Minimization of Sequential Transducers

Mehryar Mohri

AT & T Bell Laboratories*
e-mail address: mohri@research.att.com

Abstract. We present an algorithm for minimizing sequential transducers. This algorithm is shown to be efficient, since in the case of acyclic transducers it operates in $O(|E| + |V| + (E| - |V| + |F|).(|P_{max}| + 1)$ steps, where E is the set of edges of the given transducer, V the set of its vertices, F the set of final states, and P_{max} the longest of the greatest common prefixes of the output paths leaving each state of the transducer. It can be applied to a larger class of transducers which includes subsequential transducers.

1 Introduction

Finite automata and transducers are used in many efficient programs. They allow to produce in a very easy way lexical analyzers for complex languages. In some applications as in Natural Language Processing the involved finite-state machines can contain several hundreds of thousands of states. Reducing the size of these graphs without losing their recognition properties is then crucial.

This problem has been solved in the case of deterministic automata since any deterministic automaton has an equivalent automaton with the minimal number of states which classic algorithms help to compute from it in an easy way ([1]).

For sequential transducers, the existence of minimal transducers has already been shown in the case of group transducers (see [3]), but no algorithm has been proposed for their construction. Here we shall characterize minimal sequential transducers by means of an equivalence relation and present algorithms which allow to compute, given a sequential transducer, an equivalent minimal sequential transducer. These algorithms can be easily extended to the case of subsequential transducers.

The minimization algorithm for sequential transducers involves the computation of the *prefix of a non-deterministic automaton*. We present this algorithm first, as it is independent of the concept of sequential transducer. We then give a characterization of minimal sequential transducers and describe the entire algorithm allowing to obtain these transducers.

* I thank Maxime Crochemore and Dominique Perrin for interesting and helpful comments.

2 Prefix of a non-deterministic automaton

2.1 Definition

Let $G = (V, i, F, A^*, \delta)$ be a non-deterministic automaton where
- V is the set of states or vertices;
- $i \in V$ the initial state;
- $F \subseteq V$ the set of final states;
- A a finite alphabet;
- δ the state transition function which maps $V \times A^*$ to the set of subsets of V.[2]

We denote by
- G^T the transpose of G, namely the automaton obtained from G by reversing each transition;
- $Trans[u]$ the set of transitions leaving $u \in V$;
- $Trans^T[u]$ the set of transitions entering $u \in V$;
- $t.v$ the vertex reached by (resp. source of) t and $t.l$ its label, for any transition t in $Trans[u]$ (resp. in $Trans^T[u]$), $u \in V$;
- $out - degree[u]$ the number of edges leaving $u \in V$;
- $in - degree[u]$ the number of edges entering $u \in V$;
- E the set of edges of G.

In the following, we shall consider automata such that from any state there exists at least one path leading to a final state. We denote by $x \wedge y$ the greatest common prefix of x and y in A^*, and by ϵ the empty word of A^*. Let P be the function mapping V to A^* defined by the following:

if $u \in F$ $P(u) = \epsilon$,
else $P(u) =$ greatest common prefix of the labels of all paths leading from u to F.

P is well defined as for any u in V there exists at least one path leading from u to F. This path is finite, thus the greatest common prefix of the labels of all paths leading from $u \in V - F$ to F is well defined and in A^*. P can be equivalently defined by the following recursive definition:

$$\begin{array}{ll} \forall u \in F & P(u) = \epsilon; \\ \forall u \in V - F & P(u) = \bigwedge_{t \in Trans[u]} t.l\ P(t.v)\ . \end{array}$$

Given a non-deterministic automaton G, we define $p(G)$, *the prefix of* G, as the non-deterministic automaton which has the same set of vertices and edges as G, the same initial state and final states, and, which only differs from G by the labels of its transitions in the following way: for any edge $e \in E$ with starting

[2] Notice that the input alphabet of G is A^*. Hence, labels of transitions can be words.

state u and destination state[3] v,

$$label_{p(G)}(e) = label_G(e)P(v) \qquad if \ \text{u} = \text{i},$$
$$label_{p(G)}(e) = [P(u)]^{-1}label_G(e)P(v) \qquad else,$$

where $label_{p(G)}(e)$ is the label of the edge e in $p(G)$ and $label_G(e)$ its label in G. $p(G)$ is well defined as $P(u)$ is by definition a prefix of $label_G(e)P(v)$. It is easy to show that $p(G)$ recognizes the same language as G. $p(G)$ is obtained from G by *pushing* as much as possible labels from final states towards the initial state.

2.2 Computation

The definition of the function P suggests a recursive algorithm to compute $p(G)$ from G. This means to proceed by calculating P for all states of the adjacent list of $u \in V$ before calculating $P(u)$. However, in general, there does not exist a linear ordering of all states of G such that if the adjacent list of u contains v, then v appears before u in the ordering. Two states can indeed belong to a cycle. Hence, P cannot be computed in such a way.

In case G is a dag (directed acyclic graph) such an ordering exists. The reverse ordering of a *topological sort* of a dag meets this condition ([7], [4]).[4] Therefore, in case G is acyclic, we can consider states in such an order and compute for each of them the greatest common prefix corresponding to the recursive definition of P above. In this way, the greatest common prefix computation is performed at most once for each state of G. The computation of the automaton $p(G)$ from G can be performed in a similar way by considering the states of G in the same ordering.

In case G is not acyclic, we consider SCC's (strongly connected components) of G. There exists a linear ordering of SCC's such that if the adjacent list of a state in a SCC scc_1 contains a state of another SCC scc_2, then scc_2 appears before scc_1 in the ordering: the reverse ordering of a topological sort of the dag G^{SCC}, the *component graph* of G[5].

In order to compute $p(G)$ from G, we can gradually modify the transitions of G by considering its SCC's according to this ordering. Each time a SCC scc is considered, all the transitions leaving states of scc are transformed into those

[3] Notice that there can be several edges with starting state u and destination state v in G. These edges can even bear the same labels.

[4] This ordering can be obtained in linear time $O(|V|+|E|)$ since it corresponds to the increasing ordering of the finishing times in a depth-first search.

[5] Recall that the component graph of G is the dag which contains one state for each SCC of G, and which contains a transition $(u, a, v) \in V \times A^* \times V$ if there exists a transition from the SCC of G corresponding to u, to the SCC of G corresponding to v. It can be obtained in linear time $O(|E|+|V|)$ like strongly connected components of G ([1], [7], [4]).

of $p(G)$, and all the transitions from another SCC entering a state u of scc are modified in a way such that:

$$\forall t \in Trans^T[u],\ \ t.l = label_G(t)\ P(u).$$

Thanks to the choice of the ordering, these transformations are operated only once for each SCC. Once all SCC's of G have been considered one obviously obtains $p(G)$.

Now we need to indicate how these transformations are performed. Suppose that all the SCC's visited before scc have been correctly modified[6]. Then, according to the definition of the function P, the following system of equations:

$$\forall u \in scc,\ X_u = \Big(\bigwedge_{\substack{t \in Trans[u] \\ t.v \in scc}} t.l\ X_{t.v}\Big) \wedge \Big(\bigwedge_{\substack{t \in Trans[u] \\ t.v \notin scc}} t.l\Big),$$

has a unique solution corresponding to the greatest common prefixes of each state of scc ($\forall u \in scc, X_u = P(u)$). In order to solve this system, we can proceed in the following way:

1- For each u in scc, we compute π_u, the greatest common prefix of all its leaving transitions:

$$\pi_u \leftarrow \Big(\bigwedge_{\substack{t \in Trans[u] \\ t.v \in scc}} t.l\ X_{t.v}\Big) \wedge \Big(\bigwedge_{\substack{t \in Trans[u] \\ t.v \notin scc}} t.l\Big) \quad if\ u \notin F,$$
$$\pi_u \leftarrow \epsilon \qquad\qquad else;$$

2- If $\pi_u \neq \epsilon$, we can make a change of variables: $Y_u \leftarrow \pi_u\ X_u$.

This second step is equivalent to storing the value π_u and solving the system modified by the following operations:

2'- $\forall t \in Trans[u],\quad t.l \leftarrow \pi_u^{-1}\ t.l$
$\forall t \in Trans^T[u],\ \ t.l \leftarrow t.l\ \pi_u.$

We can limit the number of times these two operations are performed by storing in a array N the number of empty labels leaving each state u of scc. As long as $N[u] \neq 0$, there is no use performing these operations as the value of π_u is ϵ.

[6] This is necessarily true for the first SCC considered as no transition leaves it to join a distinct SCC.

Also, if $N[u] = 0$ right after the computation of π_u, then π_u will remain equal to ϵ, as changes of variables will only affect suffixes of the transitions leaving u. We can store this information using an array F, in order to avoid performing step 1 in such situations or when u is a final state. We use a queue Q containing the set of states u with $N[u] = F[u] = 0$ for which the two operations above need to be performed, and an additional array INQ indicating for each state u whether it is in Q.
We start the above operations by initializing N and F to 0 for all states in scc, and by enqueuing in Q an arbitrarily chosen state u of scc. Each time the transition of a state v of $Trans^T[u]$ is modified, v is added to Q if $N[v] = F[u] = 0$. The property of SCC's and the initialization of N and F assure that each state of scc will be enqueued at least once. Steps 1 and 2' are operated until $Q = \emptyset$. This necessarily happens as, except for the first time, step 1 is performed for a state u if $N[u] = 0$. After the computation of the greatest common prefix we have or $N[u] = 0$ and then u will never be enqueued again, or $N[u] \neq 0$ and then a new non empty factor π_u of $P(u)$ has been identified. Thus, each state u is enqueued at most $(|P(u)| + 2)$ times in Q, and after at most $(|P_{max}| + 2)$ steps we have $Q = \emptyset$.

```
PREFIX_COMPUTATION(G)
1  for each u ∈ V(G^SCC) ▷ considered in order of increasing finishing times of
                                  a DFS of G^SCC
2  do for each v ∈ SCC[u]
3       do N[v] ← INQ[v] ← F[v] ← 0
4       Q ← v ▷ v arbitrarily chosen in SCC[u]
5       INQ[v] ← 1
6       while Q ≠ ∅
7       do v ← head[Q]
8            DEQUEUE(Q)
9            INQ[v] ← 0
10           p ← GCP(G, v)
11           for each t ∈ Trans^T[v]
12           do if (p ≠ ε)
13                then if (t.v ∈ SCC[v] and N[t.v] > 0 and t.l = ε and
                              F[t.v] = 0)
14                     then N[t.v] ← N[t.v] − 1
15                     t.l ← t.l p
16                if (N[t.v] = 0 and INQ[t.v] = 0 and F[t.v] = 0)
17                then ENQUEUE(Q, t.v)
18                     INQ[t.v] = 1
```

Once $Q = \emptyset$, it is easy to notice that the system of equations has a trivial solution: $\forall u \in scc, X_u = \epsilon$. As noticed above, it has a unique solution. Therefore, the system is resolved. Concatenating the factors π_u involved in the changes of variables corresponding to the state u gives the value of $P(u)$. The set of operations $2'$ are thus equivalent to multiplying the label of each transition joining

the states u and v, ($v \in scc$), at right by $P(v)$ and at left by $[P(u)]^{-1}$ if u is in scc. This shows that the transformations described above do modify the transitions leaving or entering states of scc as desired. Thus, the above pseudocode gives an algorithm computing $p(G)$ from G.

In this algorithm, $V(G^{SCC})$ represents the set of states of the component graph of G, and, for each u in $V(G^{SCC})$, $SCC[u]$ stands for the strongly connected component corresponding to u. The function $GCP(G,u)$ called in the algorithm is such that it returns p the greatest common prefix of all the transitions leaving u ($p = \epsilon$ if $u \in F$), replaces each of these transitions by dividing them at left by p, counts and stores in $N[u]$ the number of empty transitions, and, if $N[u] = 0$ after the computation of the greatest common prefix or if u is a final state gives $F[u]$ the value 1.

2.3 Complexity

Notice that the computation of the greatest common prefix of n ($n > 1$) words requires at most $(|p| + 1).(n-1)$ comparisons, where p is the result of this computation. Indeed, this operation consists in comparing the letters of the first word to those of the $(n-1)$ others until a mismatch or an end of word occurs. The same comparisons allow to obtain the division at left by p and the number of empty transitions. In case only one transition leaves v, the computation of the greatest common prefix can be assumed to be in $O(1)$. Hence, the cost of a call of the function GCP for a state v ($\in V - F$) is in $O((|p|+1)(out-degree(v)-1)+1)$ where p is the greatest common prefix of the transitions leaving v.

As noticed above, p is a factor of the greatest common prefix of v. Thus, the loop of lines 6-18 is performed at most $(|P(u)| + 2)$ times. The cost of each iteration of the loop of lines 11-18 can be considered as being in $O(1)$ as it requires a constant number of comparisons[7]. This loop is iterated $in - degree(v)$ times inside the loop of lines 11-18 when state v is considered at line 7. All other operations (initialization, computation of the $SCC's$ and of G^{SCC} , and the definition of the reverse topological sort of G^{SCC}) can be done in $O(|V| + |E|)$. Therefore, the total cost of the algorithm above is in

$$O(|V| + |E| + \sum_{u \in V'} (out - degree(u) - 1).|P(u)| + \sum_{u \in V} in - degree(u).|P(u)|),$$

where $V' = \{u \in V \mid out - degree(u) > 1\}$,
hence in:

$$O(|V| + |E|(|P_{max}| + 1)),$$

where P_{max} is the longest of the greatest common prefixes of the states of G. In

[7] The test $t.v \in SCC[v]$ of line 13 can be performed in constant time if we assume that we have designed an array indicating for each state v in G its corresponding state in G^{SCC} . This can be done in linear time $O(|V| + |E|)$ from the $SCC's$ of G.

case G is acyclic, each SCC is reduced to one state. Therefore, the loop of lines 7-18 is performed once for each state of G. The cost of the algorithm is then in

$$O(|V| + |E| + \sum_{u \in V'} (out - degree(u) - 1).|P(u)| + \sum_{u \in V} in - degree(u)),$$

hence in:

$$O(|V| + |E| + (|E| - (|V| - |F|)).|P_{max}|)$$

At worse, if all labels of the automaton are identical for instance, $|P_{max}|$ can reach the length of the longest path from the initial state to a final state without going through cycles. The following figure represents an acyclic automaton with identical labels we shall assume equal to an element $a \in A$, in which all states except the final one have an $out - degree$ of 2. Here, we have: $|P_{max}| = |V|/2$.

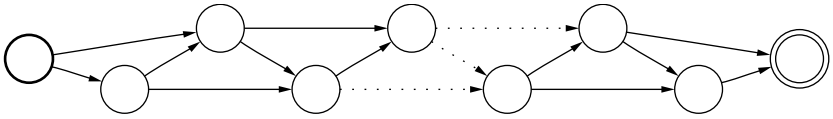

Fig. 1. Automaton G.

At each state, the number of comparisons needed for the computation of the greatest common prefix in the algorithm described above is proportional to $(d + 1)$, where d is its distance to the final state. It can be proved easily that PREFIX_COMPUTATION runs in $O(|V|^2)$ in this case. However, in most practical cases, $|P_{max}|$ is very small compared to $|V|$, and the algorithm can be considered to be very efficient. In the following section, we shall indicate an interesting application of this algorithm.

3 Minimized sequential transducers

3.1 Definitions

A sequential transducer (ST) T is a 7-tuple $(V, i, F, A, B^*, \delta, \sigma)$ where:
- V is the set of states;
- $i \in V$ the initial state;
- $F \subseteq V$ the set of final states;
- A and B finite sets corresponding respectively to the input and output alphabets of the transducer;
- δ the state transition function which maps $V \times A$ to V;
- σ the output function which maps $V \times A$ to B^*.

The functions δ and σ can be extended to map $V \times A^*$, by the following recursive relations:

$$\forall s \in V, \forall w \in A^*, \forall a \in A,\ \delta(s,\epsilon) = s, \delta(s,wa) = \delta(\delta(s,w),a);$$
$$\sigma(s,\epsilon) = \epsilon, \sigma(s,wa) = \sigma(s,w)\sigma(\delta(s,w),a).$$

In the following, we shall consider sequential transducers such that every state is reachable from the initial state, and such that for any state there exists a path leading to a final state. A sequential function f is a function which can be represented by a ST. Namely, if f is represented by $T = (V, i, F, A, B^*, \delta, \sigma)$, then for any $w \in A^*$ such that $\delta(i,w) \in F$, $f(w) = \sigma(i,w)$. We denote by $Dom(f)$ the set of words w for which f is defined.

3.2 Theorems and computation

For any sequential function f one can define the following relation on A^*:

$$\begin{aligned}\forall (u,v) \in A^*,\ uR_f v \Longleftrightarrow\ & \exists (u',v') \in B^* \times B^*\ / \\ & \forall w \in A^*,\ uw \in Dom(f) \Leftrightarrow vw \in Dom(f), \\ & \text{and,} \qquad uw \in Dom(f) \Rightarrow u'^{-1} f(uw) = v'^{-1} f(vw).\end{aligned}$$

It is easy to show that R_f is an equivalence relation. The following lemma shows that if there exists a ST T computing f with a number of states equal to the number of equivalence classes of R_f, then T is a minimal transducer computing f.

Lemma 1. *If f is a sequential function, R_f has a finite number of equivalence classes. This number is inferior or equal to the number of states of any ST computing f.*

Proof. Let $T = (V, i, F, A, B^*, \delta, \sigma)$ be a ST computing f. Choosing $u' = \sigma(i,u)$ and $v' = \sigma(i,v)$ in the above relation allows to show easily that:

$$\forall (u,v) \in (A^*)^2,\ \delta(i,u) = \delta(i,v) \Rightarrow u\ R_f\ v.$$

$\delta(i,u) = \delta(i,v)$ also defines an equivalence relation on A^*. Hence, the number of equivalence classes of this relation, namely the number of states of T, is superior or equal to the number of classes of R_f. This proves the lemma.

In the following, we define for any sequential function f a ST whose number of states is equal to the number of equivalence classes of R_f.

Theorem 1 *For any sequential function f, there exists a minimal ST computing it. Its number of states is equal to the number of equivalence classes of R_f.*

Proof. Let f be a sequential function. Let g be the function mapping A^* to B^* defined as follows:

$$\forall u \in A^*, g(u) = \bigwedge_{\substack{w \in A^* \\ uw \in Dom(f)}} f(uw).$$

We can define a transducer $T = (V, i, F, A, B^*, \delta, \sigma)$ by the following relations[8]:
- $V = \{\overline{u} | u \in A^*\}$;
- $i = \overline{\epsilon}$;
- $F = \{\overline{u} | u \in A^* \cap Dom(f)\}$
- $\forall u \in A^*, \forall a \in A, \delta(\overline{u}, a) = \overline{ua}$;
- $\forall u \in A^*, \forall a \in A, \sigma(\overline{u}, a) = [g(u)]^{-1} g(ua)$.

The definitions of V and F are correct, as, according to the previous lemma the number of equivalence states of R_f is finite. To show that δ and σ are correctly defined we need to prove that their definitions do not depend on the choice of the element u in $\overline{u}$. This is clearly true in the case of δ, as, if u and v belong to the same class, then ua and va are also equivalent for the relation R_f. The expression defining $\sigma(\overline{u}, a)$ is well-formed since, by definition[9]:

$$\begin{array}{l} \forall w \in A^* / \ uw \in Dom(f), \quad g(u) \leq_p f(uw) \\ \Longrightarrow \forall w \in A^* / u(aw) \in Dom(f),\ g(u) \leq_p f(u(aw)) \Rightarrow g(u) \leq_p g(ua). \end{array}$$

If u and v are equivalent, according to the definition of R_f we have:

$$\begin{array}{l} \exists (u', v') \in B^* \times B^* / \forall w \in A^*, \\ \qquad uw \in Dom(f) \Leftrightarrow vw \in Dom(f), \\ \quad \text{and, } uw \in Dom(f) \Rightarrow u'^{-1} f(uw) = v'^{-1} f(vw), \end{array}$$

$$\begin{array}{ll} \text{and:} & u(aw) \in Dom(f) \Leftrightarrow v(aw) \in Dom(f), \\ & \text{and, } u(aw) \in Dom(f) \Rightarrow u'^{-1} f(u(aw)) = v'^{-1} f(v(aw)). \end{array}$$

Considering the greatest common prefix of each member of the above identities leads to:

$$\begin{array}{l} u'^{-1} g(u) = v'^{-1} g(v) \text{ and } u'^{-1} g(ua) = v'^{-1} g(va), \\ \text{hence: } [g(u)]^{-1} g(ua) = [u' v'^{-1} g(v)]^{-1} u' v'^{-1} g(va) = [g(v)]^{-1} g(va). \end{array}$$

Therefore, the definition of the output function is correct. The set of words recognized by the left automaton of T is exactly $Dom(f)$:

$$\forall u \in A^*, u \in Dom(f) \Leftrightarrow \overline{u} \in F \Leftrightarrow \delta(i, u) \in F.$$

Also, notice that:

$$\forall (a, b) \in (A^*)^2, \sigma(\overline{\epsilon}, ab) = \sigma(\overline{\epsilon}, a)\sigma(\overline{a}, b) = [g(\epsilon)]^{-1} g(a) [g(a)]^{-1} g(ab) = g(ab).$$

---

[8] We denote by $\overline{u}$ the equivalence class of u. In order to shorten this presentation we shall assume in the following without reducing the generality of the theorem that $g(\epsilon) = \epsilon$.

[9] The notation $u \leq_p v$ means that u is a prefix of v.

A recursive application of these identities leads to: $\forall u \in A^*$, $\sigma(\overline{\epsilon}, u) = g(u)$. Now, if $u \in Dom(f)$:

$$g(u) = \bigwedge_{\substack{w \in A^* \\ uw \in Dom(f)}} f(uw) \leq_p f(u).$$

Since f is sequential, we also have:

$$f(u) \leq_p g(u),$$

hence:

$$u \in Dom(f) \Rightarrow \sigma(\overline{\epsilon}, u) = g(u) = f(u).$$

Thus, T is a ST representing f which has the minimal number of states. This proves the theorem.

Considered as an $A \times B'$-automaton, where $B' \subseteq B^*$ is the set of its output labels, a ST $T = (V, i, F, A, B^*, \delta, \sigma)$ can be minimized. However, the corresponding algorithm ([1]) does not necessarily lead to a minimal transducer. We here prove that once the algorithm described in the previous section has been applied to the output automaton of T, namely to the automaton which has the same states and edges as T and whose labels are the output labels of T, the minimization of a ST in the sense of automata leads to the minimal transducer as defined above.

Given a ST $T = (V, i, F, A, B^*, \delta, \sigma)$, the application of the PREFIX_COMPUTATION algorithm to the output automaton of T has no effect on the states of T, nor on its transition function. Only its output function σ is changed. We can denote by $T_2 = (V, i, F, A, B^*, \delta, \sigma_2)$ the resulting transducer. Let P be the function which maps V to B^* defined by the following recursive relations:

$$\forall s \in F, \quad P(s) = \epsilon;$$
$$\forall s \in V - F, P(s) = \bigwedge_{a \in A} \sigma(s, a)\ P(\delta(s, a)).$$

$P(s)$ is thus the greatest common prefix of the outputs of all words accepted by the left automaton of T when read from the state s. In order to simplify this presentation we shall assume that $P(i) = \epsilon$ [10]. According to the previous section, σ_2 is defined by:

$$\forall s \in V, \quad \sigma_2(s, a) = [P(s)]^{-1}\ \sigma(s, a)\ P(\delta(s, a)),$$
and, $\forall u \in A^*, \sigma_2(i, u) = \sigma(i, u)\ P(\delta(i, u))$.[11]

If $\delta(i, u) \in F$, we have: $\sigma_2(i, u) = \sigma(i, u)$. Therefore, T and T_2 compute the same sequential function. Let $T_3 = (V_3, i_3, F_3, A, B^*, \delta_3, \sigma_3)$ be the ST obtained from T_2 by minimization in the sense of automata. This operation can be performed

[10] This is equivalent to the assumption made above: $g(\epsilon) = \epsilon$.

[11] Notice that this is equivalent to: for any u in A^*, $\sigma_2(i, u) = g(u)$.

by merging the equivalent states of T_2 considered as an $A \times B'$-automaton, where $B' \subseteq B^*$ is the set of all output labels of T_2. Two states s_1 and s_2 of T_2 are equivalent in this sense iff:

$\forall w = w_1 ... w_n \in A^*$,
$\delta(s_1, w) \in F \Leftrightarrow \delta(s_2, w) \in F$ and $(w_0 = \epsilon)$:
$\delta(s_1, w) \in F \Rightarrow \forall i \in [0, n-1], \sigma_2(\delta(s_1, w_0 ... w_i), w_{i+1}) = \sigma_2(\delta(s_2, w_0 ... w_i), w_{i+1})$.

Let f be the sequential function computed by T_1 and T_2. The following lemma helps to prove that T_3 is the minimal ST computing f as defined in the previous section.

Lemma 2. *For any (u, v) in $(A^*)^2$, if $\overline{u} = \overline{v}$, then the states $\delta(i, u)$ and $\delta(i, v)$ are equivalent.*

Proof. Let (u, v) be in $(A^*)^2$, $s_1 = \delta(i, u)$ and $s_2 = \delta(i, v)$. It is easy to show that $\overline{u} = \overline{v}$ implies that:

$$\forall w \in A^*, \delta(s_1, w) \in F \Rightarrow \sigma_2(s_1, w) = \sigma_2(s_2, w).$$

Since for any i in $[0, n-1]$, $u \; R_f \; v$ implies $uw_0 ... w_i \; R_f \; vw_0 ... w_i$, for any $w = w_1 ... w_n \in A^*$ such that $\delta(s_1, w) \in F$, we also have:

$$\forall i \in [0, n-1], \; \sigma_2(\delta(s_1, w_0 ... w_i), w_{i+1} ... w_n) = \sigma_2(\delta(s_2, w_0 ... w_i), w_{i+1} ... w_n),$$

which is equivalent to:

$$\forall i \in [0, n-1], \; \sigma_2(\delta(s_1, w_0 ... w_i), w_{i+1}) = \sigma_2(\delta(s_2, w_0 ... w_i), w_{i+1}).$$

Therefore, s_1 and s_2 are equivalent states. This ends the proof of the lemma.

Now, since T_3 is obtained by merging the equivalent states of T_2 the lemma is equivalent to:

$$\forall (u, v) \in A^*, u R_f v \Rightarrow \delta_3(i_3, u) = \delta_3(i_3, v).$$

This implies that the number of states of T_3 is inferior or equal to the number of equivalence classes of R_f. As T_3 is a ST which computes the same function as T_2, by lemma 1 we prove that these two numbers are equal. T_3 is a minimal ST computing f, its states can be identified with the equivalence classes of R_f, and it is easy to show that it is exactly the minimal transducer as defined in the previous section. This proves the following theorem.

Theorem 2 *Given a ST T, a minimal ST computing the same function as T can be obtained by applying the PREFIX_COMPUTATION algorithm to the output automaton of T, and the minimization in the sense of automata to the resulting transducer. This minimal ST is the one defined in the previous section.*

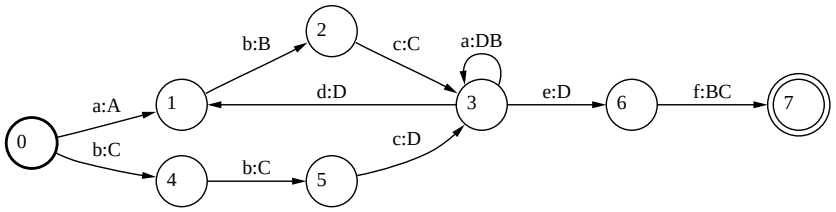

Fig. 2. Sequential transducer T.

Figures 2-4 illustrate the minimization of sequential transducers in a particular case. Consider the ST T represented in figure 2. This transducer is minimal in the sense of automata. Still, it can be minimized following the process described above.

The application of the PREFIX_COMPUTATION algorithm leads to the transducer T_2 (figure 3) which computes the same function. Only outputs differ from those of T.

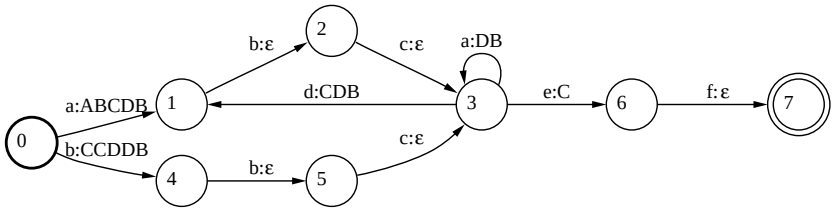

Fig. 3. Transducer T_2.

This ST is not minimal in the sense of automata. The corresponding minimization leads to the reduced transducer represented in figure 4 which is the minimal ST as defined previously.

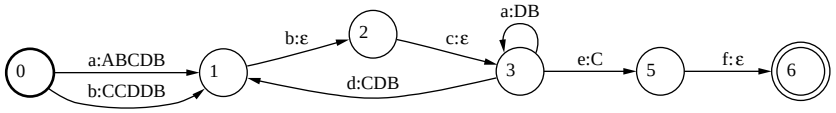

Fig. 4. Transducer T_3.

3.3 Complexity

In the case of acyclic transducers one may use a specific minimization algorithm for automata ([6]) which runs in linear time. Therefore, in this case, the whole

process of minimization of a ST T can be done in $O(|V| + |E| + (|E| - (|V| - |F|)).|P_{max}|)$ steps, where P_{max} is the longest of the greatest common prefixes of the output paths leaving each state of T.

In the general case, the classic automata minimization algorithm (see [1]) runs in $O(|A|.|V|.\log|V|)$. It can be shown that a better implementation of the algorithm described in [1] makes it independent of the size of the alphabet. It then depends only on the in-degree of each state. Thus, a better evaluation of the running time of this algorithm is $O(|E|.\log|V|)$. And, the general minimization of sequential transducers runs in $O(|V| + |E|.(\log|V| + |P_{max}|))$.

4 Conclusion

The algorithm described above can obviously be also applied to subsequential transducers[12]. Indeed, any subsequential transducer $T = (V, i, F, A, B^*, \delta, \sigma, \varphi)$ can be transformed into a ST simply by adding a new special symbol \$ to the alphabet A, a new state to V being the single final state of T, and by replacing φ by transitions to this final state labeled by \$ and the value of φ. Several implementations of this algorithm in order to minimize the phonetic subsequential transducer of French, namely the transducer which associates with words of French their phonemic pronunciations, have proved it to be very efficient.

References

1. Aho, Alfred; John Hopcroft; Jeffrey Ullman. The design and analysis of computer algorithms. Reading, Mass.: Addison Wesley, (1974).
2. Berstel, Jean. Transductions and Context-Free Languages. Stuttgart: B.G.Teubner, (1979).
3. Choffrut, Christian.Contributions à l'étude de quelques familles remarquables de fonctions rationnelles. Université Paris 7, thèse de doctorat d'Etat, Paris: LITP, (1978).
4. Cormen, Thomas H.; Charles E. Leiserson; Ronald L. Rivest. Introduction to Algorithms. 2nd edition, Cambridge, Mass.: The MIT Press, New York: MacGraw-Hill Book Company, (1990).
5. Eilenberg, S.. Automata, Languages, and Machines. Volume A, New York: Academic Press, (1974).
6. Revuz, Dominique. Dictionnaires et lexiques, méthodes et algorithmes. Université Paris 7, thèse de doctorat, Paris: LITP, (1991).
7. Sedgewick, Bob. Algorithms. 2^{nd} edition, Reading, Mass.: Addison Wesley, (1988).

[12] Subsequential transducers can be represented by an 8-tuple $(V, i, F, A, B^*, \delta, \sigma, \varphi)$, where $(V, i, F, A, B^*, \delta, \sigma)$ is a ST and φ a final function which maps F to B^* ([2]).