Natural Language Engineering **13** (4): 353–381. © 2006 Cambridge University Press
doi:10.1017/S1351324905004092 First published online 15 February 2006 Printed in the United Kingdom

Efficient dictionary-based text rewriting using subsequential transducers†

S. MIHOV
Institute for Parallel Processing, Bulgarian Academy of Sciences, Bulgaria
e-mail: stoyan@lml.bas.bg

K. U. SCHULZ
CIS, University of Munich, Munich, Germany
e-mail: schulz@cis.uni-muenchen.de

(*Received 30 September 2004; revised 16 August 2005*)

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 rewriting 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 rewriting dictionary D, a subsequential transducer $\mathscr{T}$ that accepts any 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 a deterministic manner 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 (Roche and Schabes 1997): using a collection of background rules and resources, a given input text has to be rewritten to another form. Examples include phonetic conversion (Laporte 1997), stemming and lemmatization (Porter 1980; Krovetz 1993; Maier-Meyer 1995) and related forms of text normalization, tagging (Brill 1992; Kupiec 92), shallow parsing and information extraction (Appelt *et al.* 1993; Abney 1997), semantic annotation and automated hyperlinking (Carr *et al.* 2001), automated text correction (Kukich 1992), translation (Brown 2000) and others.

† This work was supported by VolkswagenStiftung and DFG.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

A simple, yet very useful variant of text rewriting is based on rewriting dictionaries that list strings that have to be replaced ("originals") plus their substitutes. Rewriting 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. Rewriting dictionaries in memory-based translation list words, phrases and sentences of a source language together with translations into a target language. In automated hyperlinking, rewriting dictionaries translate natural language expressions into ontological concepts, links and pointers. For shallow parsing and information extraction, rewriting dictionaries may be used that attach syntactic categories and semantic labels to words and 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 (Abney 1997; Vogel and Ney 2000), 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-longest match" replacement strategy with skips. Basically this means that there is a preference for replacing originals that start first in the natural (left-to-right) 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 *naïve algorithm* for text rewriting with a given rewriting dictionary D under the leftmost-longest 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 naïve algorithm is $O(m \cdot |t| + |t'|)$, where m is the maximal length of an original string in the rewriting dictionary, $|t|$ is the length of the text t and t' denotes the resulting output text. When using the naïve algorithm, such backward steps cannot be avoided. As a simple example, assume that the dictionary contains the pairs

$$(\textit{finite state automaton} \mapsto FSA)$$
$$(\textit{state transition diagram} \mapsto STD).$$

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

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

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

be extended to "*finite state automaton*". Furthermore, the naïve 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 rewriting dictionary may represent proper prefixes of other entries, as in the following example.

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

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-longest match strategy with skips. The new algorithm does not use backward steps, and no memory device is needed: in our approach we compute, given the rewriting dictionary D, a subsequential finite state transducer $\mathscr{T}$ that always produces the desired transformation result in one deterministic run when applied to an input text t.[2]

The construction of $\mathscr{T}$ represents a preprocessing step that is performed once. The number of states and transitions of $\mathscr{T}$ is linear in the size $|D|$ of D. If implemented in a naïve 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 $\mathscr{T}$ 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 (Aho and Corasick 1975)[3].

The advantage that is gained from the new method is twofold. Our algorithm for dictionary-based text rewriting is more efficient than the naïve algorithm. Furthermore, using standard tools, the above mentioned transducer $\mathscr{T}$ 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 above mentioned applications – can be naturally integrated into more ambitious text transformation systems. The method has been implemented and successfully tested with rewriting dictionaries with over 200,000 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 rewriting 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

[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. Mihov and Maurel, 2001).

[3] Due to the use of the leftmost-longest match strategy with skips, which is less natural for pattern matching, our algorithm generalizes a special variant of the Aho-Corasick algorithm usually not described in the literature. Hence, as it comes to details, a direct comparison is difficult.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

details. In Section 7 we evaluate the method. Section 8 comments on related work, focusing 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, Motwani and Ullman (2000) or Kozen (1997). More background on transducers and their use can be found in Roche and Schabes (1997). 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.

2.1 Dictionary-based text rewriting with leftmost-longest-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-longest* 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 a unique leftmost-longest occurrence, denoted $lml_D(t)$.

Example 2.4
Consider the rewriting dictionary D given by the mapping

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

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

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-longest-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 a 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:

$$\langle \epsilon, ab, cbbbabccb\rangle$$
$$\langle ab, c, bbbabccb\rangle$$
$$\langle abcbb, babc, cb\rangle$$
$$\langle abcbbbabc, c, b\rangle$$

Definition 2.8
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 2.9
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 2.10
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 2.11
Let $a_1 \in \Sigma$. Assume that no prefix of $t = a_1 \cdot t'$ is in the domain of D. Then $D^{lmls}(t) = a_1 \cdot D^{lmls}(t')$.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

2.2 Subsequential transducers

Definition 2.12
A *subsequential transducer* is a tuple $\mathscr{T} = \langle \Sigma, \Delta, Q, q_0, F, \delta, \lambda, \Psi \rangle$, 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:

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

Since δ is partial, the same holds for δ^*, the definition is restricted to input strings that can be consumed by the transducer. Similarly we define the *extended transition output function* $\lambda^* : Q \times \Sigma^* \to \Delta^*$ in the following way:

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

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

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

Definition 2.13
A subsequential transducer $\mathscr{T}$ is said to *represent* the function $f : \Sigma^* \to \Delta^*$ iff $O_{\mathscr{T}} = f$.

3 Informal description of the algorithm

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

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

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

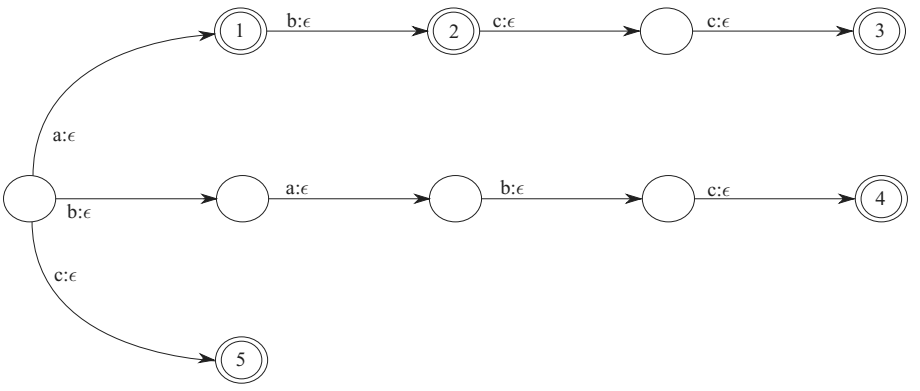


Fig. 1. Subsequential transducer trie for the rewriting dictionary D introduced in Example 2.4.

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-shaped transducer that represents (in the sense of Definition 2.13) the dictionary. To this end, we compute the trie for the domain $Dom(D)$ 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 emits the output $D(w)$. Figure 1 shows the subsequential transducer trie $\mathscr{T}_D$ for the rewriting dictionary D introduced in Example 2.4.

Obviously the number of states in the resulting transducer $\mathscr{T}_D$ does not exceed the total number of symbols in the list of all dictionary entries. In what follows, if q is any state of $\mathscr{T}_D$, then α_q denotes the sequence of input transition labels of $\mathscr{T}_D$ for the unique path leading from the start state q_0 to q.

Second Step. Given $\mathscr{T}_D$ we construct a transducer $\mathscr{T}$ that represents (in the sense of Definition 2.13) the canonical lmls-extension D^{lmls} of D. The set of states of $\mathscr{T}_D$ is not modified. Similarly the output of the final states of $\mathscr{T}_D$ is not changed. Furthermore, each transition of $\mathscr{T}_D$ remains a transition of $\mathscr{T}$. Hence, $\mathscr{T}_D$ is only extended. Since $\mathscr{T}$ 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[4] 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 emitted if we reach q at the *end* of the input string.

To introduce missing transitions and to define suitable outputs, we visit all states q of $\mathscr{T}_D$. For any state p, let $d(q_0, p)$ denote the number of transitions in $\mathscr{T}_D$ that are needed to reach p, starting from q_0. When we visit a state p we always assume

[4] Failure transitions describe how to proceed with a letter σ where the naïve algorithm cannot continue with the present forward move and has to continue from a new offset point. Since in the transducer approach the text is scanned in one traveral, strictly proceeding from left to right, a failure transition of the transducer encodes several steps of the naïve algorithm, see below.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

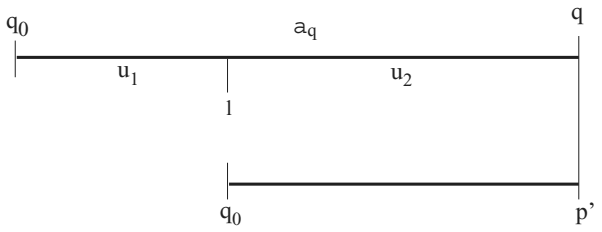


Fig. 2. Scanning a string α_q representing a prefix of an entry of D.

that all states p' such that $d(q_0, p') < d(q_0, p)$ have been visited before. We 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 $q_0 = p_1$.

We begin with q_0. When reading a text t, we want to be in state q_0 at text position k only if k corresponds to an offset point where the naïve algorithm starts a new subscan. This means that position k is not inside a relevant left-most occurrence of an entry of D. Furthermore, the output for the prefix of the text ending at k is completed. State q_0 is declared to be final and the output is the empty string. For all letters σ where we do not have a transition from q_0 in $\mathscr{T}_D$, we add a transition σ with output σ to the transducer. Recall that the naïve algorithm outputs σ and immediately starts a new scan from position $k + 1$. In our example, we have to add a transition for input letter d.

Definition of state outputs

The state output for a state $q \neq q_0$, reached with the non-empty string α_q, encodes the translation result $D^{lmls}(\alpha_q)$. A compositional description of $D^{lmls}(\alpha_q)$ can be obtained, analyzing how the naïve algorithm processes the string α_q. Since α_q represents a prefix of an entry of D, the algorithm proceeds in single forward move until the end of the string is found. At this point, a suitable output v is produced and the algorithm continues from a new offset point $l > 0$. The offset point l defines a partitioning of $\alpha_q = u_1 u_2$ into substrings u_1, u_2 in the obvious way, cf. Figure 2.

Both v and l depend on the form of α_q. If some prefix of α_q represents an entry of D, and if u_1 is the maximal prefix of this form, then $v = D(u_1)$ and l marks the end of u_1 (for $u_1 = \alpha_q$, position l marks the end of the text.) Otherwise, if no prefix of α_q is an entry of D, then v is the first letter of α_q and $l = k + 1$. Note that in both cases we have $v = D^{lmls}(u_1)$.

The output v represents a first component of the state output $D^{lmls}(\alpha_q)$ for q. The second component of $D^{lmls}(\alpha_q)$ corresponds to $D^{lmls}(u_2)$, the output of the naïve algorithm when processing u_2. The important point to note is that u_2 is shorter than α_q. This guarantees that starting from q_0, all transitions for input u_2 are defined, and also the state output for the final state p' that is reached is already encoded in the present transducer. This leads to the following compositional description of the state output for q:

$$\begin{aligned}&\text{state output for } q\\ &= D^{lmls}(\alpha_q)\\ &= \text{output of naïve algorithm for } \alpha_q\end{aligned}$$

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

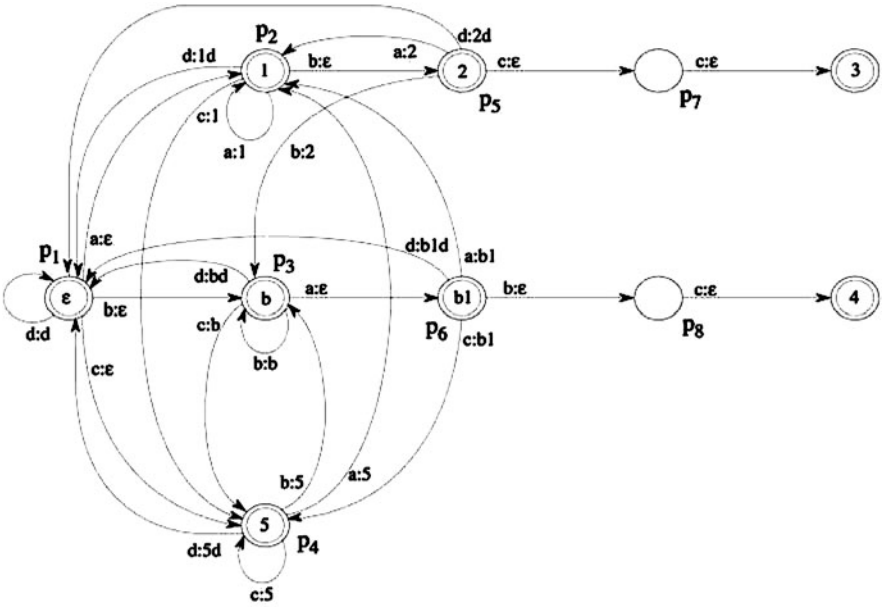


Fig. 3. Construction of the subsequential transducer for the canonical lmls extension of the dictionary given in Example 2.4. Situation after states $q_0 = p_1, \ldots, p_6$ have been visited.

$$= D^{lmls}(u_1) \cdot D^{lmls}(u_2)$$
$$= D^{lmls}(u_1) \cdot$$
transition output of transducer for input u_2 starting from q_0 ·
state output of transducer for p'.

Using the partially defined transducer, all components are already available.

As an example, consider the situation in Figure 3, where states $q_0 = p_1, \ldots, p_6$ have been visited. For $q = p_7$ we have $\alpha_q = abc$, $u_1 = ab$, $u_2 = c$. The state p' reached with $u_2 = c$ is $p' = p_4$. Following the above equations, the state output for p_7 is $D(ab) \cdot \epsilon \cdot 5 = 25$. For $q = p_8$ we have $\alpha_q = bab$, $u_1 = b$, $u_2 = ab$, $p' = p_5$. Hence the output of p_8 is $b \cdot \epsilon \cdot 2 = b2$.

Definition of target states and outputs for failure transitions. Consider a state q and a symbol σ where we do not have a transition from q with σ in $\mathcal{T}_D$. We first explain how to define the target state of the transition. When scanning $\alpha_q\sigma$, the naïve algorithm proceeds in a forward move until it finds the letter σ. Since by assumption $\alpha_q\sigma$ is not a prefix of an entry of D, the algorithm produces some output v and continues from a new offset point $l > 0$. The offset point l defines a partition of $\alpha_q = u_1 \cdot u_2$ in the obvious way. The situation is depicted in Figure 5. As above we have $v = D^{lmls}(u_1)$. Now assume that we process input u_2 (resp. $u_2\sigma$) with the transducer defined so far. Since u_2 is shorter than α_q all needed transitions are already defined. We arrive at state p' (resp. q'). In this situation, q' represents the target state for the failure transition from q with σ. Note that in general the path from q_0 to p' and q' may contain several failure transitions.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

It may be shown that the string $\alpha_{q'}$ represents a suffix of $\alpha_q\sigma = w \cdot \alpha_{q'}$ (cf. Part 1 of Lemma 5.2 below). The translation result of the corresponding prefix w, $D^{lmls}(w)$, represents the output of the failure transition. As above, for the algorithm we need a compositional description of $D^{lmls}(w)$.

Clearly, u_1 is a prefix of $w = u_1 \cdot w_2$. The choice of l as a backtracking point guarantees that $D^{lmls}(w) = D^{lmls}(u_1) \cdot D^{lmls}(w_2)$. We shall see in Lemma 5.2, Part 2, that $D^{lmls}(w_2)$ is the transition output of the transducer for input $w_2 \cdot \alpha_{q'} = u_2 \cdot \sigma$. Hence we obtain the following description of the output for the transition with letter σ from state q:

Output for transition with σ from q
$= D^{lmls}(w)$
$= D^{lmls}(u_1) \cdot D^{lmls}(w_2)$
$= D^{lmls}(u_1) \cdot$
transition output of transducer for input u_2 starting from $q_0 \cdot$
output of transition with σ from p'

As above, using the partially defined transducer, all components are already available.

Again consider the situation in Figure 3, where states $q_0 = p_1, \ldots, p_6$ have been visited. For $q = p_7$ and $\sigma = d$ we have $\alpha_q = abc$, $u_1 = ab$, $u_2 = c$. The state p' reached with $u_2 = c$ (resp. $u_2\sigma$) is p_4 (resp. q_0). Hence the failure transition for q, d leads to q_0. Following the above equations, the transition output for the transition is $2 \cdot \epsilon \cdot 5d = 25d$. For $q = p_8$ and $\sigma = d$ we have $\alpha_q = bab$, $u_1 = b$, $u_2 = ab$. The state p' reached with u_2 (resp. $u_2\sigma$) is $p' = p_5$ (resp. q_0). Hence the failure transition for q, d again leads to q_0. The transition output is $b \cdot \epsilon \cdot 2d = b2d$.

The transducer $\mathscr{T}$ that is obtained after visiting all states of $\mathscr{T}_D$ is shown in Figure 4.

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.

4.1 Construction of the subsequential transducer trie for the dictionary

The following construction of the subsequential transducer trie for a given rewriting 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 $\mathscr{T}_0$, which is defined as the tuple

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

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

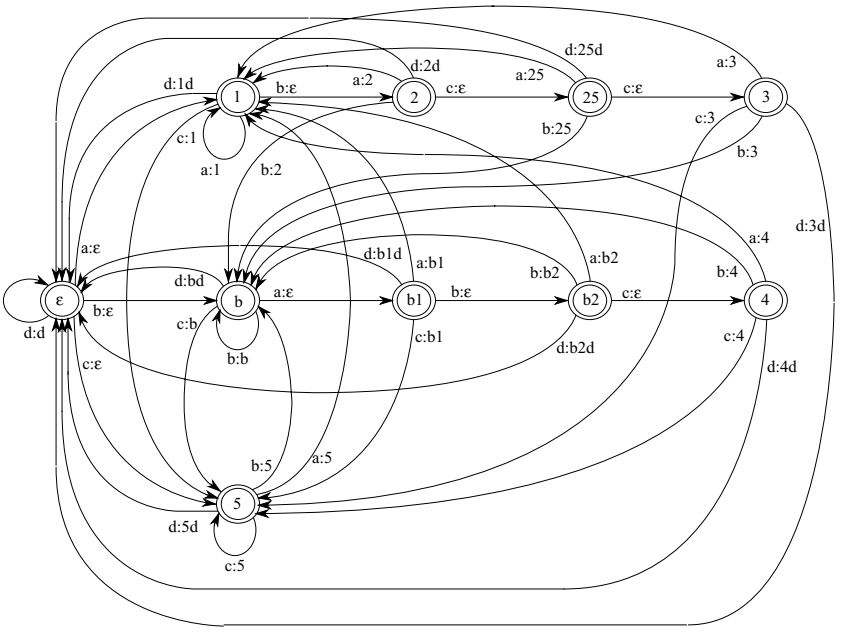


Fig. 4. The subsequential transducer for the canonical lmls extension of the dictionary given in Example 2.4.

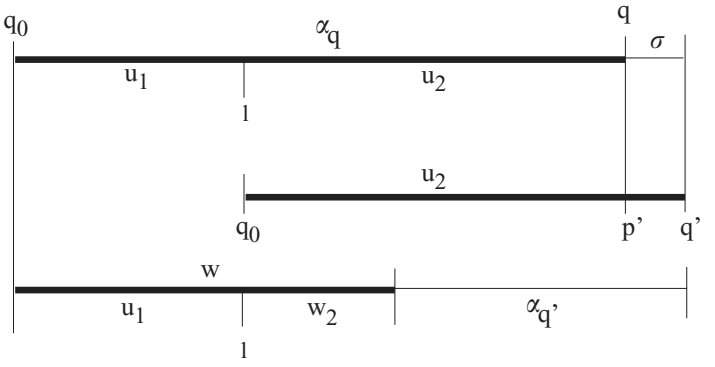


Fig. 5. Scanning substring $\alpha_q \cdot \sigma$.

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

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

representing the dictionary $D_{k-1} = \{\langle \alpha_i, \beta_i \rangle \in D \mid i \in \{1, 2, \ldots, k-1\}\}$ 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

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

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

where

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

and

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

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

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

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

which represents D. $\mathscr{T}_D$ 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. As before, by α_q we denote the unique string in Σ^* such that $\delta^*_D(q_0, \alpha_q) = q$ and $d(q_0, q) = |\alpha_q|$ denotes the number of transitions in $\mathscr{T}_D$ that are needed to reach q, starting from q_0 ($q \in Q_D$).

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

To define the transducer $\mathscr{T}$ for text rewriting based on the given dictionary D we visit all states of $\mathscr{T}_D$. 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 above, 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 assume that a list $(p_1, p_2, \dots, p_h)$ is given that describes a 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

$$\mathscr{T}'_1 := \langle \Sigma, \Delta, Q_D, q_0, F_D \cup \{q_0\}, \delta'_1, \lambda'_1, \Psi_D \cup \langle q_0, \varepsilon\rangle\rangle$$

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

$$\mathscr{T}'_i = \langle \Sigma, \Delta, Q_D, q_0, F'_i, \delta'_i, \lambda'_i, \Psi'_i\rangle.$$

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

Case 1: Path $q_0, q_1, \dots, q_r$ does not contain a final state of $\mathscr{T}_D$. In this case, $p' := \delta'^*_i(q_0, a_2 a_3 \dots a_r)$ is defined, by choice of the visiting order. Let $\gamma' := a_1 \cdot \lambda'^*_i(q_0, a_2 a_3 \dots a_r)$. We define

$$\mathscr{T}'_{i+1} := \left\langle \Sigma, \Delta, Q_D, q_0, F'_i \cup \{q\}, \delta'_{i+1}, \lambda'_{i+1}, \Psi'_i \cup \langle q, \gamma' \cdot \Psi'_i(p')\rangle\right\rangle$$

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

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: Path $q_0, q_1, \ldots, q_r$ contains a final state of $\mathscr{T}_D$. Let q_f denote the last final state of $\mathscr{T}_D$ on this path.[5] This means that $\delta^*_D(q_0, a_1a_2\ldots a_f) = q_f \in F_D$ 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

$$\mathscr{T}_{i+1} = \left\langle \Sigma, \Delta, Q_D, q_0, F'_i \cup \{q\}, \delta'_{i+1}, \lambda'_{i+1}, \Psi'_i \cup \langle q, \gamma'\cdot\Psi'_i(p')\rangle\right\rangle$$

where

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

The transducer

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

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 rewriting dictionary. Then the transducer $\mathscr{T}$ computed by the above algorithm represents the canonical lmls-extension D^{lmls} of D, in the sense of Definition 2.13.

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. $D^{lmls}(t) = \lambda^*(q_0, t)\cdot\Psi(q)$.

[5] Note that we can have $q_f = q$. In this case we will have $\Psi'_{i+1}(q) = \Psi_D(q)$.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

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

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

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

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 $\mathscr{T}_D$ from q_0 to q'.

Subcase 1.1. First assume that there is no final state of $\mathscr{T}_D$ 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 2.11 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 $\mathscr{T}_D$ 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-longest occurrence within $\alpha' = a_1 \dots a_r$ it follows from Corollary 2.10 (or from the recursive Definition 2.8) that $D^{lmls}(\alpha') = D^{lmls}(a_1 \dots a_r) = D(a_1 \dots a_f) \cdot D^{lmls}(a_{f+1} \dots a_r)$. Since the output of the final states of $\mathscr{T}_D$ is not changed in $\mathscr{T}$ 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 $\mathscr{T}_D$ on the path from q_0 to q. Let $a_1 \dots a_r$ denote the sequence of labels on the path from q_0

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

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 2.11 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 $D^{lmls}(\alpha'') = \lambda^*(q_0, a_2 \dots a_r \sigma_{n+1})$. From the definition of the λ'_i-functions we get $a_1 \cdot \lambda^*(q_0, a_2 \dots a_r \sigma_{n+1}) = \lambda(q, \sigma_{n+1})$, hence 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 2.11 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 $\mathscr{T}_D$ 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, let q_f denote the last final state on the path and let $a_1 \dots a_f$ denote the sequence of labels of the path from q_0 to q_f.

In this case we have $q' = \delta(q, \sigma_{n+1}) = \delta^*(q_0, a_{f+1} \dots a_r \sigma_{n+1})$ and $\lambda(q, \sigma_{n+1}) = \Psi_D(q_f) \cdot \lambda^*(q_0, a_{f+1} \dots a_r \sigma_{n+1})$. Let α' denote the unique word such that $\delta_D(q_0, \alpha') = q'$. The induction hypothesis applied to $a_{f+1} \dots a_r \sigma_{n+1}$ shows that α' is a suffix of $a_{f+1} \dots a_r \sigma_{n+1} = \alpha'' \cdot \alpha'$, hence α' is a suffix of $t' = t_0 \cdot a_1 \dots a_f \cdot \alpha'' \cdot \alpha'$, which shows that Claim 1 holds. Let $t'_0 := t_0 \cdot a_1 \dots a_f \cdot \alpha''$. Applying the induction hypothesis (Claim 3, $\alpha_0 := a_1 \dots a_f \cdot \alpha''$) we get $D^{lmls}(t'_0) = D^{lmls}(t_0) \cdot D^{lmls}(a_1 \dots a_f \cdot \alpha'')$. Our assumption implies that $a_1 \dots a_f$ represents a longest leftmost occurrence within $a_1 \dots a_f \cdot \alpha''$. Corollary 2.10 shows that $D^{lmls}(a_1 \dots a_f \cdot \alpha'') = D^{lmls}(a_1 \dots a_f) \cdot D^{lmls}(\alpha'') = \Psi_D(q_f) \cdot D^{lmls}(\alpha'')$, 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 $a_{f+1} \dots a_r \sigma_{n+1}$ yields $D^{lmls}(\alpha'') = \lambda^*(q_0, a_{f+1} \dots a_r \sigma_{n+1})$. We obtain

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

which proves Claim 2.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

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 $D^{lmls}(t'_0 \cdot \alpha_0) = D^{lmls}(t_0 \cdot a_1 \dots a_f \cdot \alpha'' \cdot \alpha_0) = D^{lmls}(t_0) \cdot D^{lmls}(a_1 \dots a_f \cdot \alpha'' \cdot \alpha_0)$. Our assumption and Corollary 2.11 show that $D^{lmls}(a_1 \dots a_f \cdot \alpha'' \cdot \alpha_0) = \Psi_D(q_f) \cdot D^{lmls}(\alpha'' \cdot \alpha_0)$. Induction hypothesis (3) for $a_{f+1} \dots a_r \sigma_{n+1}$ implies that $D^{lmls}(\alpha'' \cdot \alpha_0) = D^{lmls}(\alpha'') \cdot D^{lmls}(\alpha_0)$. Hence we have $D^{lmls}(t'_0 \cdot \alpha_0) = D^{lmls}(t_0) \cdot \Psi_D(q_f) \cdot D^{lmls}(\alpha'') \cdot D^{lmls}(\alpha_0)$. 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 $q' = \delta^*(q_0, a_{f+1} \dots a_r \sigma_{n+1})$ the induction hypothesis for $a_{f+1} \dots a_r \sigma_{n+1}$ yields $D^{lmls}(\alpha') = \Psi(q')$, which shows that Claim 4 is satisfied. □

6 Complexity analysis and implementation details

In this section we show that a naïve representation of the transducer output function needs cubic space. We then introduce a pointer-based representation with linear space requirement. Even with this step, a straightforward implementation would lead to quadratic running time. In the last subsection we present a queue-based iterative transducer construction which computes the intended rewriting transducer in linear time and space.

6.1 Cubic space complexity of transducer output function

It is simple to see that the number of states of the resulting transducer $\mathscr{T}$ for an input dictionary D is bounded by the size (number of symbols) of the domain of D. In fact, states of $\mathscr{T}$ represent states of the transducer trie, which correspond to prefixes of entries of D. Since the alphabet is fixed, the number of transitions of $\mathscr{T}$ 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 rewriting dictionary D of the form

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

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 $n = 4$, which illustrates this phenomenon, is shown in Figure 6.

The following proposition shows that the size of the output functions is at most cubic in the size of D. As before, for any state q, by α_q we denote the unique word in Σ^* such that $\delta^*_D(q_0, \alpha_q) = q$.

Proposition 6.2
Let λ and Ψ respectively denote the transition and state output function of $\mathscr{T}$, let $q \in Q, \sigma \in \Sigma$. Let $i = |\alpha_q|$. Then

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

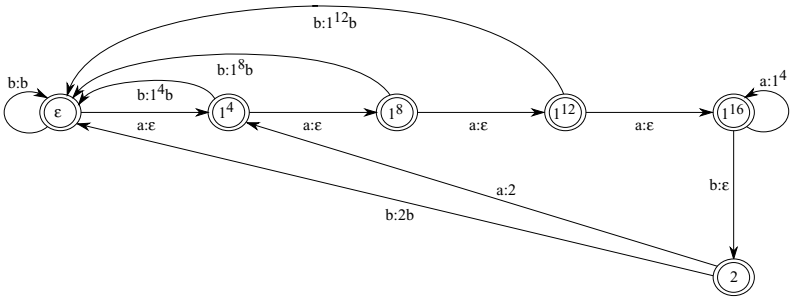


Fig. 6. The subsequential transducer for the canonical lmls-extension of the dictionary given in Example 6.1.

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

6.2 *Linear representation of the output function*

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 $\mathscr{T}'_{i+1}$ (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 of the transducer construction (cf. Section 4.2), "new" state and transition outputs for q are respectively defined as

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

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

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

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

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

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 conventional string, a symbol 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 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 one points to the record pointed by $\gamma'_{q_{r-1}}$ (Cases 1,2). The second pointer points to the record pointed by $\lambda(\delta^*(q_0, a_x \ldots, a_{r-1}), a_r)$ where $x = 2$ in Case 1 and $x = f + 1$ in Case 2. If $\gamma'_{q_{r-1}}$ or $\lambda(\delta^*(q_0, a_x \ldots, a_{r-1}), a_r)$ is empty,

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

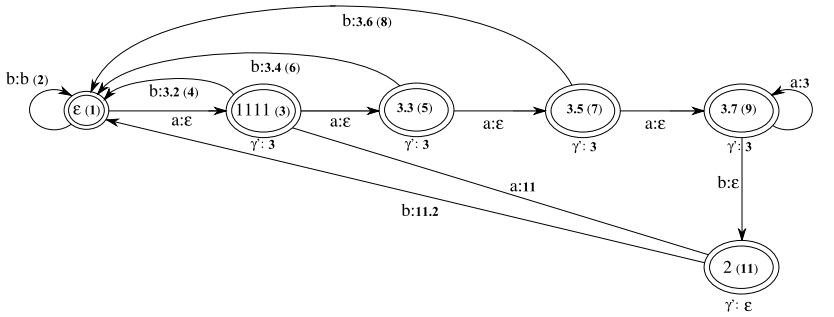


Fig. 7. The subsequential transducer for the canonical lmls-extension of the dictionary given in Example 6.1 with the pointer-based representation of output functions.

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 $\lambda(\delta^*(q_0, a_x \ldots, a_{r-1}), a_r)$ are empty, then γ'_q is redirected to point to the empty word record.

Example 6.3

In Figure 7 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 face denote pointers. Pointer addresses/values are written in brackets. For example, the state reached from the initial state with input a has state output 1111. Pointer **3** points to 1111. The state output of the state q reached with input aa, which is written **3.3** in the pointer-based representation, thus has the form $1111 \cdot 1111$. The output of the transition from q with input letter b is written **3.4**. Decoding pointers we obtain $\mathbf{3.4} = \mathbf{3.3.2} = 1111 \cdot 1111 \cdot b$.

In 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 to produce 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.

6.3 Linear computation of the rewriting transducer

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 naïve 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)

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

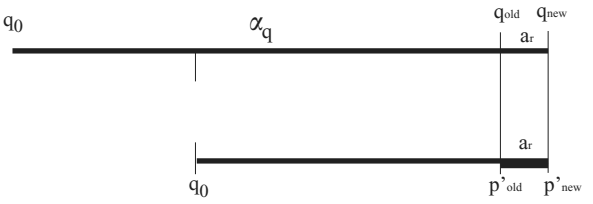


Fig. 8. Calculation of a triple from the predecessor triple.

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 α_q. The length of α_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 equations given in the previous subsection.

Consider the situation presented in Figure 8. The triple $\langle q_{old}, p'_{old}, \gamma'_{old} \rangle$ has been calculated. Reaching state q_{new} with letter a_r in the trie, we want to define the new triple. We distinguish two cases.

Situation I: If q_{new} is not a final state of $\mathscr{T}_D$ (situation depicted in Figure 8), then $p'_{new} = \delta(p'_{old}, a_r)$ and $\gamma'_{new} = \gamma'_{old} \cdot \lambda(p'_{old}, a_r)$. The equation for γ'_{new} follows directly from the above equivalences (cases $r > 1$, $r > f$). The equation for p'_{new} directly follows from the definition of the state p' in Section 4.2. It is important to note that the transition from p'_{old} to p'_{new} may be a failure transition.

Situation II: If q_{new} is a final state in the trie, then we define $p'_{new} = q_0$ and $\gamma'_{new} = \Psi(q_{new})$. Both equations follow directly from the second case distinguished in Section 4.2 where $f = r$.

We give the pseudo-code of the algorithm in a Pascal-like language (like the language used in (Aho *et al.* 1983)). 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.

On Transducer State we will need the following types and operations:

1. FIRST_CHAR, LAST_CHAR – these are the first and the last character in the input alphabet;
2. STATE – a pointer to a structure representing a transducer state;
3. **function** NEW_STATE : STATE; – function returning a new state;

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

4. **function** FINAL(STATE) : boolean; – returns true if the state is final and false otherwise;
5. **procedure** SET_FINAL(STATE, boolean); – defines the finality of the state in accordance with the boolean parameter;
6. **function** TRANSITION(STATE, char) : STATE; – returns the state to which the transducer transits from the parameter state with the parameter char;
7. **procedure** SET_TRANSITION(STATE, char, STATE); – the procedure defines the transition from the first state by the char to the second state;
8. **function** STATE_OUTPUT(STATE) : POutString; – returns the output string of a final state;
9. **procedure** SET_STATE_OUTPUT(STATE, POutString); – defines the output string of a final state;
10. **function** OUTPUT(STATE, char) : POutString; – returns the output string for the transition given by the state and the char;
11. **procedure** SET_OUTPUT(STATE, char, POutString); – defines the output string for the transition given by the state and the char;
12. **procedure** SAVE_TRANSDUCER(file, STATE); – saves the transducer to the file, starting from the initial state.

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

```
1   program Dictionary_lmls_Extension(input,output); {Main program}
2   const MAX_QUEUE_SIZE = 1000000;
```

Below the special representation of the output strings is given. An output string is either a character, or a substitute for a dictionary entry, or a concatenation of two already defined strings represented by pointers. In line 11 we define the type for the queue records as triples containing two states and one pointer to OutString.

```
3   type
4      StrType = (letter, out, conc);
5      POutString = ↑ OutString; {POutString is a pointer to OutString}
6      OutString = record case StrType of
7         letter : (C:char);
8         out : (S:string);
9         conc : (S1, S2 : POutString) {S1 and S2 are pointers to OutString}
10     end;
11     QueueRec = record q,p' : STATE; γ' : POutString end;
```

Here we define the global variables used in the algorithm.

```
12  var
13     q,q1,q0 : STATE;
14     Queue : array [1..MAX_QUEUE_SIZE] of QueueRec;
15     queue_begin, queue_end, i, j : integer;
```

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

```
16      c : char;
17      EmptyString : POutString;
```

The next three functions below are used to efficiently concatenate two OutStrings, save an ordinary string in an OutString, and to represent a character as an OutString. 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.

```
18  function ConcatOutStrings (S1, S2 : POutString) : POutString;
19  begin
20      if (S1 = EmptyString) and (S2 = EmptyString) then
21          ConcatOutStrings := EmptyString
22      else if (S1 = EmptyString) then
23          ConcatOutStrings := S2
24      else if (S2 = EmptyString) then
25          ConcatOutStrings := S1
26      else begin
27          new(ConcatOutStrings);
28          ConcatOutStrings↑.S1 := S1;
29          ConcatOutStrings↑.S2 := S2
30      end
31  end;

32  function StringToOutString (S : string) : POutString;
33  begin
34      new(StringToOutString);
35      StringToOutString↑.S := S
36  end;

37  function CharToOutString (C : char) : POutString;
38  begin
39      new(CharToOutString);
40      CharToOutString↑.C := C
41  end;
```

The following procedure describes the treatment of one triple from the queue. Processing a triple means to declare q to be final, to add state output of q, failure transitions for state q and to maintain the queue by adding triples for the trie successor states. For the definition of state outputs and failure transitions see

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

Section 4.2 and the above equations. In lines 46–56, for each letter c an appropriate action is taken: either a failure transition is added or a new triple is calculated and added to the queue.

```
42  procedure Process_State(q : STATE, p' : STATE, γ' : POutString);
43  begin
44      SET_FINAL(q,true);
45      SET_STATE_OUTPUT(q,ConcatOutStrings(γ',STATE_OUTPUT(p')));
46      for c := FIRST_CHAR to LAST_CHAR do
47          if TRANSITION(q,c) = NULL then begin
                {We add a failure transition}
48              SET_TRANSITION(q,c,TRANSITION(p',c));
49              SET_OUTPUT(q,c,ConcatOutStrings(γ',OUTPUT(p',c))
50          end else begin
51              if FINAL(TRANSITION(q,c)) then
                {Situation II, trie transition to a final state}
52                  Queue[queue_end] :=
                            [TRANSITION(q,c),q0,STATE_OUTPUT(TRANSITION(q,c))]
53              else {Situation I, trie transition to a non-final state}
54                  Queue[queue_end] := [TRANSITION(q,c),TRANSITION(p',c),
                                         ConcatOutStrings(γ',OUTPUT(p',c))];
55              queue_end := queue_end + 1
56          end {if}
57  end; {Process_State}
```

The main program has three parts.

First (lines 58–79), the transducer trie is computed by reading the input dictionary line by line.

Second (lines 80–95), the initial state q_0 is treated. We set its output to the empty string and add loops for all symbols σ of the alphabet for which there are no outgoing transitions from the starting state. The transition outputs are identical to the corresponding input symbol σ (line 87). If there is already an outgoing transition with the given symbol σ in the trie, 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), depending on the situation.

Third (lines 96–99), the triples in the queue are processed. As seen above the procedure Process_State adds new triples to the queue. The queue is used in order to process the states in ascending distance from the starting state.

```
58  begin
59      EmptyString := StringToOutString('');
60      q0 := NEW_STATE;
61      while not eof(input) do begin
62          readln(input,CurrentWord,CurrentOutput);
```

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

```
63          i := 1;
64          q := q0;
65          while (i<=length(CurrentWord))
                 and (TRANSITION(q,CurrentWord[i]) <> NULL) do begin
66              q := TRANSITION(q,CurrentWord[i]);
67              i := i + 1
68          end;
69          if i<=length(CurrentWord) then begin
70              for j := i to length(CurrentWord) do begin
71                  q1 := NEWSTATE;
72                  SET_TRANSITION(q,CurrentWord[j],q1);
73                  SET_OUTPUT(q,CurrentWord[j],EmptyString);
74                  q := q1
75              end
76          end;
77          SET_FINAL(q,true);
78          SET_STATE_OUTPUT(q,StringToOutString(CurrentOutput))
79      end; {while}
80      queue_begin := 1;
81      queue_end := 1;
82      SET_FINAL(q0,true);
83      SET_STATE_OUTPUT(q0,EmptyString);
84      for c := FIRST_CHAR to LAST_CHAR do
85          if TRANSITION(q0,c) = NULL then begin
86              SET_TRANSITION(q0,c,q0);
87              SET_OUTPUT(q0,c,CharToOutString(c))
88          end else begin
89              if FINAL(TRANSITION(q0,c)) then
                 {Situation II, trie transition to a final state}
90                  Queue[queue_end] := [TRANSITION(q0,c), q0,
                                             STATE_OUTPUT(TRANSITION(q0,c))]
91              else
                 {Situation I, trie transition to a non-final state}
92                  Queue[queue_end] := [TRANSITION(q0,c), q0,
                                                   CharToOutString(c)];
93              queue_end := queue_end + 1
94          end {if}
95      end; {for}

96      while queue_begin<queue_end do begin
97          Process_State(Queue[queue_begin].q,
                                  Queue[queue_begin].p',Queue[queue_begin].γ');
98          queue_begin := queue_begin + 1
99      end; {while}
100      SAVE_TRANSDUCER(output,q0)
101  end.
```

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

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.

7 Empirical results

We implemented both the algorithm described in Section 4 and the naïve 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 naïve 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 naïve algorithm. For three of the entries the rewriting transducer was too large to be hold in the operating memory.

The algorithm is currently applied in the context of OCR-correction (Strohmaier *et al.* 2003a; Strohmaier *et al.* 2003b), using rewriting dictionaries that rewrite erroneous words into correction suggestions. We successfully compiled transducers for a variety of rewriting dictionaries. For the experiments we used a conventional PC (Intel(R), Pentium IV, CPU 3.2 GHz with 1 GB RAM main memory running Linux). The largest dictionary had 221,681 entries. The number of states of the transducer was 737,792. The number of transitions was $737,792 \times 42$ (alphabet size 42). The construction time for the transducer was 33.60 seconds. Using the constructed transducer, 134 MB of text (18,470,101 tokens) were rewritten in 7.71 seconds. For the sake of comparison, the same text was rewritten with the naïve algorithm, which took 12.79 seconds (speed-up factor 1.66x). The size of the rewriting subsequential transducer is 278 MB. The size of the minimal subsequential transducer representing the dictionary without rewriting (constructed using the method presented in Mihov and Maurel (2001)) is 4.3 MB.

8 Related work

Applications of finite state transducers for different problems of natural language processing are described, e.g., in Kaplan and Kay (1994), Mohri (1996, 1997) and

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

Table 1. *Results for text rewriting using random rewriting dictionaries, speed-up factors (ratio of times) w.r.t. naïve algorithm*

Alphabet size	Dictionary size	Entry size 10	Entry size 30	Entry size 100
5	1,000	1·61x	1·69x	2·02x
5	10,000	1·35x	1·34x	1·50x
5	100,000	1·21x	1·25x	1·27x
30	1,000	1·31x	1·40x	1·17x
30	10,000	1·33x	1·39x	1·44x
30	100,000	1·22x	1·23x	—
60	1,000	1·31x	1·17x	1·04x
60	10,000	1·29x	1·31x	1·38x
60	100,000	1·29x	—	—

Roche and Schabes (1997). Rewriting dictionaries as discussed in this paper can be considered as a special form of rewriting rules in the sense of Kaplan and Kay (1994). Ignoring weighted versions (Mohri 2001), 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 rewriting 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 (1994) proved that all these rewriting relations are rational, i.e., they can be encoded by a finite state transducer. Kartunnen (1997) describes a similar approach. In Karttunen (1996), also the difference between longest match versus shortest match strategies is considered. For the leftmost-longest match strategy, the construction described in Kaplan and Kay (1994) has been improved by Gerdemann and van Noord (1999). Another improvement of Kaplan and Kay (1994), applicable also for weighted rewrite rules, is presented by Mohri and Sproat (1996) and Mohri *et al.* (1996). It should be noted that the above approaches lead to non-deterministic transducers.

Theoretically the above techniques can be applied for the construction of a subsequential transducer that represents D^{lmls}. To this end, a determinization procedure (Roche and Schabes 1995) must be used. 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 (1995) show how the contextual rewrite rules used in Brill's tagger (Brill 1992) 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 rewriting relation represented as a finite state transducer $\mathscr{T}_1$ can be transformed into a transducer $\mathscr{T}_2$ that represents the "local extension" of $\mathscr{T}_1$: If $\mathscr{T}_1$ produces output w for input v,

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

and if u and x are any strings where $\mathscr{T}_1$ does not apply, then $\mathscr{T}_2$ rewrites uvx into uwx. Hence, $\mathscr{T}_2$ is used to perform *all* possible $\mathscr{T}_1$-replacements inside a given text. Recall that our transducer uses the more restricted leftmost-longest replacement strategy with skips. From the theoretical results proved at the end of Roche and Schabes (1995) it follows that the lmls-extension D^{lmls} of a finite rewriting dictionary D can be represented as a subsequential transducer. But at this point no explicit construction is given.

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-longest replacement strategy with skips). For this specific problem, a simple and efficient solution directly leading to a deterministic transducer is given.

The problem of how to represent a given rewriting dictionary D (as opposed to the canonical lmls-extension D^{lmls}) as a minimal subsequential transducer is discussed in Mohri (1996). 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 Mihov and Maurel (2001), building up on ideas from Mihov (1998) and Daciuk *et al.* (2000).

The wide variety of text rewriting applications mentioned in the introduction, which could be extended by further examples from compiler construction, computer virus/worm detection, and network traffic intrusion detection, explains that many techniques have been established for solving algorithmic tasks similar to the one studied above. An exhaustive comparison is outside the scope of the paper. Yet, subtle differences have to be taken into account that explain why methods from one domain cannot be naively transferred to the present task. For example, text rewriting in translation and compiler construction is based on tokenized text, hence problems with overlapping originals do not occur.

9 Conclusion

In this paper we have shown how to compute for a given rewriting dictionary, D, a subsequential finite state transducer $\mathscr{T}$ that represents the canonical lmls-extension D^{lmls} of D and thus can be used for text-rewriting based on D using the leftmost-longest replacement strategy with skips. The resulting speed of the text-rewriting procedure can hardly be improved since the text is deterministically rewritten in one linear scan of the input. Theoretically, the size of the output functions of $\mathscr{T}$ 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 $\mathscr{T}$ in time and space linear in the size of D.

The entries of our rewriting dictionaries represent non-contextual rewrite rules. Naively, a contextual rule of the form $v \mapsto w / l_r$ (where v, w, l, r are strings) can be written as a non-contextual rule $lvr \mapsto lwr$. However, this changes the semantics of

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

the contextual rule because in the new version the right context r can no longer serve as left context to the next rule application. 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-longest 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.

Acknowledgements

The authors thank the referees of the Journal of Natural Language Engineering. Their comments helped to improve the presentation.

References

Abney, S. (1997) Partial parsing via finite-state cascades. *Natural Language Engineering*, **2**(3): 337–344.

Aho, A. and Corasick, M. (1975) Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, **18**: 333–340.

Appelt, D., Hobbs, J. R., Bear, J., Israel, D. and Tyson, M. (1993) A finite state processor for information extraction from real world text. *Proceedings International Joint Conference on Artificial Intelligence*, pp. 1172–1178.

Aho, A., Hopcroft, J. and Ullman, J. (1983) *Data Structures and Algorithms*. Addison-Wesley, Reading, Massachutes.

Brill, E. (1992) A simple rule-based part of speech tagger. *Proceedings of the Third Conference on Applied Natural Language Processing*, pp. 152–155.

Brown, R. (2000) Example-based machine translation in the pangloss system. *Proceedings of COLING'96: The 16th International Conference on Computational Linguistics*, pp. 169–174.

Carr, L., Hall, W., Bechhofer, S. and Goble, C. (2001) Conceptual linking: Ontology-based open hypermedia. *Proceedings of the 12th International World Wide Web Conference*, pp. 334–342.

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

Gerdemann, D. and van Noord, G. (1999) Transducers from rewrite rules with backreferences. *Proceedings of the 9th Conference of the European Chapter of the Association for Computational Linguistics (EACL 99)*, pp. 126–133.

Hopcroft, J., Motwani, R. and Ullman, J. (2000) *Introduction to Automata Theory, Languages, and Computation*, 2nd ed. Addison-Wesley.

Karttunen, L. (1996) Directed replacement. *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pp. 108–115.

Karttunen, L. (1997) The replace operator. In: E. Roche and Y. Schabes (editors), *Finite-State Language Processing*, pp. 117–147. MIT Press.

Kaplan, R. and Kay, M. (1994) Regular models of phonological rule systems. *Computational Linguistics*, **20**(3): 331–279.

Kozen, D. (1997) *Automata and Computability*. Springer.

Krovetz, B. (1993) Viewing morphology as an inference process. *Proceedings of the 16th ACM SIGIR Conference*, pp. 191–202.

Kukich, K. (1992) Techniques for automatically correcting words in texts. *ACM Comput. Surv.* 377–439.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press

Kupiec, J. (1992) Robust part-of-speech tagging using a hidden Markov model. *Computer Speech and Language*, **6**: 225–242.

Laporte, E. (1997) Phonetic conversion and phonology. In: E. Roche and Y. Schabes (editors), *Finite-State Language Processing*, pp. 407–429. MIT Press.

Mihov, S. (1998) Direct building of minimal automaton for given list. *Annuaire de l'Université de Sofia "St. Kl. Ohridski"*, **91**(1): 212–225.

Maier-Meyer, P. (1995) *Lexikon und automatische Lemmatisierung*. PhD thesis, CIS, Universität München.

Mihov, S. and Maurel, D. (2001) Direct construction of minimal acyclic subsequential transducers. *Proceedings of the Conference on Implementation and Application of Automata CIAA'2000*, number 2088 in LNCS, pp. 217–229. Springer.

Mohri, M. (1996) On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, **2**: 1–20.

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

Mohri, M. (2001) Weighted grammar tools: the GRM library. In: J. C. Junqua and G. van Noord (editors), *Robustness in Language and Speech Technology*, pp. 165–186. Kluwer Academic.

Mohri, M., Fernando, P. and Michael, R. (1996) Weighted automata in text and speech processing. *Proceedings of the 12th biennial European Conference on Artificial Intelligence (ECAI-96), Workshop on Extended Finite State Models of Language*, pp. 152–155.

Mohri, M. and Sproat, R. (1996) An efficient compiler for weighted rewrite rules. *Proceedings of the 34th Meeting of the Association for Computational Linguistics (ACL'96)*, pp. 231–238.

Porter, M. An algorithm for suffix stripping. *Program*, **14**(3): 130–137.

Roche, E. and Schabes, Y. (1995) Deterministic part-of-speech tagging with finite state transducers. *Computational Linguistics*, **22**(2): 227–253.

Roche, E. and Schabes, Y. (1997) Introduction. In: E. Roche and Y. Schabes (editors), *Finite-State Language Processing*, pp. 1–66. MIT Press.

Strohmaier, C., Ringlstetter, C., Schulz, K. U. and Mihov, S. (2003a) Lexical postcorrection of OCR-results: The web as a dynamic secondary dictionary? *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR-03*, pp. 1133–1137.

Strohmaier, C., Ringlstetter, C., Schulz, K. U. and Mihov, S. (2003b) A visual and interactive tool for optimizing lexical postcorrection of OCR-results. *Proceedings of DIAR-03*.

Vogel, S. and Ney, H. Translation with cascaded finite-state transducers. *Proceedings of the 38th Annual Meeting of the Association for Computational Linguistics, ACL-2000*, pp. 23–30.

https://doi.org/10.1017/S1351324905004092 Published online by Cambridge University Press