Theoretical Computer Science 638 (2016) 171–178

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Linear-size suffix tries

Maxime Crochemore [a,b], Chiara Epifanio [c], Roberto Grossi [d,*,1], Filippo Mignosi [e]

[a] King's College London, UK
[b] Université Paris-Est, France
[c] Dipartimento di Matematica e Informatica, Università di Palermo, Italy
[d] Dipartimento di Informatica, Università di Pisa, Italy
[e] Dipartimento di Matematica e Applicazioni, Università degli Studi dell'Aquila, Italy

ARTICLE INFO

Article history:
Received 15 April 2015
Received in revised form 8 February 2016
Accepted 1 April 2016
Available online 7 April 2016

Keywords:
Factor and suffix automata
Pattern matching
Suffix tree
Text indexing

ABSTRACT

Suffix trees are highly regarded data structures for text indexing and string algorithms [MCreight 76, Weiner 73]. For any given string w of length $n = |w|$, a suffix tree for w takes $O(n)$ nodes and links. It is often presented as a compacted version of a suffix trie for w, where the latter is the trie (or digital search tree) built on the suffixes of w. Here the compaction process replaces each maximal chain of unary nodes with a single arc. For this, the suffix tree requires that the labels of its arcs are substrings encoded as pointers to w (or equivalent information). On the contrary, the arcs of the suffix trie are labeled by single symbols but there can be $\Theta(n^2)$ nodes and links for suffix tries in the worst case because of their unary nodes. It is an interesting question if the suffix trie can be stored using $O(n)$ nodes. We present the *linear-size suffix trie*, which guarantees $O(n)$ nodes. We use a new technique for reducing the number of unary nodes to $O(n)$, that stems from some results on antidictionaries. For instance, by using the linear-size suffix trie, we are able to check whether a pattern p of length $m = |p|$ occurs in w in $O(m \log |\Sigma|)$ time and we can find the longest common substring of two strings w_1 and w_2 in $O((|w_1| + |w_2|) \log |\Sigma|)$ time for an alphabet Σ.

1. Introduction

One of the seminal results in pattern matching is that the size of the minimal automaton accepting the substrings of a string, called directed acyclic word graph (DAWG), is linear according to the length of the string, and it can be built in linear time [1]. This result is surprising since the maximal number of substrings that may occur in a string is quadratic, and DAWG's arcs are labeled with single symbols. Suffix trees also contain a linear number of nodes, but they need to store substrings in the arcs (e.g. pointers to the input string).

Further work has been aimed at this direction. For example, a compact version of the DAWG and a direct algorithm to construct it are given in [8]. An algorithm for the online construction of DAWGs is described in [15] and [16]. Space-efficient implementations of compact DAWGs are presented in [13] and [14]. When compared to the well-known suffix trees, Blumer et al. [2] proved that, given an arbitrary string w of length $n = |w|$ and its reversal $\tilde{w}$, the following holds

* Corresponding author.
E-mail addresses: Maxime.Crochemore@kcl.ac.uk (M. Crochemore), epifanio@math.unipa.it (C. Epifanio), grossi@di.unipi.it (R. Grossi), mignosi@di.univaq.it (F. Mignosi).

[1] Partially supported by Project 2012C4E3KT AMANDA (Algorithmics for MAssive and Networked DAta).

http://dx.doi.org/10.1016/j.tcs.2016.04.002
0304-3975/

$$n < e_{ST}(w) = e_{DAWG}(\tilde{w}) \leq 2n,$$

where $e_{ST}(w)$ and $e_{DAWG}(\tilde{w})$ represent, respectively, the number of nodes of the suffix tree for w and those of the DAWG for $\tilde{w}$. Moreover, considering the size of the two structures with respect to the same string w, they prove that

$$e_{ST}(w) \simeq e_{DAWG}(w)$$

where $\simeq$ indicates a property that holds on the average, as $\sum_{w\in\Sigma^n} e_{DAWG}(w) = \sum_{w\in\Sigma^n} e_{DAWG}(\tilde{w})$. For comparisons and results on this subject, we refer the reader to [3,4].

Most of the research on suffix trees took instead different directions, for example, by compacting chains of single-child nodes as arcs labeled by substrings. As far as we know, the natural question of having suffix tries with $o(n^2)$ nodes in the worst case has not been investigated in the literature. Indeed it is not difficult to build some examples of input strings that require $\Theta(n^2)$ nodes for the suffix trie; however, this does not imply that all of these nodes are strictly necessary for efficient pattern searching. The trick of encoding the input string as a path and its substrings as subpaths[2] does not answer the question as conceptually the arcs of the suffix tree are still (implicitly) labeled with substrings. Morrison [22] posed the above issue for Patricias but required to know the length of the substring to be skipped during searching. Some related questions have been addressed for other variants of suffix trees, such as position heaps [9], which were invented few years after our results in [5]. They can be seen as pruned suffix tries [10,19,23] with $n+1$ nodes and n "maximal-reach" pointers: they succeed in reducing the number of nodes of the suffix trie to $O(n)$ but require additional arrays using $O(n)$ words to perform pattern searching.[3]

In this paper we present a new technique for reducing the number of nodes in suffix tries, which provides a totally different alternative to that of encoding substrings in suffix trees. We call the resulting data structure a *linear-size suffix trie*, shortly LST, for an input string w of n symbols. Denoting by $e_{LST}(w)$ the number of nodes of the LST of w, we prove that

$$e_{ST}(w) \leq e_{LST}(w) \leq e_{ST}(w) + n.$$

The LST can be built in $O(n)$ time from the suffix tree, and checking whether a pattern p of length $m = |p|$ occurs as a substring in w takes $O(m)$ time, by traversing a small part of the LST.[4] Since the arcs in the LST are labeled with single symbols and only a subset of the nodes are kept, we must recover the symbols which are lacking somehow without accessing the text w. We exploit suffix links on nodes and some of the arcs for this purpose. We think that the LST is as powerful as the suffix tree in most applications because it retains its topological structure. For example, we can find the longest common substring of two strings w_1 and w_2 in $O(|w_1| + |w_2|)$ time. We show some applications in Section 3 and give further discussion in the last section.

We remark that our contributions are at the moment mainly of theoretical nature and that a preliminary version of LSTs can be found in [5]. Anyhow, they may have some practical applications. First of all, a quick space analysis in the last section shows that LSTs can be made space-efficient. Second, the ideas and applications of LSTs can be extended to compact DAWGs. This is interesting as the size of compact DAWGs can be exponentially smaller than the size of the text for families of highly compressible strings.

This paper is organized as follows. Section 2 presents our data structure, the linear-size suffix trie. Section 3 contains our new search algorithms for locating a pattern with the compact tries and related automata and shows hot to use LST in order to find the longest common substring of two strings. Section 4 draws open problems and conclusions.

2. Linear-size suffix trie (LST)

Given a string $w = a_1a_2\ldots a_n$ of n symbols drawn from an alphabet Σ, we denote by $w[j, j+k-1]$ the substring of w of length k that appears at position j in w. The length of w is denoted by $|w| = n$. The empty string ϵ is a substring of any string. A substring of the form $w[j, |w|]$ (resp. $w[1, j]$) is called a suffix (resp. prefix) of w. We adopt the notation for strings defined in [20].

The suffix trie $ST(w)$ of a string w is a trie where the set of leaves is the set of suffixes of w that do not appear previously as substrings in w. The suffix tree $S(w)$ is the compacted version of $ST(w)$ where each maximal chain of unary nodes is replaced by a single arc whose label is the substring of w obtained by concatenating the symbols on the chain. It is often required in the suffix tree definition that the last symbol of w is a terminator not appearing elsewhere; here we do not need this requirement, assuming that for any suffix $w[j, |w|]$ of w there is a node v storing $w[j, |w|]$, even if $w[j, |w|]$ has appeared as substring in another position. We assume that the reader is familiar with suffix trees and related notions,

[2] Specifically, the input string w is encoded as a path π of $n+1$ nodes with arcs labeled by single symbols, where the first node is the root of the suffix tree and the last node is the leaf corresponding to the suffix w. When encoding arcs by substrings $w[j, i]$, they are not represented by integer pairs (j, i); rather, pointer pairs (p_j, p_i) can be employed, so that p_k points to the kth node in π for $k = j, i$.

[3] A constructive way of defining position heaps is the following one. Start with a suffix trie and label the root 0, where its n leaves are initially non-marked and labeled from 1 to n as usual. For each child of the root, run the recursive pruning. Consider the current node u. If all the descending leaves from u are marked, then prune u and its subtree. Otherwise, let k be the smallest label among the non-marked leaves descending from u: store k in u, mark the corresponding leaf, and continue the recursion individually on each child of u. See [10] for a description of the additional pointers and arrays required to perform pattern searching.

[4] We assume that the alphabet of the symbols in w is constant-size, although the result can be extended to any alphabet Σ by multiplying the time complexity by a factor of $\log|\Sigma|$, as it is customary in the literature.