Minimizing local automata

Marie-Pierre Béal
Institut Gaspard-Monge
University of Marne-la-Vallée, CNRS
77454 Marne-la-Vallée Cedex 2, France
beal@univ-mlv.fr

Maxime Crochemore
Institut Gaspard-Monge
University of Marne-la-Vallée, CNRS
77454 Marne-la-Vallée Cedex 2, France
and King's College London
mac@univ-mlv.fr

***Abstract*— We design an algorithm that minimizes irreducible deterministic local automata by a sequence of state mergings. Two states can be merged if they have exactly the same outputs. The running time of the algorithm is $O(\min(m(n-r+1), m\log n))$, where m is the number of edges, n the number of states of the automaton, and r the number of states of the minimized automaton. In particular, the algorithm is linear when the automaton is already minimal and contrary to Hopcroft's minimisation algorithm that has a $O(kn\log n)$ running time in this case, where k is the size of the alphabet, and that applies only to complete automata. (Note that $kn \geq m$.)**

While Hopcroft's algorithm relies on a "negative strategy", starting from a partition with a single class of all states, and partitioning classes when it is discovered that two states cannot belong to the sam class, our algorithm relies on a "positive strategy", starting from the trivial partition for which each class is a singleton. Two classes are then merged when their leaders have the same outputs.

The algorithm applies to irreducible deterministic local automata, where all states are considered both initial and final. These automata, also called covers, recognize symbolic dynamical shifts of finite type. They serve to present a large class of constrained channels, the class of finite memory systems, used for channel coding purposes. The algorithm also applies to irreducible deterministic automata that are left-closing and have a synchronizing word. These automata present shifts that are called almost of finite type. Almost-of-finite-type shifts make a meaningful class of shifts, intermediate between finite type shifts and sofic shifts.

I. Introduction

Local automata, also called definite automata or definite covers [16], present a large class of systems: the class of finite memory systems used in coding for constrained channels [1]. These channels have a canonical minimal deterministic presentation, called the Fischer cover, which can be computed from an n-state local deterministic presentation using Hopcroft's minimisation algorithm that runs in time $O(kn\log n)$ [9], where k is the alphabet size.

Hopcroft's algorithm computes the Nerode partition of the set of states with a "negative strategy". It starts from a partition with a single class of all states, and partitions classes when it is discovered that two states cannot belong to a same class. It applies to any complete finite-state automaton. For local automata, as well as also for almost of finite type (AFT) automata, the minimal presentation can be obtained with a sequence of state mergings.

In this paper we design an algorithm to compute the minimal automaton of a deterministic local automaton that relies on a "positive strategy". We start from the trivial partition for which each class is a singleton. Two classes are then merged when their leaders have the same outputs. The running time of this algorithm is $O(\min(m(n-r+1), m\log n))$, where m is the number of edges of the automaton. In particular, it is linear when the automaton is already minimal contrary to Hopcroft's algorithm that has an $O(kn\log n)$ complexity in this case, and that requires a complete automaton. Hence, it is faster than Hopcroft's algorithm when few states are to be merged, which is a frequent situation. The algorithm consists first in building a digital tree, called the signature trie, which stores the lexicographically-sorted outputs of states. This technique, called multiset discrimination in [15], [6] is used to avoid hashing and to produce deterministic algorithms. In the second step the tree is updated after each state merging.

Our algorithm applies to the class of almost-of-finite-type automata, which includes finite memory automata. These are deterministic irreducible automata that are also left-closing (or co-deterministic with a finite delay), and synchronizing. The automata present channels called almost of finite type (AFT). They were introduced by Marcus [13] for coding purposes. Indeed, the theory of modulation codes provides many natural AFT examples. The AFT sofic shifts is a meaningful class, intermediate between shifts of finite type and sofic shifts from the point of view of symbolic dynamics. Indeed, it was shown by Boyle, Kitchens and Marcus in [5] that the class of AFT shifts is the unique class of shifts having a minimal cover in the sense of symbolic dynamics. It is possible to encode an unconstrained source into an AFT constrained channel having a capacity not less that the entropy of the source with a sliding block decoder [10], while it is not possible to build such a code for a general finite-state constrained channel in the case of equality of the capacity and entropy.

From the algorithmic point of view, our method is a solution to multiset discrimination with updates.

Below we first introduce the type of automata considered in the paper and then describe their minimisation algorithm.

II. Local and almost-of-finite-type automata

In this article, an *automaton* (or *cover*) is a pair $\mathcal{A} = (Q, E)$, where Q is a finite set of states, and E is a finite set of edges labeled by letters of a finite alphabet A. (Edges are triples of the form (p, a, q), $p, q \in Q$, $a \in A$.) No initial nor final states are specified in this notation. Actually, all states have to

be considered as both initial and final states. We say that an automaton is *irreducible* if it has a strongly connected graph. The set of bi-infinite words labeling a bi-infinite path in $\mathcal{A}$ is called the *sofic shift presented by* $\mathcal{A}$.

The automaton is *deterministic* if two edges with the same origin carry different labels.

The word w is said to be a *synchronizing word* of $\mathcal{A}$ if there are nonnegative integers m and a such that whenever two paths $((p_i, a_i, p_{i+1}))_{0\leq i<(m+a)}$ and $((p'_i, a_i, p'_{i+1}))_{0\leq i<(m+a)}$ of length $m+a$ have the same label w, then $p_m = p'_m$ (m stands for memory, and a for anticipation). An automaton is *synchronizing* if it has at least one synchronizing word.

Let m and a be nonnegative integers. We say that the automaton is (m,a)-*local* (or (m,a)-*definite*) if whenever two paths $((p_i, a_i, p_{i+1}))_{0\leq i<(m+a)}$ and $((p'_i, a_i, p'_{i+1}))_{0\leq i<(m+a)}$ of length $m+a$ have the same label, then $p_m = p'_m$. We say that an automaton is local (or definite, or has finite memory) if it is (m,a)-local for some integers m and a. An irreducible automaton has finite-memory if and only two distinct cycles carry different labels. Note that a deterministic and local automaton is always $(m,0)$-local for some integer m. A sofic shift that can be presented by a local automaton is said to be a *shift of finite type*.

Let m be a nonnegative integer. We say that an automaton is m-*right-closing* if whenever two paths $((p_i, a_i, p_{i+1}))_{0\leq i<m}$ and $((p'_i, a_i, p'_{i+1}))_{0\leq i<m}$ of length m have the same label and the same origin then they share the same first edge. The notion of an a-*left-closing* automaton is defined similarly when a is a nonnegative integer. Note that a deterministic automaton is 1-right-closing.

We say that an automaton is *almost of finite type* (AFT) if it is an irreducible and synchronizing automaton that is m-right-closing and a-left-closing for some integers m and a. Note that any irreducible local automaton is AFT. A deterministic automaton is AFT if and only if it is left-closing and has a synchronizing word. A sofic shift that can be presented by an AFT automaton is said to be *almost of finite type*.

Examples of a local automaton and of a non-AFT automaton are given in Figure 1 and Figure 2.

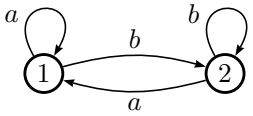

Fig. 1. A $(0,1)$-local automaton.

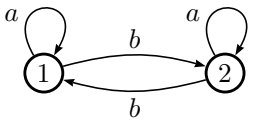

Fig. 2. A non-AFT automaton. It is both right-closing and left-closing but it is not synchronizing.

III. Minimisation of deterministic AFT automata

A. Minimisation by a sequence of state mergings

Let $\mathcal{A} = (Q, E)$ be a deterministic automaton. It is known that, when $\mathcal{A}$ is irreducible, the shift presented by $\mathcal{A}$ has a minimal deterministic irreducible automaton called the *right Fischer cover* of the shift (see for instance [12], [11], [4]). It is unique up to a renumbering of its states. It can be obtained from $\mathcal{A}$ by computing the Nerode partition of the states: two states p, q belong to the same class of this partition if and only if they have the same future, i.e., $F(p) = F(q)$, where $F(s) = \{u \in A^* \mid \text{ there is a path from } s \text{ labeled by } u \text{ in } \mathcal{A}\}$.

We say that two states p and q of $\mathcal{A}$ can be *merged* if $(p,a,r) \in E$ is equivalent to $(q,a,r) \in E$. A *state merging* identifies two mergeable states.

In general, it is not true that a deterministic non-minimal automaton has mergeable states (see the automaton of Figure 2). However, this property is true for the class of deterministic AFT automata. As a consequence, it is possible to minimize such automata by a sequence of state mergings.

The following lemma from [3, p.41] provides a efficient characterization of a left-closing automata. Efficient characterizations of AFT shifts where obtained by Boyle, Kitchens, and Marcus [5] and by Nasu [14].

The lemma uses the notion of product of two automata defined as follows. The product of the automaton $\mathcal{A}$ by itself is the automaton $\mathcal{A}^2 = \mathcal{A} \times \mathcal{A} = (Q \times Q, F)$ where $((p,q),a,(r,s)) \in F$ if and only if both $(p,a,r) \in E$ and $(q,a,s) \in E$.

Lemma 1: The automaton $\mathcal{A}$ is left-closing if and only if the automaton $\mathcal{A}^2$ has no cycle going through a state (p,q), $p \neq q$, which is co-accessible from a state of the form (r,r).

The following proposition already appeared in [3, Proposition 2.16 p.60].

Proposition 2: If a deterministic AFT automaton $\mathcal{A}$ is not minimal, then at least two of its states can be merged.

Proof: Assume that no two states of $\mathcal{A}$ can be merged, and let p and q be two distinct states having the same future.

Let C be the set of states of $\mathcal{A}^2$ accessible from (p,q). Since p and q have the same future, any states r, s such that (r,s) is in C also have the same future. We construct an infinite sequence $(p_i, q_i)_{i\geq 0}$ ($p_i \neq q_i$) of states in C defined as follows. We choose $(p_0, q_0) = (p, q)$. Hence $p_0 \neq q_0$. For $i \geq 1$, since p_i and q_i cannot be merged, for some letter a_i, (p_{i-1}, a_i, r) and (q_{i-1}, a_i, s) are edges with $r \neq s$. We choose $(p_i, q_i) = (r, s)$. Hence C contains a cycle of states (r,s) with $r \neq s$. Let (r,s) be a state on this cycle.

Since $\mathcal{A}$ is irreducible and synchronizing, there is a path starting at r labeled by w, where w is a synchronizing word. Since $F(r) = F(s)$, there is a path in $\mathcal{A}^2$ starting from (r,s) and labeled by w. This path ends in a state (t,t) since w is synchronizing. This contradicts the fact that $\mathcal{A}$ is left-closing by Lemma 1, which ends the proof. ∎

B. First step of the algorithm: building a tree

We describe below a minimisation algorithm that applies to deterministic AFT automata, and hence to irreducible deterministic local automata. The parameters of the algorithm are the number n of states of the automaton $\mathcal{A} = (Q, E)$, its number m of edges, the size k of the underlying alphabet, and

the number r of states of the minimal automaton. Of course, $m \leq kn$ since the automaton is deterministic.

We assume that $A = \{a_1, a_2, .., a_k\}$ are that the letters are ordered: $a_1 < a_2 < \cdots < a_k$. We associate with each state $p \in Q = \{1, 2 .., n\}$ its *signature* $\sigma(p) = a_1 p_1 a_2 p_2 .. a_l p_l$, where $(p, a_1, p_1), .., (p, a_l, p_l)$ are the edges starting from p in lexicographic order. A *partial signature* of state p is a prefix $a_1 p_1 a_2 p_2 .. a_r p_r$ of its signature.

In the first step of the algorithm we build a tree representing the set of all signatures. Is is called the *signature trie*. Denoted by T, it is defined as follows. Each node represents the set of states whose partial signature is $a_1 p_1 a_2 p_2 .. a_r p_r$. The root of the tree represents the set of all states. An *arc* labeled by $a_{r+1} p_{r+1}$ links the node associated with the partial signature $a_1 p_1 a_2 p_2 .. a_r p_r$ to the node associated with the partial signature $a_1 p_1 a_2 p_2 .. a_{r+1} p_{r+1}$. The leaves of the trie are the nodes associated with a complete signature. Each leaf contains the set of states having this signature.

The trie can be constructed after a lexicographical sort of the signatures. This technique is analogue to the multiset discrimination described in [15], [6], which avoids using hashing. It is used in [17] to minimize acyclic automata.

The structure of a node x of the trie is the following. It contains the (non-sorted) list *succ*(x) of the arcs leaving x. The list is indexed by all (a, p) for which ap is the label of some arc of the trie. We denote by *succ*$(x)(a, p)$ the arc labeled by ap originated from x. It contains the address of its target node. The *size* of a node of the trie is the number of states contained in the leaves of the subtree rooted at this node.

In addition to the trie T, we also maintain a non-sorted list *arc*, indexed by all (a, p) such that ap is the label of some arc of the trie, and such that *arc*(a, p) is the list of all arcs labeled by ap contained in some list *succ*(x)[1]. We also assume that each arc from x labeled by ap contains the address of the element in *arc*(a, p) pointing to it. We denote by *size*(a, p) the size of the list *arc*(a, p). It is defined as the sum of sizes of all the target nodes of arcs labeled by ap. Hence it is the number of states of the automaton with an outgoing edge labeled by a and ending in p. The implementation of the lists *succ*(x), *arc*, and *arc*(a, p) is described in Section III-D. A sparse list implementation is used for the lists *succ*(x) and *arc*. All the data structures lead to an efficient update of the trie T after two states are merged, as described in the next section.

An example of a deterministic AFT-automaton $\mathcal{A}$ is given in Figure 3. Its minimal automaton is displayed in Figure 4 and its initial signature trie is shown in Figure 5.

C. Second step of the algorithm: updating the tree

The second step of the algorithm can be shortly described as follows. The signature trie allows one to detect easily a pair (x, y) of states with the same signature (*i.e.* , with the same outputs). Indeed, x and y belong to a same leaf in this case. Then the states x and y are merged. Hence the signatures containing x or y have to be updated inside the trie. This update is the delicate point of the algorithm.

[1]Actually, in the implementation, *arc*(a, p) is the list of addresses of all nodes x such that *succ*(x) contains an arc labeled by ap.

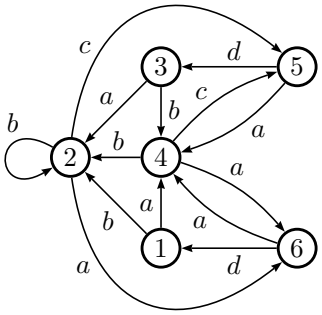

Fig. 3. A deterministic AFT automaton.

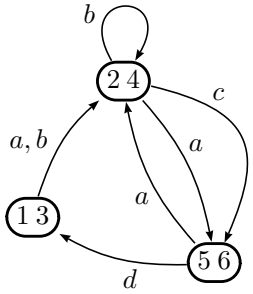

Fig. 4. The minimal automaton corresponding to the automaton of Figure 3.

For each letter $a \in A$, we denote by $\mathrm{Im}(a)$ the set of states of $\mathcal{A}$ ending edges labeled by a. For each letter a, we maintain the partition of $\mathrm{Im}(a)$ for which two states of $\mathrm{Im}(a)$ having already been merged belong to a same class. Each class has a leader. Assume that the states x and y have to be merged and both belong to $\mathrm{Im}(a)$. Let p (resp. q) be the leader of the class of x (resp. y). We keep q as the new leader of the union class when *the number of edges labeled by a and ending in p is smaller than the number of edges labeled by a and ending in q*. This smaller-half strategy, applied for each letter, will guarantee the overall running time of the algorithm. The edges labeled by a and ending in p are then changed into edges labeled by a and ending in q.

We update the edges ending in p for each letter a_i in the

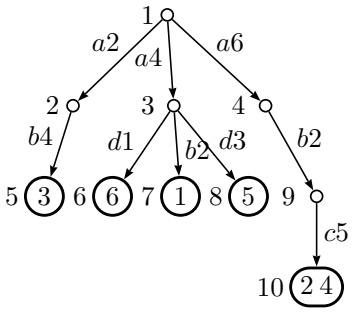

Fig. 5. The initial signature trie of the automaton of Figure 3. It is rooted at the node 1. Only one leaf contains more than one state, namely states $2, 4$. They have the same signature $a6b2c5$.

decreasing order of the a_i. Once this update is done, the same process is iterated until no more pair of states with the same outputs is detected.

In order to change all edges labeled by a and ending in p into edges labeled by a and ending in q, the procedure MERGE(a, p, q) updates the nodes x having an outgoing arc e labeled by ap. Two cases may appear according to whether x has no arc labeled by aq or it has an arc f labeled by aq. In the former case, its arc e becomes an arc labeled by aq and the lists *arc*(a, p), *arc*(a, q) are changed accordingly. In the latter case, the procedure FUSION(x_1, x_2), where x_1 (resp. x_2) is the target of e (resp. f), makes a fusion of the subtrees rooted at x_1 and x_2 by inserting x_1 into x_2. The procedures MERGE and FUSION are described below.

The main procedure is the procedure MINIMISATION-AFT-AUTOMATON which starts with the initial trie construction and sets up the lists *arc*. The leaves of the trie are then scanned. For each leaf of size greater than one containing states $q_1, \ldots, q_l$, each pair (q_i, q_{i+1}), $1 \leq i \leq l-1$, is pushed onto a stack denoted *toMerge*. This stack contains pairs of states to be merged.

MERGE (a, p, q)
1. **for** each arc e in *arc*(a, p) **do**
2. **let** x be the node origin of e
3. **if** x has no arc labeled by aq **then**
4. change the label of e into aq
5. transfer it from *arc*(a, p) to *arc*(a, q)
6. **else** (x has an arc f labeled by aq)
7. **let** x_1 be the target of e and x_2 be the target of f
8. remove the arc e from *succ*(x) and from *arc*(a, p)
10. FUSION(x_1, x_2)

FUSION (node x_1, node x_2)
1. **if** x_1 and x_2 are leaves **then**
2. **let** s be a state in x_1 and t a state in x_2
3. push (s, t) onto the stack *toMerge*
4. concatenate the list of states in x_1 to the one in x_2
5. **else**
6. **for** each arc e labeled by ap in *succ*(x_1) **do**
8. **if** x_2 has no arc labeled by ap **do**
9. transfer the arc e from *succ*(x_1) to *succ*(x_2)
10. **else** (x_2 has an arc f labeled by ap)
11. **let** y_1 (resp. y_2) be the target of e (resp. f)
12. FUSION(y_1, y_2)

In the minimisation procedure described below, CLASS(a, x) denotes a call to the procedure computing the leader of the class of x for the letter a.

MINIMISATION-AFT-AUTOMATON ($\mathcal{A} = (Q, E)$, $A = \{a_1, .., a_k\}$)
1. build the signature trie T and the lists *arc*
2. **for** each leaf $(p_1, p_2, .., p_l)$ of T **do**
3. push (p_i, p_{i+1}) onto the stack *toMerge* for $1 \leq i \leq l-1$
4. compute *size*(a, p) the size of each list *arc*(a, p)
5. **while** the stack *toMerge* is non-empty **do**
6. remove a pair (x, y) from *toMerge*
7. **for** i from k downto 1 **do**
8. **if** x and y belong to $\mathrm{Im}(a_i)$ **then**
9. **let** $(p, q) \leftarrow$(CLASS(a_i, x),CLASS(a_i, y))
10. **if** *size*$(a_i, p) \leq$*size*(a_i, q) **then**
11. q becomes the leader of the union
12. of the classes of p and q for the letter a_i
13. MERGE(a_i, p, q)
14. **else**
15. p becomes the leader of the union
16. of the classes of p and q for the letter a_i
17. MERGE(a_i, q, p)
18. **return** T

D. Complexity of the algorithm

We analyze the complexity of the algorithm. The initial signature trie and the lists *arc* are built in time $O(m)$ using radix sort for sorting the list whose elements belong to the integer interval $[1, 2, \ldots, n]$. The implementation of lists *arc* and *succ*(x) for all nodes x is done with a sparse list implementation (see [2] Exercise 2.12 p. 71 and [8] Exercise 1.14 "Implantation de fonctions partielles" Chapter 1). These lists are indexed by (a, p) where ap is the label of some arc of the trie. As a consequence, it is possible to find, add, or remove an arc in a list *succ*(x) or in a list *arc* in time $O(1)$. Initialization is done in constant time too. The space required for the implementation is $O(kn^2)$.

It remains to evaluate the time complexity of all updates. Assume that the states x and y in $\mathrm{Im}(a)$ have to be merged. Classes of the partition of $\mathrm{Im}(a)$ are implemented as trees where the leader of the class is at the root of the tree. A sequence of m Union-Find operations with path compression [7] is performed in time $O(m \min\{\alpha(m), \log n\})$, where α is the inverse of the Ackermann function. Let us denote by p (resp. q) the leader of the class of x (resp. the class of y). Assume that the number of edges labeled by a and ending in p is smaller than the number of edges labeled by a and ending in q. Then q becomes the leader of the union of the two classes. The arcs of the trie labeled by ap are changed into arcs labeled by aq through the procedure MERGE(a, p, q). The time complexity of this call is proportional to the size of the subtrees rooted at the targets of all arcs of the trie labeled by ap. Up to a multiplicative constant, it is at most the number of leaves of these subtrees, that is, *size*(a, p), which is equal to the number of edges labeled by a and ending in p in the automaton. Since *size*(a, p) is no more than *size*(a, q) before the merging operation, *size*(a, q) after the merging has at least twice the size of *size*(a, p) before the merging. Hence each edge labeled by a is changed at most $\log n$ times. Furthermore, if $n - r$ is the number of pairs of states being merged, each edge can be changed also at most $n - r$ times. Hence the cost of all these updates for all letters is $O(\min(m \log n, m(n + 1 - r)))$. As a consequence, we get the following proposition.

Proposition 3: The overall running time of the procedure MINIMISATION is $O(\min(m \log n, m(n + 1 - r)))$, where r is the number of states of the minimal automaton.

The execution of the algorithm on the automaton of Figure 3 is described in Figure 6. The forests F_a, F_b, F_c, and F_d corresponding to the partitions of $\mathrm{Im}(a)$, $\mathrm{Im}(b)$, $\mathrm{Im}(c)$, and $\mathrm{Im}(d)$ respectively, are represented in Figure 8. The lists *arc* are described in Figure 7.

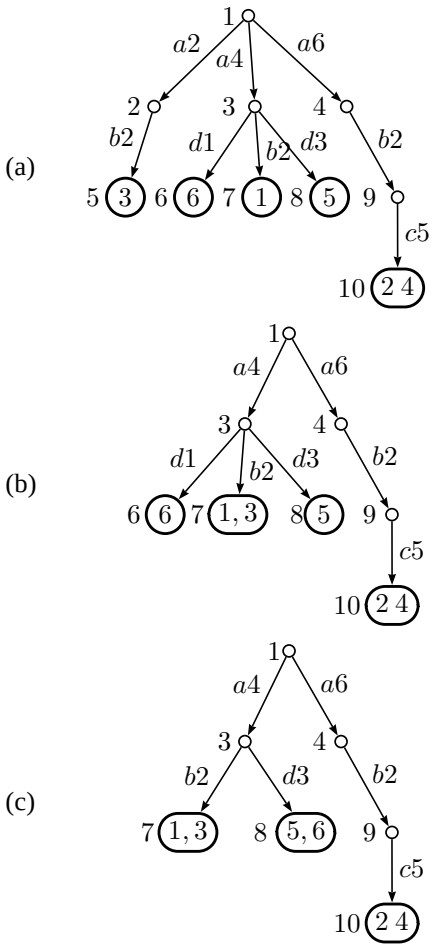

Fig. 6. Updates of the signature trie of Figure 5. The initial trie is given in Figure 5. (a) After updating arcs labeled by b. (b) After updating arcs labeled by a. This is the result of merging 2 and 4. (c) Result of merging 1 and 3: update of the arcs labeled by d. The merging of states 5 and 6 does not change the trie. The leaves of the trie correspond to the states of the minimal automaton of Figure 4.

arc	step 1	step 2	step 3	step 4
$a2$	$1(1)$		$-$	
$a4$	$1(3)$		$1(4)$	
$a6$	$1(2)$		$1(2)$	
$b2$	$3,4(3)$	$3,4,2(4)$	$3,4(4)$	$3,4(4)$
$b4$	$2(1)$	$-$		
$c5$	$9(2)$			
$d1$	$3(1)$			$-$
$d3$	$3(1)$			$3(2)$

Fig. 7. The lists *arc* during the four steps (step 1 corresponds to the initial trie of Figure 5, step 2 to Figure 6 (a), step 3 to Figure 6 (b), and step 3 to Figure 6 (c)). The size of each list $arc(a,p)$ is given in parenthesis. Each list $arc(a,p)$ contains the addresses of the nodes x of the trie for which $succ(x)$ contains an arc labeled by ap.

F_d	F_c	F_b	F_a
3		2	4
$\downarrow$	5	$\downarrow$	$\downarrow$ 6
1		4	2

Fig. 8. The forests F_a, F_b, F_c and F_d correspond to the partitions of $\mathrm{Im}(a)$, $\mathrm{Im}(b)$, $\mathrm{Im}(c)$, and $\mathrm{Im}(d)$ respectively at the end of the execution.

REFERENCES

[1] R. L. ADLER, D. COPPERSMITH, AND M. HASSNER, *Algorithms for sliding block codes*, IEEE Trans. Inform. Theory, IT-29 (1983), pp. 5–22.

[2] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[3] M.-P. BÉAL, *Codage symbolique*, Masson, 1993.

[4] M.-P. BÉAL AND D. PERRIN, *Symbolic dynamics and finite automata*, in Handbook of formal languages, Vol. 2, Springer, Berlin, 1997, pp. 463–505.

[5] M. BOYLE, B. P. KITCHENS, AND B. H. MARCUS, *A note on minimal covers for sofic shifts*, Proc. Amer. Math. Soc., 95 (1985), pp. 403–411.

[6] J. CAI AND R. PAIGE, *Using multiset discrimination to solve language processing problems without hashing*, Theoret. Comput. Sci., 145 (1995), pp. 189–228.

[7] T. H. CORMEN, C. E. LEISERSON, R. L. RIVEST, AND C. STEIN, *Introduction to algorithms*, MIT Press, Cambridge, MA, second ed., 2001.

[8] M. CROCHEMORE, C. HANCART, AND T. LECROQ, *Algorithmique du texte*, Vuibert, 2001. 347 pages.

[9] J. E. HOPCROFT, *An n log n algorithm for minimizing states in a finite automaton*, in Theory of Machines and Computations, Z. Kohavi, ed., Academic Press, New York, 1971, pp. 189–196.

[10] R. KARABED AND B. H. MARCUS, *Sliding-block coding for input-restricted channels*, IEEE Trans. Inform. Theory, IT-34 (1988), pp. 2–26.

[11] B. P. KITCHENS, *Symbolic Dynamics: one-sided, two-sided and countable state Markov shifts*, Springer-Verlag, 1997.

[12] D. A. LIND AND B. H. MARCUS, *An Introduction to Symbolic Dynamics and Coding*, Cambridge University Press, Cambridge, 1995.

[13] B. H. MARCUS, *Sofic systems and encoding data*, IEEE Trans. Inform. Theory, 31 (1985), pp. 266–377.

[14] M. NASU, *An invariant for bounded-to-one factor maps between transitive sofic subshifts*, Ergodic Theory Dynam. Systems, 5 (1985), pp. 89–105.

[15] R. PAIGE AND R. E. TARJAN, *Three partition refinement algorithms*, SIAM J. Comput., 16 (1987), pp. 973–989.

[16] M. PERLES, M. O. RABIN, AND E. SHAMIR, *The theory of definite automata*, IEEE Trans. Electronic Computers, EC-12 (1963), pp. 233–243.

[17] D. REVUZ, *Minimisation of acyclic deterministic automata in linear time*, Theoret. Comput. Sci., 92 (1992), pp. 181–189. Combinatorial Pattern Matching School (Paris, 1990).