Theoretical Computer Science 678 (2017) 22–39

Contents lists available at ScienceDirect

Theoretical Computer Science

www.elsevier.com/locate/tcs

Inducing enhanced suffix arrays for string collections ☆

Felipe A. Louza [a,*], Simon Gog [b], Guilherme P. Telles [a]

[a] Institute of Computing, University of Campinas, Brazil
[b] Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany

ARTICLE INFO

Article history:
Received 9 November 2016
Received in revised form 31 March 2017
Accepted 31 March 2017
Available online 6 April 2017
Communicated by R. Giancarlo

Keywords:
Data structures
Suffix array
LCP array
Document array
String collections

ABSTRACT

Constructing the suffix array for a string collection is an important task that may be performed by sorting the concatenation of all strings. In this article we present algorithms gSAIS and gSACA-K, which extend SAIS (Nong et al., 2011) [8] and SACA-K (Nong, 2013) [10] to construct the suffix array for a string collection maintaining their theoretical bounds, respecting the order among all suffixes, and improving their practical performance. gSACA-K is an optimal suffix array construction algorithm for string collections from constant alphabets. Moreover, we show how to modify gSAIS and gSACA-K to also compute the longest common prefix array and the document array as a byproduct of suffix sorting, with the same theoretical bounds. We performed experiments that showed that our algorithms have an outstanding practical performance.

© 2017 Elsevier B.V. All rights reserved.

1. Introduction

Suffix array construction is a well studied problem [2–4] and currently several linear time solutions exist [5–9]. In 2013, Nong [10] presented an optimal suffix array construction algorithm (SACA) that runs in linear time using constant workspace for constant alphabet size. By workspace we mean the extra space needed in addition to input and output.

The suffix array is frequently enhanced with the longest common prefix (LCP) array [11]. The LCP array can be constructed in linear time given the string and its suffix array as input [12–14], or alternatively during the suffix array construction [15]. Recently, Louza et al. [16] introduced a modification of Nong's algorithm to also compute the LCP array with the same bounds.

In many applications we are interested in constructing the enhanced suffix array for a collection of strings [17,18]. For example, when an index for a document database is needed, each document may be regarded as a string and the task may be performed by using a standard construction algorithm over the concatenation of such strings [2]. However, such approach may deteriorate both the theoretical bounds and the practical behavior of many construction algorithms.

Let $\mathcal{T} = T_1, T_2, \ldots, T_d$ be a collection of d strings of total length N. There are two common approaches used to concatenate all strings in $\mathcal{T}$ into a single string T^{cat}. The first alternative uses d pairwise distinct symbols $\$_i$ as separators, one for each $T_i \in \mathcal{T}$, such that $\$_i < \$_j$ if and only if $i < j$. The second alternative uses the same symbol \$ as separator for every $T_i \in \mathcal{T}$. T^{cat} ends with an end marker symbol # in both approaches, such that # is smaller than any other symbol.

Although both approaches are straightforward and have been used in different applications (e.g. [19–28]), they have some drawbacks. The first alternative increases the alphabet size of T^{cat} by the number of strings, which may deteriorate

☆ A preliminary version of this work appeared in DCC 2016 [1].

* Corresponding author.
E-mail addresses: louza@ic.unicamp.br (F.A. Louza), gog@kit.edu (S. Gog), gpt@ic.unicamp.br (G.P. Telles).

http://dx.doi.org/10.1016/j.tcs.2017.03.039
0304-3975/© 2017 Elsevier B.V. All rights reserved.

the theoretical bounds of many algorithms. For instance, the workspace of [10,16] would increase from $O(1)$ to $O(d \log N)$ bits, which is not optimal for a constant size input alphabet. On the other side, for strings T_i and T_j, $i < j$, the second alternative will not guarantee that equal suffixes of T_i and T_j will be sorted with respect to i and j, in other words, ties will not be broken by the string rank, what may cause unnecessary comparisons in the suffix sorting, depending on the order of the strings in the collection. Moreover, this alternative may cause standard algorithms to incorrectly compute the LCP array, because lcp-values may exceed the length of the strings.

Less emphasis has been put on specific SACAs and LCP array construction algorithms for string collections (e.g. [17,1]).[1] In this article we show how to modify SAIS [8] and SACA-K [10] to receive as input the concatenation of all strings using the same symbol \$ as separator, while guaranteeing that suffixes that are equal up to \$ will be sorted by string rank.[2]

Our Contributions. In this article we make the following three contributions:

1. We propose the general versions of SAIS and SACA-K, called gSAIS and gSACA-K in this article. The new suffix array construction algorithms work for string collections and have the same theoretical bounds as SAIS and SACA-K.
2. An extended version, called gSAIS+LCP and gSACA-K+LCP, computes the LCP array along with the suffix array construction. This was achieved by adapting ideas from Fischer [15] and Louza et al. [16].
3. Finally, we show how the document array (DA) can also be computed along with the suffix array construction. We call these algorithms gSAIS+DA and gSACA-K+DA.

Experimental evaluation with different string collections have shown that the practical performance of gSAIS and gSACA-K is better than the performance of the original versions applied to sort strings using both concatenation alternatives above. Experiments have also shown that gSAIS+LCP, gSACA-K+LCP, gSAIS+DA and gSACA-K+DA outperform the best known alternatives.

2. Background

Let T be a string of length $|T| = n$ over an alphabet Σ of size σ. A constant alphabet has size $\sigma = O(1)$ and an integer alphabet has size $\sigma = n^{O(1)}$. We denote the concatenation of strings or symbols by the dot operator $(\cdot)$. We use the symbol $<$ for the lexicographic order relation between strings and suffixes.

The i-th symbol of T is denoted by $T[i]$ and the substring $T[i] \cdot T[i+1] \cdot \ldots \cdot T[j]$ is denoted by $T[i, j]$, for $1 \leq i \leq j \leq n$. A prefix of T is a substring of the form $T[1, i]$ and a suffix is a substring of the form $T[i, n]$. We assume that T always ends with a special end marker symbol $T[n] = \#$, called sentinel, which is not present in elsewhere in T and precedes every symbol in Σ.

The *suffix array* (SA) [35,36] of a string $T[1, n]$ is an array of integers in the range $[1, n]$ that gives the lexicographic order of all suffixes of T, such that $T[\mathsf{SA}[1], n] < T[\mathsf{SA}[2], n] < \ldots < T[\mathsf{SA}[n], n]$. We denote the inverse of SA as ISA, such that $\mathsf{ISA}[\mathsf{SA}[i]] = i$.

The LCP *array* stores the lengths of the longest common prefix (lcp) of adjacent suffixes in SA. We define $\mathsf{LCP}[1] = 0$ and $\mathsf{LCP}[i] = \mathsf{lcp}(T[\mathsf{SA}[i], n], T[\mathsf{SA}[i-1], n])$, for $1 < i \leq n$. A range minimum query (rmq) on LCP is the smallest lcp-value in a given interval, that is, $\mathsf{rmq}(i, j) = \min_{i<k\leq j}\{\mathsf{LCP}[k]\}$, for $1 \leq i < j \leq n$. It is easy to see that $\mathsf{lcp}(T[\mathsf{SA}[i], n], T[\mathsf{SA}[j], n]) = \mathsf{rmq}(i, j)$.

Let $\mathcal{T} = T_1, T_2, \ldots, T_d$ be a collection of d strings over Σ, of lengths $n_1, n_2, \ldots, n_d$. The suffix array for $\mathcal{T}$ is the SA of the concatenated string T^{cat}, which can be created by two alternatives that replace the sentinel of each string by a separator symbol, as follows: (1) using pairwise distinct separator symbols, or (2) using the same separator symbol [2].

Concatenating alternatives.

1. $T^{cat} = T_1[1, n_1 - 1] \cdot \$_1 \cdot T_2[1, n_2 - 1] \cdot \$_2 \cdots T_d[1, n_d - 1] \cdot \$_d \cdot \#$
2. $T^{cat} = T_1[1, n_1 - 1] \cdot \$ \cdot T_2[1, n_2 - 1] \cdot \$ \cdots T_d[1, n_d - 1] \cdot \$ \cdot \#$

where $\$, \$_1, \$_2, \ldots, \$_d$ are not in Σ, and $\# < \$ < \$_1 < \$_2 < \ldots < \$_d$ are smaller than any other symbol in Σ. The total length of T^{cat} is $(\Sigma_{i=1}^{d} n_i) + 1 = N$.

The usage of d distinct separators may be disadvantageous in many applications due to the increased alphabet size $\sigma + d$ [2]. Note that in the second alternative the alphabet size is $\sigma + 1$. In the text that follows the alternative of T^{cat} that is referred to will be clear by the context.

The suffix array of $T^{cat}[1, N]$ is commonly accompanied by the document array (DA), where $\mathsf{DA}[i]$ stores the index of the string which suffix $T^{cat}[\mathsf{SA}[i], N]$ came from. We define $\mathsf{DA}[1] = d + 1$ as the suffix $T^{cat}[N, N] = \#$ is always in $\mathsf{SA}[1]$.

[1] Our interest, herein, is limited to main memory algorithms. There exists alternatives to sort all suffixes of a string collection in external memory (e.g. [29–31]) and in parallel (e.g. [32–34]).

[2] In other words, we obtain the same results one would get using distinct separators, but without increasing the size of the alphabet.

Download English Version:

https://daneshyari.com/en/article/4952116

Download Persian Version:

https://daneshyari.com/article/4952116

Daneshyari.com