Basic Research in Computer Science

The Suffix Tree of a Tree and Minimizing Sequential Transducers

Dany Breslauer

BRICS Report Series RS-95-47

ISSN 0909-0878 September 1995

Copyright © 1995, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

The Suffix Tree of a Tree and Minimizing Sequential Transducers

Dany Breslauer*

Abstract

This paper gives a linear-time algorithm for the construction of the suffix tree of a tree. The suffix tree of a tree is used to obtain an efficient algorithm for the minimization of sequential transducers.

1 Introduction

The suffix tree of a string is a compact trie representing all suffixes of the string. The suffix tree has proven to be an extremely useful data structure in a wide variety of string processing algorithms [3, 6]. Kosaraju [10] defined the generalized suffix tree of all suffixes of a set of strings which are represented by a tree. Kosaraju mentions that Weiner's [14] suffix tree construction algorithm can be easily modified to construct the suffix tree of a tree in $O(n \log n)$ time, and that it might even be possible to do so in $O(n)$ time. In this paper we give an $O(n)$ time algorithm for the construction of the suffix tree of a tree, if the input symbols are drawen from a constant size alphabet. We then use the new suffix tree construction algorithm in the minimization of sequential transducers.

The minimization problem of deterministic finite automata is a well studied problem with the best algorithmic solution, due to Hopcroft [9], dating back to the beginning of the seventies. Recently, Mohri [11] considered the minimization of sequential transducers, or finite automata with output. Mohri proved that after transforming any given sequential transducer into an equivalent transducer that writes its output as soon as possible, the minimization of the resulting transducer, considered as a finite automaton, gives the minimal equivalent sequential transducer. We observe that Mohri's transformation algorithm essentially solves a single source shortest path problem. The shortest path formulation of the problem and the new generalized suffix tree data structure are used to obtain a simple and efficient implementation of Mohri's algorithm.

*BRICS – Basic Research in Computer Science – a centre of the Danish National Research Foundation, Department of Computer Science, University of Aarhus, DK-8000 Aarhus C, Denmark. Partially supported by the ESPRIT Basic Research Action Program of the EC under contract #7141 (ALCOM II).

2 Suffix trees

The *suffix tree* of the string w is a rooted tree with edges and vertices that are labeled with substrings of w. The suffix tree satisfies the following properties:

1. Edges leaving (leading away from the root) any given vertex are labeled with non-empty strings that start with different symbols.

2. Each vertex is labeled with the string formed by the concatenation of the edge labels on the path from the root to the vertex.

3. Each internal (non-leaf) vertex has at least two descendants.

4. For each substring v of w, there exists a vertex labeled u, such that v is a prefix of u.

It is a common practice to append a special alphabet symbol \$, which does not appear anywhere within w, at the end of w. This guarantees that the suffix tree has exactly $|w| + 1$ leaves that are labeled with all the distinct non-empty suffixes of $w\$$. Observe that the edge and vertex labels of the suffix tree can be represented by their starting position in w and their length, and thus, the suffix tree can be stored in $O(|w|)$ space.

Theorem 2.1 (Weiner [5, 14]) *The suffix tree of a string w can be constructed in $O(|w| \log |\Sigma|)$ time and $O(|w|)$ space, where Σ is the ordered alphabet that the symbols of w are taken from.*

We shall use the following data structure together with suffix trees.

Lemma 2.2 (Harel and Tarjan [7]; see also [12]) *Given a rooted tree, it is possible to pre-process the tree in $O(n)$ time and space, such that queries about the lowest common ancestor of any pair of vertices can be answered in constant time.*

One of the important applications of suffix trees is in finding the longest common prefix (abbreviate hereafter LCP) of any two suffixes. This can be done in constant time for any LCP query after an $O(|w|)$ time preprocessing of the suffix tree, by using Theorem 2.2 and by observing that the longest common prefix of two suffixes is the label of the vertex in the suffix tree which is the lowest common ancestor of the leaves labeled with the two suffixes. For more properties and applications of suffix trees see [3, 6].

2.1 The suffix tree of a tree

Given a set of strings $\{w_1, \ldots, w_h\}$, we can represent these strings by the *common-suffix tree* (abbreviate hereafter CS-tree), which is a rooted trie that is defined as follows:

1. Each edge is labeled by a single alphabet symbol.
2. Each vertex is labeled with the string formed by the concatenation of the edge labels on the path from the vertex to the root. (These labels are not stored explicitly.)
3. Edges leaving (leading away from the root) any given vertex are labeled with different alphabet symbols.
4. There are h leafs which are labeled $w_1, \ldots, w_h$.

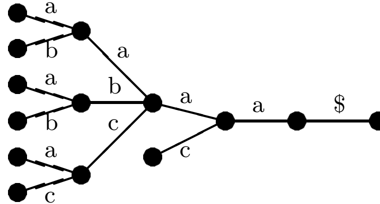

Figure 1: The CS-tree of the strings *aaaa*\$, *baaa*\$, *abaa*\$, *bbaa*\$, *acaa*\$, *ccaa*\$, *ca*\$.

Clearly, the number of vertices in the CS-tree is equal to the number of different suffixes of the strings $w_1, \ldots, w_h$ (including the empty suffix). Therefore, the size of the CS-tree is smaller than or equal to the sum of the length of the strings $w_1, \ldots, w_h$ plus one, and much smaller if the strings share long suffixes. Similarly to the case of a single string, we can append a special alphabet symbol \$ that does not appear anywhere within any of the strings $w_1, \ldots, w_h$, at the end of these strings, guaranteeing that no suffix is a proper prefix of another.

Kosaraju [10] observed that similarly to the suffix tree of a string, the suffix tree of a CS-tree can be defined to represent all substrings of the strings $w_1\$, \ldots, w_h\$$. The suffix tree of the CS-tree contains a leaf that corresponds to each vertex (suffix) in the CS-tree (except the CS-tree root that represents the empty suffix), and therefore, its size is linear in the size of the CS-tree, regardless of the alphabet size. The edge and vertex labels can be represented by the position of their start in the CS-tree (CS-tree vertex) and their length, and thus, the suffix tree of an n-vertex CS-tree can be stored in $O(n)$ space. Observe that also the suffix tree of a CS-tree can be used to find the LCP of two suffixes in the CS-tree in constant time, after an $O(n)$ time preprocessing of the suffix tree, using Lemma 2.2.

Remark. If we start with the strings $w_1, \ldots, w_h$, we do not need to construct the CS-tree of these strings in order to build the suffix tree. However, if we are given directly the CS-tree, then we shall see that we can construct the suffix tree of the CS-tree in time that is proportional to the size of the CS-tree and not to the length of leaf labels.

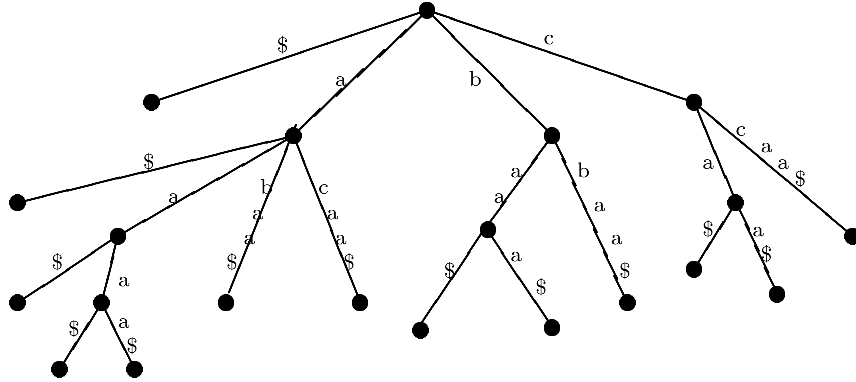

Figure 2: The suffix tree of the CS-tree in Figure 1.

2.2 Useful data structures

We define next the *semi-dynamic nearest marked ancestor* problem and quote the complexity bounds of a data structure for its solution. The problem is concerned with the maintenance of a rooted tree with marked vertices. The following operations are supported:

1. $u := create()$; build a new tree with a single vertex u being its root.
2. $insert(u, w)$; insert a new leaf w as a descendent of u.
3. $insert(uv, w)$; insert a new vertex w between u and v by breaking the tree edge uv.
4. $mark(v)$; mark the vertex v. Newly inserted vertices are not marked.
5. $u := nma{-}query(v)$; return the vertex u that is the nearest marked ancestor of v.

Lemma 2.3 (Amir et al. [2] and Westbrook [15]) *The semi-dynamic nearest marked ancestor problem can be solved in constant amortized time for* insert *and* mark*, and constant time for* create *and* nma-query*, using linear space.*

We shall use also the following data structure.

Lemma 2.4 (Berkman and Vishkin [4]) *Given a rooted tree, it is possible to pre-process the tree in $O(n)$ time and space, such that level-ancestor queries that find the l-th vertex on the path from a vertex to the root can be answered in constant time.*

2.3 Generalizing Weiner's algorithm

Weiner's [14] suffix tree construction algorithm takes $O(n \log |\Sigma|)$ time and uses $O(n)$ space, for strings of length n over an ordered alphabet Σ. Chen and Seiferas [5] simplified Weiner's algorithm and eliminated the need for some of the information maintained by the algorithm while constructing the tree. We show next that Weiner's original algorithm can be modified to build the suffix tree of a CS-tree.

Theorem 2.5 *The suffix tree of an n-vertex CS-tree can be constructed in $O(n \log |\Sigma|)$ time and space.*

Proof: We modify Weiner's algorithm to handle CS-trees. We refer the reader to [5, 14] for some properties of suffix trees that are used here without proof. Recall that Weiner's algorithm inserts a suffix v into the suffix tree only after all the suffixes of v have been inserted. Namely, it inserts the suffixes of the string into the tree in increasing order of their lengths. The modified algorithm also inserts a suffix v into the suffix tree only after all its suffixes have been inserted. Only now, this does not mean that suffixes are inserted in increasing order of their lengths, since we have suffixes of different strings. We assume for simplicity that the suffixes are inserted in non-decreasing order of their length, by a breadth-first traversal of the CS-tree. We also assume that the special symbol \$ has been appended to the CS-tree, so that each suffix in the CS-tree corresponds to a leaf of the suffix tree.

We show how to insert efficiently a new suffix au into the suffix tree, once the suffix u has already been inserted. We first check separately if the new suffix au has to be inserted as a child of the root. This can be done in constant time. From now on assume that au is not inserted as a child of the root.

Let au' be the longest vertex label in the suffix tree that is a prefix of au, and let au'' be the longest prefix of au that is also a prefix of some vertex label $au''y$ in the suffix tree ($au''y$ being the shortest vertex label with prefix au''). Then, clearly, au' is a prefix of au''. There are two cases:

1. If $au' = au''$, then the new suffix au is inserted as a child of the vertex au'.

2. If $au' \neq au''$, then let $au'x = au''$. The suffix tree contains an edge between the vertices au' and $au''y = au'xy$. This edge is broken by the insertion of the new vertex $au'x$ and the leaf au is inserted as a child of $au'x$.

The main task of the algorithm, therefore, is to find the vertex au', the edge between au' and $au''y = au'xy$ (specified by the first symbol of x), and the length of au''. Weiner proved that both u' and u'' must be vertices in the suffix tree, if for every vertex label v the suffix tree contains also all suffixes of v. His algorithm maintains the following information for each vertex v in the suffix tree:

1. $v.link[a]$, for each $a \in \Sigma$, is a pointer to the vertex labeled av if such a vertex exists in the suffix tree.

2. $v.indicator[a]$, for each $a \in \Sigma$, indicates if av is a prefix of any vertex label in the suffix tree.

This information allows to find au' and au'' by observing that u' is the nearest ancestor of u with $u'.link[a]$ defined, and u'' is the nearest ancestor of u with a positive $u''.indicator[a]$. Then, the vertex au' is given by $u'.link[a]$, and au'' is the prefix of au of length $1 + |u''|$.

The changes that are made to the *link* and *indicator* information are the same as is Weiner's algorithm. (If au was inserted as a child of the root, the links and indicators are set similarly.)

1. $u.link[a] := au$;

2. $v.indicator[a]$ is set for all ancestors v of u, up to u''.

3. If $u'' \neq u'$, then $u''.link[a] := au''$.

4. If $u'' \neq u'$, then $au''.indicator[b] := au''y.indicator[b]$, for all $b \in \Sigma$.

Weiner's [5, 14] algorithm takes $O(n \log |\Sigma|)$ time and $O(n)$ space since the number of defined links and set indicators is $O(n)$, since the links, the indicators and the edges are stores by a binary searched tree that gives $O(\log |\Sigma|)$ access time, and since the work spent in tracing u' and u'' up the tree is amortizes against the depth reduction between u and au. In the suffix tree of a CS-tree, the number of defined links is still $O(n)$, but the number of set indicators might be larger. Also, we can not amortize the time spent tracing u' and u'' against the depth reduction from u to au, since u might be used again to insert other suffixes bu, such that $b \neq a$.

We assume for the moment that the alphabet $\Sigma = \{1, \ldots, c\}$, for some constant c, so that the suffix tree edges, the links and the indicators can be stored in an array and accessed in constant time. For each alphabet symbol $a \in \Sigma$, we maintain a copy of the suffix tree where the vertices v with defined $v.link[a]$ are marked by the nearest marked ancestor data structure from Lemma 2.3. Then, given the vertex u, we can find the vertex u' by a single nca-query. We still trace up the path from u to u'' since the indicators of the vertices along this path must be set. The space requirements of maintaining the nearest marked ancestor data structures, the links, the indicators, and the suffix tree edges are $O(n|\Sigma|)$. Each structural change in the suffix tree has to be made also in the $|\Sigma|$ copies of the nearest marked ancestor data structure, taking $O(|\Sigma|)$ time per change, or $O(n|\Sigma|)$ in total. The time spent tracing up the vertices u'' accounted over all steps is also $O(n|\Sigma|)$, since in each step we set an indicator that was not set before. The rest of the construction takes constant time in each step, and therefore, the overall time and space complexities are $O(n|\Sigma|)$.

When constructing the suffix tree of a CS-tree we need to pay attention also to the maintenance of the edge and vertex labels. When an edge is broken by inserting the vertex $au'' = au'x$ between au' and $au'xy$, the label of the edge between au' and $au'x$ is set to x and on the edge between $au'x$ and $au'xy$ is set to y. Since we choose to represent the labels by a pointer to their start in the CS-tree and their length, the label x can be obtained by reducing the length of the label xy that was on the broken edge. The label y, which is obtained easily in the case of a string by adding $|x|$ to the starting position of the label xy, is now obtained by finding the $|x|$-level ancestor of the CS-tree vertex. By Lemma 2.4, after preprocessing the CS-tree in $O(n)$ time, each such level ancestor query is answered in constant time. The vertex labels can be handled similarly.

If the alphabet is larger, we can encode in $O(n \log |\Sigma|)$ time each alphabet symbol as sequences of $\lceil \log_2 |\Sigma| \rceil$ symbols from the alphabet $\{0, 1\}$. Then, the CS-tree can be modified to represent the encoded alphabet strings, the suffix tree is built for the modified CS-tree (whose size is at most $n \log |\Sigma|$), and the suffix tree is modified to represent the strings over the original alphabet CS-tree. All this takes $O(n \log |\Sigma|)$ time and space. Notice that after the construction of the suffix tree has been completed, the suffix tree can be stored in only $O(n)$ space. □

3 Automata and sequential transducers

We give next a brief summary of the definitions and the terminology of finite automata and sequential transducers. See Harrison's book [8] and Mohri's article [11] for more information.

Definition 3.1 *A deterministic finite automaton is the tuple* (S, i, F, A, δ)*, such that:*

- S *is the finite set of states of the automaton.*
- $i \in S$ *is the initial state.*
- $F \subseteq S$ *is the set of final states.*
- A *is the finite input alphabet.*
- $\delta(s, a) \in S$ *is the transition function mapping each state* $s \in S$ *and alphabet symbol* $a \in A$ *to the next state.*

The transition function δ can be extended to strings from A^* by defining:

$$\begin{aligned} \delta(s, \epsilon) &= s \quad \text{and} \\ \delta(s, aw) &= \delta(\delta(s, a), w) \quad \text{for } s \in S,\ a \in A \text{ and } w \in A^*. \end{aligned}$$

The finite automaton is said to accept a string w if and only if $\delta(i, w) \in F$. The languages (sets of strings) accepted by finite automata are called *regular*.

Definition 3.2 *A sequential transducer is the tuple* $(S, i, F, A, B, \delta, \sigma)$*, where:*

- (S, i, F, A, δ) *is a deterministic finite automaton.*
- B *is the finite output alphabet.*
- $\sigma(s, a) \in B^*$ *is the output string produced on the transition from the state* $s \in S$ *on the input symbol* $a \in A$*.*

The output function σ can be extended to strings by defining:

$$\begin{aligned} \sigma(s, \epsilon) &= \epsilon \quad \text{and} \\ \sigma(s, aw) &= \sigma(s, a)\sigma(\delta(s, a), w) \qquad \text{for } s \in S,\ a \in A \text{ and } w \in A^*. \end{aligned}$$

The sequential transducer can be thought of computing the partial function $f(w) = \sigma(i, w) : A^* \to B^*$, that is defined for those $w \in A^*$, such that $\delta(i, w) \in F$. The functions computed by sequential transducers are called *sequential functions*.

Finite automata and sequential transducers can be represented by graphs. The graph representation of a sequential transducer consists of the set of vertices S which are the states of the transducer. There is an edge $u \to v$ between the vertices $u, v \in S$, labeled with the input symbol $a \in A$ and the output string $b \in B^*$, if $\delta(u, a) = v$ and $\sigma(u, a) = b$. We allow the graph representation to be a partial representation that does not necessarily include at every vertex an outgoing edge for all input alphabet symbols in A. Thus, we can assume that in the graph representation of a sequential transducer, every vertex (state) is reachable from the initial state i and has some final state that can be reached from it. One can trivially remove vertices and edges in order to satisfy these requirements in time that is linear in the size of the input graph. We shall use the following notation for the graph representation of a sequential transducer:

- S is the set of states of the sequential transducer and $n = |S|$ is their number.
- E is the set of defined transitions and $m = |E|$ is their number.
- $uv.in \in A$ is the input label of the transition $uv \in E$.
- $uv.out \in B^*$ is the output label of the transition $uv \in E$ and $\mathcal{L} = \sum_{uv \in E} |uv.out|$ is the sum of the output label lengths.

3.1 Minimal sequential transducers

Two deterministic finite automata are said to be equivalent if they accept the same language. Classical automata theory [8] characterizes the unique finite automaton with minimal number of states that is equivalent to a given finite automaton. There are efficient algorithm that minimize a finite automaton:

Theorem 3.3 (Hopcroft [1, 9]) *Given a deterministic finite automaton, the minimal equivalent automaton can be found in $O(|A| n \log n)$ time.*

Mohri [11] observes that if a finite automaton is given by its partial graph representation, then the complexity bounds above can be expressed in the size of the graph.

Corollary 3.4 *Given the partial graph representation of a deterministic finite automaton, it can be minimized in $O(m \log n)$ time (provided that the transition labels can be compared in constant time).*

Similarly, two sequential transducers are said to be *equivalent* if they accept the same languages and their partial output functions are the identical on the language they accept. Mohri [11] characterized minimal sequential transducers, with the smallest number of states, that are equivalent to a given sequential transducer, and gave an algorithm for the minimization of sequential transducers. His algorithm consists of the following steps:

1. The sequential transducer is converted into an equivalent transducer in *prefix form*. The prefix form equivalent transducer has the same graph representation, except that the output labels are changed so that the transducer writes its output as soon as possible.
2. The prefix form transducer is minimized as if it was a finite automaton. Namely, the finite automaton with the same graph representation of the sequential transducer and with transition labels that consist of both the input labels and the output labels, is minimized using Corollary 3.4. (Now transition labels are strings and it should become clear why we need to be able to compare the transition labels in constant time.)

We give next the shortest path formulation of the problem of converting a sequential transducer into its prefix form. Using this formulation and the suffix tree of a tree we give an efficient algorithm for computing the prefix form of a transducer and the minimal equivalent transducer.

4 The prefix form

Given a sequential transducer $\mathcal{T} = (S, i, F, A, B, \delta, \sigma)$, define the prefix function $P : S \to B^*$, such that $P(s)$ is the LCP of the output labels on all paths from s to vertices in F (the output label on a path is the concatenation of the output labels of the edges along the path). The function P is well defined since for for any $s \in S$, there exists at least one path to a final state and the LCP of the labels is also well defined. An equivalent formal definition is:

$$\begin{array}{lcll} P(s) & = & \epsilon & s \in F \\ P(s) & = & \bigwedge_{sv \in E} sv.out\ P(v) & \text{otherwise.} \end{array}$$

Where $\bigwedge$ denotes the LCP.

Mohri [11] defines the equivalent transducer to $\mathcal{T}$ in *prefix form* to be the transducer $\mathcal{P}$ with the same graph description as $\mathcal{T}$, changing only the output labels, so that $\mathcal{P}$ write its output as soon as possible. Namely, for any edge $u \to v$, the output function is defined as:

$$uv.out_{\mathcal{P}} = [P(u)]^{-1}\ uv.out_{\mathcal{T}}\ P(v),$$

if $P(i) = \epsilon$. If $P(i) \neq \epsilon$, then the transducer minimization problem can be solved similarly and the details are omitted here [11]. Observe that we used above the notation $x^{-1}x = \epsilon$. This notation is justified since $P(u)$ is surely a prefix of $uv.out_{\mathcal{T}}\ P(v)$.

4.1 Efficient implementation

Let $\mathcal{T} = (S, i, F, A, B, \delta, \sigma)$ be a sequential transducer given by its graph description. The most time consuming computation is that of the prefix function $P(s)$, for all $s \in S$, which we address first. We denote by $\mathcal{L}_{in}$ the sum of the output label lengths in the input transducer $\mathcal{T}$ and by $\mathcal{L}_{out}$ the sum of the output label lengths in the output minimized transducer.

Let $\mathcal{F}$ be a rooted forest in the graph describing $\mathcal{T}$, such that the final states in F are the roots in $\mathcal{F}$, and there is a unique directed path in $\mathcal{F}$ from each vertex $s \in S \setminus F$ to some vertex in F. Let $L(s)$ be the output label on the path connecting the vertex s to the root of its tree in $\mathcal{F}$. Define $C(s)$ as:

$$\begin{array}{lcll} C(s) & = & \epsilon & s \in F \\ C(s) & = & \bigwedge_{sv \in E} sv.out\ L(v) & \text{otherwise.} \end{array}$$

Then, clearly, $P(s)$ is a prefix of $C(s)$, and $C(s)$ is a prefix of $L(s)$. Letting $l(s) = |P(s)|$ and $c(s) = |C(s)|$, it should be clear that for any $s \in S$,

$$l(s) = \min\{c(s);\quad |sv.out| + l(v) \quad \forall sv \in E\},$$

since either $l(s)$ is the length of the shortest output label on a path from s to F, or there are at least two paths from s to F with longer output labels, but the LCP of the output labels along these paths is exactly $P(s)$; if these two path start with the same edge sv, then $l(s) = |sv.out| + l(v)$ and if these two path start with different edges, then $l(s) = c(s)$.

Since $c(s)$ and $|sv.out|$, for $sv \in E$, are all non-negative integers, $l(s)$ is well defined. In fact, the equations in $l(s)$ can be modeled as a single source shortest path problem to a special vertex R in the graph consisting of the vertices $S \cup \{R\}$, the edges $sv \in E$ weighted $|sv.out|$, and extra edges from each vertex $s \in S$ to R with weight $c(s)$.

The algorithm to compute $P(s)$ proceeds as follows:

1. Compute the forest $\mathcal{F}$ by a breadth-first-search;

2. Compute $c(s)$, for all $s \in S$;
3. Solve a single source shortest path problem in the graph with $n+1$ vertices and $m+n$ edges that is defined above.

Mohri's algorithm essentially computes $c(s)$ by brute force comparison of the labels while solving the shortest path problem simultaneously. His algorithm takes $O(n + m(P_{\max} + 1))$ time or $O(n + m + (m - n + |F|)P_{\max})$ time if the transducer graph is acyclic, where $P_{\max} = \max\{|P(s)| \mid s \in S\}$.

The single source shortest path problem with integer edge weights can be solved by an implementation of Dijkstra's algorithm using Thorup's [13] integer priority queue in $O(m \log \log m)$ time. In our case, we can use even a simple weighted breadth-first-search that takes $O(n + m + \mathcal{L}_{in})$ time.

We show next that $c(s)$ can be computed in $O(n + m + \mathcal{L}_{in} \log |B|)$ time using the suffix tree of a tree.

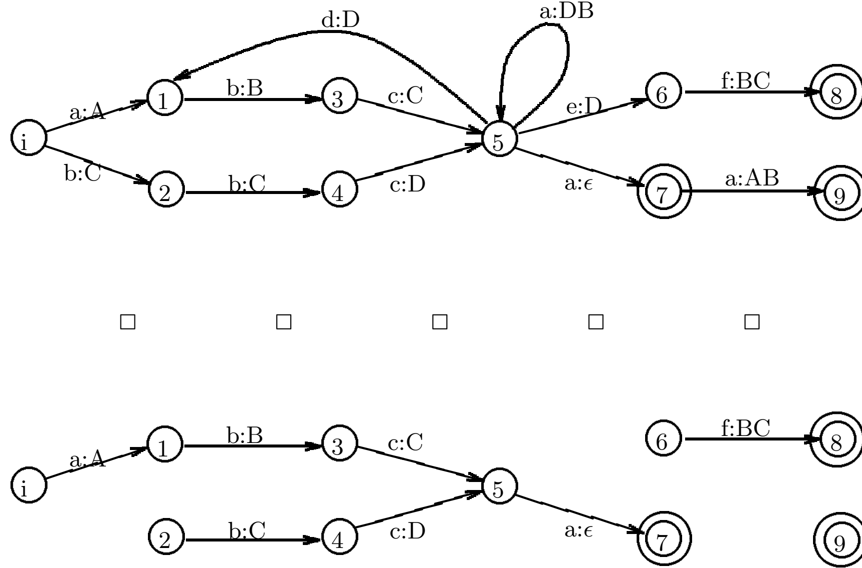

Figure 3: A transducer $\mathcal{T}$ and a forest $\mathcal{F}$.

1. Given the graph description of the transducer $\mathcal{T}$ and the forest $\mathcal{F}$, create a tree $\hat{T}$ with at most $m+1$ vertices as follows:
   - The tree $\hat{T}$ is rooted at a special vertex R that is connected to all the vertices in F, which are roots of the trees in $\mathcal{F}$, with edges labeled ϵ.
   - The tree $\hat{T}$ includes all edges $u \to v$ that are in $\mathcal{F}$, with their labels $uv.out$.

- For each edge $u \to v$ that is not in F, there is a new vertex $\tilde{uv}$ in the tree $\hat{T}$, connected to v with an edge that is labeled $uv.out$.

Clearly, the sum of the lengths of the edge labels in $\hat{T}$ is $\mathcal{L}_{in}$.

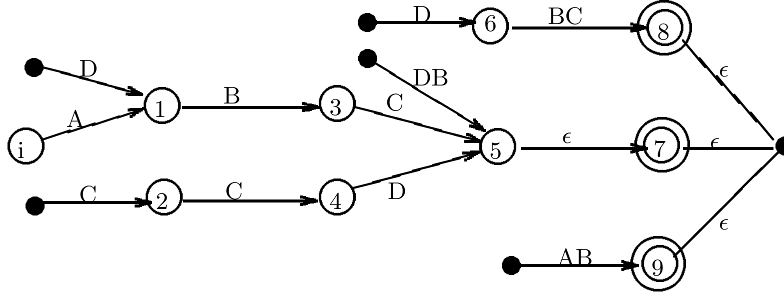

Figure 4: The tree $\hat{T}$ built for $\mathcal{T}$ and $\mathcal{F}$.

2. Construct the CS-tree of $\hat{T}$, that is the CS-tree of the strings $uv.out\ C(v)$, which are the labels in $\hat{T}$ on the paths from each vertex to the root. This can be done in in $O(\mathcal{L}_{in} \log |B|)$ time, while maintaining for each edge $uv \in E$, the corresponding CS-tree vertex that represents the path label $uv.out\ C(v)$. The resulting CS-tree has size $O(\mathcal{L}_{in})$.

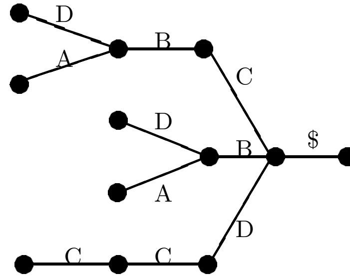

Figure 5: The CS-tree constructed from $\hat{T}$.

3. Construct the suffix tree of the CS-tree in $O(\mathcal{L}_{in} \log |B|)$ time and space.

4. Compute $c(s)$, for all $s \in S$, by lowest common ancestor queries on the suffix tree in $O(n + m)$ time.

The overall computation is summarized in the following theorem:

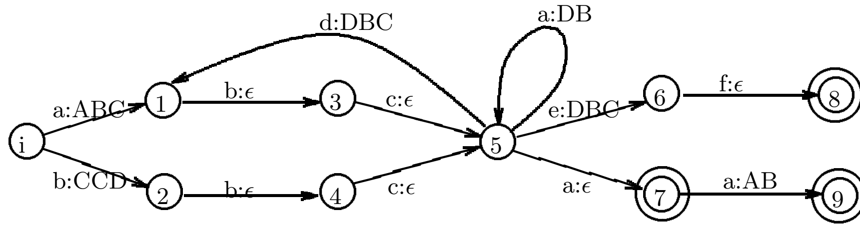

Figure 6: The prefix form of $\mathcal{T}$.

Theorem 4.1 *Given a sequential transducer, the equivalent minimal sequential transducer can be computed in* $O(n + m \log n + \mathcal{L}_{in} \log |B| + \mathcal{L}_{out})$ *time and* $O(n + m + \mathcal{L}_{in} \log |B|)$ *space.*

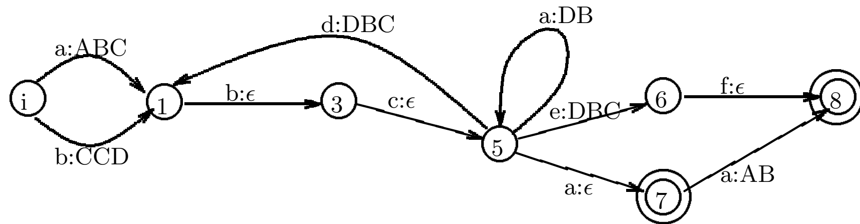

Figure 7: The minimal transducer equivalent to $\mathcal{T}$.

Proof: The algorithm first finds the forest $\mathcal{F}$ in $O(n + m)$ time, computes $c(s)$, for all $s \in S$, in $O(n + m + \mathcal{L}_{in} \log |B|)$ time and solves the single source shortest path problem in $O(n+m+\mathcal{L}_{in})$ time. The prefix form of the transducer is then found in $O(n + m)$ time, if the output labels are represented by their starting position within the CS-tree and their length, and by using the level-ancestor data structure from Lemma 2.4. (Notice that the sum of the length of the output labels in the prefix form transducer can be larger. This is not a problem, however, since these edge output labels are represented by the CS-tree.) Then, the algorithm applies Hopcroft's automata minimization algorithm from Corollary 3.4, using the fact the edge (output) labels in their CS-tree representation can be compared in constant time by consulting the suffix tree. Finally, $O(n + m + \mathcal{L}_{out})$ time is required to output the minimized transducer. □

5 Conclusions

The minimization of the number of states in sequential transducers is not the only aspect of their size. In many cases, it is possible that by minimizing the number of states, the sum of the lengths of the output labels is increased substantially. From a practical point of view, if one is interested in the storage requirements of sequential transducers, it might make more sense to minimize the label lengths, or some function of the label length and the number of states.

We believe that the suffix tree of a tree is of independent interest and that other applications of this suffix tree will be found in the future. It would be interesting to reduce the space requirements of the suffix tree construction algorithm to $O(n)$.

6 Acknowledgments

I thank Pino Italiano, Peter Bro Miltersen and Mehryar Mohri for several discussions.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[2] A. Amir, M. Farach, R.M. Idury, J.A. La Poutré, and A.A. Schäffer. Improved Dynamic Dictionary-Matching. In *Proc. 4nd ACM-SIAM Symp. on Discrete Algorithms*, pages 392–401, 1993.

[3] A. Apostolico. The Myriad Virtues of Subword Trees. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 85–96. Springer-Verlag, Berlin, Germany, 1985.

[4] O. Berkman and U. Vishkin. Finding Level-Ancestors in Trees. *J. Comput. System Sci.*, 48:214–230, 1994.

[5] M.T. Chen and J. Seiferas. Efficient and elegant subword-tree construction. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words*, volume 12 of *NATO ASI Series F*, pages 97–107. Springer-Verlag, Berlin, Germany, 1985.

[6] R. Grossi and G.F. Italiano. Suffix Trees and their Applications in String Algorithms. Manuscript, 1995.

[7] D. Harel and R.E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13:338–355, 1984.

[8] M. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley, Reading, MA., U.S.A., 1978.

[9] J.E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computation*, pages 189–196. Academic Press, New York, U.S.A., 1971.

[10] S.R. Kosaraju. Fast Pattern Matching in Trees. In *Proc. 30th IEEE Symp. on Foundations of Computer Science*, pages 178–183, 1989.

[11] M. Mohri. Minimization of Sequential Transducers. In *Proc. 5rd Symp. on Combinatorial Pattern Matching*, number 807 in Lecture Notes in Computer Science, pages 151–163. Springer-Verlag, Berlin, Germany, 1994.

[12] B. Schieber and U. Vishkin. On finding Lowest common ancestors: simplification and parallelization. *SIAM J. Comput.*, 17(6):1253–1262, 1988.

[13] M. Thorup. An $O(\log \log n)$ Priority Queue. Manuscript, 1995.

[14] P. Weiner. Linear pattern matching algorithms. In *Proc. 14th Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[15] J. Westbrook. Fast Incremental Planarity Testing. In *Proc. 19th International Colloquium on Automata, Languages, and Programming*, number 623 in Lecture Notes in Computer Science, pages 342–353. Springer-Verlag, Berlin, Germany, 1992.

Recent Publications in the BRICS Report Series

RS-95-47 **Dany Breslauer.** ***The Suffix Tree of a Tree and Minimizing Sequential Transducers*****. September 1995. 15 pp.**

RS-95-46 **Dany Breslauer, Livio Colussi, and Laura Toniolo.** ***On the Comparison Complexity of the String Prefix-Matching Problem*****. August 1995. 39 pp. Appears in Leeuwen, editor,** ***Algorithms - ESA '94: Second Annual European Symposium proceedings*****, LNCS 855, 1994, pages 483–494.**

RS-95-45 **Gudmund Skovbjerg Frandsen and Sven Skyum.** ***Dynamic Maintenance of Majority Information in Constant Time per Update*****. August 1995. 9 pp.**

RS-95-44 **Bruno Courcelle and Igor Walukiewicz.** ***Monadic Second-Order Logic, Graphs and Unfoldings of Transition Systems*****. August 1995. 39 pp. To be presented at CSL '95.**

RS-95-43 **Noam Nisan and Avi Wigderson.** ***Lower Bounds on Arithmetic Circuits via Partial Derivatives (Preliminary Version)*****. August 1995. 17 pp. To appear in** ***36th Annual Conference on Foundations of Computer Science*****, FOCS '95, IEEE, 1995.**

RS-95-42 **Mayer Goldberg.** ***An Adequate Left-Associated Binary Numeral System in the λ-Calculus*****. August 1995. 16 pp.**

RS-95-41 **Olivier Danvy, Karoline Malmkjær, and Jens Palsberg.** ***Eta-Expansion Does The Trick*****. August 1995. 23 pp.**

RS-95-40 **Anna Ingólfsdóttir and Andrea Schalk.** ***A Fully Abstract Denotational Model for Observational Congruence*****. August 1995. 29 pp.**

RS-95-39 **Allan Cheng.** ***Petri Nets, Traces, and Local Model Checking*****. July 1995. 32 pp. Full version of paper appearing in Proceedings of AMAST '95, LNCS 936, 1995.**

RS-95-38 **Mayer Goldberg.** ***Gödelisation in the λ-Calculus*****. July 1995. 7 pp.**