Incremental Construction of Minimal Acyclic Sequential Transducers from Unsorted Data

Wojciech Skut
Rhetorical Systems Ltd
4 Crichton's Close
Edinburgh EH8 8DT, Scotland
wojciech.skut@rhetorical.com

Abstract

This paper presents an efficient algorithm for the incremental construction of a minimal acyclic sequential transducer (ST) for a dictionary consisting of a list of input and output strings. The algorithm generalises a known method of constructing minimal finite-state automata (Daciuk et al., 2000). Unlike the algorithm published by Mihov and Maurel (2001), it does not require the input strings to be sorted. The new method is illustrated by an application to pronunciation dictionaries.

1 Introduction

Sequential transducers constitute a powerful formalism for storing and processing large dictionaries. Each word in the dictionary is associated with an annotation, e.g., phonetic transcription or a collection of syntactic features. Since STs are deterministic, lexical lookup can be performed in linear time. Space efficiency can be achieved by means of minimisation algorithms (Mohri, 1994; Eisner, 2003).

In the present paper, we consider the following problem. Given a list of strings $w^{(1)} \dots w^{(m)}$ associated with annotations $o^{(1)} \dots o^{(m)}$, we want to construct a minimal ST T implementing the mapping $f(w^{(j)}) = o^{(j)}, j = 1 \dots m$.

The naïve way of doing that would be first to create a (non-minimal) ST implementing f and then to minimise it. As pointed out by Daciuk et al. (2000), this can be inefficient, especially for large m. Instead, the same task can be performed more efficiently in an *incremental* way, i.e., by constructing a sequence of transducers $T_1 \dots T_m$ such that each T_j is the minimal ST implementing the restriction of the original mapping f to the first j words ($f|_{\{w^{(1)} \dots w^{(j)}\}}$). Since the insertion of a new word w^{j+1} typically affects only few states of the transducer, T_{j+1} can be constructed from T_j by changing only a small part of its structure.

Daciuk et al. (2000) show how to incrementally construct a minimal finite-state automaton for a list of words $w_1 \dots w_m$. Their algorithm can also be applied to transducers, but fails to produce a minimal ST in the general case. Mihov and Maurel (2001) describe an algorithm that handles the ST case correctly, but requires the words to be sorted in advance. In some applications, this requirement is unrealistic as lexical entries may be added dynamically to an already constructed dictionary.[1] The present paper presents an algorithm that does not make assumptions about the order of the list $w^{(1)} \dots w^{(m)}$.

The paper is structured as follows. Section 2 introduces definitions and notation. The original algorithm for finite automata is described in section 3. Section 4 explains why the algorithm does not work for transducers. The required generalisation is introduced in section 5; an algorithm based on this generalisation is the topic of section 6. Section 7 illustrates the new method with a practical application.

2 Definitions

Definition 1. (Deterministic FSA) *A deterministic finite-state automaton (DFSA) over an alphabet Σ is a quintuple $A = (\Sigma, Q, q_0, \delta, F)$ such that Q is a finite set of states, $q_0 \in Q$ is the initial state, $F \subset Q$ the set of final states, and $\delta : Q \times \Sigma \to Q$ is a (partial) transition function. δ can be extended to the domain $Q \times \Sigma^*$ by the following definition: $\delta^*(q, \epsilon) = q$, $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$.*

A accepts a string $w \in \Sigma^*$ if $q = \delta^*(q_0, w)$ is defined and $q \in F$. The set of all strings accepted by A is called the language of A and

[1] Many systems employ a built-in lexicon that is constructed off-line, but may be extended with user dictionaries merged in at any time in any order. The only way to use the sorted-data algorithm would be to unfold the already minimised lexicon, add the new entries, sort the data and re-apply the construction method.

written $\mathcal{L}(A)$. *An FSA is called* trim *if every state belongs to a path from* q_0 *to a final state.*

Definition 2. (Right Language) *Let* $A = (\Sigma, Q, q_0, \delta, F)$ *be a DFSA. The right language* $\overrightarrow{\mathcal{L}}_A(q)$ *of a state* q *in* A *is the set of all strings* w *such that* $\delta^*(q, w) \in F$. *If* $\delta^*(q_0, u)$ *is defined for* $u \in \Sigma^*$, *the right language of* u *is defined as* $\overrightarrow{\mathcal{L}}_A(u) \stackrel{\text{def}}{=} \overrightarrow{\mathcal{L}}_A(\delta^*(q_0, u))$. *We omit the subscript whenever* A *can be inferred from the context.*

Definition 3. (Sequential Transducers) *A sequential transducer (ST) over an input alphabet* Σ *and an output alphabet* Δ *is a 7-tuple* $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ *such that* $A = (\Sigma, Q, q_0, \delta, F)$ *is a DFSA and* $\sigma : Q \times \Sigma \to \Delta^*$ *is a function that labels transitions with emissions from* Δ^* *(*$Dom(\sigma) = Dom(\delta)$*).*

Function σ *can be extended to* $Q \times \Sigma^*$ *according to the following recursive definition:* $\sigma^*(q, \epsilon) = \epsilon, \sigma^*(q, wa) = \sigma^*(q, w)\sigma(\delta^*(q, w), a)$.

Unless indicated otherwise, the definitions formulated above for DFSAs also apply to STs ($\mathcal{L}(T)$, $\overrightarrow{\mathcal{L}}(u)$, $\overrightarrow{\mathcal{L}}(q)$*).*

Each ST T *realises a function* $f_T : \Sigma^* \to \Delta^*$ *such that* $Dom(f_T) = \mathcal{L}(T)$ *and* $f_T(u) = \sigma^*(q_0, u)$ *for* $u \in Dom(f_T)$. *A* sequential *function is one that can be realised by an ST.*

Definition 4. (Minimal FSA/ST) *A DFSA* $A = (\Sigma, Q, q_0, \delta, F)$ *is called minimal if* $|Q| \leq |Q'|$ *for all DFSA* $A' = (\Sigma, Q', q_0', \delta', F')$ *such that* $\mathcal{L}(A') = \mathcal{L}(A)$.

An ST $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ *is called minimal if* $|Q| \leq |Q'|$ *for any ST* $T' = (\Sigma, \Delta, Q', q_0', \delta', \sigma', F')$ *such that* $f_{T'} = f_T$.

2.1 String Notation

For $u, v \in \Sigma^*$, uv and $u \cdot v$ denote the concatenation of u and v. $u \wedge v$ is the longest common prefix of u and v. If u is a prefix of v (written $u \leq_p v$), $u^{-1}v$ denotes the remainder of v: $u \cdot u^{-1}v = v$. $|u|$ denotes the length of u, while $u_1 \dots u_{|u|}$ are its letters. $u_{[j \dots k]}$ denotes the substring $u_j \dots u_k$ of u ($u_{[j \dots k]} = \epsilon$ if $j > k$). If T is an ST, $w \wedge T$ stands for the longest prefix $u \leq_p w$ such that $\delta^*(q_0, u)$ is defined.

3 Minimisation of FSAs

The algorithm decribed by Daciuk et al. (2000) is iterative. In each iteration, given a minimal acyclic trim FSA $A = (\Sigma, Q, q_0, \delta, F)$ and a word $w \in \Sigma^*$, the algorithm creates a DFSA $A' = (\Sigma, Q', q_0, \delta', F')$ such that A' is the minimal automaton for the language $\mathcal{L}(A) \cup \{w\}$. A' then serves as input to the next iteration.

The key notion here is the *equivalence of states*. Two states $q_1, q_2 \in Q$ are considered equivalent (written $q_1 \equiv q_2$) iff $\overrightarrow{\mathcal{L}}(q_1) = \overrightarrow{\mathcal{L}}(q_2)$. A well-known result states that a trim FSA is minimal iff it does not contain a pair q_1, q_2 of distinct but equivalent states:

$$\forall_{q_1, q_2 \in Q} : q_1 \neq q_2 \Rightarrow q_1 \not\equiv q_2 \qquad (1)$$

Each iteration consists of two steps, which can be called *insertion* and *local minimisation.*[2]

3.1 Insertion

The insertion operation identifies the longest prefix $w_1 \dots w_l$ of w in A and the corresponding states $q_0 \dots q_l$. Some of them may be *confluence states*, i.e., states q_i such that $in - degree(q_i) > 1$. In order to prevent overgeneration, the algorithm identifies the first confluence state q_k and clones the path $q_k \dots q_l$. The cloned states $\hat{q}_k \dots \hat{q}_l$ are copies of the original ones, i.e., $\hat{\delta}(q_i, a) = \delta(q_i, a)$ for all $a \in \Sigma$ such that $\delta(q_i, a)$ is defined – with the only exception of the transition consuming the next symbol of w: $\hat{\delta}(\hat{q}_i, w_{i+1}) = \hat{q}_{i+1}$. Furthermore, $\hat{\delta}(q_{k-1}, w_k) := \hat{q}_k$.

In the next step, a chain of states $\hat{q}_{l+1} \dots \hat{q}_t$, $\hat{q}_t \in \hat{F}$, consuming the remainder of w (i.e., $w_{l+1} \dots w_t$) is appended to $\hat{q}_l$ (if it has been created) or to q_l. If $l = t$, the remainder is the empty string, and we make sure that $q_l / \hat{q}_l \in \hat{F}$.

Formally, this step creates an automaton $\hat{A} = (\Sigma, \hat{Q}, q_0, \hat{\delta}, \hat{F})$ such that, for $i \in \{k, \dots, t\}$:[3]

$$\begin{aligned} \hat{Q} &= Q \cup \{\hat{q}_k \dots \hat{q}_t\} \\ \hat{F} &= F \cup \{\hat{q}_i : q_i \in F\} \\ \hat{\delta}(q, a) &= \begin{cases} \hat{q}_k & : q = q_{k-1}, a = w_k \\ \hat{q}_{i+1} & : q = \hat{q}_i, a = w_{i+1} \\ \delta(q, a) & : \text{ otherwise.} \end{cases} \end{aligned}$$

This completes the insertion step. The new automaton $\hat{A}$ obviously accepts the language $\mathcal{L}(A) \cup \{w\}$ and preserves the right languages of all states except $q_0 \dots q_{k-1}$.

3.2 Local Minimisation

The situation after insertion is that $\hat{A}$ contains

- a path $q_0 \dots q_{k-1}$ of states whose right languages may have changed,

[2] The following description refers to a simpler variant of the algorithm rather than to the optimised version described in the pseudocode in the original publication. Optimisation is discussed separately in section 6.5.

[3] We set $k := l + 1$ if there are no confluence states.

- a path $\hat{q}_k \dots \hat{q}_t$ of newly created (partly cloned) states, and
- the remaining states of A, whose right languages have not changed (i.e., (1) still holds for $Q \backslash \{q_0 \dots q_{k-1}\}$).

In order to make the new FSA minimal, the algorithm must enforce condition (1) for the states $q_0 \dots q_{k-1}\hat{q}_k \dots \hat{q}_t$ by replacing them, if possible, by their equivalents in a set $Q_{\not\equiv}$, which is initially set to $Q \backslash \{q_0 \dots q_{k-1}\}$.

The sequence is traversed in reverse order, starting from $\hat{q}_t$. In the j-th iteration ($j = 1 \dots t-2$), the algorithm checks if there is already a state $q' \in Q_{\not\equiv}$ equivalent to the current state q. If such a q' exists, q is replaced by q' (i.e., q is deleted and all transitions reaching q are redirected to q'). Otherwise, q is added to $Q_{\not\equiv}$. In this way, the algorithm gets rid of duplicates w.r.t. the equivalence relation $\equiv$. The automaton left after the last iteration satisfies condition (1), i.e., it is minimal.

3.3 The State Register

The efficiency of the algorithm depends on its ability to quickly check the equivalence of states. This check is fast because $\delta^*(q, u) \in Q_{\not\equiv}$ for all $q \in Q_{\not\equiv}$ (at any stage of the local minimisation step). In effect, $q_1 \equiv q_2 \iff Out(q_1) = Out(q_2) \wedge (q_1, q_2 \in F \vee q_1, q_2 \notin F)$, where $Out(q) = \{(a, q') : \delta(q, a) = q'\}$ is the set of transitions leaving q. Thus, for each $q \in Q_{\not\equiv}$, the pair $(Out(q), q)$ is put on a *register*, i.e., an associative container that maps sets of pairs $(input, state)$ (uniquely identifying a right language) to the corresponding states in $Q_{\not\equiv}$.

4 Application to Transducers

The problem for sequential transducers can be stated as follows: given a minimal ST T implementing a sequential function f, we want to insert into T a string w associated with an emission o, creating a minimal ST for $f \cup \{(w, o)\}$. Daciuk et al. (2000) state on this topic:

> This new algorithm can also be used to construct transducers. The alphabet of the (transducing) automaton would be $\Sigma_1 \times \Sigma_2$, where Σ_1 and Σ_2 are the alphabets of the levels. Alternatively, as previously described, elements of Σ_2^* can be associated with the final states of the dictionary and only output once a valid word from Σ_1^* is recognised.

Unfortunately, both suggested solutions are problematic. They require that we commit ourselves to a particular alignment of input and output symbols in the transitions in advance, before running the algorithm. For instance, consider the fragment of a pronunciation dictionary shown below.

```
but | b uh t
bite | b ai t
cut | k uh t
cite | s ai t
```

Obviously, there are several string-to-string transducers that implement this dictionary. One possibility would be to encode the mapping in a phonologically motivated way, i.e., associating each phonetic symbol with the grapheme(s) it corresponds to. Unfortunately, the result of applying an FSA minimisation algorithm is nondeterministic (figure 1).

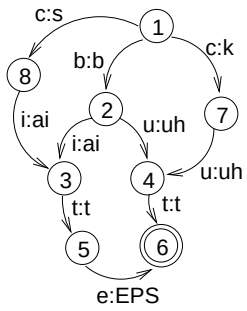

Figure 1: A "phonologically motivated" alignment of input and output symbols.

The second suggestion made by Daciuk et al. (2000), the use of final emissions, can be emulated using a special end-of-string symbol \$. The result of FSA minimisation, shown at the top of figure 2, is an ST, but not minimal, since it has more states than the ST shown below.

5 ST Minimality Criteria

In order to adapt the original algorithm to transducers, we need to find an ST counterpart of the $\equiv$ relation defined for finite-state automata. The following proposition constitutes a good point of departure.

Proposition 1. *(Mohri, 1994) If $f : \Sigma^* \to \Delta^*$ is a sequential function, there exists a minimal FST $T = (\Sigma, \Delta, Q, i, \delta, \sigma, F)$ realising f. The size $|T|$ $(= |Q|)$ of T is equal to the count of the equivalence relation R_f defined as*

$$uR_f v \iff \overrightarrow{\mathcal{L}}(u) = \overrightarrow{\mathcal{L}}(v) \wedge \exists_{u', v' \in \Delta^*} \forall_{w \in \overrightarrow{\mathcal{L}}(u)} u'^{-1} f(uw) = v'^{-1} f(vw) \quad (2)$$

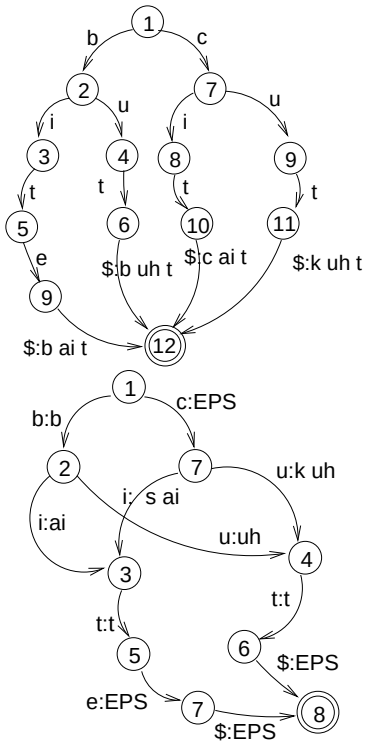

Figure 2: An ST with final emissions and a minimal ST.

R_f is defined on the set $(\Sigma^*)^2$. In order to adapt the original algorithm, we must define an equivalence relation on Q, analogous to $\equiv$. It turns out that this is possible for transducers that are *prefix-normalised*, as stated in the following definition.

Definition 5. *A sequential transducer* $T = (\Sigma, \Delta, Q, i, \delta, \sigma, F)$ *is prefix-normalised if*

$$\forall_{u \in \Sigma^*, \vec{\mathcal{L}}(u) \neq \emptyset} \sigma(q_0, u) = \bigwedge_{z \in \vec{\mathcal{L}}(u)} f_T(uz) \qquad (3)$$

The following proposition allows us to define an equivalence relation on Q analogous to R_f.

Proposition 2. *If a trim sequential transducer* $T = (\Sigma, \Delta, Q, i, \delta, \sigma, F)$ *that realises a function* $f : \Sigma^* \to \Delta^*$ *is prefix-normalised, then the count of* R_f *is equal to the count of the relation* $\equiv_{\mathrm{ST}} \subset Q^2$ *defined as follows:*

$$q \equiv_{\mathrm{ST}} q' \iff \vec{\mathcal{L}}(q) = \vec{\mathcal{L}}(q') \wedge \forall_{w \in \vec{\mathcal{L}}(q)} \sigma^*(q, w) = \sigma^*(q', w)$$

Proof. Since T is trim, it is sufficient to prove

$$uR_f v \iff \delta^*(q_0, u) \equiv_{\mathrm{ST}} \delta^*(q_0, v)$$

$\Leftarrow$: follows immediately with $u' = \sigma^*(q_0, u)$, $v' = \sigma^*(q_0, v)$, since $\vec{\mathcal{L}}(u) \stackrel{\mathrm{def}}{=} \vec{\mathcal{L}}(\delta^*(q_0, u))$.
$\Rightarrow$: Let $q := \delta^*(q_0, u)$, $q' := \delta^*(q_0, v)$. If $uR_f v$ then $\vec{\mathcal{L}}(q) \stackrel{(2)}{=} \vec{\mathcal{L}}(q')$ and there exist $u', v' \in \Delta$ such that:

$$\forall_{w \in \vec{\mathcal{L}}(q)} : u'^{-1} f(uw) = v'^{-1} f(vw)$$

Since $f(uw) = \sigma^*(q_0, u)\sigma^*(q, w)$, this is equivalent to:

$$u'^{-1}\sigma^*(q_0, u)\sigma^*(q, w) = v'^{-1}\sigma^*(q_0, v)\sigma^*(q', w)$$

Furthermore, u' and v' must be prefixes of $\sigma^*(q_0, u)$ and $\sigma^*(q_0, v)$, respectively (otherwise, T would not be prefix-normalised), thus there exist u'', v'' such that $u'u'' = \sigma^*(q_0, u), v'v'' = \sigma^*(q_0, v)$ and

$$\forall_{w \in \vec{\mathcal{L}}(q)} : u''\sigma^*(q, w) = v''\sigma^*(q', w)$$

This holds only if u'' is a prefix of v'' or vice versa. Without loss of generality assume $v'' = u''z$. Then it follows:

$$\forall_{w \in \vec{\mathcal{L}}(u)} \sigma^*(q, w) = z\sigma^*(q', w)$$

Therefore, for all $w \in \vec{\mathcal{L}}(q')$, z is a prefix of $\sigma^*(\delta^*(q_0, v), w)$. Since T is prefix-normalised, this implies $z = \epsilon$, i.e. $u'' = v''$, hence $\forall_{w \in \vec{\mathcal{L}}(u)} \sigma^*(\delta^*(q_0, u), w) = \sigma^*(\delta^*(q_0, v), w)$, i.e., $q \equiv_{\mathrm{ST}} q'$. □

6 The Algorithm

According to proposition 2, a modification of the original algorithm (by Daciuk et al. (2000)) shall produce a minimal ST if $\equiv$ is replaced by $\equiv_{\mathrm{ST}}$, and the transducer being constructed is prefix-normalised in each iteration. As in the original approach, each iteration is a two-step operation: first, a new word-output pair (w, o) is inserted into a minimal, trim and prefix-normalised transducer T, creating a prefix-normalised, "almost minimal" transducer $\hat{T}$. In the second step, the "redundant" states on the path of w are merged with equivalent states in $\hat{T}$, resulting in a minimal, trim and prefix-normalised ST implementing $f_T \cup \{(w, o)\}$.

The modifications to the FSA algorithm required in order to adapt it to sequential transducers are discussed in sections 6.1 and 6.2. The pseudocode is presented in section 6.3.

6.1 Insertion

Like the original algorithm, the modified one identifies $w \wedge T = q_0 \dots q_l$ and creates new states $\hat{q}_{l+1} \dots \hat{q}_t$. It also clones the path $q_k \dots q_l$ from the first confluence state (if there is one).

The main complication is due to the insertion of the output sequence o. In order for the ST to be well-formed, the output $\hat{\sigma}^*(q_0, w \wedge T)$ generated by the prefix $w \wedge T$ must be a prefix of o. However, the original output $\sigma^*(q_0, w \wedge T)$ in T might not meet this requirement.

The solution is to "push" some of the outputs away from the path of the prefix $w \wedge T = w_{[1\dots l]}$. However, one must be careful not to change the right languages (and their translations) of the states that are not on the path of w. Furthermore, proposition 2 requires the resulting ST to be prefix-normalised.

All this can be achieved by the following recursive definition of $\hat{\sigma}$.[4]

$$\begin{aligned} \hat{\sigma}(r_i, w_{i+1}) &:= (\hat{\sigma}^*(q_0, w_{[1\dots i]}))^{-1} \\ &\quad (o \wedge \sigma^*(q_0, w_{[1\dots i+1]})) \\ \hat{\sigma}(r_i, a) &:= (\hat{\sigma}^*(q_0, w_{[1\dots i]}))^{-1} \\ &\quad \sigma^*(q_0, w_{[1\dots i]}a), a \neq w_{i+1} \end{aligned}$$

An example of insertion is shown in figure 3.

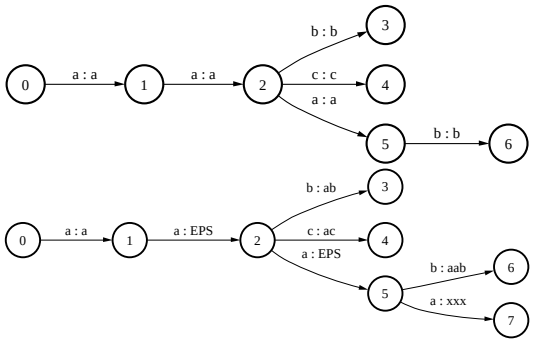

Figure 3: Insertion of pair $(aaaa, axxx)$.

As for the suffix $w_{[l+1\dots t]}$, we associate the output remainder $\hat{\sigma}^*(q_0, w_{[1\dots l]})^{-1}o$ with the first transition. The remaining ones emit ϵ.

This insertion mechanism ensures that $\hat{T}$ implements the mapping $f_T \cup \{(w, o)\}$ without changing the equivalence of states not on the path $q_0 \dots q_{k-1}$.

6.2 Local Minimisation

As in the FSA algorithm, the path $q_0 \dots q_{k-1}\hat{q}_k \dots \hat{q}_t$ is traversed in reverse order. Each state q for which there exists an equivalent state $q' \in Q_{\not\equiv}$ is replaced by q'.

What changes is the way the equivalence of two states $q_1, q_2,\ q_2 \in Q_{\not\equiv}$ is determined. Obviously, now it is no longer sufficient to compare the target states and input labels of all transitions leaving q_1 and q_2. However, since $\hat{T}$ is prefix-normalised, the only extra bit to check is the output, i.e., we keep the original formula $q_1 \equiv_{\mathrm{ST}} q_2 \iff Out(q) = Out(q') \wedge (q_1, q_2 \in F \vee q_1, q_2 \notin F)$, but redefine $Out(q)$ as $\{(a, o, q') : \delta(q, a) = q' \wedge \sigma(q, a) = o\}$.

6.3 Pseudocode

In each iteration of the main loop, the algorithm takes a pair (*Word*, *Output*) and calls the procedures INSERT() and REMOVE_DUPLICATES(), responsible respectively for the insertion and the local minimisation step. The former traverses the word from left to right, clones the path from the first confluence state down (if there is any), and ends up in the state $\delta^*(q_0, w \wedge T)$, or its copy. From there, INSERT_SUFFIX() is called, creating a chain of states corresponding to the remainder of *Word* (The pseudocode of INSERT_SUFFIX() is omitted for space reasons, but its functionality is very simple: the remainder of *Output* is emitted in the first transition of this new chain of states).[5] In each iteration of the `for`-loop, the variable *Output* holds the remainder of the original output that has not been emitted so far. PUSH_OUTPUTS(*State*, *Residual*) takes care of making the path outputs in $\hat{T}$ compatible with $o = f_{\hat{T}}(o)$. After i iterations of the loop, the argument *Residual* holds the value of $(o \wedge \sigma^*(q_0, w_{[1\dots i-1]}))^{-1}\sigma^*(q_0, w_{[1\dots i]})$, i.e., the remainder of $\sigma^*(q_0, w_{[1\dots i-1]})$ after subtracting the longest common prefix with o. This prefix is prepended to the output labels of all transitions starting in *State*.[6]

The procedure REMOVE_DUPLICATES() traverses the path of w in $\hat{T}$ (in reverse order) and removes those states for which there is an equivalent state in the register. The remaining states are added to the register (note that the procedure INSERT_SUFFIX() de-registers all states from the root down to the first confluence state – if such a state exists – or to the end of $w \wedge T$).

[4] The symbol r_i ranges over $q_i, \hat{q}_i$ (whenever the latter is defined). If there is no confluence state, $k := l + 1$.

[5] If $w \wedge T = w$ and there is some output left, then $f_T \cup \{(w, o)\}$ is not sequential.

[6] Note that if $State \in F$ and $Output \neq \epsilon$, $f_T \cup \{(w, o)\}$ is not sequential.

Algorithm 6.1: CONSTRUCTMINST()

```
main
 Register ← ∅
 while there is a word-output pair
   (Word, Output) ← next pair
   INSERT(Word, Output)
   REMOVE_DUPLICATES(Word)

procedure INSERT(Word, Output)
 State ← q0
 FoundConfluence ← false
 for i ← 1 to size(Word)
   if State ∈ Register
     Register ← Register\{State}
   Symbol ← Word[i]
   Child ← δ(State, Symbol)
   if Child = ∅ break
   if InDegree(Child) > 1
     FoundConfluence ← true
   if FoundConfluence
     δ(State, Symbol) ← CLONE(Child)
   OutputPrefix ← Output ∧ σ(State, Symbol)
   OutputSuffix ←
         OutputPrefix^-1 σ(State, Symbol)
   Output ← OutputPrefix^-1 Output
   σ(State, Symbol) ← OutputPrefix
   State ← δ(State, Symbol)
   PUSH_OUTPUTS(State, OutputSuffix)
 rof
 INSERT_SUFFIX(State, Word[i..size(Word)], Output)

procedure REMOVE_DUPLICATES(Word)
 for i ← size(Word) − 1 downto 1
 State ← δ*(q0, Word[1...i])
 Symbol ← Word[i + 1]
 Child ← δ(State, Symbol)
 if ∃q ∈ Register, q ≡ST Child
   δ(State, Symbol) ← q
  else Register ← Register ∪ {Child}
 rof

procedure PUSH_OUTPUTS(State, Residual)
 for each a ∈ Σ
 if δ(State, a) is defined
   σ(State, a) ← Residual · σ(State, a)
```

6.4 Extensions

The algorithm can be extended to the more general case of *subsequential transducers* (SST). An SST is an ST that emits *final outputs* when halting in a final state (Mohri, 1997). It can be emulated in the ST framework by appending a special end-of-string character \$ to each string, making each final state q in the transducer non-final and adding a transition from q via \$ to a new final state q_f. The output associated with this transition is the ST equivalent of a final output. In this encoding, the new algorithm can be used to construct minimal subsequential transducers.

Alternatively, the algorithm can be directly modified to cope with final outputs. In this case, the equivalence relation $\equiv_{\mathrm{ST}}$ needs to be refined by requiring that two equivalent final states must also have identical final outputs.

In some cases, the mapping $f : w \to o$ is not a function. For example, a word in a pronunciation dictionary may have two or more transcriptions. Such cases of bounded ambiguity can be handled by another extension of the ST framework, namely *p-subsequential transducers*, in which each final state is associated with up to p final outputs (Mohri, 1997). The present algorithm can be extended to this case by employing p different end-of-string symbols $\$_1 \ldots \$_p$, as in the case of SSTs. This technique was used in the application described in section 7.

6.5 Complexity and Optimisation

For a dictionary of m words, the main loop of the algorithm executes m times. The loops in procedures INSERT() (including the call to INSERT_SUFFIX()) and REMOVE_DUPLICATES() are each executed $|w|$ times for each word w. Putting a state on a register may be done in constant time when using a hash map.

Compared to the FSA algorithm, the ST generalisation has one more complexity component, namely the procedure PUSH_OUTPUTS(), which is executed in each iteration of the loop in function INSERT(). Each call to PUSH_OUTPUTS(q) involves $OutDegree(q)$ operations.

In practical implementation, there is also some overhead stemming from the use of more complex data structures (because of the need to store transition outputs). This mainly affects the efficiency of the register lookup and the INSERT() procedure.

The algorithm can be optimised by reducing the number of times states are registered/deregistered during the processing of the prefix of w in T (main loop of INSERT()). More precisely, the idea is to deregister a state only if there is any residual output pushed down the trie (i.e., the previous value of *OutputSuffix* was other than ϵ). As a result, some states $q_1 \ldots q_s$, $s < k$, may stay registered after the call to INSERT(). The loop in REMOVE_DUPLICATES() must then check whether or not $\delta(q_i, w_{i+1}) = q_{i+1}$. If not, q_{i+1} must have been replaced by an equivalent state. In such a case, we must de-register q_i and check if there are equivalent states in the register. As soon as one of the q_i's is not replaced,

there is no need to perform this check for the remaining states $q_{i-1} \ldots q_1$.

This optimisation idea is used in the original algorithm. As for STs, the speed-up achieved is moderate because PUSH_OUTPUTS() typically changes most of the outputs on the path of w.

7 Applications and Evaluation

The new algorithm has been employed to construct pronunciation lexica in the rVoice text-to-speech system.[7] In languages such as English, where the relation between orthography and pronunciation is not straightforward, it is often advantageous to store all known words in the dictionary, rather than rely on letter-to-sound rules (Fackrell and Skut, 2004). The algorithm makes it possible to store large amounts of data in such dictionaries without affecting the efficiency and flexibility of the system: the resulting representations are very compact, words can be looked up deterministically in linear time, and user-defined entries can be inserted into the dictionary at any time in any order (unlike in Maurel and Mihov's approach). This last feature in particularly important as rVoice users can control the behaviour of the system by dynamically inserting their own entries into the dictionary at runtime.

The performance of the algorithm has been evaluated by constructing a minimal ST for a pronunciation dictionary comprising the 50,000 most frequent British surnames. The size and the construction time for the ST were compared to the equivalent parameters for the sorted-data ST algorithm (Mihov and Maurel, 2001) and the unsorted-data FSA algorithm (Daciuk et al., 2000). The dictionary was not sorted, so there was an extra sorting step in the case of Maurel and Mihov's algorithm (sorting took less than 1 sec. and is not included in the reported execution time). In the FSA, the phonetic transcriptions were encoded as final emissions (i.e., the FSA encoded the language $\{w^{(1)}o^{(1)}, \ldots, w^{(m)}o^{(m)}\}$, each phonetic symbol serving as an additional symbol of the input alphabet). The ST encoding used a special end-of-string symbol \$ appended to each word in order to make sure the resulting mapping was rational.[8] The results are shown in table 1.

	ST-unsorted	ST-sorted	FSA
states	22,211	22,211	161,592
arcs	67,129	67,129	211,327
time	19 sec	12 sec	22 sec

Table 1: Comparison of three construction methods (unsorted-data ST, sorted-data ST and unsorted-data FSA) applied to a pronunciation lexicon on a Pentium 4 1.7 GHz processor.

The comparison demonstrates that STs are superior to FSAs as an encoding method for lexica annotated with rich respresentations. FSA minimisation is obviously of little help if every (or almost every) input is associated with a different annotation; almost no states are merged in the part of the FSA encoding the $w^{(i)}$'s.[9] Since the FSA is much larger, construction takes longer than in the ST case although the FSA algorithm is faster on structures of equal size.

Not surprisingly, the sorted-data algorithm is faster than the unsorted-data version, even including the actual sorting time. However, its limited flexibility restricts its applicability, leaving the new unsorted-data algorithm as the preferable option in a range of applications.

References

Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. 2000. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16.

Jason Eisner. 2003. Simpler and more general minimization for weighted finite-state automata. In *Proceedings of the Joint Meeting of the Human Language Technology Conference and the North American Chapter of the ACL*, Edmonton.

Justin Fackrell and Wojciech Skut. 2004. Improving pronunciation dictionary coverage of names by modelling spelling variation. In *Proceedings of the 5th Speech Synthesis Workshop*, Pittsburgh.

Stoyan Mihov and Denis Maurel. 2001. Direct construction of minimal acyclic subsequential transducers. *Lecture Notes in Computer Science*, 2088:217–229.

Mehryar Mohri. 1994. Minimization of sequential transducers. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM '94)*, pages 156–163.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311.

[7] `www.rhetorical.com/tts-en/technical/rvoice.html`

[8] See section 6.4. Transitions consuming \$ correspond to final emissions in a subsequential transducer. There were 6,096 such transitions in the minimal ST.

[9] In contrast, the results reported by Mihov and Maurel (2001) for a grammatical dictionary of Bulgarian show little difference between the minimal FSA (47K states) and the minimal ST (43K states). Clearly, this is due to the fact that the number of grammatical classes is substantially smaller than the number of word forms.