# On the Complexity of Hopcroft's State Minimization Algorithm

Jean Berstel[1] and Olivier Carton[2]

[1] Institut Gaspard Monge,
Université de Marne-la-Vallée
http://www-igm.univ-mlv.fr/~berstel
[2] LIAFA, Université Paris 7
http://www.liafa.jussieu.fr/~carton

**Abstract.** Hopcroft's algorithm for minimizing a deterministic automaton has complexity $O(n \log n)$. We show that this complexity bound is tight. More precisely, we provide a family of automata of size $n = 2^k$ on which the algorithm runs in time $k2^k$. These automata have a very simple structure and are built over a one-letter alphabet. Their sets of final states are defined by de Bruijn words.

## 1 Introduction

Efficient state minimization algorithms are an important issue for tools involving finite state automata, as they arise e.g. in computational linguistics. The elementary minimization algorithm usually credited to Moore (see also [1]) has been improved by Hopcroft [2]. In the special case of finite sets, minimal automata can be constructed and maintained even more efficiently (see [3, 4] and [5] for a recent survey). Extensions to more general situations of Hopcroft's algorithm are considered in [6, 7, 8].

Hopcroft's algorithm is known to run in time $O(n \log n)$ for an automaton with $n$ states. We show here that this bound is tight, that is that this running time is reached for an infinite family of automata. For that purpose we define a class of automata over a unary alphabet. These automata have a very simple structure since they are just made of a single cycle. The final states of these automata are defined by a pattern given by de Bruijn words. The simple structure of the automaton and the special layout of the final states allows us to control precisely how some particular execution of the algorithm runs.

We should point out that Hopcroft's algorithm has a degree of freedom because, in each step of its main loop, it allows a free choice of a set of states to be processed. Hopcroft has proved that any sequence of choices can be processed in time $O(n \log n)$. Our family of examples results in showing that there exists some "unlucky" sequence of choices that slows down the computation to achieve the lower bound $\Omega(n \log n)$. Partial results on another family of examples have been obtained in [9].

M. Domaratzki et al. (Eds.): CIAA 2004, LNCS 3317, pp. 35–44, 2005.

The paper is organized as follows. After some general definitions we outline Hopcroft's algorithm. We next present de Bruijn words, and then introduce our family of automata. These are simply one letter automata with $n = 2^k$ states organized as a cycle. The key property is the choice of final states. Exactly one half of the states are final, and they are chosen according to the occurrence of the symbol 1 in a de Bruijn word of order $k$.

Given such a cyclic automaton, we next present the strategy used to choose the sets in Hopcroft's algorithm. We then prove that this choice indeed leads to a running time in $O(n \log n)$. It should be observed that minization of one-letter automata can be performed in linear time by another algorithm [7].

## 2 Minimal Automaton

In this section, we fix some notation and we give some basic definitions.

We only use deterministic and complete automata. An *automaton* $\mathcal{A}$ over a finite alphabet $A$ is composed of a finite state set $Q$, a distinguished state called the initial state, a set $F \subseteq Q$ of final states, and of a next-state function $Q \times A \to Q$ that maps $(q, a)$ to a state denoted by $q \cdot a$.

A *partition* of a set $Q$ is a family $\{Q_1, \ldots, Q_n\}$ of nonempty subsets of $Q$ that are pairwise disjoint (that is $Q_i \cap Q_j = \emptyset$ for $i \neq j$) and cover $Q$, (that is $Q = Q_1 \cup \cdots \cup Q_n$). The subsets $Q_i$ are called the *classes* of the partition.

If $Q$ is the state set of an automaton $\mathcal{A}$, a *congruence* of $\mathcal{A}$ is a partition which is compatible with the transitions of $\mathcal{A}$. This means that if $q$ and $q'$ are in the same class, then $q \cdot a$ and $q' \cdot a$ are also in the same class for any $q, q' \in Q$ and any $a \in A$.

A partition of $Q$ *saturates* a subset $F$ of $Q$ if $F$ is the union of some of its classes. This also means that in a class either all elements or none belong to $F$. A partition $\{Q_1, \ldots, Q_n\}$ is *coarser* than a partition $\{Q'_1, \ldots, Q'_m\}$ if the partition $\{Q'_1, \ldots, Q'_m\}$ saturates each class $Q_i$. This relation defines a partial order on partitions.

It is well known that any regular set $L$ of finite words is accepted by a unique minimal deterministic automaton.

It should be noticed that the minimal automaton of $\mathcal{A}$ does not depend on the initial state of $\mathcal{A}$ as long as any state is reachable from it. In what follows, we often omit to specify the initial state since it does not matter.

## 3 Hopcroft's Algorithm

Hopcroft [2] has given an algorithm that computes the minimal automaton of a given deterministic automaton. The running time of the algorithm is $O(|A| \times n \log n)$ where $|A|$ is the cardinality of the alphabet and $n$ is the number of states of the given automaton. The algorithm has been described and re-described several times [2, 10, 11, 12, 13, 14].

The algorithm is outlined below, and it is explained then in some more detail.

It is convenient to use the shorthand $T^c = Q \setminus T$ when $T$ is a subset of the set $Q$ of states. We denote by $\min(B, C)$ the set of smaller size of the two sets $B$ and $C$, and any one of them if they have the same size.

```
 1: P ← {F, F^c}
 2: for all a ∈ A do
 3:    Add((min(F, F^c), a), S)
 4: while S ≠ ∅ do
 5:    (C, a) ← Some(S)                    ▷ takes some element in S
 6:    for each B ∈ P split by (C, a) do
 7:       B', B'' ← Split(B, C, a)
 8:       Replace B by B' and B'' in P
 9:       for all b ∈ A do
10:          if (B, b) ∈ S then
11:             Replace (B, b) by (B', b) and (B'', b) in S
12:          else
13:             Add((min(B', B''), b), S)
```

**Algorithm 1.** HopcroftMinimization

Given a deterministic automaton $\mathcal{A}$, Hopcroft's algorithm computes the coarsest congruence which saturates the set $F$ of final states. It starts from the partition $\{F, F^c\}$ which obviously saturates $F$ and refines it until it gets a congruence. These refinements of the partition are always obtained by splitting some class into two classes.

Before explaining the algorithm in more detail, some notation is needed. For a set $B$ of states, we note by $B \cdot a$ the set $\{q \cdot a \mid q \in B\}$. Let $B$ and $C$ be two sets of states and let $a$ be a letter. We say that the pair $(C, a)$ *splits* the set $B$ if both sets $(B \cdot a) \cap C$ and $(B \cdot a) \cap C^c$ are nonempty. In that case, the set $B$ is split into the two sets $B' = \{q \in B \mid q \cdot a \in C\}$ and $B'' = \{q \in B \mid q \cdot a \notin C\}$ that we call the *resulting sets*. Note that a partition $\{Q_1, \ldots, Q_n\}$ is a congruence if and only if for any $1 \leq i, j \leq n$ and any $a \in A$, the pair $(Q_i, a)$ does not split $Q_j$.

The algorithm proceeds as follows. It maintains a current partition $\mathcal{P} = \{B_1, \ldots, B_n\}$ and a current set $\mathcal{S}$ of pairs $(C, a)$ where $C$ is a class of $\mathcal{P}$ and $a$ is a letter that remain to be processed. The set $\mathcal{S}$ is called the *waiting* set. The algorithm stops when the waiting set $\mathcal{S}$ becomes empty. When it stops, the partition $\mathcal{P}$ is the coarsest congruence that saturates $F$. The starting partition is the partition $\{F, F^c\}$ and the starting set $\mathcal{S}$ contains all pairs $(\min(F, F^c), a)$ for $a \in A$.

The main loop of the algorithm takes one pair $(C, a)$ out of the waiting set $\mathcal{S}$ and performs the following actions. Each class $B$ of the current partition (including the class $C$) is checked whether it is split by the pair $(C, a)$. If $(C, a)$ does not split $B$, then nothing is done. Otherwise, the class $B$ is replaced in the partition $\mathcal{P}$ by the two resulting sets $B'$ and $B''$ of the split. For each letter $b$, if the pair $(B, b)$ is in $\mathcal{S}$, it is replaced in $\mathcal{S}$ by the two pairs $(B', b)$ and $(B'', b)$, otherwise only the pair $(\min(B', B''), b)$ is added to $\mathcal{S}$.

The main ingredient in the analysis of the running time of the algorithm is that the splitting of all classes of the current partition according to a pair $(C, a)$ takes a time proportional to the size of $C$. Therefore, the global running time of the algorithm is proportional to the sum of the sizes of the classes processed in the main loop. Note that a pair which is added to the waiting set $\mathcal{S}$ is not necessarily processed later because it can be split by the processing of another pair before it is considered.

It should be noted that the algorithm is not really deterministic because it has not been specified which pair $(C, a)$ is taken from $\mathcal{S}$ to be processed at each iteration of the main loop. This means that for a given automaton, there are many executions of the algorithm. It turns out that all of them produce the right partition of the states. However, different executions may give rise to different sequences of splitting and also to different running time. Hopcroft has proved that the running time of any execution is bounded by $O(|A| \times n \log n)$.

In this paper, we show that this bound is tight. More precisely, we show that there exist automata over a one-letter alphabet and of size $n$ and there exist executions on these automata that give a running time of magnitude $O(n \log n)$. Actually, we will not give automata for all integers $n$ but those of the form $2^k$.

## 4 De Bruijn Words

The family of automata that we use to show the lower bound on the running time of Hopcroft's algorithm are based of de Bruijn words. We recall their definition.

Let $w = w_1 \dots w_m$ a word of length $m$. By a slight abuse, we use the notation $w_i$ even if the integer $i$ is greater than $m$. We denote by $w_i$ the letter $w_{i'}$ where $i'$ is the unique integer such that $1 \leq i' \leq m$ and $i' = i \mod m$. A *circular occurrence* of a word $u = u_1 \dots u_p$ of length $p$ in $w$ is an integer $k$ in the interval $[1; m]$ such that $w_{k+i-1} = u_i$ for each $i$ in $[1; p]$.

A *de Bruijn word* of order $n$ over the alphabet $B$ is a word $w$ such that each word of length $n$ over $B$ has exactly one circular occurrence in $w$. Since there are $|B|^n$ words of length $n$, the length of a de Bruijn word of order $n$ is $|B|^n$.

Set for instance the alphabet $B = \{0, 1\}$. The word $w = 1100$ is a de Bruijn word of order 2 since each of the words $\{00, 01, 10, 11\}$ has a circular occurrence in $w$. The word $w = 11101000$ is a de Bruijn word of order 3.

De Bruijn words are widely investigated (see for instance [15]). It is well known that for any alphabet, there are de Bruijn words for all orders. We recall here a short proof of this fact. Let $B$ be a fixed alphabet and let $n$ be a fixed integer. We recall the definition of the *de Bruijn graph* $\mathcal{B}_n$ of order $n$. Its vertex set is the set $B^{n-1}$ of all words of length $n-1$. The edges of $\mathcal{B}_n$ are the pairs of the form $(bu, ua)$ for $u \in B^{n-2}$ and $a, b \in B$. This graph is often presented as a labeled graph where each edge $(bu, ua)$ is labeled by the letter $a$. Note that the function which maps each word $w = bua$ of length $n$ to the edge $(bu, ua)$ is one to one. Therefore, a de Bruijn word of order $n$ corresponds to an Eulerian circuit in $\mathcal{B}_n$. Since there are exactly $|B|$ edges entering and leaving each vertex of $\mathcal{B}_n$, the graph $\mathcal{B}_n$ has Eulerian circuits [15] and there are de Bruijn words

of order $n$. In Fig. 1 below we show the de Bruijn graph of order 4. Taking an Eulerian circuit from it, one obtains the de Bruijn word $w = 0000100110101111$ of order 4.

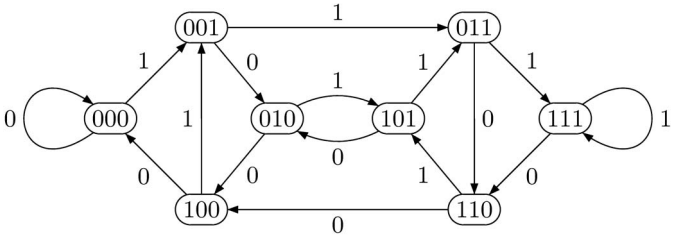


**Fig. 1.** The de Bruijn graph of order 4 over the alphabet $\{0, 1\}$

## 5 Cyclic Automata

In what follows, we only consider de Bruijn words over the binary alphabet $\mathbb{B} = \{0, 1\}$. Let $w$ be a de Bruijn word of order $n$. Recall that the length of $w$ is $2^n$. We define an automaton $\mathcal{A}_w$ over the unary alphabet $\{a\}$ as follows. The state set of $\mathcal{A}_w$ is $\{1, \ldots, 2^n\}$ and the next state function is defined by $i \cdot a = i+1$ for $i < 2^n$ and $2^n \cdot a = 1$. Note that the underlying labeled graph of $\mathcal{A}_w$ is just a cycle of length $2^n$. The final states really depend on $w$. The set of final states of $\mathcal{A}_w$ is $F = \{1 \leq i \leq 2^n \mid w_i = 1\}$.

For a word $u$ over $\mathbb{B}$, we define a subset $Q_u$ of states of $\mathcal{A}_w$. By definition the set $Q_u$ is the set of positions of circular occurrences of $u$ in $w$. If the length of $u$ is $n$, the set $Q_u$ is a singleton since the de Bruijn word $w$ has exactly one circular occurrence of $u$. More generally, if the length of $u$ is less than $n$, the cardinality of $Q_u$ is $2^{n-|u|}$ since there are as many circular occurrences of $u$ as there are words $v$ such that $|uv| = n$. If $u$ is the empty word, then $Q_u$ is by convention the set $Q$ of all states of $\mathcal{A}_w$. By definition, the set $F$ of final states of $\mathcal{A}_w$ is $Q_1$ while its complement $F^c$ is $Q_0$.

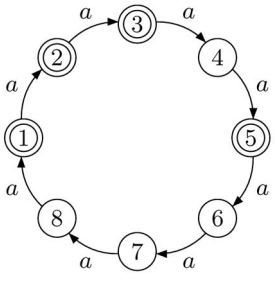


**Fig. 2.** Cyclic automaton $\mathcal{A}_w$ for $w = 11101000$

Let $w$ be the de Bruijn word 11101000. The automaton $\mathcal{A}_w$ is pictured in Fig. 2. The sets $Q_1$, $Q_{01}$ and $Q_{011}$ of states are respectively $\{1, 2, 3, 5\}$, $\{4, 8\}$ and $\{8\}$.

Since any circular occurrence of $u$ in $w$ is followed by either 0 or 1, the equality $Q_u = Q_{u0} \cup Q_{u1}$ holds. If a word $u = bu'$ has a circular occurrence $k$ in $w$, its suffix $u'$ has a circular occurrence $k + 1$ in $w$. If follows that if $u$ is factorized $u = bu'$ where $b \in \mathbb{B}$, then $Q_u \cdot a \subset Q_{u'}$.

## 6 Hopcroft's Algorithm on Cyclic Automata

We claim that the running time of Hopcroft's algorithm on a cyclic automaton $\mathcal{A}_w$ may be of order $n2^n$. Before giving the proof of this claim, we give an example of an execution on the automaton pictured in Fig. 2. Since cyclic automata are over the unary alphabet $A = \{a\}$, we merely say that a class $C$ splits a class $B$ to mean that the pair $(C, a)$ splits the class $B$.

- The starting partition is $\mathcal{P} = \{F, F^c\} = \{Q_0, Q_1\}$ and $\mathcal{S} = \{Q_1\}$.
- The class $Q_1$ is processed.
  - The class $Q_0$ is split into $Q_{00}$ and $Q_{01}$, and $Q_{01}$ is added to $\mathcal{S}$.
  - The class $Q_1$ is split into $Q_{10}$ and $Q_{11}$, and $Q_{11}$ is added to $\mathcal{S}$.

  Then $\mathcal{P} = \{Q_{00}, Q_{01}, Q_{10}, Q_{11}\}$ and $\mathcal{S} = \{Q_{01}, Q_{11}\}$.
- The class $Q_{01}$ is processed.
  - The class $Q_{00}$ is split into $Q_{000}$ and $Q_{001}$, and $Q_{001}$ is added to $\mathcal{S}$.
  - The class $Q_{10}$ is split into $Q_{100}$ and $Q_{101}$, and $Q_{101}$ is added to $\mathcal{S}$.

  Then $\mathcal{P} = \{Q_{000}, Q_{001}, Q_{01}, Q_{100}, Q_{101}, Q_{11}\}$ and $\mathcal{S} = \{Q_{11}, Q_{001}, Q_{101}\}$.
- The class $Q_{11}$ is processed.
  - The class $Q_{01}$ is split into $Q_{010}$ and $Q_{011}$, and $Q_{011}$ is added to $\mathcal{S}$.
  - The class $Q_{11}$ is split into $Q_{110}$ and $Q_{111}$, and $Q_{111}$ is added to $\mathcal{S}$.

  Then $\mathcal{P} = \{Q_{000}, Q_{001}, Q_{010}, Q_{011}, Q_{100}, Q_{101}, Q_{110}, Q_{111}\}$ and $\mathcal{S} = \{Q_{001}, Q_{011}, Q_{101}, Q_{111}\}$.
- Classes $Q_{001}, Q_{011}, Q_{101}, Q_{111}$ are processed but this gives no further splitting since the partition is made of singletons.

Let us point out some properties of this particular execution of the algorithm. The classes that appear during the the execution are all of the form $Q_u$ for some word $u$. Every time a class $Q_u$ is split, it is split into the classes $Q_{u0}$ and $Q_{u1}$. Since these two classes have the same cardinality, the algorithm may either add one or another one to $\mathcal{S}$. In this execution we have always assumed that it chooses $Q_{u1}$.

When the algorithm processes $Q_{01}$, it could have chosen to process $Q_{11}$ instead. The algorithm would have run differently because the class $Q_{01}$ would have been split by $Q_{11}$.

We now describe the worst case strategy which we use to prove that the $O(n \log n)$ bound of Hopcroft's algorithm is tight. Given $n$ and the automaton

$\mathcal{A}_w$, we construct a sequence $(\mathcal{P}_k, \mathcal{S}_k)$ for $k = 1, \ldots, n$ where $\mathcal{P}_k$ and $\mathcal{S}_k$ are the partition and the waiting set given by

$$\mathcal{P}_k = \{Q_u \mid u \in \mathbb{B}^k\} \quad \text{and} \quad \mathcal{S}_k = \{Q_v \mid v \in \mathbb{B}^{k-1}1\}.$$

In particular, $\mathcal{P}_1 = \{Q_0, Q_1\}$ is the starting partition of Hopcroft's algorithm and $\mathcal{S}_1 = \{Q_1\}$ is the starting content of the waiting set. The pair $(\mathcal{P}_{k+1}, \mathcal{S}_{k+1})$ is obtained from the pair $(\mathcal{P}_k, \mathcal{S}_k)$ by obeying to the following strategy: choose the sets $Q_v$ of $\mathcal{S}_k$ in such an order that $Q_v$ does not split any set in the current waiting set $\mathcal{S}$.

More precisely, a linear order $<$ on $\mathcal{S}_k$ is said to be *non-splitting* if whenever $Q_{v'}$ splits $Q_v$ then $Q_v < Q_{v'}$. In other terms the strategy we choose is to process sets in $\mathcal{S}_k$ in some order which avoids splitting. We call such a strategy a *non-splitting strategy*. We will see in Proposition 1 that during this process, each removal of an element of $\mathcal{S}_k$ contributes to two elements in $\mathcal{S}_{k+1}$. It happens, as we will prove, that the new sets are not split by the currently processed set either. We will see in Proposition 2 that non-splitting orders do exist.

The transition from $(\mathcal{P}_k, \mathcal{S}_k)$ to $(\mathcal{P}_{k+1}, \mathcal{S}_{k+1})$ involves $2^{k-1}$ iterations of the main loop of the algorithm. Each iteration removes one set from the waiting set, and as we will show splits exactly two sets in the current partition and adds exactly two sets to the waiting set. These latter sets are of the form $Q_v$ for $v \in \mathbb{B}^k 1$.

**Proposition 1.** *If Hopcroft's algorithm starts from $(\mathcal{P}_k, \mathcal{S}_k)$ and processes the sets in $\mathcal{S}_k$ in a non-splitting order, it yields the pair $(\mathcal{P}_{k+1}, \mathcal{S}_{k+1})$.*

**Proposition 2.** *Each $\mathcal{S}_k$ admits non-splitting orders.*

We start with several lemmas. Some properties of the splitting of the sets of the form $Q_u$ are needed. They are stated in the following lemma.

**Lemma 1.** *Let $u$ and $v$ be two words of length smaller than $n$. The pair $(Q_v, a)$ splits $Q_u$ if and only if there are $b \in \mathbb{B}$ and $s \in \mathbb{B}^+$ such that $us = bv$. If $(Q_v, a)$ splits $Q_u$, the resulting sets are $Q_{us}$ and $Q_u \setminus Q_{us}$. In particular if $|u| > |v|$, then $Q_v$ does not split $Q_u$.*

*Proof.* Assume that $u = bu'$ where $b \in \mathbb{B}$. Then the inclusion $Q_u \cdot a \subset Q_{u'}$ holds. Therefore if $v$ is not equal to $u's$ for some $s \in \mathbb{B}^*$, the intersection $(Q_u \cdot a) \cap Q_v$ is empty and $(Q_v, a)$ does not split $Q_u$. Assume now that $v = u's$ for some $s$. If $s$ is the empty word, the intersection $(Q_u \cdot a) \cap Q_v^c$ is empty and $(Q_v, a)$ does not split $Q_u$. It follows that $s$ is not empty and that $us = bv$. □

**Corollary 1.** *If $u$ and $v$ are two words of the same length, the pair $(Q_v, a)$ splits $Q_u$ if and only if there are $b, b' \in \mathbb{B}$ such that $ub' = bv$. If $(Q_v, a)$ splits $Q_u$, the resulting sets are $Q_{u0}$ and $Q_{u1}$.*

In other terms, if $u$ and $v$ are two words of the same length $k$, then $Q_v$ splits $Q_u$ iff there is an edge $(u, v)$ in the de Bruijn graph $\mathcal{B}_{k+1}$.

We are now ready for the proof of Proposition 1.

*Proof.* (of Proposition 1) We consider how the execution goes according to our non-splitting strategy from the pair $(\mathcal{P}_k, \mathcal{S}_k)$ to the pair $(\mathcal{P}_{k+1}, \mathcal{S}_{k+1})$. We denote by $\mathcal{P}$ and $\mathcal{S}$ the current values of the partition and of the waiting set when we process the classes in $\mathcal{S}_k$ in a fixed non-splitting order. At the beginning of the execution, $\mathcal{P} = \mathcal{P}_k$ and $\mathcal{S} = \mathcal{S}_k$ and at the end $\mathcal{P} = \mathcal{P}_{k+1}$ and $\mathcal{S} = \mathcal{S}_{k+1}$. By Corollary 1, each class $Q_u$ of $\mathcal{P}_k$ is split by exactly one class $Q_v$ in $\mathcal{S}_k$ and each class $Q_v$ splits two classes $Q_u$ and $Q_{u'}$ in $\mathcal{P}_k$. Moreover, $Q_v$ does not split any other class in the current partition. By the choice of the ordering, both classes $Q_u$ and $Q_{u'}$ do not belong to $\mathcal{S}$ when $Q_v$ is processed. The class $Q_u$ is split into the classes $Q_{u0}$ and $Q_{u1}$. Since these two classes have the same cardinality, either $Q_{u0}$ or $Q_{u1}$ may be added to $\mathcal{S}$. Similarly the class $Q_{u'}$ is split into the classes $Q_{u'0}$ and $Q_{u'1}$. The execution of our strategy adds the classes $Q_{u1}$ and $Q_{u'1}$ to the set $\mathcal{S}$. The execution continues until all classes in $\mathcal{S}_k$ have been processed. While this is done, classes $Q_{u1}$ for $u \in \mathbb{B}^k$ are added to $\mathcal{S}$. When all classes $Q_u$ from $\mathcal{S}_k$ have been processed, the partition $\mathcal{P}$ and the set $\mathcal{S}$ are $\mathcal{P}_{k+1}$ and $\mathcal{S}_{k+1}$. □

We now proceed to proof of the existence of non-splitting orders on $\mathcal{S}_k$.

*Proof.* (of Proposition 2) Let $G_k = (V_k, E_k)$ be the graph where the vertex set is $V_k = \mathbb{B}^{k-1}1$ and the set of edges is $E_k = \{(u, v) \mid Q_v \text{ splits } Q_u\}$. By Corollary 1, the graph $G_k$ is actually the subgraph of the de Bruijn $\mathcal{B}_{k+1}$ defined by the set $V_k$ of vertices. The main property of that graph $G_k$ is to be almost acyclic: For each $k \geq 0$, the only cycle in $G_k$ is the edge $(1^k, 1^k)$.

It is easy to see that if there is a path of length $\ell$ from some node to $v$ in $G$, then the word $v$ belongs to $\mathbb{B}^{k-\ell-1}1^{\ell+1}$. It follows from the claim that the vertex $1^k$ is the only vertex which can appear in a cycle.

Since this graph is acyclic, the words of $\mathbb{B}^{k-1}1$ can be topologically ordered. Thus a non-splitting order on $\mathcal{S}_k$ is defined by $Q_u < Q_v$ iff $u < v$ in the previous topological order. □

The graph $G_3$ of the previous proof is pictured in Fig. 3.

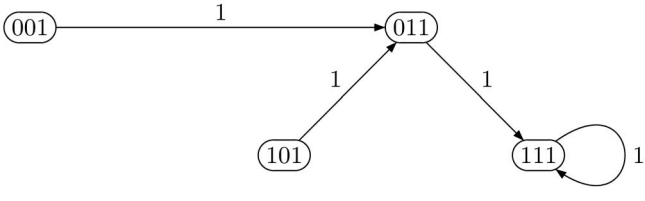


**Fig. 3.** The graph $G_3$

Let us come back to the execution given at the beginning of that section. After $Q_1$ is processed, the partition $\mathcal{P}$ and the set $\mathcal{S}$ are $\mathcal{P} = \{Q_{00}, Q_{01}, Q_{10}, Q_{11}\}$ and $\mathcal{S} = \{Q_{01}, Q_{11}\}$. The class $Q_{11}$ splits the class $Q_{01}$ while the class $Q_{01}$ does not split the class $Q_{11}$. A non-splitting order on $\mathcal{S}_2$ is given by $Q_{01} < Q_{11}$. The class

$Q_{01}$ is therefore processed before the class $Q_{11}$. The partition $\mathcal{P}$ and the set $\mathcal{S}$ become $\mathcal{P} = \{Q_u \mid u \in \mathbb{B}^3\}$ and $\mathcal{S} = \{Q_{u1} \mid u \in \mathbb{B}^2\}$.

We finally analyze the running time of the algorithm. The following result shows that the $O(n \log n)$ upper bound of the running time of Hopcroft's algorithm is tight.

**Theorem 1.** *The non-splitting strategy requires $n2^n$ operations for the minimization of the automaton $\mathcal{A}_w$ of size $2^n$ for any de Bruijn word $w$ of order $n$.*

*Proof.* The time needed to process a class $C$ is proportional to the size of $C$. In the execution that we give the algorithm processes all classes $Q_{u1}$ for $|u| < n$. Summing all the sizes, we get that the running time of the algorithm is $n2^n$ whereas the size of the automaton $\mathcal{A}_w$ is $2^n$. □

## 7 Conclusion

We have shown that Hopcroft's algorithm may have executions running in time $O(n \log n)$. These executions run on the cyclic automata that we have defined. It is not very difficult to see that there are also executions that run in linear time for the same automata. It is still open whether there are automata on which all executions of Hopcroft's algorithm do not run in linear time.

These different executions depend on the choice of the class which is processed at each iteration of the main loop of the algorithm. Defining strategies which specify which class is processed might be of interest from a theoretical and practical point of view.

*Acknowledgment.* We would like to thank Luc Boasson and Isabelle Fagnot for fruitful discussions and the anonymous referees for their helpful comments.

## References

1. Hopcroft, J.E., Ullman, J.D.: Formal Languages and their Relation to Automata. Addison-Wesley (1969)
2. 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, Academic Press (1971) 189–196
3. Krivol, S.L.: Algorithms for minimization of finite acyclic automata and pattern matching in terms. Cybernetics **27** (1991) 324– 331 translated from Kibernetika, No 3, May-June 1991, pp. 11–16.
4. Revuz, D.: Minimisation of acyclic deterministic automata in linear time. Theoret. Comput. Sci. **92** (1992) 181–189
5. Daciuk, J.: Comparison of construction algorithms for minimal, acyclic, deterministic finite-state automata from sets of strings. In Champarnaud, J.M., Maurel, D., eds.: 7th Implementation and Application of Automata (CIAA 2002). Volume 2608 of Lect. Notes in Comput. Sci., Springer Verlag (2002) 255–261
6. Cardon, A., Crochemore, M.: Partitioning a graph in $O(|A| \log_2 |V|)$. Theoret. Comput. Sci. **19** (1982) 85–98

7. Paige, R., Tarjan, R.E., Bonic, R.: A linear time solution for the single function coarsest partition problem. Theoret. Comput. Sci. **40** (1985) 67–84
8. Paige, R., Tarjan, R.E.: Three partition refinement algorithms. SIAM J. Comput. **18** (1987) 973–989
9. Gai, A.T.: Algorithmes de partionnement : minimisation d'automates et applications aux graphes. Mémoire de DEA, Université Montpellier II (2003)
10. Gries, D.: Describing an algorithm by Hopcroft. Acta Inform. **2** (1973) 97–109
11. Aho, A., Hopcroft, J., Ullman, J.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
12. Beauquier, D., Berstel, J., Chrétienne, P.: Éléments d'algorithmique. Masson (1992)
13. Blum, N.: A $O(n \log n)$ implementation of the standard method for minimizing $n$-state finite automata. Inform. Proc. Letters **57** (1996) 65–69
14. Knuutila, T.: Re-describing an algorithm by Hopcroft. Theoret. Comput. Sci. **250** (2001) 333–363
15. Tutte, W.T.: Graph Theory. Volume 21 of Encyclopedia of Mathematics and its Applications. Addison-Wesley (1984)