Efficient Dictionary-Based Text Rewriting using Subsequential Transducers*

Stoyan Mihov
Linguistic Modelling Department
IPP – Bulgarian Academy of Sciences
stoyan@lml.bas.bg

Klaus U. Schulz
CIS
University of Munich
schulz@cis.uni-muenchen.de

Abstract

Problems in the area of text and document processing can often be described as *text rewriting tasks*: given an input text, produce a new text by applying some fixed set of rewriting rules. In its simplest form, a rewriting rule is given by a pair of strings, representing a source string (the "original") and its substitute. By a rewrite dictionary, we mean a finite list of such pairs; dictionary-based text rewriting means to replace in an input text occurrences of originals by their substitutes. We present an efficient method for constructing, given a rewrite dictionary D, a subsequential transducer $\mathcal{T}$ that accepts a text t as input and outputs the intended rewriting result under the so-called "leftmost longest match" replacement with skips, t'. The time needed to compute the transducer is linear in the size of the input dictionary. Given the transducer, any text t of length $|t|$ is rewritten in time $O(|t| + |t'|)$, where t' denotes the resulting output text. Hence the resulting rewriting mechanism is very efficient. As a second advantage, using standard tools, the transducer can be directly composed with other transducers to efficiently solve more complex rewriting tasks in a single processing step.

1 Introduction

An impressive number of problems in the areas of natural language processing and document analysis and transformation can be viewed as text rewriting tasks [RS97]: using a collection of background rules and ressources, a given input text has to be rewritten to another form. Examples include phonetic conversion [Lap97], stemming and lemmatization [Por80, Kro93, MM95] and related forms of text normalization, tagging [Bri92, Kup92], shallow parsing and information extraction [AHB$^+$93, Abn97], semantic annotation and automated hyperlinking [CHBG01], automated text correction [Kuk92], translation [Bro00] and others.

A simple, yet very useful variant of text rewriting is based on rewrite dictionaries that list strings that have to be replaced ("originals") plus their substitutes. Rewrite dictionaries in this sense do not specify additional contextual conditions. Dictionary-based text rewriting simply means to replace occurrences of originals in an input text by their corresponding substitutes. Rewrite dictionaries in memory-based translation, e.g., list words, phrases and sentences of a source language together with translations into a target language. In automated hyperlinking, rewrite dictionaries translate natural language expressions into ontological concepts, links and pointers. For shallow parsing and information extraction, rewrite dictionaries may be used that attach syntactic categories and semantic labels to words and

*This work was supported by VolkswagenStiftung and DFG.

phrases. Other rewriting dictionaries may help to substitute correct words for erroneous variants, or to replace synonyms by canonical forms. In cascaded approaches to natural language processing [Abn97, VN00], rewriting dictionaries may be used for transduction tasks that arise on one level.

¿From an algorithmic point of view, dictionary based text rewriting is complicated by the fact that occurrences of originals in a text may overlap. A strategy has to be given for resolving this kind of conflicts. Most actual text rewriting systems use the so-called "leftmost largest match" replacement strategy with skips. Basically this means that there is a preference for replacing originals that start first in the natural reading order. Among distinct originals with the same starting point, the longest one is replaced.[1] No overlap between replaced parts is possible.

It is straightforward to describe a *naive algorithm* for text rewriting with a given rewrite dictionary D under the leftmost-largest strategy with skips: for each possible offset position k in the text t, we read the suffix s of t that starts at position k. As soon as we find a prefix p of s that does not represent a prefix of any original in D, or if we reach the end of t, we stop. If at this point we have already found an occurrence of an original string starting at position k, then we replace the longest of these originals by its substitute, using D. We then go back to the position immediately after the occurrence, k', and continue the search from k'. In the other case, if no occurrence starting at k has been observed, the first letter of s is passed to the output and we continue the search at position $k' = k + 1$ of t.

Due to steps where we return to an earlier text position k', the worst-case time complexity of the naive algorithm is $O(m \cdot |t|)$, where m is the maximal length of an original string in the rewrite dictionary and $|t|$ is the length of the text t. When using the naive algorithm, such backward steps cannot be avoided. As a simple example, assume that the dictionary contains the pairs

$$\begin{array}{rcl} (\textit{finite state automaton} & \mapsto & FSA) \\ (\textit{state transition diagram} & \mapsto & STD). \end{array}$$

When reading the text "*a finite state transition diagram ...*" the algorithm will be forced to return to an earlier offset point when it finds that "*finite state t...*" cannot be extended to "*finite state automaton*". Furthermore, the naive algorithm needs some kind of special memory device that keeps track of the longest original string starting at offset position k found so far. This is due to the effect that originals of the rewrite dictionary may represent proper prefixes of other entries, as in the following example.

$$\begin{array}{rcl} (\textit{New York} & \mapsto & \textit{http.//www.nyc.gov/, http.//www.state.ny.us/}) \\ (\textit{New York City} & \mapsto & \textit{http.//www.nyc.gov/}) \\ (\textit{New York City Opera} & \mapsto & \textit{http.//www.nycopera.com/}) \end{array}$$

In this paper, we present an algorithm for dictionary-based rewriting that rewrites a given text t of length $|t|$ in time $O(|t| + |t'|)$, using the leftmost largest match strategy with skips. Here t' denotes the resulting output text. The new algorithm does not use backward steps, and no memory device is needed: in our approach we compute, given the rewrite dictionary D, a subsequential finite state transducer $\mathcal{T}$ that always produces the desired transformation result when applied to an input text t.[2]

[1] In many applications, long originals carry more specific information, which explains the intuitive appeal of this decision.

[2] The construction of a subsequential transducer for dictionary based *text rewriting* considered in this paper should not be confused with the construction of a subsequential transducer for the dictionary D itself. Transducers of the latter type just rewrite originals into substitutes but cannot be used for text rewriting (cf., e.g., [MM01]).

The construction of $\mathcal{T}$ represents a preprocessing step that is performed once. The number of states and transitions of $\mathcal{T}$ is linear in the size $|D|$ of D. If implemented in a naive way, output functions may have size cubic in $|D|$. Special string representation techniques help to reduce this to linear space without affecting the efficiency of the text rewriting. In this way, the time and space needed to compute $\mathcal{T}$ is linear in $|D|$. Our method for dictionary-based text rewriting can be described as a natural generalization of the well-known Aho-Corasick algorithm for multiple pattern search in texts [AC75].

The advantage that is gained from the new method is twofold. Our algorithm for dictionary-based text rewriting is more efficient than the naive algorithm. Furthermore, using standard tools, the abovementioned transducer $\mathcal{T}$ can be composed with other transducers, which helps to efficiently solve more complex text rewriting tasks in a single transduction step. In this way, a schematic translation of large vocabularies — which arises as a subtask in many of the abovementioned applications — can be naturally integrated into more ambitious text transformation systems. The method has been implemented and successfully tested with rewrite dictionaries with over 100,000 entries.

The paper is structured as follows. Section 2 provides some formal background. In Section 3 we illustrate the construction using a simple example. The algorithm for computing the subsequential transducer for text rewriting based on a given rewrite dictionary D is formally described in Section 4. In Section 5 we prove that the algorithm is correct. Section 6 presents a complexity analysis and implementation details. In Section 7 we evaluate the method. Section 8 comments on related work, focussing on finite state approaches. The Conclusion gives some ideas for further research.

2 Formal Background

We assume that the reader is familiar with the basic notions of formal language theory as described, e.g., by Hopcroft and Ullman [HU79] or Kozen [Koz97]. More background on transducers and their use can be found in [RS97]. In the remainder of the paper, the symbols Σ and Δ always denote finite alphabets. As usual, Σ^* (resp. Σ^+) denotes the set of all (resp. all non-empty) strings over Σ, and similarly for Δ. The symbol ϵ denotes the empty word. If w is any string, then $|w|$ denotes the length of w. The concatenation of the strings u and w is written $u \cdot w$ or uw. If f is any function, then $Dom(f)$ denotes the domain of f.

Dictionary-based text rewriting with leftmost-largest-skip strategy

Definition 2.1 A *rewriting dictionary* is a partial function $D : \Sigma^+ \rightarrow \Sigma^*$ with finite domain. By an *entry* of D, we mean an element w of $Dom(D)$. If w is an entry of D, the string $D(w)$ is called the *substitute* for w. The set $CoDom(D) := \{D(w) \mid w \in Dom(D)\}$ is called the *codomain* of D.

In this paper, the terms "string" and "text" are used synonymously.

Definition 2.2 By an *occurrence* of a dictionary entry w in the text $t \in \Sigma^*$, we mean a triple $\langle u, w, v\rangle \in (\Sigma^*)^3$ such that $t = uwv$. The strings u and v are respectively called the *pre-text* and the *post-text* of the occurrence.

Definition 2.3 The occurrence $\langle u, w, v\rangle \in (\Sigma^*)^3$ is called *leftmost* iff its pre-text is minimal with respect to all pre-texts of occurrences of dictionary entries in t. It is called *leftmost largest* if it is leftmost and if w is of maximal length among all leftmost occurrences of dictionary entries in t. If there is any occurrence of a dictionary entry in t, then there exists an unique leftmost largest occurrence, denoted $lml_D(t)$.

Example 2.4 Consider the rewrite dictionary D given by the mapping

$$\begin{array}{rcl} (a & \mapsto & 1) \\ (ab & \mapsto & 2) \\ (abcc & \mapsto & 3) \\ (babc & \mapsto & 4) \\ (c & \mapsto & 5) \end{array}$$

and the text $t = abcbbbabccb$. Among nine distinct occurrences of dictionary entries in t, there are two occurrences of ab, which respectively have the form $\langle \epsilon, ab, cbbbabccb \rangle$ and $\langle abcbbb, ab, ccb \rangle$. We have $\langle \epsilon, ab, cbbbabccb \rangle = lml_D(t)$.

Definition 2.5 The set of all *leftmost-largest-skip occurrences* of dictionary entries in t, denoted $Lmls_D(t)$ is recursively defined in the following way:

$$Lmls_D(t) := \begin{cases} \emptyset \text{ iff there is no occurrence of an entry of } D \text{ in } t; \\ \{\langle u, w, t' \rangle\} \ \cup \ \{\langle uwu', w', v' \rangle \mid \langle u', w', v' \rangle \in Lmls_D(t')\} \\ \text{iff } lml_D(t) = \langle u, w, t' \rangle. \end{cases}$$

Remark 2.6 It is simple to see that two distinct occurrences $\langle u, w, v \rangle, \langle u', w', v' \rangle \in Lmls_D(t)$ are always disjoint in the sense that either $|u'| \geq |uw|$ or $|u| \geq |u'w'|$. Hence there exists an unique representation of t in the form $t = u_1 \cdot w_1 \cdot \ldots \cdot u_k \cdot w_k \cdot u_{k+1}$ where w_i are entries of D $(1 \leq i \leq k)$ and $u_j \in \Sigma^*$ $(1 \leq j \leq k+1)$ such that

$$\{\langle u_1 w_1 u_2 w_2 \ldots u_i, w_i, u_{i+1} w_{i+1} \ldots u_k w_k u_{k+1} \rangle \mid 1 \leq i \leq k\} = Lmls_D(t).$$

Example 2.7 Let D denote the dictionary from Example 2.4, let $t = abcbbbabccb$. Then $Lmls_D(t)$ is the set with the following tuples:

$$\begin{array}{r} \langle \epsilon, ab, cbbbabccb \rangle \\ \langle ab, c, bbbabccb \rangle \\ \langle abcbb, babc, cb \rangle \\ \langle abcbbbabc, c, b \rangle \end{array}$$

The *canonical lmls-extension* of D is the function $D^{lmls} : \Sigma^* \to \Sigma^*$ recursively defined in the following way:

$$D^{lmls}(t) := \begin{cases} t \text{ if there is no occurrence of an entry of } D \text{ in } t; \\ u \cdot D(w) \cdot D^{lmls}(t') \text{ otherwise, if } lml_D(t) = \langle u, w, t' \rangle. \end{cases}$$

Example 2.8 Let D denote the dictionary given in Example 2.4, let $t = abcbbbabccb$. Then $D^{lmls}(t) = \epsilon \cdot 2 \cdot \epsilon \cdot 5 \cdot bb \cdot 4 \cdot \epsilon \cdot 5 \cdot b = 25bb45b$.

The following corollaries follow immediately.

Corollary 2.9 *Let $\langle u, w, v \rangle \in Lmls_D(t)$ where $t = u \cdot w \cdot v$. Then $D^{lmls}(t) = D^{lmls}(u \cdot w) \cdot D^{lmls}(v)$.*

Corollary 2.10 *Assume that no prefix of $t = a_1 \cdot t'$ is in the domain of D. Then $D^{lmls}(t) = a_1 \cdot D^{lmls}(t')$.*

Subsequential transducers

Definition 2.11 A *subsequential transducer* is a tuple $\mathcal{T} = \langle \Sigma, \Delta, Q, q_0, F, \delta, \lambda, \Psi \rangle$, where:

- Σ is a finite *input alphabet*;
- Δ is a finite *output alphabet*;
- Q is a finite set of *states*;
- $q_0 \in Q$ is the *start state*;
- $F \subseteq Q$ is the set of *final states*;
- $\delta : Q \times \Sigma \to Q$ is a partial function called the *transition function*;
- $\lambda : Q \times \Sigma \to \Delta^*$ is a partial function called the *transition output function*; we have $Dom(\lambda) = Dom(\delta)$;
- $\Psi : F \to \Delta^*$ is the *state output function*.

The *extended transition function* $\delta^* : Q \times \Sigma^* \to Q$ is defined in the usual way:

$$\begin{array}{rl} \forall q \in Q & \delta^*(q, \varepsilon) := q \\ \forall q \in Q\ \forall t \in \Sigma^*\ \forall \sigma \in \Sigma & \delta^*(q, t\sigma) := \delta(\delta^*(q, t), \sigma). \end{array}$$

Similarly we define the *extended transition output function* $\lambda^* : Q \times \Sigma^* \to \Delta^*$ in the following way:

$$\begin{array}{rl} \forall q \in Q & \lambda^*(q, \varepsilon) := \varepsilon \\ \forall q \in Q\ \forall t \in \Sigma^*\ \forall \sigma \in \Sigma & \lambda^*(q, t\sigma) = \lambda^*(q, t) \cdot \lambda(\delta^*(q, t), \sigma). \end{array}$$

The set $L(\mathcal{T}) = \{t \in \Sigma^* \mid \delta^*(q_0, t) \in F\}$ is called the *input language* of the transducer $\mathcal{T}$. The subsequential transducer maps each word of the input language to an output word. The *output function* $O_{\mathcal{T}} : L(\mathcal{T}) \to \Delta^*$ of the transducer is defined as follows:

$$\forall t \in L(\mathcal{T}) \qquad O_{\mathcal{T}}(t) = \lambda^*(q_0, t) \cdot \Psi(\delta^*(q_0, t)).$$

Definition 2.12 A subsequential transducer $\mathcal{T}$ is said to *represent* the function $f : \Sigma^* \to \Delta^*$ iff $O_{\mathcal{T}} = f$.

3 Informal description of the algorithm

Before we turn to the formal details of the algorithm, we use the dictionary

$$\begin{array}{rcl} (a & \mapsto & 1) \\ (ab & \mapsto & 2) \\ (abcc & \mapsto & 3) \\ (babc & \mapsto & 4) \\ (c & \mapsto & 5) \end{array}$$

introduced in Example 2.4 to illustrate the main idea. The construction of the transducer for text-rewriting based on the dictionary D proceeds in two steps. The first step is straightforward.

First Step. We construct a tree-formed transducer that represents (in the sense of Definition 2.12) the dictionary. To this end, we compute the trie for the domain

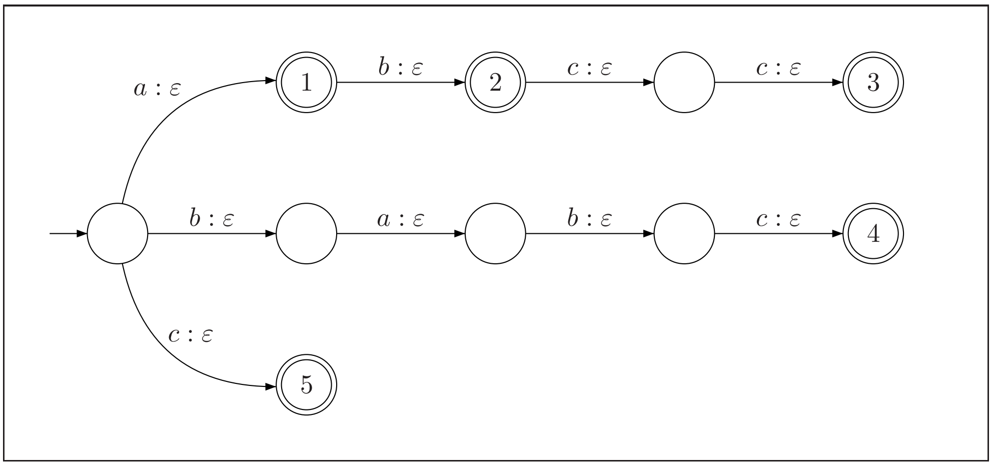

Figure 1: Subsequential transducer trie for the rewrite dictionary D introduced in Example 2.4.

$Dom(D)$ of D. The output of each transition is set to the empty word. If w is an entry of D, then the final state corresponding to w receives the output $D(w)$. Figure 1 shows the subsequential transducer trie $\mathcal{T}_D$ for the rewrite dictionary D introduced in Example 2.4. Obviously the number of states in the resulting transducer $\mathcal{T}_D$ does not exceed the total number of symbols in the list of all dictionary entries.

Second Step. Given $\mathcal{T}_D$ we construct a transducer $\mathcal{T}$ that represents (in the sense of Definition 2.12) the canonical lmls-extension D^{lmls} of D. The set of states of $\mathcal{T}_D$ is not modified. Similarly the output of the final states of $\mathcal{T}_D$ not changed. Furthermore, each transitions of $\mathcal{T}_D$ remains a transition of $\mathcal{T}$. Hence, $\mathcal{T}_D$ is only extended. Since $\mathcal{T}$ must accept any text, we must be able to depart from any state with any symbol of the input alphabet. This means that suitable failure transitions have to be added. Furthermore, each state has to be treated as a final state. The main difficulty is the definition of the output of failure transitions and final states. The reader should keep in mind that the output of a final state q is only used if we reach q at the *end* of the text.

In order to introduce missing transitions and to define suitable outputs, we visit all states q of $\mathcal{T}_D$. For any state p, let $d(q_0, p)$ denote the number of transitions in $\mathcal{T}_D$ that are needed to reach p, starting from q_0. When we visit a state p we always assume that all states p' such that $d(q_0, p') < d(q_0, p)$ have been visited before.

We begin with q_0. When reading a text t, we want to be in state q_0 when we are sure that the current position is not inside a relevant left-most occurrence of an entry of D. We also assume that no output obligations are left open. In terms of the naive algorithm we capture the situation where we begin a new scan, starting from a certain offset point k. Here q_0 is declared to be final and the output is the empty string. We have to add a transition for input letter d. Since no entry of D starts with d the transition returns to q_0, passing d to the output. Note that in this situation the naive algorithm would immediately begin a new scan, starting from the next offset point $k + 1$.

For simplicity, let us assume that we have visited all states p such that $d(q_0, p) \leq 2$. The transitions and state outputs that have been added to the visited six states of $\mathcal{T}_D$ can be seen in Figure 1.

Let q denote the state that is reached from q_0 with input bab. We declare q to be a final state. Imagine that we process a text with prefix $t' \cdot bab$, using the naive algorithm. Starting from the offset position k that is marked by the dot we have read bab. Here we have not found any relevant leftmost occurrence that starts at position k so far. Correspondingly, in $\mathcal{T}_D$ there is no final state on the path from q_0 to q.

First, if the text ends after bab, then the naive algorithm passes b to the output, goes back to the next offset position $k+1$ and starts to scan ab. In the current version of the transducer, by assumption, we have encoded the result of all scans of length 2. Hence we already know the result: as we see in Figure 1, a final traversal with ab yields (state) output 2. Hence, the appropriate state output for q is $b2$.

It remains to add missing transitions for input letters a, b, d. Imagine the text does not end after bab, but continues, say, with letter a. Then the naive algorithm passes b to the output, goes back to the next offset position $k+1$ and starts to scan aba. Again, looking at the transducer we already know the result: as we see in Figure 1, a traversal with aba yields (transition) output 2 and leads to the state q' with output 1. Hence the correct output for the transition from q with letter a is $b2$, and the transition leads to q'. The situation for input letter b is similar. For input letter d we again use the result of a traversal with abd to find the correct transition output and target state. Figure 1 shows that we reach the initial state with transition output $2d$. Hence, from q we add a transition with output $b2d$ to the initial state.

Now let q denote the state that is reached from q_0 with input abc. This time the situation is different since we have two final states on the path of $\mathcal{T}_D$ from q_0 to q. Imagine that we process a text with prefix $t' \cdot abc$, using the naive algorithm. Starting from the offset point k that is marked by the dot we have read abc. Here ab is the maximal relevant occurrence starting at position k that we found so far.

First, if the text ends after abc, then the naive algorithm passes $D(ab) = 2$ to the output and continues with a final scan of c. In the transducer, by assumption, we have encoded the result of all scans of length 2. Hence we already know the result of the final scan: as we see in Figure 1, a final traversal with c yields (state) output 5. Hence, the appropriate state output for q is 25.

It remains to add missing transitions for input letters a, b, d. Imagine the text does not end after abc, but continues, say, with letter a. Then the naive algorithm outputs $D(ab) = 2$, goes back to the offset position k' given by letter c and starts to scan ca. Again, in the transducer we already know the result: as we see in Figure 1, a traversal with ca yields (transition) output 1 and leads to the state q' with output 1. Hence the correct output for the transition from q with letter a is 21, and the transition leads to q'. The situation for input letter b is similar. For input letter d we use the output $D(ab) = 2$ plus the result of a traversal with cd to find the correct transition output and target state. Figure 1 shows that a traversal with cd leads to q_0 with transition output $5d$. Hence, from q we add a transition with output $25d$ to q_0.

The transducer $\mathcal{T}$ that is obtained after visiting all states of $\mathcal{T}_D$ is shown in Figure 2.

4 The algorithm

We now formally describe the construction of the subsequential transducer for text rewriting based on a dictionary D. As we made clear in the previous section, the constructions proceeds in two steps.

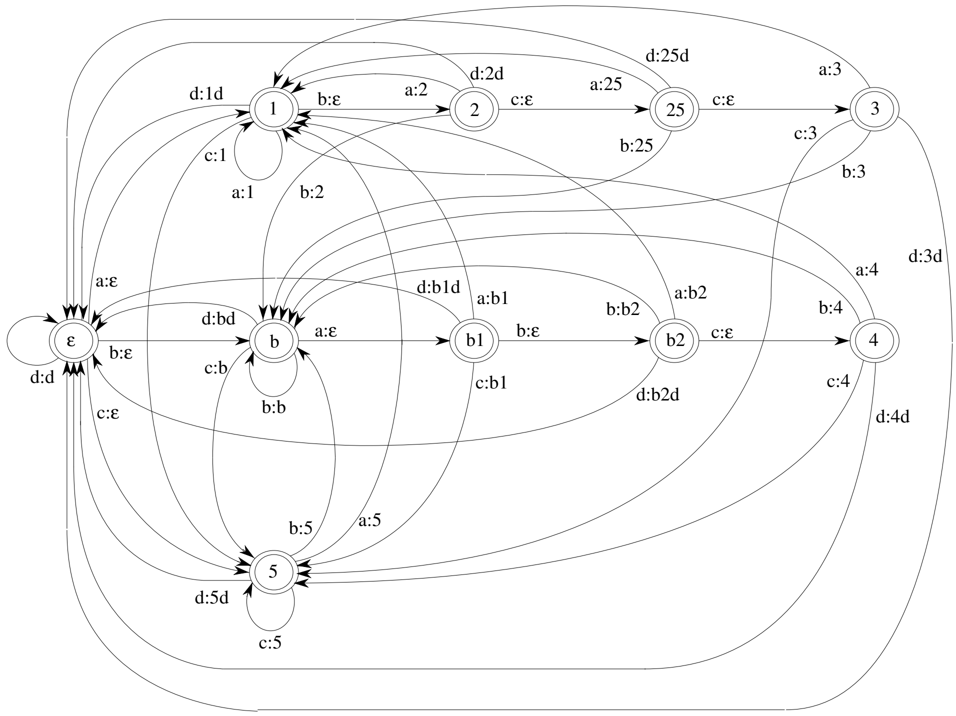

Figure 2: The subsequential transducer for the local extension of the dictionary given in Example 2.4.

4.1 Construction of the subsequential transducer trie for the dictionary

The following construction of the subsequential transducer trie for a given rewrite dictionary D is well-known and only described for convenience and to fix notation. We assume that D is represented as a finite list of pairs of words of the form

$$D = \{(\alpha_i \mapsto \beta_i) \mid i = 1, \ldots, n\}.$$

Here $\alpha_i \in \Sigma^+$, $\beta_i \in \Delta^*$ and $\alpha_i \neq \alpha_j$ for $1 \leq i, j \leq n$, $i \neq j$. We start with the empty transducer $\mathcal{T}_0$, which is defined as the tuple

$$\mathcal{T}_0 = \langle \Sigma, \Delta, \{q_0\}, q_0, \emptyset, \emptyset, \emptyset, \emptyset \rangle .$$

Each of the following steps adds a new word pair to the subsequential transducer. Let us assume that we have constructed the subsequential transducer

$$\mathcal{T}_{k-1} = \langle \Sigma, \Delta, Q_{k-1}, q_0, F_{k-1}, \delta_{k-1}, \lambda_{k-1}, \Psi_{k-1} \rangle .$$

representing the dictionary $D_{k-1} = \{\langle \alpha_i, \beta_i \rangle \in D \,|\, i \in \{1, 2, \ldots, k-1\}\}$ for some $k, 1 \leq k \leq n$. We next have to add the pair $\langle \alpha_k, \beta_k \rangle$ to the transducer. Let $\alpha_k = a_1 \ldots a_m$ and let $a_1 \ldots a_l$ denote the maximal prefix of α_k such that $\delta^*_{k-1}(a_1 \ldots a_l)$ is defined. Here $l \leq m$. We introduce a set Q' with $m - l$ new states $q_{l+1}, q_{l+2}, \ldots, q_m$ and define

$$\mathcal{T}_k := \langle \Sigma, \Delta, Q_{k-1} \cup Q', q_0, F_{k-1} \cup \{f^*\}, \delta_k, \lambda_k, \Psi_{k-1} \cup \langle q_m, \beta_k \rangle \rangle$$

where

$$f^* := \begin{cases} q_m \text{ iff } l < m, \\ \delta^*_{k-1}(a_1 \ldots a_m) \text{ otherwise} \end{cases}$$

and

$$\delta_k := \delta_{k-1} \cup \{\langle q_{r-1}, a_r, q_r \rangle \mid r \in \{l+1, \ldots, m\}\},$$
$$\lambda_k := \lambda_{k-1} \cup \{\langle q_{r-1}, a_r, \varepsilon \rangle \mid r \in \{l+1, \ldots, m\}\}.$$

Obviously, $\mathcal{T}_k$ represents the dictionary $D_k = \{\langle \alpha_i, \beta_i \rangle \in D \mid 1 \le i \le k\}$. Hence, after n steps we receive the transducer

$$\mathcal{T}_D = \langle \Sigma, \Delta, Q_D, q_0, F_D, \delta_D, \lambda_D, \Psi_D \rangle,$$

which represents D. $\mathcal{T}_D$ is tree-shaped, output is only produced when reaching a final state after completely consuming the input. Furthermore, each state $q \in Q_D$ can be reached with exactly one word. Henceforth, by α_q we denote the unique string in Σ^* such that $\delta_D^*(q_0, \alpha_q) = q$. Furthermore, $d(q_0, q) = |\alpha_q|$ denotes the number of transitions in $\mathcal{T}_D$ that are needed to reach q, starting from q_0 ($q \in Q_D$).

4.2 Canonical lmls-extension of $\mathcal{T}_D$

In order to define the transducer $\mathcal{T}$ for text rewriting based on the given dictionary D we visit all states of $\mathcal{T}_D$. When visiting a state q we declare q to be final, define the state output and add missing transitions such that for each $\sigma \in \Sigma$ we have exactly one transition departing from q with input letter σ. As described in the previous section, when visiting q we always assume that all states q' such that $d(q_0, q') < d(q_0, q)$ have been visited before. We may assume that a list $(p_1, p_2, \ldots, p_h)$ is given that describes one possible visiting order for the states of Q_D that respects this constraint. Note that $q_0 = p_1$.

The transducer that results from our visit at q_0 is

$$\mathcal{T}_1' := \langle \Sigma, \Delta, Q_D, q_0, F_D \cup \{q_0\}, \delta_1', \lambda_1', \Psi_D \cup \langle q_0, \varepsilon \rangle \rangle$$

where

$$\begin{aligned} \delta_1' &:= \delta_D \cup \{\langle q_0, \sigma, q_0 \rangle \mid \sigma \in \Sigma, \langle q_0, \sigma \rangle \notin \mathit{Dom}(\delta_D)\}, \\ \lambda_1' &:= \lambda_D \cup \{\langle q_0, \sigma, \sigma \rangle \mid \sigma \in \Sigma, \langle q_0, \sigma \rangle \notin \mathit{Dom}(\delta_D)\}. \end{aligned}$$

Let us assume that after visiting p_i (where $1 \le i < h$) we have constructed the transducer

$$\mathcal{T}_i' = \langle \Sigma, \Delta, Q_D, q_0, F_i', \delta_i', \lambda_i', \Psi_i' \rangle.$$

Let $q = p_{i+1}$ denote the next state on our list of states to be visited. Let $\alpha_q = a_1 \ldots a_r$. For $1 \le j \le r$, let $q_j := \delta_D^*(q_0, a_1 \ldots a_j)$. Hence $q_0, \ldots, q_r$ denotes the sequence of states on the path of $\mathcal{T}_D$ leading from q_0 to q. We distinguish two cases:

Case 1: $F_D \cap \{q_0, q_1, \ldots, q_r\} = \emptyset$. Here $\delta_i^*(q_0, a_2 a_3 \ldots a_r)$ is defined, by choice of the visiting order. Let $p' := \delta_i^*(q_0, a_2 a_3 \ldots a_r)$ and $\gamma' := a_1 \cdot {\lambda_i'}^*(q_0, a_2 a_3 \ldots a_r)$. We define

$$\mathcal{T}_{i+1}' := \left\langle \Sigma, \Delta, Q_D, q_0, F_i' \cup \{q\}, \delta_{i+1}', \lambda_{i+1}', \Psi_{i+1}' \cup \langle q, \gamma' \cdot \Psi_i'(p') \rangle \right\rangle$$

where

$$\begin{aligned} \delta_{i+1}' &:= \delta_i' \cup \{\langle q, \sigma, \delta_i'(p', \sigma) \rangle \mid \sigma \in \Sigma, \langle q, \sigma \rangle \notin \mathit{Dom}(\delta_D)\}, \\ \lambda_{i+1}' &:= \lambda_i' \cup \{\langle q, \sigma, \gamma' \cdot \lambda_i'(p', \sigma) \rangle \mid \sigma \in \Sigma, \langle q, \sigma \rangle \notin \mathit{Dom}(\delta_D)\}. \end{aligned}$$

Case 2: $F_D \cap \{q_0, q_1, \ldots, q_r\} \ne \emptyset$. Let q_f denote the last final state of $\mathcal{T}_D$ on the path leading from q_0 to q.[3] This means that $\delta_D^*(q_0, a_1 a_2 \ldots a_f) = q_f \in F_D$

[3] Note that we can have $q^f = q$. In this case we will have $\Psi_{i+1}'(q) = \Psi_D(q)$.

and $\{q_{f+1}, q_{f+2}, \ldots, q_r\} \cap F_D = \emptyset$. Let $p' := \delta_i'^*(q_0, a_{f+1}a_{f+2}\ldots a_r)$ and $\gamma' := \Psi_D(q_f) \cdot \lambda_i'^*(q_0, a_{f+1}a_{f+2}\ldots a_r)$. We define

$$\mathcal{T}_{i+1} = \left\langle \Sigma, \Delta, Q_D, q_0, F_i' \cup \{q\}, \delta_{i+1}', \lambda_{i+1}', \Psi_i' \cup \langle q, \gamma' \cdot \Psi_i'(p')\rangle \right\rangle$$

where

$$\begin{aligned} \delta_{i+1}' &:= \delta_i' \cup \{\langle q, \sigma, \delta_i'(p', \sigma)\rangle \mid \sigma \in \Sigma, \langle q, \sigma\rangle \notin Dom(\delta_D)\}, \\ \lambda_{i+1}' &:= \lambda_i' \cup \{\langle q, \sigma, \gamma' \cdot \lambda_i'(p', \sigma)\rangle \mid \sigma \in \Sigma, \langle q, \sigma\rangle \notin Dom(\delta_D)\}. \end{aligned}$$

The transducer

$$\mathcal{T} = \langle \Sigma, \Delta, Q, q_0, Q, \delta, \lambda, \Psi\rangle$$

obtained after the final step h represents the output of our algorithm.

5 Correctness of the algorithm

We now show that the above algorithm is correct.

Theorem 5.1 *Let D be a rewrite dictionary. Then the transducer $\mathcal{T}$ computed by the above algorithm represents the canonical lmls-extension D^{lmls} of D, in the sense of Definition 2.12.*

The theorem follows immediately from Part 5 of the following Lemma.

Lemma 5.2 *Let $t = \sigma_1 \ldots \sigma_n \in \Sigma^*$. Let $q := \delta^*(q_0, t)$. Let $\alpha = \alpha_q$ denote the unique word such that $\delta_D^*(q_0, \alpha) = q$. Then*

1. *α is a suffix of t.*

Furthermore, if $t = t_0 \cdot \alpha$, then

2. *$D^{lmls}(t_0) = \lambda^*(q_0, t)$,*
3. *for all $\gamma \in \Sigma^*$ and all prefixes α_0 of $\alpha \cdot \gamma$ we have $D^{lmls}(t_0 \cdot \alpha_0) = D^{lmls}(t_0) \cdot D^{lmls}(\alpha_0)$,*
4. *$D^{lmls}(\alpha) = \Psi(q)$,*
5. *$D^{lmls}(t) = \lambda^*(q_0, t) \cdot \Psi(q)$.*

Proof. We first remark that Property 5 is a consequence of the given partitioning of t and Properties 2-4. In fact, with $\alpha_0 := \alpha$ we obtain

$$D^{lmls}(t) = D^{lmls}(t_0 \cdot \alpha) \overset{3.}{=} D^{lmls}(t_0) \cdot D^{lmls}(\alpha) \overset{2.,4.}{=} \lambda^*(q_0, t) \cdot \Psi(q).$$

In order to verify Properties 1-4 we use induction on the length n of the text t. If $n = 0$, then $t = t_0 = \alpha$ is the empty string and all claims are trivial. Assume that the claims are correct for n.

Let $t' := \sigma_1 \ldots \sigma_n \sigma_{n+1}$ and $q' := \delta^*(q_0, t')$. Let $t = \sigma_1 \ldots \sigma_n, q, t_0$ and α as above.

Case 1. $\delta_D(q, \sigma_{n+1})$ is defined. Then $q' = \delta_D(q, \sigma_{n+1})$ and $\delta_D(q_0, \alpha \cdot \sigma_{n+1}) = q'$. Obviously $\alpha' = \alpha \cdot \sigma_{n+1}$ is a suffix of $t' = t_0 \cdot \alpha'$, hence Claim 1 is satisfied. In what follows we consider the partitioning $t' = t_0 \cdot \alpha'$ of t'. We have $\lambda(q, \sigma_{n+1}) = \epsilon$. Induction hypothesis 2 for t yields

$$D^{lmls}(t_0) = \lambda^*(q_0, t) = \lambda^*(q_0, t) \cdot \lambda(q, \sigma_{n+1}) = \lambda^*(q_0, t').$$

which shows that Claim 2 is satisfied. Claim 3 is trivial. We prove Claim 4. Let α' have the form $a_1 \dots a_r$. This means that $a_1, \dots, a_r$ is the sequence of input transition labels on the path of $\mathcal{T}_D$ from q_0 to q'.

Subcase 1.1. First assume that there is no final state of $\mathcal{T}_D$ on the path from q_0 to q'. In this case our construction implies that

$$\Psi(q') = a_1 \cdot \lambda^*(q_0, a_2 \dots a_r) \cdot \Psi(\delta^*(q_0, a_2 \dots a_r)).$$

Furthermore, no prefix of $a_1 \dots a_r$ belongs to the domain of D. Hence Corollary 2.10 yields $D^{lmls}(\alpha') = D^{lmls}(a_1 \dots a_r) = a_1 D^{lmls}(a_2 \dots a_r)$. Applying the induction hypothesis to the text $a_2 \dots a_r$ we get (cf. Claim 5)

$$D^{lmls}(a_2 \dots a_r) = \lambda^*(q_0, a_2 \dots a_r) \cdot \Psi(\delta^*(q_0, a_2 \dots a_r)).$$

Hence $D^{lmls}(\alpha') = \Psi(q')$, which shows that Claim 4 is satisfied.

Subcase 1.2. There exists a final state of $\mathcal{T}_D$ on the path from q_0 to q'. Let q_f denote the last state on the given path that is final. Let $a_1 \dots a_f$ denote the sequence of input transition labels from q_0 to q_f. In this case our construction implies that

$$\Psi(q') = \Psi(q_f) \cdot \lambda^*(q_0, a_{f+1} \dots a_r) \cdot \Psi(\delta^*(q_0, a_{f+1} \dots a_r)).$$

Furthermore, since $a_1 \dots a_f$ is a leftmost largest occurrence within $\alpha' = a_1 \dots a_r$ it follows from Corollary 2.9 that $D^{lmls}(\alpha') = D^{lmls}(a_1 \dots a_f) = D(a_1 \dots a_f) \cdot D^{lmls}(a_{f+1} \dots a_r)$. Since the output of the final states of $\mathcal{T}_D$ is not changed in $\mathcal{T}$ we have $D(a_1 \dots a_f) = \Psi(q_f)$. Since $q_0 \neq q_f$, the induction hypothesis (Claim 5) applied to $a_{f+1} \dots a_r$ yields $D^{lmls}(a_{f+1} \dots a_r) = \lambda^*(q_0, a_{f+1} \dots a_r) \cdot \Psi(\delta^*(q_0, a_{f+1} \dots a_r))$. Hence $D^{lmls}(\alpha') = \Psi(q')$, which shows that Claim 4 is satisfied.

Case 2.1. $\delta_D(q, \sigma_{n+1})$ is not defined and there is no final state of $\mathcal{T}_D$ on the path from q_0 to q. Let $a_1 \dots a_r$ denote the sequence of labels of the path from q_0 to q. In this case our construction yields $q' = \delta(q, \sigma_{n+1}) = \delta^*(q_0, a_2 \dots a_r \sigma_{n+1})$ and $\lambda(q, \sigma_{n+1}) = a_1 \cdot \lambda^*(q_0, a_2 \dots a_r \sigma_{n+1})$. Let α' denote the unique word such that $\delta_D(q_0, \alpha') = q'$. The induction hypothesis applied to $a_2 \dots a_r \sigma_{n+1}$ shows that α' is a suffix of $a_2 \dots a_r \sigma_{n+1} = \alpha'' \cdot \alpha'$, hence α' is a suffix of $t' = t_0 \cdot a_1 \cdot \alpha'' \cdot \alpha'$, which shows that Claim 1 holds. Let $t_0' := t_0 \cdot a_1 \cdot \alpha''$. Applying the induction hypothesis (Claim 3, $\alpha_0 := a_1 \cdot \alpha''$) we get $D^{lmls}(t_0') = D^{lmls}(t_0) \cdot D^{lmls}(a_1 \cdot \alpha'')$. Our assumption implies that no prefix of $a_1 \cdot \alpha''$ is in the domain of D. Corollary 2.10 shows that $D^{lmls}(a_1 \cdot \alpha'') = a_1 \cdot D^{lmls}(\alpha'')$, hence $D^{lmls}(t_0') = D^{lmls}(t_0) \cdot a_1 \cdot D^{lmls}(\alpha'')$. The induction hypothesis (Claim 2) for t yields $D^{lmls}(t_0) = \lambda^*(q_0, t)$. The induction hypothesis (Claim 2) for $a_2 \dots a_r \sigma_{n+1}$ yields $D^{lmls}(\alpha'') = \lambda^*(q_0, a_2 \dots a_r \sigma_{n+1})$. We obtain

$$\begin{aligned} D^{lmls}(t_0') &= \lambda^*(q_0, t) \cdot a_1 \cdot \lambda^*(q_0, a_2 \dots a_r \sigma_{n+1}) \\ &= \lambda^*(q_0, t) \cdot \lambda(q, \sigma_{n+1}) \\ &= \lambda^*(q_0, t'). \end{aligned}$$

which proves Claim 2.

We prove Claim 3. Let $\gamma \in \Sigma^*$, let α_0 be any prefix of $\alpha' \cdot \gamma$. By induction hypothesis (3) for t we have $D^{lmls}(t_0' \cdot \alpha_0) = D^{lmls}(t_0 \cdot a_1 \cdot \alpha'' \cdot \alpha_0) = D^{lmls}(t_0) \cdot D^{lmls}(a_1 \cdot \alpha'' \cdot \alpha_0)$. Our assumption implies that no prefix of $a_1 \cdot \alpha'' \cdot \alpha_0$ is in the domain of D. Corollary 2.10 shows that $D^{lmls}(a_1 \cdot \alpha'' \cdot \alpha_0) = a_1 \cdot D^{lmls}(\alpha'' \cdot \alpha_0)$. Induction hypothesis (3) for $a_2 \dots a_r \sigma_{n+1}$ implies that $D^{lmls}(\alpha'' \cdot \alpha_0) = D^{lmls}(\alpha'') \cdot D^{lmls}(\alpha_0)$. Hence we have $D^{lmls}(t_0' \cdot \alpha_0) = D^{lmls}(t_0) \cdot a_1 \cdot D^{lmls}(\alpha'') \cdot D^{lmls}(\alpha_0)$. As above it follows that $D^{lmls}(t_0) \cdot a_1 \cdot D^{lmls}(\alpha'') = D^{lmls}(t_0')$, which proves Claim 3.

Since $q' = \delta^*(q_0, a_2 \dots a_r \sigma_{n+1})$ the induction hypothesis for $a_2 \dots a_r \sigma_{n+1}$ yields $D^{lmls}(\alpha') = \Psi(q')$, which shows that Claim 4 is satisfied.

Case 2.2. $\delta_D(q, \sigma_{n+1})$ is not defined and there is a final state of $\mathcal{T}_D$ on the path from q_0 to q. Let $a_1 \ldots a_r$ denote the sequence of labels of the path from q_0 to q, let q_f denote the last final state on the path and let $a_1 \ldots a_f$ denote the sequence of labels of the path from q_0 to q_f.

In this case we have $q' = \delta(q, \sigma_{n+1}) = \delta^*(q_0, a_{f+1} \ldots a_r \sigma_{n+1})$ and $\lambda(q, \sigma_{n+1}) = \Psi_D(q_f) \cdot \lambda^*(q_0, a_{f+1} \ldots a_r \sigma_{n+1})$. Let α' denote the unique word such that $\delta_D(q_0, \alpha') = q'$. The induction hypothesis applied to $a_{f+1} \ldots a_r \sigma_{n+1}$ shows that α' is a suffix of $a_{f+1} \ldots a_r \sigma_{n+1} = \alpha'' \cdot \alpha'$, hence α' is a suffix of $t' = t_0 \cdot a_1 \ldots a_f \cdot \alpha'' \cdot \alpha'$, which shows that Claim 1 holds. Let $t'_0 := t_0 \cdot a_1 \ldots a_f \cdot \alpha''$. Applying the induction hypothesis (Claim 3, $\alpha_0 := a_1 \ldots a_f \cdot \alpha''$) we get $D^{lmls}(t'_0) = D^{lmls}(t_0) \cdot D^{lmls}(a_1 \ldots a_f \cdot \alpha'')$. Our assumption implies that $a_1 \ldots a_f$ represents a largest leftmost occurrence within $a_1 \ldots a_f \cdot \alpha''$. Corollary 2.9 shows that $D^{lmls}(a_1 \ldots a_f \cdot \alpha'') = D^{lmls}(a_1 \ldots a_f) \cdot D^{lmls}(\alpha'') = \Psi_D(q_f) \cdot D^{lmls}(\alpha'')$, hence $D^{lmls}(t'_0) = D^{lmls}(t_0) \cdot \Psi_D(q_f) \cdot D^{lmls}(\alpha'')$. The induction hypothesis (Claim 2) for t yields $D^{lmls}(t_0) = \lambda^*(q_0, t)$. The induction hypothesis (Claim 2) for $a_{f+1} \ldots a_r \sigma_{n+1}$ yields $D^{lmls}(\alpha'') = \lambda^*(q_0, a_{f+1} \ldots a_r \sigma_{n+1})$. We obtain

$$\begin{aligned} D^{lmls}(t'_0) &= \lambda^*(q_0, t) \cdot \Psi_D(q_f) \cdot \lambda^*(q_0, a_{f+1} \ldots a_r \sigma_{n+1}) \\ &= \lambda^*(q_0, t) \cdot \lambda(q, \sigma_{n+1}) \\ &= \lambda^*(q_0, t'), \end{aligned}$$

which proves Claim 2.

We prove Claim 3. Let $\gamma \in \Sigma^*$, let α_0 be any prefix of $\alpha' \cdot \gamma$. By induction hypothesis (3) for t we have $D^{lmls}(t'_0 \cdot \alpha_0) = D^{lmls}(t_0 \cdot a_1 \ldots a_f \cdot \alpha'' \cdot \alpha_0) = D^{lmls}(t_0) \cdot D^{lmls}(a_1 \ldots a_f \cdot \alpha'' \cdot \alpha_0)$. Our assumption and Corollary 2.10 show that $D^{lmls}(a_1 \ldots a_f \cdot \alpha'' \cdot \alpha_0) = \Psi_D(q_f) \cdot D^{lmls}(\alpha'' \cdot \alpha_0)$. Induction hypothesis (3) for $a_{f+1} \ldots a_r \sigma_{n+1}$ implies that $D^{lmls}(\alpha'' \cdot \alpha_0) = D^{lmls}(\alpha'') \cdot D^{lmls}(\alpha_0)$. Hence we have $D^{lmls}(t'_0 \cdot \alpha_0) = D^{lmls}(t_0) \cdot \Psi_D(q_f) \cdot D^{lmls}(\alpha'') \cdot D^{lmls}(\alpha_0)$. As above it follows that $D^{lmls}(t_0) \cdot \Psi_D(q_f) \cdot D^{lmls}(\alpha'') = D^{lmls}(t'_0)$, which proves Claim 3.

Since $q' = \delta^*(q_0, a_{f+1} \ldots a_r \sigma_{n+1})$ the induction hypothesis for $a_{f+1} \ldots a_r \sigma_{n+1}$ yields $D^{lmls}(\alpha') = \Psi(q')$, which shows that Claim 4 is satisfied. $\square$

6 Complexity analysis and implementation details

It is simple to see that the number of states of the resulting transducer $\mathcal{T}$ for an input dictionary D is bounded by the size (number of symbols) of the domain of D. In fact, states of $\mathcal{T}$ represent states of the transducer trie, which correspond to prefixes of entries of D. Since the alphabet is fixed, the number of transitions of $\mathcal{T}$ is linear in the size of $Dom(D)$. The following example shows that the size of the output functions can become cubic in the size (number of symbols) of D when using the standard representation for output strings.

Example 6.1 Let $n \geq 1$. Consider the rewrite dictionary D of the form

$$\begin{array}{rcl} (a & \mapsto & 1^n) \\ (a^n b & \mapsto & 2) \end{array}$$

where 1^n (a^n) denotes a sequence of n symbols 1 (a). Then the size of a single transition/state output is quadratic, which means that the size of the transition/state output function is cubic. The corresponding transducer for $n = 4$ is shown in Figure 3.

The following proposition shows that the size of the output functions is at most cubic in the size of D.

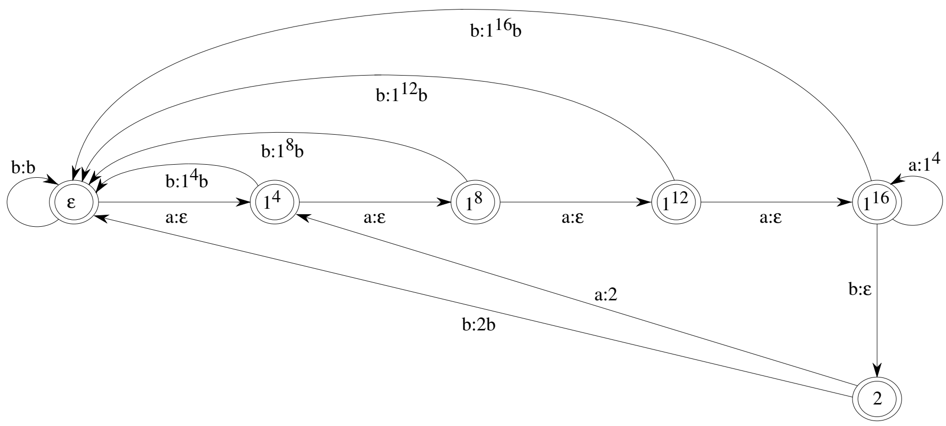

Figure 3: The subsequential transducer for the local extension of the dictionary given in Example 6.1.

Proposition 6.2 *Let λ and Ψ respectively denote the transition and state output function of $\mathcal{T}$, let $q \in Q, \sigma \in \Sigma$. Assume that q is reached with i transitions from the initial state q_0. Then*

1. *$\Psi(q)$ can be represented as a concatenation of $\leq i$ strings representing either elements of CoDom(D) or symbols from Σ, and*
2. *$\lambda(q, \sigma)$ can be represented as a concatenation of $\leq i+1$ strings representing either elements of CoDom(D) or symbols from Σ.*

Proof. Part 1 directly follows from Part 4 of Lemma 5.2. Here α represents the input of length i leading from q_0 to q. Clearly $D^{lmls}(\alpha)$ can be represented in the described form.

Part 2: Let α_0 denote the the unique word such that $\delta^*_D(q_0, \alpha_0) = q$. It is simple to see that $\lambda^*(q_0, \alpha_0) = \epsilon$ since the relevant transitions are already present in the transducer trie for D, $\mathcal{T}_D$. Hence we have $\lambda^*(q, \alpha_0\sigma) = \lambda(q, \sigma)$. We consider the text $t = \alpha_0\sigma$. Let $q' = \delta^*(q_0, t)$ and let α denote the unique word such that $\delta^*_D(q_0, \alpha) = q'$. By Part 1 of Lemma 5.2, α is a suffix of t. The corresponding prefix t_0 of $t = t_0 \cdot \alpha$ has length $\leq i+1$. Part 2 of Lemma 5.2 now yields $\lambda(q, \sigma) = \lambda^*(q, t) = D^{lmls}(t_0)$. Clearly $D^{lmls}(t_0)$ can be represented in the described form. □

A closer inspection of the recursive definition of the output functions given in Section 4.2 shows that a linear representation of the output function is possible. The key idea is to use pointers to represent the strings γ' defined in Cases 1 and 2 of the construction of $\mathcal{T}'_{i+1}$ (here and in what follows we use the notation introduced in Section 4.2). We write γ'_q for γ' in order to make the correspondence to q (the state for which γ' is defined) more obvious.

In the algorithm, "old" outputs of the final states of the transducer trie and the empty transition outputs already defined in the trie are not modified. In Case 1, "new" state and transition outputs for q are respectively defined as

$$\begin{aligned}
\Psi(q) &= \gamma'_q \cdot \Psi(p') \\
&= a_1 \cdot \lambda^*(q_0, a_2 \ldots a_r) \cdot \Psi(p') \\
&= \begin{cases} \gamma'_{q_{r-1}} \cdot \lambda(\delta^*(q_0, a_2 \ldots, a_{r-1}), a_r) \cdot \Psi(p') \text{ if } r > 1, \\ a_1 \cdot \Psi(p') \text{ if } r = 1. \end{cases} \\
\lambda(q, \sigma) &= \gamma'_q \cdot \lambda(p', \sigma)
\end{aligned}$$

$$
\begin{aligned}
&= a_1 \cdot \lambda^*(q_0, a_2 \dots a_r) \cdot \lambda(p', \sigma) \\
&= \begin{cases} \gamma'_{q_{r-1}} \cdot \lambda(\delta^*(q_0, a_2 \dots, a_{r-1}), a_r) \cdot \lambda(p', \sigma) \text{ if } r > 1, \\ a_1 \cdot \lambda(p', \sigma) \text{ if } r = 1. \end{cases}
\end{aligned}
$$

In Case 2, new state and transition outputs for q are respectively defined as

$$
\begin{aligned}
\Psi(q) &= \gamma'_q \cdot \Psi(p') \\
&= \Psi(q_f) \cdot \lambda^*(q_0, a_{f+1} \dots a_r) \cdot \Psi(p') \\
&= \begin{cases} \gamma'_{q_{r-1}} \cdot \lambda(\delta^*(q_0, a_{f+1} \dots, a_{r-1}), a_r) \cdot \Psi(p') \text{ if } r > f, \\ \Psi(q_f) \cdot \Psi(p') \text{ if } r = f. \end{cases} \\
\lambda(q, \sigma) &= \gamma'_q \cdot \lambda(p', \sigma) \\
&= \Psi(q_f) \cdot \lambda^*(q_0, a_{f+1} \dots a_r) \cdot \lambda(p', \sigma) \\
&= \begin{cases} \gamma'_{q_{r-1}} \cdot \lambda(\delta^*(q_0, a_{f+1} \dots, a_{r-1}), a_r) \cdot \lambda(p', \sigma) \text{ if } r > f, \\ \Psi(q_f) \cdot \lambda(p', \sigma) \text{ if } r = f. \end{cases}
\end{aligned}
$$

Recall that the states p' and q_{r-1} are closer to the start state than q. From these equations we see that the following representation is possible. We will construct the output string using a record, which contains either a convential string, a symbols from Σ or two pointers to other records. Let π_1 and π_2 be pointers. Assume that π_1 points to the location of π_2. By *redirecting* π_1 we mean the process where we let π_1 point to the same address as π_2.

- "Old" outputs of the final states of the transducer trie and the empty transition outputs already defined in the transducer trie are represented as pointers to records containing conventional strings.

- To represent a new state output $\Psi(q)$ we use a pointer to a new record with two pointers. The first (second) pointer points to a record representing γ'_q (resp. $\Psi(p')$). If γ'_q or $\Psi(p')$ is empty, the corresponding pointer is suppressed and $\Psi(q)$ is redirected to the record pointed by the other pointer. If both γ'_q and $\Psi(p')$ are empty, $\Psi(q)$ is redirected to point to the empty word record.

- To represent the output of a new transition, $\lambda(q, \sigma)$, we use a pointer to a new record with two pointers. The first (second) pointer points to a record representing γ'_q (resp. $\lambda(p', \sigma)$). If γ'_q or $\lambda(p', \sigma)$ is empty, the corresponding pointer is suppressed and $\lambda(q, \sigma)$ is redirected to to the record pointed by the other pointer. If both γ'_q and $\lambda(p', \sigma)$ are empty, $\lambda(q, \sigma)$ is redirected to point to the empty word record.

- In the case where $r = 1$ (Case 1) or $r = f$ (Case 2) γ'_q is represented by a pointer to a new record containing the letter a_1 (Case 1) or a pointer to the record pointed by $\Psi(q_f)$ (Case 2). For $r > 1$ (Case 1) or $r > f$ (Case 2) γ'_q is represented by a pointer to a new record which contains two pointers. The first points to the record pointed by $\gamma'_{q_{r-1}}$ (Cases 1,2). The second pointer points to record pointed by $\lambda(\delta^*(q_0, a_x \dots, a_{r-1}), a_r)$ where $x = 2$ in Case 1 and $x = f + 1$ in Case 2. If $\gamma'_{q_{r-1}}$ or $\lambda(\delta^*(q_0, a_x \dots, a_{r-1}), a_r)$ is empty, the corresponding pointer is suppressed and γ'_q is redirected to the record pointed by the other pointer. If both $\gamma'_{q_{r-1}}$ and $\lambda(\delta^*(q_0, a_x \dots, a_{r-1}), a_r)$ are empty, then γ'_q is redirected to point to the empty word record.

Example 6.3 In Figure 4 we see the subsequential transducer for the local extension of the dictionary given in Example 6.1 with the pointer-based representation of output functions. The string γ'_q for each state q is written below q. Small numbers in bold denote pointers.

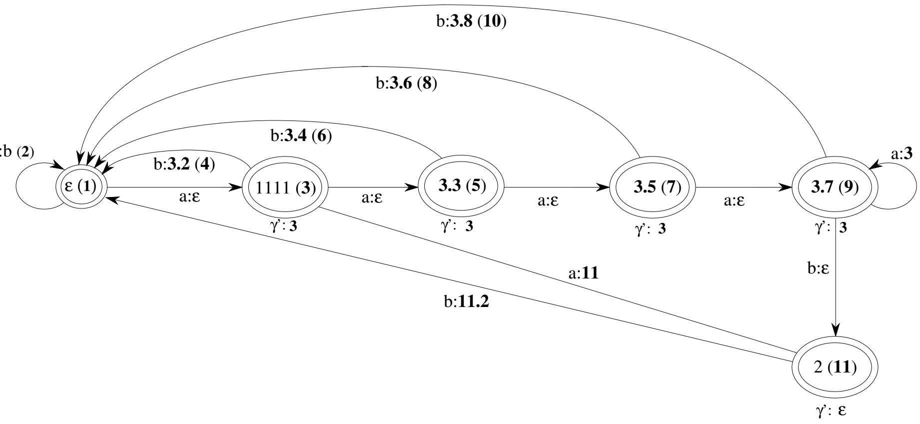

Figure 4: The subsequential transducer for the local extension of the dictionary given in Example 6.1 with the pointer-based representation of output functions.

As a matter of fact, concatenating the strings that are found when following the pointers in the obvious order yields the correct output strings. As to the representation size, the important point to note is that we need a constant number of pointers per state q. Besides the pointers, each output of the trie transducer (i.e., each element of the codomain of D) and each letter of the alphabet needs to be represented once. It follows that the representation size of the transducer is linear in the size of D.

Since each member of a pair of pointers always points to a non-empty string, and since there are no chains of single pointers, when applying the transducer to a text t the number of pointers that have to be processed for producing a partial output string β is bounded by the length of β plus 1 (1 for empty output). Hence the complexity of text rewriting $O(|t| + |t'|)$ is not affected by this special technique for representing outputs.

It is not completely obvious how to modify the algorithm described in Section 4 in order to obtain the correct pointer representation for the outputs. Furthermore, a naive implementation of the algorithm leads to a time complexity which is quadratic in the worst case. The problem is that the calculation of p' and γ' (cf. Cases 1 and 2) requires a traversal of the transducer with a suffix $a_2 \dots a_r$ (Case 1) or $a_{f+1} \dots a_r$ (Case 2) of the word a_q. The length of a_q, as the length of the suffix, may be linear in the input, and it seems necessary to do the traversal for each state.

To overcome these problems, during the breadth first traversal of the trie we maintain a queue. The queue contains triples with the following entries:

1. the state to be processed – q,
2. the state p' which is reached by traversing the transducer with the string obtained by removing the first symbol, or the string on the path from the last final state, depending on the case, and
3. a pointer to the representation of γ'_q.

In the algorithm, triples are calculated in constant time from predecessor triples, using the above equivalences.

We give the pseudo-code of the algorithm in a Pascal-like language (like the language used in [AHU83]). We assume that an implementation for the Abstract Data Type (ADT) "Transducer State" is given. Later we presume that NULL is the null constant for an arbitrary abstract data type. For the implementation of the efficient output string representation we introduce a new type called `OutString`. This type represents a record of either a char, a string, or a pair of pointers to other Outstrings.

Now we are ready to present the pseudo-code of our algorithm.

```
program Dictionary_Local_Extension(input, output);
const MAXSTATES = 1000000;
type
   QueueRec = record q,p' : STATE; γ' : ↑OutString; end;
   StrType = (letter, out, conc);
   POutString = ↑ OutString;
   OutString = record case StrType of
      letter : (C:char);
      out : (S:string);
       conc : (S1, S2 : POutString);
    end;
 var
    q,q1,q0 : STATE;
    Queue : array [1..MAXSTATES] of QueueRec;
    queue_begin, queue_end, i, j : integer;
    c : char;
    EmptyString : POutString;

 function ConcatOutStrings (S1, S2 : POutString) : POutString;
 begin
    if (S1 = EmptyString) and (S2 = EmptyString) then
       ConcatOutStrings := EmptyString
    else if (S1 = EmptyString) then
       ConcatOutStrings := S2
    else if (S2 = EmptyString) then
       ConcatOutStrings := S1
    else begin
       new(ConcatOutStrings);
       ConcatOutStrings↑.S1 := S1;
       ConcatOutStrings↑.S2 := S2;
    end;
 end;

 function StringToOutString (S : string) : POutString;
 begin
    new(StringToOutString);
    StringToOutString↑.S := S;
 end;

 function CharToOutString (C : char) : POutString;
 begin
    new(CharToOutString);
    CharToOutString↑.C := C;
 end;

 procedure Complement_State(q : STATE, p' : STATE, γ' : POutString);
 begin
    SET_FINAL(q,true);
```

```
    SET_STATE_OUTPUT(q,ConcatOutStrings(γ',STATE_OUTPUT(p')));
    for c := FIRST_CHAR to LAST_CHAR do
        if TRANSITION(q,c) = NULL then begin
            SET_TRANSITION(q,c,TRANSITION(p',c));
            SET_OUTPUT(q,c,ConcatOutStrings(γ',OUTPUT(p',c));
        end else begin
            if FINAL(TRANSITION(q,c)) then
                Queue[queue_end] :=
                           [TRANSITION(q,c),q0,STATE_OUTPUT(TRANSITION(q,c))];
            else
                Queue[queue_end] :=
      [TRANSITION(q,c),TRANSITION(p',c),ConcatOutStrings(γ',OUTPUT(p',c))];
            queue_end := queue_end + 1;
        end; {if}
end; {Complement_State}

begin
    EmptyString := StringToOutString('');
    q0 := NEW_STATE;
    while not eof(input) do begin
        readln(input,CurrentWord,CurrentOutput);
        i := 1;
        q := q0;
        while (i<=length(CurrentWord))
             and (TRANSITION(q,CurrentWord[i]) <> NULL) do begin
            q := TRANSITION(q,CurrentWord[i]);
            i := i + 1;
        end;
        if i<=length(CurrentWord) then begin
            for j := i to length(CurrentWord) do begin
                q1 := NEWSTATE;
                SET_TRANSITION(q,CurrentWord[j],q1);
                SET_OUTPUT(q,CurrentWord[j],EmptyString);
                q := q1;
            end;
        end;
        SET_FINAL(q,true);
        SET_STATE_OUTPUT(q,StringToOutString(CurrentOutput));
    end; {while}

    queue_begin := 1;
    queue_end := 1;
    SET_FINAL(q0,true);
    SET_STATE_OUTPUT(q0,EmptyString);
    for c := FIRST_CHAR to LAST_CHAR do
        if TRANSITION(q0,c) = NULL) then begin
            SET_TRANSITION(q0,c,q0));
            SET_OUTPUT(q0,c,CharToOutString(c));
        end else
            if FINAL(TRANSITION(q0,c)) then
                Queue[queue_end] := [TRANSITION(q0,c), q0, STATE_OUTPUT(TRANSITION(q0,c))];
            else
                Queue[queue_end] := [TRANSITION(q0,c), q0, CharToOutString(c)];
            queue_end := queue_end + 1;
```

```
94          end; {if}
95       end; {for}
96       while queue_begin<queue_end do begin
97          Complement_State(Queue[queue_begin].q,
                              Queue[queue_begin].p',Queue[queue_begin].γ');
98          queue_begin := queue_begin + 1;
99       end; {while}

100      PRINT_TRANSDUCER(output,q0);
101   end.
```

The first three functions in the code are defining the construction procedures for OutStrings. The first of the three functions – `ConcatOutStrings` – checks the two parameters. If both of them point to the empty string, then the empty string is returned. If one of the parameters points to the empty string, then a pointer to the other one is returned. In the remaining case, the function returns a pointer to a new structure with exactly two pointers pointing to the two input parameters respectively. The second function – `StringToOutString` – returns a pointer to a structure containing a given input string. The last of the three functions – `CharToOutString` returns a pointer to a structure containing a given input character.

The algorithm first constructs (lines 58-79) the transducer trie by reading the input dictionary line by line. Afterwards we proceed to the second phase. In order to process the states in ascending distance from the start we organize a queue with triples of the form described above. In lines 80–95 we first initialize the queue and process the starting state. This is done be setting its output to the empty string and adding loops for all symbols σ of the alphabet for which there are no outgoing transitions from the starting state. The transition outputs, represented with pointers, are identical to the corresponding input symbol σ (line 87). If there is already an outgoing transition with the given symbol σ, then we put the following triple in the queue: the state q reached from q_0 with σ, the starting state (here playing the role of p'), and the representation for γ', which here is either σ (line 92) or the output of q if this state represents a final state of the trie (line 90). Afterwards in lines 96–99 we proceed by processing each triple of the queue. This is done using the function *Complement_State*.

In the procedure *Complement_State* we set the current state q as final and define its output. The next loop in lines 46–56 treats all alphabet symbols. If there is no outgoing transition from the current state q with the given symbol σ, we introduce a new transition from q to the state p'_{new} reached from p' with σ. The output is set to the concatenation of the current γ' with the output of the transition from p' with σ. In the other case we push a new triple to the queue consisting of (1) the state q_{new} reached from q with σ, (2) the state p'_{new} reached with σ from p' and (3) the current γ' concatenated with the output of the transition from p' with σ, or the state output of q_{new} if this state represents a final state of the trie. This completes the *Complement_State* procedure.

Clearly the above algorithm implements the method described in the previous sections. Reading the input dictionary takes time proportional to the size of the dictionary. The procedure for building the trie has time complexity proportional to the number of trie states. The number of steps required for processing a single state in the *Complement_State* procedure is proportional to the alphabet size. The space requirements for a state are proportional to the alphabet size as well. Thus the space and time complexity of the given algorithm is $O(|D| + m + |\Sigma| \cdot m)$, where m is the number of states in the trie transducer. Obviously $m < \Sigma_{\langle\alpha_i,\beta_i\rangle\in D}|\alpha_i|$ thus the time complexity of the algorithm is proportional to the size of the dictionary for a fixed alphabet.

Alphabet size	Dictionary size	Entry size 10	Entry size 30	Entry size 100
5	1000	1.61x	1.69x	2.02x
5	10000	1.35x	1.34x	1.50x
5	100000	1.21x	1.25x	1.27x
30	1000	1.31x	1.40x	1.17x
30	10000	1.33x	1.39x	1.44x
30	100000	1.22x	1.23x	—-
60	1000	1.31x	1.17x	1.04x
60	10000	1.29x	1.31x	1.38x
60	100000	1.29x	—-	—-

Table 1: Results for dictionary rewriting by random dictionaries, speed-up factors (ratio of times) w.r.t. naive algorithm.

7 Empirical Results

We implemented both the algorithm described in Section 4 and the naive algorithm described in the introduction. For the latter implementation, a trie is used to encode the dictionary. In this way we may efficiently check if a string is a prefix of a dictionary word. When the naive algorithm is applied to a text, any new input symbol gives rise to a single transition step in the trie. We obtain an efficient realization since we have to go back only in the cases where we went too far in the trie.

To compare both methods,we made a series of experiments, varying the size of the input alphabet, the number of dictionary entries, and the average length of dictionary entries. All dictionary words were randomly generated, varying both the length and the form using a regular distribution. We then measured the time for rewriting a randomly generated text. Table 1 presents the speed-up factor (ratio of times) of our method compared to the naive algorithm.

The algorithm is currently applied in the context of OCR-correction [SRSM03a, SRSM03b], using rewrite dictionaries that rewrite erroneous words into correction suggestions. We successfully compiled transducers for a variety of rewrite dictionaries. For the experiments we used a conventional PC (Intel(R), Pentium III, CPU 0.5 GHz with 527,827 KB main memory). The largest dictionary had $111,071$ entries. The number of states of the transducer was $454,151$. The number of transitions was $454,151 \times 44$ (alphabet size 44). The construction time for the transducer was 68.91 seconds. Using the constructed transducer, a text with 134 MB (18,470,101 tokens) was rewritten in 33.7 seconds.

8 Related work

Applications of finite state transducers for different problems of natural language processing are described, e.g., in [KK94, Moh96, Moh97, RS97]. Rewrite dictionaries as discussed in this paper can be considered as a special form of rewriting rules in the sense of [KK94]. Ignoring weighted versions [Moh01], such a rewriting rule has the form $\alpha \mapsto \beta/\lambda\_\rho$. The rule expresses that occurrences of α that are immediately preceded by an occurrence of λ (left context) and directly followed by an occurrence of ρ (right context) are replaced by β. Here α, β, λ and ρ are regular expressions, which implies that in general we have a proper rewriting relation (as opposed to a rewriting function). In order to formally define the rewrite relation, distinct modes of applying the rule must be distinguished: left-to-right versus right-to-left application, optional versus obligatory application. Kaplan and

Kay [KK94] proved that all these rewrite relations are rational, i.e., they can be encoded by a finite state transducer. Kartunnen [Kar97] describes a similar approach. In later improvements, also the difference between longest match versus shortest match strategies is considered [Kar96]. For the leftmost longest match strategy, the construction described in [KK94] has been improved by Gerdemann and van Noord [GvN99]. Another improvement of [KK94], applicable also for weighted rewrite rules, is presented by Mohri et al. [MS96, MPR96].

Theoretically the above techniques can be applied for the construction of a subsequential transducer that represents D^{lmls}. This would lead, however, to a much more indirect and complex construction, which in practice could be applied only for very small dictionaries.

Roche and Schabes [RS95] show how the contextual rewrite rules used in Brill's tagger [Bri92] can be compiled into subsequential finite state transducers. The rules have the form $w \mapsto v/\lambda\_\rho$ where w, v are tags and the context conditions λ and ρ restrict the previous and next tags. Hence the rules represent a special form of the above rewrite rules. Roche and Schabes show that any rewrite relation represented as a finite state transducer $\mathcal{T}_1$ can be transformed into a transducer $\mathcal{T}_2$ that represents the "local extension" of $\mathcal{T}_1$: If $\mathcal{T}_1$ produces output w for input v, and if u and x are any strings where $\mathcal{T}_1$ does not apply, then $\mathcal{T}_2$ rewrites uvx into uwx. Hence, $\mathcal{T}_2$ is used to perform all possible $\mathcal{T}_1$-replacements inside a given text. Recall that our transducer uses the more restricted leftmost-largest replacement strategy with skips. From the theoretical results proved at the end of [RS95] it follows that the lmls-extension D^{lmls} of a finite rewrite dictionary D can be represented as a subsequential transducer. But at this point no explicite construction is given. Roche and Schabes also describe an algorithm for determinization of finite-state-transducers, based on ideas from Mohri.

The problem considered in the present paper is more restricted in the sense that our rewrite rules ignore contextual conditions. Furthermore, we only consider one fixed application mode (obligatory leftmost-largest replacement strategy with skips). For this specific problem, a simple and efficient solution is given.

The problem of how to represent a given rewrite dictionary D (as opposed to the canonical lmls-extension D^{lmls}) as a minimal subsequential transducer is discussed in [Moh96]. This construction provides a very fast dictionary lookup as well as compact storage. An on-line algorithm for constructing the minimal subsequential transducer for a given dictionary is presented in [MM01], building up on ideas from [Mih98, DMWW00].

9 Conclusion

In this paper we showed how to compute for a given rewrite dictionary, D, a subsequential finite state transducer $\mathcal{T}$ that represents the canonical lmls-extension D^{lmls} of D and thus can be used for text-rewriting based on D using the leftmost-largest replacement strategy with skips. The resulting speed of the text-rewriting procedure can hardly be improved since the text is rewritten in one linear scan of the input. Theoretically, the size of the output functions of $\mathcal{T}$ can become cubic in the size of the input dictionary D when using a standard implementation. In practice, however, the size of the output functions is typically small and the real limitations are caused by the number of transducer states and transitions (which is linear in the size of D for fixed alphabet size). We also showed how special encoding techniques for output strings help to compute $\mathcal{T}$ in time and space linear in the size of D.

Though the entries of our rewrite dictionaries represent non-contextual rewrite rules, a contextual rule of the form $v \mapsto w/l\_r$ (where v, w, l, r are strings) can always be represented as a non-contextual rule $lvr \mapsto lwr$. It should be noted, however, that this translation affects the notion of a leftmost-largest replacement strategy with skips. As an example, consider the two contextual rules $a \mapsto 1/bb\_ab$ and $aa \mapsto 22/b\_b$, which are respectively translated into $bbaab \mapsto bb1ab$ and $baab \mapsto b22b$. Using the contextual version of the rules, the text $bbaab$ is translated into $bb22b$ under the leftmost-largest strategy. In contrast, using the translated version we obtain $bb1ab$.

In future work we intend to investigate how other replacement strategies can be realized with modified transducer constructions.

References

[Abn97] Steven Abney. Partial parsing via finite-state cascades. *Journal of Natural Language Engineering*, 2(3):337–344, 1997.

[AC75] Alfred Aho and Margarete Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18:333–340, 1975.

[AHB+93] Douglas Appelt, Jerry R. Hobbs, John Bear, David Israel, and Mabry Tyson. FASTUS: A finite state processor for information extraction from real world text. In *Proc. Int. Joint Conference on Artificial Intelligence*, pages 1172–1178, 1993.

[AHU83] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1983.

[Bri92] Eric Brill. A simple rule-based part of speech tagger. In *Proceedings of the Third Conference on Applied Natural Language Processing*, pages 152–155, Trento, Italy, 1992.

[Bro00] Ralph D. Brown. Example-based machine translation in the pangloss system. In *Proceedings of COLING'96: The 16th International Conference on Computational Linguistics*, pages 169–174, Copenhagen, Denmark, August 2000.

[CHBG01] Leslie Carr, Wendy Hall, Sean Bechhofer, and Carole Goble. Conceptual linking: Ontology-based open hypermedia. In *Proceedings of the 12th International World Wide Web Conference*, pages 334–342, 2001.

[DMWW00] Jan Daciuk, Stoyan Mihov, Bruce Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1):3–16, 2000.

[GvN99] Dale Gerdemann and Gertjan van Noord. Transducers from rewrite rules with backreferences. In *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics (EACL 99)*, pages 126–133, 1999.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[Kar96] Lauri Karttunen. Directed replacement. In Arivind Joshi and Martha Palmer, editors, *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 108–115, Santa Cruz, 1996.

[Kar97] Lauri Karttunen. The replace operator. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 117–147. MIT Press, 1997.

[KK94] Ronald Kaplan and Martin Kay. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–279, 1994.

[Koz97] Dexter C. Kozen. *Automata and Computability.* Springer, New York, Berlin, 1997.

[Kro93] Bob Krovetz. Viewing morphology as an inference process. In *Proceedings of the 16th ACM SIGIR Conference*, pages 191–202, June, July 1993.

[Kuk92] Karen Kukich. Techniques for automatically correcting words in texts. *ACM Computing Surveys*, pages 377–439, 1992.

[Kup92] Julian M. Kupiec. Robust part-of-speech tagging using a hidden markov model. *Computer Speech and Language*, 6:225–242, 1992.

[Lap97] Éric Laporte. Phonetic conversion and phonology. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 407–429. MIT Press, 1997.

[Mih98] Stoyan Mihov. Direct building of minimal automaton for given list. *Annuaire de l'Université de Sofia "St. Kl. Ohridski"*, 91, livre 1, 1998.

[MM95] Petra Maier-Meyer. *Lexikon und automatische Lemmatisierung.* PhD thesis, CIS, Universität München, 1995.

[MM01] Stoyan Mihov and Denis Maurel. Direct construction of minimal acyclic subsequential transducers. In *Proceedings of the Conference on Implementation and Application of Automata CIAA'2000*, number 2088 in LNCS, pages 217–229. Springer, 2001.

[Moh96] Mehryar Mohri. On some applications of finite-state automata theory to natural language processing. *Journal of Natural Language Engineering*, 2:1–20, 1996.

[Moh97] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, 1997.

[Moh01] Mehryar Mohri. Weighted grammar tools: the GRM library. In Jean Claude Junqua and Gertjan van Noord, editors, *Robustness in Language and Speech Technology*, pages 165–186. Kluwer Academic Publishers, The Netherlands, 2001.

[MPR96] Mehryar Mohri, Fernando C.N. Pereira, and Michael Riley. Weighted automata in text and speech processing. In *Proceedings of the 12th biennial European Conference on Artificial Intelligence (ECAI-96), Workshop on Extended Finite State Models of Language*, pages 152–155, Budapest, Hungary, 1996.

[MS96] Mehryar Mohri and Richard Sproat. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Meeting of the Association for Computational Linguistics (ACL'96)*, pages 231–238, Santa Cruz, CA, 1996.

[Por80] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

[RS95] Emmanuel Roche and Yves Schabes. Deterministic part-of-speech tagging with finite state transducers. *Computational Linguistics*, 22(2):227–253, 1995.

[RS97] Emmanuel Roche and Yves Schabes. Introduction. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, pages 1–66. MIT Press, 1997.

[SRSM03a] Christian Strohmaier, Christoph Ringlstetter, Klaus U. Schulz, and Stoyan Mihov. Lexical postcorrection of OCR-results: The web as a dynamic secondary dictionary? In *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR-03*, pages 1133–1137, Edinburgh, 2003.

[SRSM03b] Christian Strohmaier, Christoph Ringlstetter, Klaus U. Schulz, and Stoyan Mihov. A visual and interactive tool for optimizing lexical postcorrection of OCR-results. In *Proceedings of DIAR-03*, Wisconcent, CA, 2003.

[VN00] Stephan Vogel and Hermann Ney. Translation with cascaded finite-state transducers. In *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics, ACL-2000*, pages 23–30, Hong Kong, China, October 2000.