Incremental Construction of Minimal Sequential Transducers

The Unsorted-Data Algorithm for Acyclic Sequential Transducers

Wojciech Skut

Rhetorical Systems Ltd, Edinburgh, Scotland

Abstract

This paper presents an efficient algorithm for the incremental construction of a minimal acyclic sequential transducer (ST) from a list of input and output strings. The algorithm generalizes a known method of constructing minimal finite-state automata (Daciuk, Mihov, Watson and Watson 2000). Unlike the algorithm published by Mihov and Maurel (2001), it does not require the input strings to be sorted in advance. The algorithm is illustrated by an application in a text-to-speech system.

1 Introduction

Sequential transducers are a powerful framework for storing and processing large dictionaries of words associated with rich annotations such as phonetic transcriptions or syntactic features. Since STs are deterministic, lexical lookup can be performed in linear time. Space efficiency can be achieved by means of minimization algorithms (Mohri 1994).

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

One possible, although naïve, solution is first to create a non-minimal ST implementing f and then to minimize it. However, this is impractical or even impossible for large m due to the large size of the initial ST. 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 acceptor 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 describes an algorithm that does not make any assumptions about the order of $w^{(1)}, \dots, w^{(m)}$.

[1] Many systems employ a 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 minimized lexicon, add the new entries, sort the data and re-apply the construction method.

The paper is structured as follows. Section 2 introduces some notation and definitions. The algorithm of Daciuk et al. (2000) is described in section 3. Section 4 explains why it does not work for STs. The required generalization is introduced in section 5. The new algorithm is the topic of section 6, which also contains a proof of correctness and discusses the case of *subsequential transducers*. Section 7 illustrates the new method with a practical application.

2 Definitions and Notation

Definition 1. (Alphabet, String) *An alphabet Σ is a finite set of symbols. Σ^* denotes the set of all strings over Σ, i.e. finite sequences of symbols in Σ. 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: $u \cdot u^{-1}v = v$. The symbol $<_p$ denotes the* proper prefix*: $u <_p v \iff u \leq_p v$ and $u \neq v$. $|u|$ 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 $j > k$).*

Definition 2. (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: $\delta^*(q, \epsilon) = q$, $\delta^*(q, wa) = \delta(\delta^*(q, w), a)$.*

If A is a DFSA, $w \wedge A$ is the longest prefix of w in A, i.e. the longest prefix $u \leq_p w$ such that $\delta^*(q_0, u)$ is defined.

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

If $w = w_1 \dots w_t \in \Sigma^*$, $\delta^*(q_0, w)$ is defined and $q_i \stackrel{\text{def}}{=} \delta^*(q_0, w_{[1 \dots i]})$ for $i = 0, \dots, t$, then $q_0, q_1, \dots, q_t$ is called the path *of string w in A.*

Definition 3. (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 is the set of all $w \in \Sigma^*$ such that $\delta^*(q, w) \in F$. If $u \in \Sigma^*$ and $\delta^*(q_0, u)$ is defined, one can define the right language of u 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 4. (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^* ($Dom(\sigma) = Dom(\delta)$).*

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

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

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

Definition 5. (Minimal DFSA/ST) *The size of a DFSA/ST ($|A|, |T|$) is defined as the number of its states. A DFSA A is called minimal if $|A| \leq |A'|$ for all DFSA A' such that $\mathcal{L}(A') = \mathcal{L}(A)$. Accordingly, an ST T is called minimal if $|T| \leq |T'|$ for any ST T' such that $f_{T'} = f_T$.*

3 Minimization of DFSAs

The algorithm of Daciuk et al. (2000) is iterative. In each iteration, given a minimal acyclic trim DFSA $A = (\Sigma, Q, q_0, \delta, F)$ and a word $w \in \Sigma^*$, the algorithm creates a DFSA $A' = (\Sigma, Q', q_0, \delta', F')$ such that A' is the minimal acceptor 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 (cf. e.g. Hopcroft, Motwani and Ullman (2001)) states that a trim DFSA is minimal iff it does not contain a pair q_1, q_2 of distinct but equivalent states:

$$\forall_{q_1,q_2 \in Q} : q_1 \neq q_2 \quad \Rightarrow \quad q_1 \not\equiv q_2 \tag{1}$$

Each iteration of the algorithm consists of two steps, which can be called *insertion* and *local minimization*.

3.1 Insertion

The insertion operation identifies the longest prefix $w_1 \ldots w_l$ of w in A and the corresponding path $q_0 \ldots q_l$. Some of the q_i may be *confluence states*, i.e. $\mathit{InDegree}(q_i) > 1$. In order to prevent overgeneration, the algorithm identifies the first confluence state q_k and clones the path $q_k \ldots q_l$. The cloned states $\hat{q}_k \ldots \hat{q}_l$ 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, $\hat{\delta}(q_{k-1}, w_k) := \hat{q}_k$.

After cloning, a chain of states $\hat{q}_{l+1} \ldots \hat{q}_t$, $\hat{q}_t \in \hat{F}$, consuming the remainder of w (i.e. $w_{l+1} \ldots w_t$) is appended to $\hat{q}_l$ (if it has been created) or to q_l. If $l = t$, the remainder is the empty string, in which case the algorithm ensures that $\hat{q}_l \in \hat{F}$.

Formally, this step creates an automaton $\hat{A} = (\Sigma, \hat{Q}, q_0, \hat{\delta}, \hat{F})$ such that[2]

$$\begin{aligned}
\hat{Q} &= Q \cup \{\hat{q}_k \ldots \hat{q}_t\} \\
\hat{F} &= F \cup \{\hat{q}_i : q_i \in F, k \leq i \leq t\} \cup \{\hat{\delta}^*(q_0, w)\} \\
\hat{\delta}(q, a) &= \begin{cases} \hat{q}_k & : q = q_{k-1}, a = w_k \\ \hat{q}_{i+1} & : q = \hat{q}_i, a = w_{i+1}, k \leq i \leq t \\ \delta(q, a) & : \text{ otherwise.} \end{cases}
\end{aligned}$$

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 $q_0 \ldots q_{k-1}$.

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

3.2 Local Minimization

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,
- 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 minimize the new DFSA, 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'. Otherwise, q is added to $Q_{\not\equiv}$. In this way, the algorithm gets rid of duplicates w.r.t. the 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 crucially on how fast it can check the equivalence of two states. The trick is to use the fact that $\delta^*(q, u) \in Q_{\not\equiv}$ for all $q \in Q_{\not\equiv}$ at any stage of the local minimization 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, each $q \in Q_{\not\equiv}$ and the corresponding set $Out(q)$ are put on a *register*, i.e. an associative container that maps sets of input-state pairs (uniquely identifying a right language) to the corresponding start states in $Q_{\not\equiv}$.

4 Application to Transducers

The problem for STs can be stated as follows: given a minimal ST T, we want to insert into T a string w associated with an emission o, creating a minimal ST for $f_T \cup \{(w, o)\}$. 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 recognized.

Unfortunately, both suggested solutions are problematic. They require the input and output symbols to be aligned 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 transducers that implement the above 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, such a transducer might be non-sequential (fi gure 1, left).

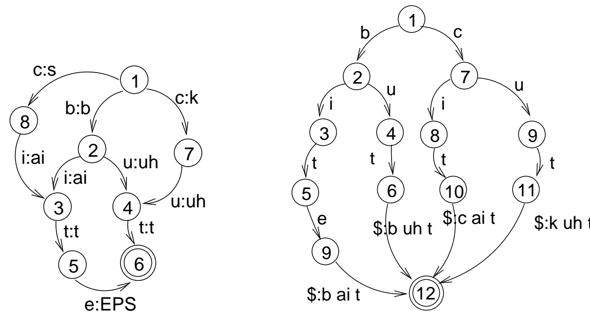

Figure 1: A “phonological” alignment of input and output symbols vs. final emissions.

The second suggestion made by Daciuk et al. (2000), i.e. the use of fi nal emissions, can be emulated by using the end-of-string symbol $. The result of applying FSA minimization, shown in fi gure 1 on the right, is an ST, but not minimal, since it has more states than the transducer shown in fi gure 2.[3]

Thus, the original algorithm requires two modifi cations. Firstly, the insertion step must not only identify the common prefi x $w \wedge T$, but also make sure the output o is actually emitted by $\hat{T}$ when it consumes $w \wedge T$. The main problem is that the output function σ of the original transducer T might be incompatible with o (i.e. $\sigma^*(q_0, w \wedge T) \neq o$). In such a case, the algorithm should only leave the common prefi xes of o and the original emissions on the path of $w \wedge T$, while pushing the incompatible suffi xes towards the fi nal states.

Secondly, the relation $\equiv$ defi ned for DFSAs is inadequate as a criterion for the equivalence of states in a sequential transducer. The following section defi nes the ST counterpart of $\equiv$ and shows how to use it in minimization.

5 Minimality Criteria for Sequential Transducers

The notion of minimality applies to STs according to the following proposition.

[3]Yet another option is to use a DFSA encoding the language $w^{(j)}o^{(j)}$. In such a case, minimization may factorize some of the common output suffi xes, especially when there are relatively few distinct $o^{(j)}$'s. However, if the mapping is injective (or almost injective), there is almost no size reduction (see section 7).

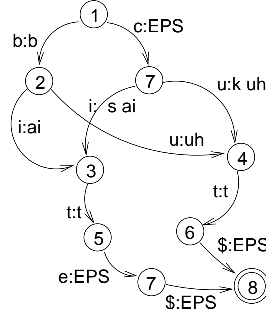

Figure 2: A minimal ST.

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

$$uR_fv \iff \vec{\mathcal{L}}(u) = \vec{\mathcal{L}}(v) \wedge \exists_{u',v' \in \Delta} \forall_{w \in \vec{\mathcal{L}}(u)} u'^{-1} f(uw) = v'^{-1} f(vw) \tag{2}$$

R_f is defined on $(\Sigma^*)^2$. In order to adapt the original algorithm, we must define an equivalence relation on Q, analogous to $\equiv$. Such a relation can indeed be defined for STs that emit output symbols as early as possible, e.g. the ST in figure 2 (the reader may check that no output symbol can be moved up towards the initial state without changing the mapping f_T). Such STs can be called *prefix-normalized*, as stated in the following definition.

Definition 6. *An ST $T = (\Sigma, \Delta, Q, i, \delta, \sigma, F)$ is prefix-normalized if*

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

The following proposition allows us to define a relation on Q analogous to $\equiv$.

Proposition 2. *If a trim transducer $T = (\Sigma, \Delta, Q, i, \delta, \sigma, F)$ that realizes a function $f : \Sigma^* \rightarrow \Delta^*$ is prefix-normalized, 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 that

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

$\Leftarrow$: follows immediately with $u' = \sigma^*(q_0, u)^{-1}, v' = \sigma^*(q_0, v)^{-1}$.

$\Rightarrow$: Let $q := \delta^*(q_0, u), q' := \delta^*(q_0, v)$. $uR_f v$ implies that $\vec{\mathcal{L}}(q) \overset{(2)}{=} \vec{\mathcal{L}}(q')$ 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 $f(uw) = \sigma(q_0, u) \cdot \sigma(q, w)$, this is equivalent to:

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

Furthermore, u' and v' must be prefixes of $\sigma^*(q_0, u)$ and $\sigma^*(q_0, v)$, respectively (otherwise, T would not be prefix-normalized), thus there exist u'', v'' such that $u' \cdot u'' = \sigma^*(q_0, u), v' \cdot v'' = \sigma^*(q_0, v)$ 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 $v'' = u'' \cdot z$. 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-normalized, 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 algorithm by Daciuk et al. (2000) shall produce a minimal ST if $\equiv$ is replaced by $\equiv_{\mathrm{ST}}$, and the transducer is prefix-normalized 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-normalized ST T, creating a prefix-normalized, “almost minimal” ST $\hat{T}$. In the second step, the states on the path of w in $\hat{T}$ are merged with equivalent states in T, resulting in a minimal, trim and prefix normalized ST for the rational mapping $f_{\hat{T}}$. The notion “almost minimal” is captured by the following definition.

Definition 7. (Minimal-except) *Let $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ be an ST, and let $u = u_1 \dots u_t$ be a string such that $q_j := \delta^*(q_0, u_1 \dots u_j)$ is defined for $j = 0 \dots t$. T is called* minimal except u *if*

$$\forall_{q, q' \in Q \setminus \{q_0 \dots q_t\}} : q \neq q' \quad \Rightarrow \quad q \not\equiv_{\mathrm{ST}} q' \tag{4}$$

$$\forall_{1 \leq j \leq t, q \in Q, a \in \Sigma} \quad : \quad \delta(q, a) = q_j \iff q = q_{j-1} \wedge a = u_j \tag{5}$$

The formalization of the two steps of the algorithm (insertion and local minimization) is treated separately in the following two subsections.

6.1 Insertion of (w, o) into T

Let $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ be a minimal, trim and prefix-normalized ST. Furthermore, let $w = w_1 \dots w_t$ be a string such that $w \notin \mathcal{L}(T)$, $w \wedge T = w_1 \dots w_l$.

In order for $f_T \cup \{(w,o)\}$ to be sequential, o must satisfy the following condition:

$$\forall_{v \in \mathcal{L}(T)} : \quad \begin{cases} v \leq_p w & \Rightarrow \quad \sigma^*(q_0, v) \leq_p o \\ w \leq_p v & \Rightarrow \quad o \leq_p \sigma^*(q_0, v) \end{cases} \tag{6}$$

Let k denote the index of the first *confluence state*: $k := min\{j \in \{1 \dots l\} : \mathit{InDegree}(q_j) > 1\}$. If there is no such state, we set $k := l + 1$.

The new transducer $\hat{T} = (\Sigma, \Delta, \hat{Q}, q_0, \hat{\delta}, \hat{\sigma}, \hat{F})$ is constructed as follows:

$$\hat{Q} \quad = \quad Q \cup \{\hat{q}_k \dots \hat{q}_t\} \text{ (new states – only if } k \leq t) \tag{7}$$

$$\hat{\delta}(q,a) \quad = \quad \begin{cases} \delta(q_j, a) & \text{if } q = \hat{q}_j, a \neq w_{j+1} \\ \hat{q}_{j+1} & \text{if } q = \hat{q}_j, a = w_{j+1} \\ \hat{q}_k & \text{if } q = q_{k-1}, a = w_k \\ \delta(q,a) & \text{if } q \in Q \text{ otherwise.} \end{cases} \tag{8}$$

$$\hat{\sigma}(q,a) \quad = \quad \begin{cases} \sigma(q,a) & \text{if } q \in Q \backslash \{q_0 \dots q_{k-1}\} \\ \hat{\sigma}^*(q_0, w_{[1..j]})^{-1} & \text{if } a \neq w_{j+1} \text{ and} \\ \quad \sigma^*(q_0, w_{[1..j]}a) & \quad q = \hat{\delta}^*(q_0, w_{[1..j]}), j \leq l \\ \hat{\sigma}^*(q_0, w_{[1..l]})^{-1} o & \text{if } a = w_{l+1}, \\ & \quad q = \hat{\delta}^*(q_0, w_{[1..l]}), l < |w| \\ \hat{\sigma}^*(q_0, w_{[1..j]})^{-1} & \text{if } a = w_{j+1} \text{ and} \\ \ (o \wedge \sigma^*(q_0, w_{[1..j+1]})) & \quad q = \hat{\delta}^*(q_0, w_{[1..j]}), j < l \\ \epsilon & \text{if } q \in \{\hat{q}_{l+1} \dots \hat{q}_t\} \end{cases} \tag{9}$$

$$\hat{F} \quad = \quad F \cup \{\hat{q}_j : q_j \in F\} \cup \{\hat{\delta}^*(q_0, w)\} \tag{10}$$

Informally, the idea behind the construction of $\hat{T}$ is to clone all states from the first confluence state q_k down to q_l, creating new states $\hat{q}_k \dots \hat{q}_l$. If $l < |w|$, a chain of new states $\hat{q}_{l+1} \dots \hat{q}_t$ is created for the suffix $w_{[l+1 \dots t]}$ and appended to $\hat{q}_l$ (if a confluence state exists) or to q_l. If $l = |w|$, we just make sure $\hat{\delta}^*(q_0, w) \in F$.

In order to make the emissions compatible with o, we may need to push some of the emissions associated with the prefix paths of $q_0 \dots q_{k-1} q_k \dots q_l$ in T towards the final states. This is done by the definition of the new output function $\hat{\sigma}$. The remainder of o not emitted on the path of the string $w_{[1..l]}$ is then emitted in the transition from $\hat{q}_l$ to $\hat{q}_{l+1}$ (or from q_l to $\hat{q}_{l+1}$ in the confluence-free case).

The construction of $\hat{T}$ implies the following properties of $\hat{\sigma}^*$.

$$\hat{\sigma}^*(q_0, u) = \begin{cases} o \wedge \sigma^*(q_0, u) : u \leq_p w \wedge T \\ \sigma^*(q_0, u) \quad\ : u \not\leq_p w \\ o \qquad\qquad\quad\ : w \wedge T <_p u \leq_p w \end{cases} \tag{11}$$

We start by proving a lemma.

Lemma 1. *If* $u \in \Sigma^*$, *then:*

$$\overrightarrow{\mathcal{L}}_{\hat{T}}(u) \quad = \quad \begin{cases} \overrightarrow{\mathcal{L}}_T(u) & : u \not\leq_p w \\ \overrightarrow{\mathcal{L}}_T(u) \cup \{u^{-1}w\} & : u \leq_p w \end{cases} \tag{12}$$

Proof. Since q_k is the first confluence state, $\delta(q,a) \in Q\backslash\{q_0 \dots q_{k-1}\}$ for any $(q,a) \in Q\backslash\{q_0 \dots q_{k-1}\} \times \Sigma$ such that $\delta(q,a)$ is defined. The definition of $\hat{\delta}$ ensures that $\hat{\delta}|_{Q\backslash\{q_0 \dots q_{k-1}\}\times\Sigma} = \delta|_{Q\backslash\{q_0 \dots q_{k-1}\}\times\Sigma}$. In effect, $\overrightarrow{\mathcal{L}}_{\hat{T}}(q) = \overrightarrow{\mathcal{L}}_T(q)$ for $q \in Q\backslash\{q_0 \dots q_{k-1}\}$, which proves the lemma for $u \not\leq_p w$.

If $u \leq_p w$, we prove the lemma by (backward) induction over $m := |u| = t \dots 0$. If $m = t$, then $u = w \notin \mathcal{L}(T)$, hence $\overrightarrow{\mathcal{L}}_T(u) = \emptyset$ and $\overrightarrow{\mathcal{L}}_{\hat{T}}(u) = \{u^{-1}w\}$.

In the induction step $m+1 \rightarrow m$ we assume that

$$\overrightarrow{\mathcal{L}}_{\hat{T}}(w_{[1\dots m+1]}) = \{w_{[m+2\dots t]}\} \cup \overrightarrow{\mathcal{L}}_T(w_{[1\dots m+1]}) \tag{13}$$

and use the already proved equality (12) for $u \not\leq_p w$ as well as identity (14):

$$\overrightarrow{\mathcal{L}}(v) = \bigcup_{a\in\Sigma} a\cdot \overrightarrow{\mathcal{L}}(va) \text{ for any } a \in \Sigma, v \in \Sigma^*. \tag{14}$$

$$\begin{aligned}
\overrightarrow{\mathcal{L}}_{\hat{T}}(u) &= \overrightarrow{\mathcal{L}}_{\hat{T}}(w_{[1\dots m]}) = \bigcup_{a\in\Sigma} a\cdot \overrightarrow{\mathcal{L}}_{\hat{T}}(ua) \\
&\overset{(14)}{=} w_{m+1}\cdot \overrightarrow{\mathcal{L}}_{\hat{T}}(w_{[1\dots m+1]}) \cup \bigcup_{a\neq w_{m+1}} a\cdot \overrightarrow{\mathcal{L}}_{\hat{T}}(ua) \\
&\overset{(13,12)}{=} w_{m+1}\cdot(\{w_{[m+2\dots t]}\} \cup \overrightarrow{\mathcal{L}}_T(w_{[1\dots m+1]})) \\
&\quad \cup \bigcup_{a\neq w_{m+1}} a\,\overrightarrow{\mathcal{L}}_T(ua) \\
&= \{w_{[m+1\dots t]}\} \cup \\
&\quad w_{m+1}\,\overrightarrow{\mathcal{L}}_T(w_{[1\dots m+1]}) \cup \bigcup_{a\neq w_{m+1}} a\,\overrightarrow{\mathcal{L}}_T(ua) \\
&= \{w_{[m+1\dots t]}\} \cup \bigcup_{a\in\Sigma} a\,\overrightarrow{\mathcal{L}}_T(ua) \overset{(14)}{=} \{u^{-1}w\} \cup \overrightarrow{\mathcal{L}}_T(u)
\end{aligned}$$

□

The following four propositions establish a relation between $\hat{T}$ and T.

Proposition 3. $f_{\hat{T}} = f_T \cup \{(w,o)\}$.

Proof. Since $\mathcal{L} = \overrightarrow{\mathcal{L}}(\epsilon)$, Lemma 1 implies $\mathcal{L}(\hat{T}) = \mathcal{L}(T) \cup \{w\}$. If $w \wedge T <_p w$, then $f_{\hat{T}}(w) = \hat{\sigma}^*(q_0,w) \overset{(11)}{=} o$. If $w \wedge T = w$, then $\sigma^*(q_0,w)$ is defined and it follows $f_{\hat{T}}(w) = \hat{\sigma}^*(q_0,w) \overset{(11)}{=} \sigma^*(q_0,w) \wedge o \overset{(6,T\text{pref.}-\text{norm.})}{=} o$.

It remains to be shown that, for $u \neq w, u \in \mathcal{L}(T)$, $f_{\hat{T}}(u) = f_T(u)$.

If $u \not\leq_p w$, then $f_{\hat{T}}(u) = \hat{\sigma}^*(q_0,u) \overset{(11)}{=} \sigma^*(q_0,u) = f_T(u)$.

If $u <_p w$, then $f_{\hat{T}}(u) = \hat{\sigma}^*(q_0,u) \overset{(11)}{=} o \wedge \sigma^*(q_0,u) \overset{(6)}{=} \sigma^*(q_0,u) = f_T(u)$.

□

Proposition 4. $\hat{T}$ *is prefix-normalized.*

Proof. Let $u \in \Sigma^*$ be a string such that $\hat{\sigma}^*(q_0, u)$ is defined. We consider three cases: $w \wedge T <_p u \leq_p w$, $u \leq_p w \wedge T$ and $u \not\leq_p w$.

If $w \wedge T <_p u \leq_p w$, then $\hat{\sigma}^*(q_0, u) \overset{(11)}{=} o = \bigwedge_{v \in \vec{\mathcal{L}}_{\hat{T}}(u)} f(uv)$ because there is no other word $w' \in \mathcal{L}(\hat{T})$ such that $w \wedge T <_p w'$.

If $u \leq_p w \wedge T$, then:

$$\hat{\sigma}^*(q_0, u) \overset{(11)}{=} \overbrace{o}^{f(w)} \wedge \overbrace{\sigma^*(q_0, u)}^{\bigwedge_{v \in \vec{\mathcal{L}}_T(u)} f_T(uv)} \overset{(L1,P3)}{=} \bigwedge_{v \in \vec{\mathcal{L}}_{\hat{T}}(u)} f_{\hat{T}}(uv).$$

Finally, if $u \not\leq_p w$, Lemma 1 states that $\mathcal{L}_{\hat{T}}(u) = \mathcal{L}_T(u)$, while (11) implies that:

$$\forall_{v : \hat{\sigma}^*(q_0, uv) \text{ is defined}} : \hat{\sigma}^*(q_0, uv) = \sigma^*(q_0, uv) \tag{15}$$

Thus, $\hat{\sigma}^*(q_0, u) \overset{(15), v=\epsilon}{=} \sigma^*(q_0, u) \overset{T \text{ pref.-norm.}}{=} \bigwedge_{v \in \vec{\mathcal{L}}_T} f_T(uv) \overset{(15)}{=} \bigwedge_{v \in \vec{\mathcal{L}}_{\hat{T}}} f_{\hat{T}}(uv)$. □

Proposition 5. $\hat{T}$ *is trim.*

Proof. Each of the states $q_0 \ldots q_{k-1}, \hat{q}_k \ldots \hat{q}_t$ is obviously on a successful path in $\hat{T}$. If $q \in Q \backslash \{q_0 \ldots q_{k-1}\}$, then q is a state of the trim ST T, and as such it belongs to a successful path $\pi = r_0 \ldots r_m$ in T accepting a string u. Let $j := |w \wedge u|$. This implies $r_i = \hat{\delta}^*(q_0, w_{[1..i]})$ for $i = 0 \ldots j$.

If $j < k$, then the definition of $\hat{\delta}$ makes sure that $\hat{\delta}(r_i, u_{i+1}) = \delta(r_i, u_{i+1})$ for $i = 0 \ldots m-1$, and hence π is also a successful path in $\hat{T}$.

If $k \geq j$, the path for u in $\hat{T}$ is $\pi' = q_0 \ldots q_{k-1}, \hat{q}_k \ldots \hat{q}_j r_{j+1} \ldots r_n$. If $q = r_i$ for an $i > j$, q belongs to π'. Thus, the only problem is $q = r_i$ such that $k \leq i \leq j$.

Since q_k is a confluence state, it must be reachable from q_0 in T by another string $v = v_1 \ldots v_n$. Let $s_0 \ldots s_n$ denote the corresponding path; obviously $s_n = r_k = q_k$ and either $s_{n-1} \neq q_{k-1}$ or $v_n \neq w_k$. Then $v \cdot u_{[k+1\ldots m]} \in \mathcal{L}(T)$, and $s_0 \ldots s_{n-1} r_k \ldots r_m$ is the path of $v \cdot u_{[k+1\ldots m]}$ in both T and $\hat{T}$, i.e. q is on a successful path in $\hat{T}$. □

Proposition 6. $\hat{T}$ *is minimal except* w.

Proof. We check the conditions of definition 7. Recall that the path in $\hat{T}$ corresponding to w is $q_0 \ldots q_{k-1} \hat{q}_k \ldots \hat{q}_t$. Condition (4) follows from Lemma 1 and from property (11). Condition (5) follows, for the states $q_0 \ldots q_{k-1}$, from q_k being the first confluence state on the path of w. For the states $\hat{q}_k \ldots \hat{q}_t$, it follows from the definition of the new transition function $\hat{\delta}$. □

6.2 Local Minimization

The (incremental) minimization step is taken care of by the following propositions.

Proposition 7. *Let $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ be an ST minimal except for $w = w_1 \dots w_t$. Let there be a $\hat{q} \in Q \backslash \{q_0 \dots q_t\}$ such that $q_t \equiv_{\mathrm{ST}} \hat{q}$. Let $\hat{T} = (\Sigma, \Delta, \hat{Q}, q_0, \hat{\delta}, \sigma, F \backslash \{q_t\})$ be an ST such that $\hat{Q} = Q \backslash \{q_t\}$ and*

$$\hat{\delta}(q, a) \quad = \quad \begin{cases} \hat{q} & : \delta(q, a) = q_t \\ \delta(q, a) & : \text{ otherwise.} \end{cases} \tag{16}$$

Then $f_{\hat{T}} = f_T$ and $\hat{T}$ is prefix-normalized and minimal except for $w_{[1 \dots t-1]}$.

Proof. Let $u = u_1 \dots u_l$ be an arbitrary string. If $\delta^*(q_0, u_{[1 \dots i]}) \neq q_t$ for any $i \leq l$ such that $\delta^*(q_0, u_{[1 \dots i]})$ is defined, then $u \in \mathcal{L}(T) \iff u \in \mathcal{L}(\hat{T})$ and $f_{\hat{T}}(u) = f_T(u)$ if $u \in \mathcal{L}(T)$.

Otherwise, there exists exactly one $k \leq l$ such that $\delta^*(q_0, u_{[1 \dots k]}) = q_t$. The definition of $\hat{T}$ implies $\hat{\sigma}^*(q_0, u_{[1 \dots k]}) = \sigma^*(q_0, u_{[1 \dots k]})$ and $\hat{\delta}^*(q_0, u_{[1 \dots k]}) = \hat{q}$. Since $\hat{q} \equiv_{\mathrm{ST}} q_t$, this means $u \in \mathcal{L}(\hat{T}) \iff u \in \mathcal{L}(T)$ and $f_{\hat{T}}(u) = f_T(u)$.

$\hat{T}$ is prefix-normalized because the construction of $\hat{T}$ preserves the right languages of all states q and strings u.

In order to show that $\hat{T}$ is minimal except for word $w = w_1 \dots w_{t-1}$, we check the conditions of definition 7. Condition (4) is satisfied because $\hat{Q} \backslash \{q_0 \dots q_{t-1}\} = Q \backslash \{q_0 \dots q_t\}$, the emission function σ remains unchanged, and the only state in T whose outgoing transitions have been changed, i.e. q_{t-1}, is not reachable from $\hat{Q} \backslash \{q_0 \dots q_{t-1}\}$. Condition (5) is satisfied trivially. □

Proposition 8. *Let $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ be an ST that is minimal except for word $w = w_1 \dots w_t$. if $q_t \not\equiv_{\mathrm{ST}} q$ for all $q \in Q \backslash \{q_0 \dots q_t\}$, then T is also minimal except for $w = w_1 \dots w_{t-1}$.*

Proof. We check the conditions of definition 7: all are obviously satisfied. □

Proposition 9. *If a trim ST $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ is minimal except for the empty word ϵ, then T is minimal.*

Proof. Condition (4) implies $q \neq q' \Rightarrow q \not\equiv_{\mathrm{ST}} q'$ for all $q, q' \in Q \backslash \{q_0\}$. Furthermore, T is trim, so $\delta^*(q_0, u) = q$ for any $q \in Q \backslash \{q_0\}$ and some $u \neq \epsilon$. Since T is acyclic, $\overrightarrow{\mathcal{L}}(q) \neq \overrightarrow{\mathcal{L}}(q_0)$, hence $q \not\equiv_{\mathrm{ST}} q_0$. Thus, condition (4) holds, too. □

6.3 Putting It All Together

Combined together, the above propositions yield the following result.

Proposition 10. *Let T be a trim, minimal, prefix-normalized ST. Let $(w, o) \in \Sigma^* \times \Delta^*$ be a pair of strings satisfying condition (6). Then the ST T' created by (1) constructing transducer $\hat{T}$ according to equations (7)-(9), and (2) successively applying the operations described by the propositions 7 and 8, is a trim, minimal and prefix-normalized ST implementing $f_T \cup \{(w, o)\}$.*

Proof. According to propositions 3-6, $\hat{T}$ is trim, prefix-normalized and minimal except for string $w = w_1 \ldots w_t$, and implements the function $f_T \cup \{(w, o)\}$. The recursive application of the transformations described in propositions 7 and 8 produces a sequence of trim and prefix-normalized transducers $\hat{T}_0 \ldots \hat{T}_t$ such that $\hat{T}_0 = \hat{T}$, $f_{\hat{T}_{i+1}} = f_{\hat{T}_i}$, each $\hat{T}_i$ being minimal except for the string $w_1 \ldots w_{t-i}$. In particular, $T' = \hat{T}_t$ implements the mapping $f_T \cup \{(w, o)\}$, and is trim, prefix-normalized and minimal except ϵ, i.e. according to proposition 9, minimal. □

6.4 Pseudocode

The pseudocode of the algorithm is shown on page 105. In each iteration of the main loop, the algorithm reads a pair $(Word, Output)$ and calls the procedures INSERT() and REMOVE_DUPLICATES(), responsible respectively for the insertion and the local minimization 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. Then INSERT_SUFFIX() is called, creating a chain of states for the remainder of $Word$ and emitting the remainder of $Output$ in the first transition (the pseudocode is omitted for space reasons).

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 $(o \wedge \sigma^*(q_0, w_{[1 \ldots i-1]}))^{-1} \sigma^*(q_0, w_{[1 \ldots i]})$, i.e. the remainder of $\sigma^*(q_0, w_{[1 \ldots i-1]})$ after subtracting the longest common prefix with o. This prefix is prepended to the output labels of all transitions starting in $State$.[4]

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 $w \wedge T$).

6.5 Extension to Subsequential Transducers

Definition 8. (Subsequential Transducers) *A subsequential transducer (SST) is an 8-tuple* $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F, \rho)$ *such that* $(\Sigma, \Delta, Q, q_0, \delta, \sigma, F)$ *is an ST and* $\rho : F \rightarrow \Delta^*$ *is a function that associates each final state q with a* final output *emitted when the SST stops in q. The translation of a string* $w \in \mathcal{L}(T)$ *is defined as* $f_T(w) = \sigma^*(q_0, w) \cdot \rho(\delta^*(q_0, w))$.

An SST $T = (\Sigma, \Delta, Q, q_0, \delta, \sigma, F, \rho)$ can be emulated by an ST $\hat{T} = (\Sigma \cup \{\$\}, \Delta, Q \cup \{\hat{q}\}, q_0, \hat{\delta}, \hat{\sigma}, \{\hat{q}\})$, where $\hat{q}$ is a new final state and $\$ \notin \Sigma$ a designated end-of-string symbol. The new transition and output functions extend the original ones: $\hat{\delta} = \delta \cup \{(q, \$, \hat{q}) : q \in F\}$, and $\hat{\sigma} = \delta \cup \{(q, \$, \rho(q)) : q \in F\}$.

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

Algorithm 6.1: CONSTRUCTMINST()

$Register \leftarrow \emptyset$
while there is a word-output pair
  $(Word, Output) \leftarrow$ next pair
  INSERT$(Word, Output)$
  REMOVE_DUPLICATES$(Word)$

procedure INSERT$(Word, Output)$
 $State \leftarrow q_0; FoundConfluence \leftarrow false$
 for $i \leftarrow 1$ **to** $size(Word)$
   $Register \leftarrow Register \backslash \{State\}$
   $Symbol \leftarrow Word[i]$
   $Child \leftarrow \delta(State, Symbol)$
   if $Child = \emptyset$ **break**
   if $InDegree(Child) > 1$
     $FoundConfluence \leftarrow true$
   if $FoundConfluence$
     $\delta(State, Symbol) \leftarrow$ CLONE$(Child)$
   $OutputPrefix \leftarrow Output \wedge \sigma(State, Symbol)$
   $OutputSuffix \leftarrow OutputPrefix^{-1}\sigma(State, Symbol)$
   $Output \leftarrow OutputPrefix^{-1} Output$
   $\sigma(State, Symbol) \leftarrow OutputPrefix$
   $State \leftarrow \delta(State, Symbol)$
   PUSH_OUTPUTS$(State, OutputSuffix)$
 rof
 INSERT_SUFFIX$(Word_{[i \ldots size(Word)]}, Output)$

procedure REMOVE_DUPLICATES$(Word)$
 for $i \leftarrow size(Word) - 1$ **downto** 1
 $State \leftarrow \delta^*(q_0, Word[1 \ldots i])$
 $Symbol \leftarrow Word[i+1]$
 $Child \leftarrow \delta(State, Symbol)$
 if $\exists q \in Register, q \equiv_{\mathrm{ST}} Child$
   $\delta(State, Symbol) \leftarrow q$
  else $Register \leftarrow Register \cup \{Child\}$
 rof

procedure PUSH_OUTPUTS$(State, Residual)$
 for each $a \in \Sigma$
 if $\delta(State, a)$ is defi ned
   $\sigma(State, a) \leftarrow Residual \cdot \sigma(State, a)$

The algorithm produces the minimal SST if $ is appended to each word, and transitions consuming $ are interpreted as final emissions. Interestingly, there is no need to check condition (6) as such a mapping is always sequential.

6.6 Complexity and Optimization

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 DFSA algorithm, the ST generalization 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 due to the use of more complex data types (transition outputs).

The algorithm can be optimized by reducing the number of times states are registered/de-registered 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 $q_1 \dots q_s$, $s < k$, 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 $q_{i-1} \dots q_1$.

This optimization idea is used in the DFSA algorithm. As for STs, the speed-up achieved is rather modest because the procedure PUSH_OUTPUTS() typically changes most of the transitions on the path of w.

7 Evaluation

The new algorithm has been employed to construct pronunciation lexica in the rVoice text-to-speech system.[5] 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. The algorithm makes it possible to build very large dictionaries without affecting the efficiency and flexibility of the system: the resulting representations are very compact, words can be looked up deterministically, and user-defined entries can be merged in at any time in any order. This last feature in particularly important as rVoice users can change the behavior of the system by inserting their own entries into the dictionary at runtime.

The algorithm has been evaluated by constructing a minimal ST for a pronunciation dictionary comprising the 50,000 most frequent British surnames. The size

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

and construction time were compared to the equivalent parameters for the sorted-data ST algorithm (Mihov and Maurel 2001) and the unsorted-data DFSA 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 DFSA, the phonetic transcriptions were encoded as final emissions (i.e., the DFSA encoded the language $\{w^{(1)}o^{(1)}, \ldots, w^{(m)}o^{(m)}\}$, each phonetic symbol serving as an additional symbol of the input alphabet). In the ST encoding, the symbol $ was appended to each word in order to ensure sequentiability.[6] The results are shown in table 1.

	ST-unsorted	ST-sorted	DFSA
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 DFSA) applied to a pronunciation lexicon on a Pentium 4 1.7 GHz processor.

The comparison demonstrates that STs are superior to DFSAs as an encoding method for lexica annotated with rich representations. DFSA minimization 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 DFSA encoding the $w^{(i)}$'s.[7] Since the DFSA is much larger, construction takes longer than in the ST case although the DFSA 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 leaves the new unsorted-data algorithm as the preferable option in a range of applications.

References

Daciuk, J., Mihov, S., Watson, B. and Watson, R.(2000), Incremental construction of minimal acyclic finite state automata, *Computational Linguistics* **26**(1), 3–16.

Hopcroft, J. E., Motwani, R. and Ullman, J. D.(2001), *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley.

Mihov, S. and Maurel, D.(2001), Direct construction of minimal acyclic subsequential transducers, *in* S. Yu (ed.), *Implementation and Application of Automata*, Vol. 2088 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 217–229.

Mohri, M.(1994), Minimization of sequential transducers, *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching* , pp. 156–163.

[6]See section 6.5. There were 6,096 such transitions in the minimal ST.

[7]In contrast, Mihov and Maurel (2001) report only a small difference between the minimal DFSA (47K states) and the minimal ST (43K states) for a grammatical dictionary of Bulgarian. Clearly, this is due to the fact that the number of grammatical classes is much smaller than the number of word forms.