Journal of Discrete Algorithms 32 (2015) 53–63

Contents lists available at ScienceDirect

Journal of Discrete Algorithms

www.elsevier.com/locate/jda

Improved and extended locating functionality on compressed suffix arrays ☆

Simon Gog [a,*], Gonzalo Navarro [b,*], Matthias Petri [c,*]

[a] Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
[b] Center of Biotechnology and Bioengineering, Department of Computer Science, University of Chile, Chile
[c] Department of Computing and Informations Systems, The University of Melbourne, Australia

ARTICLE INFO

Article history:
Available online 20 January 2015

Keywords:
Compressed suffix array
Inverted indexes
Algorithm engineering

ABSTRACT

Compressed Suffix Arrays (CSAs) offer the same functionality as classical suffix arrays (SAs), and more, within space close to that of the compressed text, and in addition they can reproduce any text fragment. Furthermore, their pattern search times are comparable to those of SAs. This combination has made CSAs extremely successful substitutes for SAs in space-demanding applications. Their weakest point is that they are orders of magnitude slower when retrieving the precise positions of pattern occurrences. SAs have other well-known shortcomings, inherited by CSAs, such as not retrieving those positions in a useful order. In this paper we present new techniques that, on the one hand, improve the current space/time tradeoffs for retrieving pattern occurrences in CSAs, and on the other, efficiently support extended pattern locating functionalities, such as retrieving occurrences in text order or limiting the occurrences to within a text window. Our experimental results display considerable savings with respect to the baseline techniques in many cases of interest: in some cases we outperform their time by a factor of two or three.

1. Introduction

Suffix arrays [16,28] are text indexing data structures that support various pattern matching functionalities. Built on a text $T[1,n]$ over an alphabet $[1,\sigma]$, the most basic functionality provided by a suffix array (SA) is to *count* the number of times a given pattern $P[1,m]$ appears in T. This can be done in $O(m\log n)$ and even $O(m+\log n)$ time [28]. Once counted, SAs can retrieve each of the *occ* positions of P in T in $O(1)$ time (this is called *reporting* or *locating* the pattern occurrences). A suffix array uses $O(n\log n)$ *bits* of space and can be built in $O(n)$ time [25,24,23].

The space usage of suffix arrays, albeit "linear" in classical terms, is asymptotically larger than the $n\lg\sigma$ bits needed to represent T itself.[1] Since the year 2000, two families of compressed suffix arrays (CSAs) have emerged [32]. One family, simply called CSAs [19,20,38,39,18], is built on the compressibility of a so-called Ψ function (see details in the next section), and simulates the basic SA procedure for pattern searching, achieving the same $O(m\log n)$ counting time of basic SAs. A second family, called FM-indexes [6–8,1], built on the Burrows–Wheeler transform [3] of T and on a new concept called

☆ An early partial version of this article appeared in *Proc. SEA 2014* [14]. Funded with Basal Funds FB0001, CONICYT, Chile and ARC grants DP110101743 and DP140103256, Australia.

* Corresponding authors.
E-mail addresses: gog@kit.edu (S. Gog), gnavarro@dcc.uchile.cl (G. Navarro), matthias.petri@unimelb.edu.au (M. Petri).

[1] We use lg to denote logarithm to the base 2.

http://dx.doi.org/10.1016/j.jda.2015.01.006
1570-8667/

backward-search, which allows $O(m \log \sigma)$ and even $O(m)$ time for counting occurrences. The counting times of all CSAs are comparable to those of SAs in practical terms as well [5]. Their space usage can be made asymptotically equal to that of the *compressed* text under the k-th order empirical entropy model, and in all cases it is $O(n \log \sigma)$ bits. Within this space, CSAs support even stronger functionalities than SAs. In particular, they can reconstruct any text segment $T[l, r]$, as well as to compute "inverse" suffix array entries (again, details in the next section), efficiently. Reproducing any text segment allows CSAs to *replace* T, further reducing space.

The weakest part of CSAs in general is that they are much slower than SAs at retrieving the *occ* positions where P occurs in T. SAs require basically *occ* contiguous memory accesses. Instead, both CSA families use a sampling parameter s that induces an extra space of $O((n/s) \log n)$ bits (and therefore s is typically chosen to be $\Omega(\log n)$); then Ψ-based CSAs require $O(s)$ time per reported position and FM-indexes require $O(s \log \sigma)$. In practical terms, all CSAs are orders of magnitude slower than SAs when reporting occurrence positions [5], even when the distribution of the queries is known [10]. Text extraction complexities for windows $T[l, r]$ are also affected by s, but to a lesser degree: they require $O(s + r - l)$ steps.

Although widely acknowledged as a powerful and flexible tool for text searching activities, the SA has some drawbacks that can be problematic in certain applications. The simplest one is that it retrieves the occurrence positions of P in an order that is not left-to-right in the text. This complicates displaying the occurrences in order (unless one obtains and sorts them all), as for example when displaying the occurrences progressively in a document viewer. A related one is that there is no efficient way to retrieve only the occurrences of P that are within a window of T unless one uses $\Omega(n \log n)$ bits of space [27,2,33,21]. This is useful, for example, to display occurrences only within some documents of a collection (T being the concatenation of the documents), for instance only recent news in a collection of news documents.

In this paper we present new techniques that speed up the basic pattern locating functionalities of CSAs, and also efficiently support extended functionalities. Our experimental results show that the new solutions outperform the baseline solutions, in many cases of interest, by a wide margin. The detailed breakdown of our contributions is as follows.

1. We unify the samplings for pattern locating and for displaying text substrings into a single data structure, by using the fact that they are essentially inverse permutations of each other. This yields improved space/time tradeoffs for locating pattern positions and displaying text substrings, especially in memory-reduced scenarios where large values of s must be used.
2. We show that CSAs, which are based on the Ψ function, can be further improved by using two methods of representing increasing lists, which were recently applied successfully to inverted indexes [44,35]. Our experiments show that adapting these techniques to CSAs results in improved space/time tradeoffs for small and large alphabet inputs. For example, for a word parsed text, the new solutions are more than twice as fast as previous state-of-the-art CSA implementations.
3. The *occ* positions of P have variable locating cost in a CSA. We use a data structure that takes $2n + o(n)$ additional bits to report the occurrences of P from cheapest to most expensive, thereby making reporting considerably faster when only some occurrences must be displayed (as in search engine interfaces, or when one displays results progressively and can show a few and process the rest in the background). Our experiments show that, when reporting less than around 15% of the occurrences, this technique is faster than reporting random occurrences, even when the baseline uses those extra $2n + o(n)$ bits to reduce s. A simple alternative that turns out to be very competitive is just to report first the occurrences that are sampled in the CSA, and thus can be reported at basically no cost.
4. Variants of the previous idea have been used for document listing [31] and for reporting positions in text order [33]. We study this latter application in practice. While for showing *all* the occurrences in order it is better to extract and partially sort them, one might need to show only the first occurrences, or might have to show the occurrences progressively. Our implementation becomes faster than the baseline when we report a fraction below 25% of the occurrences, and improves for lower fractions. For example, we report three times faster the first 5% of the occurrences, even letting the baseline spend those $2n + o(n)$ extra bits on a denser sampling.
5. Finally, we extend this second idea to report the text positions that are within a given text window. While the result is not competitive for windows located at random positions of T, our method is faster than the baseline of filtering the text positions by brute force when the window is within the first 15% of T. This is particularly useful in versioned collections or news archives, when the latest versions/dates are those most frequently queried.

The improved sampling we propose is now available in the Succinct Data Structure Library (SDSL). The library contains state-of-the-art `C++11` implementations of many succinct data structures proposed in over 40 research publications. It is available in at https://github.com/simongog/sdsl-lite.

2. Compressed suffix arrays

Let $T[0, n-1]$ be a text over the alphabet $[0, \sigma - 1]$. Then a substring $T[i, n-1]$ is called a *suffix* of T, and is identified with position i. A *suffix array* $SA[0, n-1]$ is a permutation of $[0, n-1]$ containing the positions of the n suffixes of T in increasing lexicographic order (thus the suffix array uses at least $n \lg n$ bits). Since the positions of the occurrences of $P[0, m-1]$ in T are precisely the suffixes of T that start with P, and those form a lexicographic range, *counting* the number of occurrences of P in T is done via two binary searches using SA and T, within $O(m \log n)$ time. Once we find that $SA[sp, ep]$ contains all the occurrences of P in T, their number is $occ = ep - sp + 1$ and their positions are $SA[sp], SA[sp+1], \ldots, SA[ep]$.