Information Processing Letters 114 (2014) 174–178

Contents lists available at ScienceDirect

Information Processing Letters

www.elsevier.com/locate/ipl

Faster semi-external suffix sorting

Jasbir Dhaliwal

School of Computer Science and Information Technology, RMIT University, Australia

ARTICLE INFO

Article history:
Received 11 December 2012
Received in revised form 26 November 2013
Accepted 26 November 2013
Available online 4 December 2013
Communicated by B. Doerr

Keywords:
Data structures
Suffix array
Burrows–Wheeler transform
String algorithms

ABSTRACT

Suffix array (SA) construction is a time-and-memory bottleneck in many string processing applications. In this paper we improve the runtime of a small-space — *semi-external* — SA construction algorithm by Kärkkäinen (TCS, 2007) [5]. We achieve a speedup in practice of 2–4 times, without increasing memory usage. Our main contribution is a way to implement the "pointer copying" heuristic, used in less space-efficient SA construction algorithms, in a memory-efficient way.

1. Introduction

The suffix array (SA) provides efficient solutions to many string processing problems, such as text indexing [1, 2] and repeat detection [3]. In many of these applications SA construction [4] is a time-and-memory bottleneck. To date, SA construction algorithms fall into two sets: large memory and small memory. Algorithms in the first set assume there is RAM sufficient to hold at least the input string, x, and the output SA. The fastest existing algorithms in this set [4] are also "lightweight", using little more memory than is required to hold x and SA, and use a technique called "pointer copying". The second, more recent set of algorithms are slower, but use less RAM and some external memory (disk) [5,6].

In this paper, our focus is on improving an SA construction algorithm by Kärkkäinen [5]. This algorithm is in the small-memory set: it uses only space to hold the input string and some sublinear data structures, and outputs the SA in a seek-friendly way to disk. In this sense, it is a *semi-external* algorithm. Via two algorithmic parameters, the algorithm uses $O(n \log \sigma)$ bits of working space and runs in $O(n \log_\sigma^2 n)$ time in the worst case, with $n \log n$ bits of disk used to store the output SA.

Our main improvement, is a way to incorporate the above mentioned "pointer copying" technique into Kärkkäinen's algorithm, without increasing memory usage (RAM or disk). Our improvements maintain the algorithm's worst-case behaviour but achieve a speedup in practice of 2–4 times.

This paper proceeds as follows. Section 2 sets notation and Section 3 gives an overview of Kärkkäinen's algorithm. Our algorithmic refinements are presented in Sections 4 and 6. Experimental results are in Section 7. Section 8 concludes.

2. Preliminary definitions and notation

Let x be a string of $n+1$ characters, $x = x[0..n] = x[0]x[1]\ldots x[n]$, drawn from a fixed, ordered alphabet Σ of size σ, that require $(n+1)\log\sigma$ bits. The final character, $x[n]$, is a special end-of-string character, \$, which occurs nowhere else in x and is lexicographically smaller than any other character in Σ. We write $x[i, j]$ to represent the substring of x starting at position i and ending at position j. A substring $x[0, i]$, $0 \leqslant i \leqslant n$, that begins at the start of x is called a *prefix* of x; and a substring $x[i, n]$, $0 \leqslant i \leqslant n$, that ends at the end of x is called a *suffix* of x. For brevity, we will often refer to suffix $x[i, n]$ as "suffix i".

The *suffix array* of a string x, which we write as SA, is an array SA[0..n] that contains the permutation of the

E-mail address: jasbir.dhaliwal@student.rmit.edu.au.

0020-0190/\$ – see front matter
http://dx.doi.org/10.1016/j.ipl.2013.11.017

integers $0..n$ such that $x[SA[0]..n] \prec x[SA[1]..n] \prec \cdots \prec x[SA[n]..n]$. SA requires $n \log n$ bits of storage.

Finally, the *Burrows–Wheeler transform* [7], which we write as BWT, is a permutation of x defined by SA: $BWT[i] = x[SA[i] - 1]$, unless $SA[i] = 0$, in which case $BWT[i] = \$$.

3. Kärkkäinen's algorithm

The first step in Kärkkäinen's algorithm is to construct a *difference cover sample* $DCS_v(x)$ of the suffixes of x. We refer the reader to [8,5,9] for details of the data structure, and just note its salient properties here. $DCS_v(x)$ is a data structure that takes $O(n \log n/\sqrt{v})$ bits of space, and allows the relative order of any two suffixes i and j *known to share a common prefix of length* $\ell \geqslant v$ to be determined in $O(1)$ time. Thus, $DCS_v(x)$ allows the relative order of any two suffixes of x to be determined in $O(v)$ time: compare their first v characters, and if there is a character mismatch, the order has been resolved; if there is a tie after v characters, the order is determined in $O(1)$ time by querying $DCS_v(x)$. Parameter v controls a space–time tradeoff.

The next step is to choose and sort a random set of suffixes from x called *splitters*. We enforce that the lexicographically smallest and largest suffixes in the whole string are also in the set of splitters. Once sorted, pairs of lexicographically adjacent splitters (lower- and upper-bound respectively) conceptually partition the SA into blocks. Let $b_{\max}$ be the total number of suffixes we are able to store in RAM (having allowed space for the input string and $DCS_v(x)$).

Kärkkäinen's algorithm constructs SA in rounds, one block at a time, with blocks determined by the splitters. First the leftmost block, containing all suffixes greater than or equal to the leftmost splitter, and all suffixes strictly less than the second splitter, are collected, sorted and written to disk to form a contiguous section of SA. Memory is then reused to process the next block, which will contain suffixes falling between the second and third splitters.

More specifically, a left-to-right scan is made over x while collecting and storing pointers of suffixes that have the same or higher lexicographical order (lexorder) than the lower-bound splitter but lower than the upper-bound splitter. To determine efficiently if a suffix falls between the splitters, the splitters are preprocessed using a Knuth–Morris–Pratt (KMP)-like failure function. If a suffix shares a prefix of v or more with one of the splitters, then its inclusion in the block is determined by $DCS_v(x)$. The KMP preprocessing keeps the total number of character comparisons to $O(n)$ per scan.

By the end of the scan the block contains pointers to suffixes lexicographically between the lower- and upper-bound splitters, but they are not in any particular order. So we sort them using Multikey Quicksort (MKQS) [10] to depth v, and the order of any suffixes that remained tied is determined with $DCS_v(x)$. In this way, the time required to sort b suffixes is $O(vb + b \log b)$ [5].

Of course, because we picked the splitters randomly, there can be cases for which the number of suffixes to be collected exceeds $b_{\max}$. Kärkkäinen provides a clever method for dealing with such cases. Whenever we need to add a suffix to a full block, the scan is halted and the block is sorted using the combination of MKQS and $DCS_v(x)$ described above. The lexicographically larger half of the block is then discarded and the median suffix in the block becomes the new upper-bound splitter. The scan then resumes. This method does not (asymptotically) increase the number of scans.

As noted above, v and $b_{\max}$ allow for different space–time tradeoffs. Apart from x, space usage is mainly dominated by $DCS_v(x)$ and B (a block of SA). $DCS_v(x)$ is computed in $O(|S| \log |S| + v|S|)$ time and in $O(v + |S|)$ space where S is a set of $\Theta(n/\sqrt{v})$ suffixes. In addition, $O(b_{\max}) = O(n/\sqrt{v})$ words for B and $O(n/b_{\max}) = O(\sqrt{v})$ for the splitters, which is $O(n \log n/\sqrt{v})$ in total. Setting $v = \log_\sigma^2 n$ (respectively, $v = \log^2 n$), the total space complexity becomes $O(n \log \sigma)$ (respectively, $O(n)$) bits. The runtime on other hand is dominated by the building and processing of SA blocks, which overall requires $O(n \log n + vn)$ time. With $v = \log_\sigma^2 n$ (respectively, $v = \log^2 n$) this is $O(n \log_\sigma^2 n)$ (respectively, $O(n \log^2 n)$) time.

4. Faster splitter comparisons

We now describe our first optimization. Using suffixes as splitters as Kärkkäinen described is a good general approach to dividing lexicographic space, which in turn allows the SA to be built one block at a time. In practice however, we found a significant boost to runtime is possible if one simply uses character frequencies to divide the suffixes into lexicographic blocks.

More precisely, we make a scan of x and count the frequency of each character. Let $C[0..\sigma]$ be an array containing these character frequencies. We then (conceptually) partition the suffix array by selecting k symbols, $c_0 < c_1 < \cdots < c_k$, such that, for $i < k$, $\sum_{j=c_i}^{c_{i+1}} C[j] < b_{\max}$ and $\sum_{j=c_i}^{c_{i+2}} C[j] > b_{\max}$. In other words, we use the character frequencies to partition SA into blocks containing close to $b_{\max}$ suffixes. Consecutive selected symbols are later used as (respectively) lower- and upper-bound splitters.

This approach is faster than using suffixes as splitters because suffix inclusion in a block is determined with a single symbol comparison, not up to v, as can be the case with Kärkkäinen's original method. Moreover, KMP preprocessing of the v-length prefix of suffix splitters is avoided. When inclusion cannot be determined by one symbol alone, it is easy to have the algorithm fall back to using suffix splitters. In order to enable this when counting characters, we also collect a single suffix that starts with each distinct symbol.

Of course, this idea can be generalized to q-grams, for constant q, or $q = O(\log_\sigma n)$ for packed strings. In practice, we found using bigrams ($q = 2$) provided the best balance of runtime improvement and extra memory usage for the inputs we tested.

5. Pointer copying in a lightweight setting

Pointer Copying is the name given to a class of heuristic methods for suffix sorting where a complete sort of a special subset of suffixes can be used to cheaply derive the