

In what follows we recall some notions and definitions about trees given in [6]. We define *factor* of a tree a finite complete subtree of the tree.

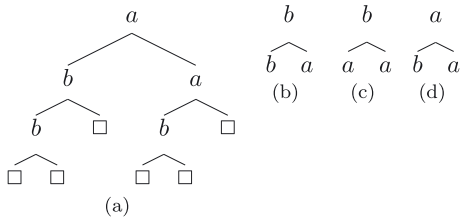

FIGURE 7. A finite tree τ and its circular factors of height 2.

Let τ be a tree, and let σ and $\bar{\sigma}$ be two factors of τ such that $\bar{\sigma}$ is the complete prefix of σ of height $h(\sigma)-1$, then σ is called an *extension of* $\bar{\sigma}$ in τ. A factor σ of a tree τ is *extendable* in τ if there exists at least one extension of σ in τ.

A factor σ of τ is 2-*special* if there exist exactly two different extensions of σ in τ.

We say that γ is a *circular factor* of τ if it is a factor of τ^ω with $h(\gamma) \leq h(\tau)$. A circular factor γ of τ is a *special circular factor* if there exist at least two different extensions of γ in τ^ω (that we can call *circular extensions* or simply extensions). A special factor is called 2-*special circular factor* if it has exactly two different extensions.

Example 2.2. In Figure 7a, a tree τ and three of its circular factors are depicted. The single node labeled by b is a 2-special circular factor indeed it has two different extensions depicted in Figures 7b and 7c. The single node labeled by a has a unique extension depicted in Figure 7d.

The concept of circular factor can be easily understood by noting that in the case of unary tree it coincides with the well-known notion of circular factor of a word.

With reference to a characterization of the notion of circular standard word given in [4], we say that a finite tree τ is a *standard tree* if for each $0 \leq h \leq h(\tau)-2$ it has only a 2-special circular factor of height h.

In this section we analyze some particular binary trees that are close related to the combinatorial notion of word. Such trees were investigated also in [6,7] and here we recall some definitions and results that are preliminary for the goal of this section.

Given two words $v = v_1 v_2 \dots v_{n-1} \in \Sigma^*$ and $w = w_1 w_2 \dots w_n \in A^*$, by $\tau_{v,w}$ we denote the labeled tree $\tau_{v,w}$ such that $dom(\tau_{v,w})$ is the set of prefixes of v and the map is defined as follows:

$$\begin{cases} \tau_{v,w}(\epsilon) = w_1 \\ \tau_{v,w}(v_1 v_2 \dots v_i) = w_{i+1} \quad \forall 1 \leq i \leq n-1. \end{cases}$$

We call *word tree* the finite labeled tree $\tau_{v,w}$. When v is obtained by taking the prefix of length $n-1$ of w and by substituting a's with 0's and b's with 1's, we use the simpler notation τ_w. In Figure 8, a word tree τ_w with $w = abaababa$ is depicted.

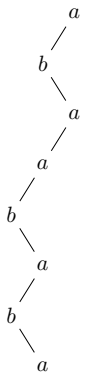

FIGURE 8. The word tree τ_w with $w = abaababa$.

In this paper we focus on word trees τ_w such that w is a standard word.

We recall the well known notion of standard word. Let $d_1, d_2, \ldots, d_n, \ldots$ be a sequence of natural integers, with $d_1 \geq 0$ and $d_i > 0$, for $i = 2, \ldots, n, \ldots$. Consider the following sequence of words $\{s_n\}_{n\geq 0}$ over the alphabet A: $s_0 = b$, $s_1 = a$, $s_{n+1} = s_n^{d_n} s_{n-1}$ for $n \geq 1$. Each finite word s_n in the sequence is called *standard word*. It is uniquely determined by the (finite) directive sequence $(d_0, d_1, \ldots, d_{n-1})$. In the special case where the directive sequence is of the form $(1, 1, \ldots, 1, \ldots)$ we obtain the sequence of Fibonacci words.

In [7] we proved the following result.

Proposition 2.3. *w is a standard word if and only if the word tree τ_w is a standard tree.*

Remark 2.4. Note that previous proposition is a consequence of a result given in [7] stating that there exists a one-to-one correspondence between the set of circular factors of a word w and the set of circular factors of τ_w, in the sense that the occurrences of a factor of τ_w univocally individuate the occurrences of the corresponding factors in w.

Given a word tree $\tau_{v,w}$ we can uniquely associate an automaton $\mathcal{A}_{\tau_{v,w}}$ called *tree-like automaton* having $\tau_{v,w}$ as skeleton and such that for each missing edge in the tree we add a transition to the root of the tree. Moreover, the root is the initial state and the states corresponding to nodes labeled by a (resp. b) are not final (resp. final) states. When v is obtained by taking the prefix of length $n-1$ of w and by substituting a's with 0's and b's with 1's, we use the simpler notation τ_w for the word tree and $\mathcal{A}_{\tau_w}$ for the corresponding automaton.

Example 2.5. In Figure 9 the automaton associated to the word tree of Figure 8 is depicted. The automaton in Figure 2 is the tree-like automaton associated to the word tree τ_w with $w = aabbb$.

We define *standard tree-like automaton* a tree-like automaton $\mathcal{A}_\tau$ associated to a standard word tree τ.

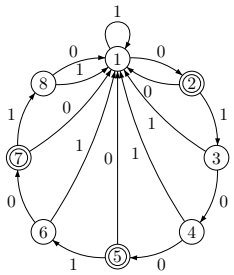

FIGURE 9. The tree-like automaton $\mathcal{A}_{\tau_w}$ associated to the word tree τ_w with $w = abaababa$.

We proved in [6,7] that for these automata the refinement process of Hopcroft's algorithm is uniquely determined whatever execution is realized. In particular we gave the exact description of the partitions of states during each execution of the algorithm. Actually, there exists a close relation between the split operation during the execution of Hopcroft's algorithm on a standard tree-like automaton and the existence of a circular 2-special circular factor of the standard word-tree associated. In few words, since there is a unique 2-special circular factor for each height in the word tree then at most a unique split of a class occurs at each iteration. We report such a result in Proposition 2.6.

Note that, in this context we do not focus our attention on the language recognized by the tree-like automaton because it is not relevant for our purpose. The relative positions of the final states and the transitions in such automata are fundamental to prove our statements, instead.

Let $\mathcal{A}_\tau = (Q, \Sigma, \delta, q_0, F)$ be a tree-like automaton. For any circular factor σ of τ, we define the subset Q_σ of states of $\mathcal{A}_\tau$ that are occurrences of σ in τ. Trivially, we have that $Q_\epsilon = Q$, $Q_b = F$ and $Q_a = Q \setminus F$.

Proposition 2.6. *Let $\mathcal{A}_\tau$ be a standard tree-like automaton. Let Q_σ and Q_γ be classes of a partition of Q. If (Q_γ, x) splits Q_σ, for some $x \in \Sigma$, with $h(\gamma) = h(\sigma)$, then σ is a 2-special circular factor of τ. The resulting classes are $Q_{\sigma'}$ and $Q_{\sigma''}$, where σ' and σ'' are the only two possible extensions of σ in τ. Viceversa, if σ is a 2-special circular factor of τ and Q_σ is a class of a partition of Q then Q_σ is split in $Q_{\sigma'}$ and $Q_{\sigma''}$ where σ' and σ'' are the unique two extensions of σ in τ.*

Each time we extract a pair from the waiting set it can either cause some splits or not. Hence, at each iteration of the main loop of the algorithm the current partition can be either equal to or different from that one of the previous iteration. Then, we call refinement process the sequence $\Pi_1, \Pi_2, \ldots \Pi_m$ of the different partitions produced during a possible execution of the algorithm, where $\Pi_1 = \{F, Q \setminus F\}$ and Π_m is the partition corresponding to Nerode equivalence. We recall the following theorem (*cf.* [6,7]) stating that in case of standard tree-like automata, the refinement process of Hopcroft's algorithm is unique whatever strategy is used for choosing and deleting any pair from the waiting set. Actually such a result has

been proved for a more general class of binary automata associated to standard trees.

Theorem 2.7. *Let $\mathcal{A}_\tau$ be a standard tree-like automaton. The refinement process $\Pi_1, \Pi_2, \dots \Pi_m$ is uniquely determined. Furthermore, $m = h(\tau) - 1$ and for each $1 \leq k \leq h(\tau) - 1$,*

$$\Pi_k = \{Q_\sigma |\ \sigma \text{ is a circular factor of } \tau \text{ with } h(\sigma) = k\}.$$

Note that from this results it follows that each standard tree-like automaton is minimal.

As mentioned before, the uniqueness of the refinement process does not necessarily imply the uniqueness of the execution.

The aim of this section is to compute the running time of Hopcroft's algorithm on standard tree-like automata in the best and worst execution. We report the following lemma proved in [7] that we use in such a computation.

Lemma 2.8. *Let τ_w be a standard word tree and let $\mathcal{A}_{\tau_w}$ be the automaton associated. Let σ and γ be two circular factors of τ_w having the same height. If $(Q_\gamma, 0)$ (resp. $(Q_\gamma, 1)$) splits Q_σ then $(Q_\gamma, 1)$ (resp. $(Q_\gamma, 0)$) either does not split Q_σ.*

We know that for each automaton $\mathcal{A}$ with n states several executions of the algorithm can exist. We denote by $t(n)$ the running time of the current execution of Hopcroft's algorithm to minimize $\mathcal{A}$.

In the following theorem we express the running time of the best and the worst execution of Hopcroft's algorithm on a standard tree-like automata in terms of the occurrences of 2-special circular factors of the standard word tree. With $sp(\tau_w)$ we denote the set of 2-special circular factors of τ_w.

Theorem 2.9. *Let w be a standard word of length n and let A_{τ_w} be the standard automaton associated to the standard tree τ_w. Each execution of Hopcroft's algorithm on this automaton has a running time satisfying the following inequalities:*

$$\sum_{\sigma \in sp(\tau_w)} \min(|Q_{\sigma'}|, |Q_{\sigma''}|) + n - 1 \leq t(n) \leq 2 \sum_{\sigma \in sp(\tau_w)} \min(|Q_{\sigma'}|, |Q_{\sigma''}|).$$

Proof. The second inequality is proved in [6,7]. We prove the first one by computing the running time of the best execution of the algorithm on the automaton.

One can verify that since τ_w is a standard word tree if Q_γ splits Q_σ in $Q_{\sigma'}$ and $Q_{\sigma''}$ then $|Q_{\sigma'}| = |Q_\gamma|$ and $|Q_{\sigma''}| = |Q_\sigma| - |Q_\gamma|$.

Let $\Pi_1, \Pi_2, \dots, \Pi_{n-1}$ be the unique sequence of refinements of the set of the states. There exists a unique set of classes $P = \{Q_{\sigma_1}, Q_{\sigma_2}, \dots, Q_{\sigma_{n-1}}\}$ such that for each $1 \leq i \leq n-1$ one has that Q_{σ_i} is never added to the waiting set and $Q_{\sigma_i} \supset Q_{\sigma_{i+1}}$.

As soon as a class $Q_{\sigma_i} \in P$ is split, the minimal class between $Q_{\sigma_i'}$ and $Q_{\sigma_i''}$ is added to the waiting set paired both with 0 and 1, the other one is a class of P too. By iterating this process, starting from $Q_{\sigma_1} = Q$ up to $Q_{\sigma_{n-1}}$ (that is a singleton

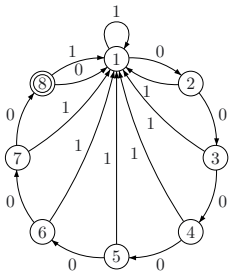

FIGURE 10. The tree-like automaton $\mathcal{A}_{\tau_w}$ associated to the word tree τ_w with $w = aaaaaaab$.

by the minimality of the automaton) we have that the sum of the cardinality of all these minimal classes resulting by these kind of splits is equal to $n-1$.

All the classes not belonging to P and included in some partition of the set of states will be added to the waiting set after some splits. The best execution of the algorithm is obtained by extracting at each iteration the unique (see Prop. 2.6 and Lem. 2.8) splitter pair. By following such a strategy, if (Q_σ, x) is in the waiting set and it is not a splitter then it will be never extracted but, when Q_σ is split, (Q_σ, x) will be replaced by $(Q_{\sigma'}, x)$ and $(Q_{\sigma''}, x)$, that are not splitter too, and the pair $(\min(Q_{\sigma'}, Q_{\sigma''}), y)$, with $y \neq x$, will be added to the waiting set. On the contrary, the pair added after the split of a class in P contributes one more time to the size of the waiting set. At each step of the refinement process during this execution, if σ is a 2-special circular factor either (Q_σ, x) is in the waiting set, for some x, or Q_σ belongs to P. In any case, the pair $(\min(Q_{\sigma'}, Q_{\sigma''}), y)$ is added with $y \neq x$, in particular, if $Q_\sigma \in P$, $(\min(Q_{\sigma'}, Q_{\sigma''}), x)$ is added too. The new class added it will be the new splitter with respect to either 1 or 0. Then, the running time is

$$t(n) \geq \sum_{\sigma \in sp(\tau_w)} \min(|Q_{\sigma'}|, |Q_{\sigma''}|) + n - 1.$$

□

The following two examples describe two family of standard tree-like automata for which the running time is $\Theta(n)$ and $\Theta(n \log n)$, respectively.

Example 2.10. Let us consider the standard word $w = a^{n-1}b$. The associated tree-like automaton is depicted in Figure 10. In this case $|F| = 1$ and $|Q \backslash F| = n-1$. Then, after each split, a singleton paired with both 0 and 1 will be added to the waiting set. One can easily verify that the execution is unique and its running time is $t(n) = 2(n-1)$.

Example 2.11. Let f_n be the n-th finite Fibonacci word and let $A_{\tau_{f_n}}$ be a standard automaton associated to the standard tree τ_{f_n}, with $n \geq 0$. We denote by F_n the number of the states of $A_{\tau_{f_n}}$, *i.e.* $F_n = |f_n|$. As stated in a theorem proved in [7], the family of standard tree-like automata associated to Fibonacci words

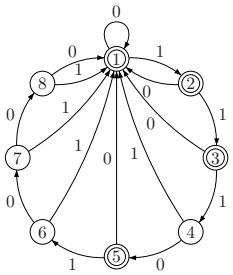

FIGURE 11. The tree-like automaton $\mathcal{A}_{\tau_w}$ associated to the word tree τ_w with $w = bbbabaaa$.

represents a worst case for Hoprcroft's algorithm. Each execution of Hopcroft's algorithm on this automaton has a running time that satisfies the following inequalities:

$$\frac{K}{\phi} F_n \log F_n + F_n - 1 \leq t(F_n) \leq 2KF_n \log F_n,$$

where $K = \frac{3}{5log\phi}$.

Note that, as we can deduce by the proof of the Theorem 2.9, the best and the worst executions do not always correspond with the implementation by stack and queue of the waiting set, respectively. On the other hand, in case of the tree-like automata associated to standard word trees, the theorem states also that the order of the running time of any execution of the algorithm on such automata does not depend by the strategy to implement the waiting set. Instead, there exist binary automata associated to word trees for which the choice of the implementation significantly influences the order of the running time. The following two theorems confirm such a statement by exhibiting two executions of Hopcroft's algorithm on tree-like automata associated to de Brujin words. The first one is performed by implementing the waiting set as a stack and the second one by using a queue. In Figure 11 the tree-like automaton associated to de Brujin word *bbbabaaa* of order 3 is depicted.

Theorem 2.12. *Let w be any de Brujin word of order n and let A_{τ_w} be the associated tree-like automaton. Then, there exists an execution that use a stack implementation of the waiting set such that $t(N) = \Theta(N)$, where $N = 2^n$ is the number of states of A_{τ_w}.*

Proof. Let w be any de Brujin word of order n and let τ_w be the associated word tree. We show an execution of Hopcroft's algorithm on the automaton A_{τ_w} by implementing the waiting set as a stack. Since each class that will be pushed in the waiting set will be paired with both 0 and 1, we choose to insert the pair $(C, 0)$ and then $(C, 1)$. In this way, $(C, 1)$ will be processed always before than $(C, 0)$.

Note that, when a class B of the partition is split into B' and B'', and the pair (B, x) belongs to the waiting set for some x, we substitute (B, x) with (B') and we push $(B', 0)$ and $(B'', 1)$ where $|B'| > |B''|$. The case $|B'| = |B''|$ will be discussed

in the following. By using the one-to-one correspondence between the set of circular factors of τ_w and the set of circular factors of w described in Remark 2.4, if u is a circular factor of w, we denote by Q_u the set of states of A_{τ_w} that are occurrences of the corresponding circular factor of τ_w. With this notation Q_b and Q_a are the set of final and not final states, respectively. Note that $|Q_a| = |Q_b| = 2^{n-1} = N/2$, where N is the number of the states of A_{τ_w}. We choose to push in the waiting set W the pair $(Q_b, 0)$ and then $(Q_b, 1)$. Then the pair $(Q_b, 1)$ is extracted. It splits Q_b into Q_{ba} and Q_{bb} with $|Q_{ba}| = |Q_{bb}| = 2^{n-2}$. We replace $(Q_b, 0)$ with $(Q_{ba}, 0)$ and we push $(Q_{bb}, 0)$ and $(Q_{bb}, 1)$. More in general, when Q_u is split and halved into Q_{ua} and Q_{ub} and $(Q_u, 0)$ belongs to the waiting set, we choose to substitute $(Q_u, 0)$ with $(Q_{ua}, 0)$ and add $(Q_{ub}, 0)$ and $(Q_{ub}, 1)$ to the stack. Then, after the first split the waiting set is $W = \{(Q_{ba}, 0), (Q_{bb}, 0), (Q_{bb}, 1)\}$ and the partial running time is $t(n) = 2^{n-1}$. Since w is any de Brujin word of order n, $(Q_{bb}, 1)$ splits Q_{bb} into Q_{bba} and Q_{bbb} with $|Q_{bba}| = |Q_{bbb}| = 2^{n-3}$. As before we substitute the pair $(Q_{bb}, 0)$ with $(Q_{bba}, 0)$ and we push $(Q_{bbb}, 0)$ and $(Q_{bbb}, 1)$. By iterating such a process, at i-th iteration the pair $(Q_{b^i}, 1)$ is extracted and the class Q_{b^i} is split and halved into $Q_{b^i a}$ and $Q_{b^{i+1}}$ with $1 \leq i \leq n-1$. Note that Q_{b^n} is a singleton and since w is any de Brujin word $(Q_{b^n}, 1)$ does not produce any split. Then the running time up to now is $t(n) = 2^{n-1} + 2^{n-2} + \cdots + 2^{n-(n-1)} + 2^{n-n} = N - 1$ and $W = \{(Q_{ba}, 0), (Q_{bba}, 0), \ldots, (Q_{b^{n-1}a}, 0), (Q_{b^n}, 0)\}$. Note that Q_{b^n} is a singleton and $(Q_{b^n}, 0)$ splits Q_a into a singleton $Q_{ab^{n-1}}$ and its complement. Then $t(n) = N - 1 + 1 = N$ and $W = \{(Q_{ba}, 0), (Q_{bba}, 0), \ldots, (Q_{b^{n-1}a}, 0), (Q_{ab^{n-1}}, 0), (Q_{ab^{n-1}}, 1)\}$. The sum of the cardinality of the sets in W except the last two is $|W| = 2^{n-2} + 2^{n-3} + \cdots + 1 = N/2 - 1$. Note that from this step on, each class of the partition will be split into two subsets one of which is a singleton. If a subset of Q_a is split by a pair $(\{p\}, 0)$, a singleton paired with both 0 and 1 will be pushed in the waiting set. This involves to add two pair of the form $(\{s\}, 0)$ and $(\{s\}, 1)$ after each split, as for $(Q_{ab^{n-1}}, 0)$ and $(Q_{ab^{n-1}}, 1)$, so the running time increases by $2(N/2 - 1)$. Instead, if a subset of Q_b is split by a pair $(\{p\}, 1)$, a class of the form $Q_{b^i a}$ will be split into a singleton $\{s\}$ and its complement $\{s\}^c$. So, the pair $(Q_{b^i a}, 0)$ will be substituted by $(\{s\}^c, 0)$ in loco and $(\{s\}, 0)$ and $(\{s\}, 0)$ will be pushed into the stack. In this way, each split of a subset of Q_b leads to add 1 to the running time until all singletons are obtained. Then the running time is increased by $2^{n-2} - 1 + 2^{n-3} - 1 + \cdots + 2 - 1 = 2^{n-1} - n = N/2 - \log N$, then $t(n) = N + N/2 - 1 + 2(N/2 - 1) + N/2 - \log N = 3N + 1 - \log N = \Theta(N)$. □

The strategy used in the proof of the previous theorem can be easily adapted to the unary case, leading to a linear execution that uses a stack implementation of Hopcroft's algorithm applied to the unary cyclic automata associated to de Brujin words. See for instance Figure 3. Indeed, the sequence of splits and refinements is the same. The substantial difference is that each class C that is added to the stack is paired only with a letter of the alphabet, *i.e.* only the pair (C, a) is inserted instead of the two pairs (C, a) and (C, b). It is easy to see that whereas in binary case the pairs (C, a) and (C, b) one class for each, in the unary case, the pair (C, a) splits both the classes.

Theorem 2.13. *Let w be any de Brujin word of order n and let A_{τ_w} be the associated tree-like automaton. Then, there exists an execution that uses a queue implementation of the waiting set such that*

$$t(N) = N \log N,$$

where $N = 2^n$ is the number of states of A_{τ_w}.

Proof. To prove the statement we adapt to the binary case the strategy used in [1] to prove that the unary cyclic automata associated to the de Brujin words of order n are a worst case of the algorithm with running time $\frac{N}{2} \log N$. In [14] it is proved that such a strategy can be realized by implementing the waiting set as a queue and it represents the absolute worst case of the algorithm. We consider a sequence $(\mathcal{P}_k, \mathcal{S}_k)$, with $k = 1 \ldots, n$, where $\mathcal{P}_k$ and $\mathcal{S}_k$ are the partition and the waiting set given by

$$\mathcal{P}_k = \{Q_u | u \in A^k\} \text{ and } \mathcal{S}_k = \{(Q_v, x) | v \in A^{k-1}b, x \in \Sigma\}.$$

In particular, $\mathcal{P}_1 = \{Q_a, Q_b\}$ and $\mathcal{S}_1 = \{(Q_b, 0), (Q_b, 1)\}$. For each class C that will be enqueued to the waiting set we choose to enqueue $(C, 0)$ and then $(C, 1)$ so $(C, 0)$ will be processed before than $(C, 1)$. Note that, differently from the unary case, each pair splits exactly one class of the partition. The configuration $(\mathcal{P}_{k+1}, \mathcal{S}_{k+1})$ is obtained from $(\mathcal{P}_k, \mathcal{S}_k)$ after 2^k iterations of the main loop of the algorithm. During each iteration a pair (Q_{ub}, x) is removed from the queue $\mathcal{S}_k$ and splits one class that either does not belong yet to $\mathcal{S}_k$ or it is no longer in $\mathcal{S}_k$. Note that, the pair $(Q_{ub}, 0)$ does not split Q_{ub} and $(Q_{ub}, 1)$ could split Q_{ub} but, when this split occurs, neither $(Q_{ub}, 0)$ nor $(Q_{ub}, 1)$ are in the waiting set anymore. To compute the running time of such an execution we sum the sizes of all sets in the waiting set. If $(Q_v, x) \in \mathcal{S}_k$ then $|Q_v| = 2^{n-k}$ and $v = ub$ is a factor of w of length k. Hence we have that $t(n) = 2 \sum_{k=1}^{n} 2^{k-1} 2^{n-k} = n2^n$. □

References

[1] J. Berstel and O. Carton, On the complexity of Hopcroft's state minimization algorithm, in *CIAA. Lecture Notes in Computer Science* **3317** (2004) 35–44.

[2] J. Berstel, L. Boasson and O. Carton, Continuant polynomials and worst-case behavior of Hopcrofts minimization algorithm. *Theoret. Comput. Sci.* **410** (2009) 2811–2822.

[3] J. Berstel, L. Boasson, O. Carton and I. Fagnot, Sturmian trees. *Theor. Comput. Syst.* **46** (2010) 443–478.

[4] J.P. Borel and C. Reutenauer, On Christoffel classes. *RAIRO-Theor. Inf. Appl.* **450** (2006) 15–28.

[5] G. Castiglione, A. Restivo and M. Sciortino, Hopcroft's algorithm and cyclic automata, in *LATA. Lecture Notes in Computer Science* **5196** (2008) 172–183.

[6] G. Castiglione, A. Restivo and M. Sciortino, On extremal cases of hopcroft's algorithm, in *CIAA. Lecture Notes in Computer Science* **5642** (2009) 14–23.

[7] G. Castiglione, A. Restivo and M. Sciortino, On extremal cases of hopcroft's algorithm. *Theoret. Comput. Sci.* **411** (2010) 3414–3422 .

[8] J.E. Hopcroft, An $n \log n$ algorithm for mimimizing the states in a finite automaton, in *Theory of machines and computations (Proc. Internat. Sympos. Technion, Haifa, 1971)*. Academic Press, New York (1971), 189–196.
[9] T. Knuutila, Re-describing an algorithm by Hopcroft. *Theoret. Comput. Sci.* **250** (2001) 333–363.
[10] M. Lothaire, *Algebraic Combinatorics on Words*, *Encyclopedia of Mathematics and its Applications* **90**. Cambridge University Press (2002).
[11] E.F. Moore, Gedaken experiments on sequential, in *Automata Studies*. Annals of Mathematical Studies **34** (1956) 129–153.
[12] R. Paige, R.E. Tarjan and R. Bonic, A linear time solution to the single function coarsest partition problem. *Theoret. Comput. Sci.* **40** (1985) 67–84 .
[13] A. Paun, M. Paun and A. Rodríguez-Patón, Hopcroft's minimization technique: Queues or stacks? in *CIAA*. *Lecture Notes in Computer Science* **5148** (2008) 78–91.
[14] A. Paun, M. Paun and A. Rodríguez-Patón, On the hopcroft's minimization technique for dfa and dfca. *Theoret. Comput. Sci.* **410** (2009) 2424–2430.
[15] B. Watson, *A taxonomy of finite automata minimization algorithms.* Technical Report 93/44, Eindhoven University of Technology, Faculty of Mathematics and Computing Science (1994).

Communicated by A. Cherubini.
Received January 22, 2010. Accepted November 18, 2010.