gsufsor t: constructing suffix arrays, LCP arrays and BWTs for string collections

Felipe A. Louza1*, Guilherme P. Telles2, Simon Gog3, Nicola Prezza4 and Giovanna Rosone5*

Abstract

Background: The construction of a suffix array for a collection of strings is a fundamental task in Bioinformatics and in many other applications that process strings. Related data structures, as the Longest Common Prefix array, the Burrows–Wheeler transform, and the document array, are often needed to accompany the suffix array to efficiently solve a wide variety of problems. While several algorithms have been proposed to construct the suffix array for a single string, less emphasis has been put on algorithms to construct suffix arrays for string collections.

Result: In this paper we introduce gsufsor t, an open source software for constructing the suffix array and related data indexing structures for a string collection with N symbols in O(N) time. Our tool is written in ANSI/C and is based on the algorithm gSACA-K (Louza et al. in Theor Comput Sci 678:22–39, 2017), the fastest algorithm to construct suffix arrays for string collections. The tool supports large fasta, fastq and text files with multiple strings as input. Experiments have shown very good performance on different types of strings.

Conclusions: gsfusor t is a fast, portable, and lightweight tool for constructing the suffix array and additional data structures for string collections.

Keywords: Suffix array, LCP array, Burrows–Wheeler transform, Document array, String collections

Background

The suffix array (SA) [1] is one of the most important data structures in string processing. It enables efficient pattern searching in strings, as well as solving many other string problems [2–4]. More space-efficient solutions for such problems are possible by replacing the suffix array with an index based on the Burrows–Wheeler transform (BWT) [5]. Many applications require additional data structures—most commonly, the longest common prefix (LCP) [6] array and the document array (DA) [7]—on top of SA or BWT. These structures, possibly stored in compressed form, serve as a basis for building modern compact full-text indices, which allow to efficiently pre-process and query strings in compact space.

There are several internal memory algorithms designed for constructing the suffix array and additional data structures when the input consists of a single string [8, 9]. While less emphasis has been put on specialized algorithms for string collections, in many applications the input is composed by many strings, and a common approach is concatenating all strings into a single one and using a standard construction algorithm. However, this approach may deteriorate either the theoretical bounds or the practical behavior of construction algorithms due to, respectively, the resulting alphabet size or unnecessary string comparisons [10–12].

Textual documents and webpages are examples of widespread large string collections. In Bioinformatics, important problems on collections of sequences may be solved rapidly with a small memory footprint using the aforementioned data structures, for example, finding suffix-prefix overlaps for sequence assembly [13], clustering...
cDNA sequences [14], finding repeats [15] and sequence matching [16].

In this paper we present gsufsort, an open source tool that takes a string collection as input, constructs its (generalized) suffix array and additional data structures, like the BWT, the LCP array, and the DA, and writes them directly to disk. This way, applications that rely on such data structures may either read them from disk or may easily include gsufsort as a component. Large collections, with up to $2^{64} - d - 2$ total letters in $d$ strings, may be handled provided that there is enough memory. This tool is an extension of previous results [10], with new implementations of procedures to obtain the BWT and the generalized suffix array (GSA) from SA during output to disk, and with the implementation of a lightweight alternative to compute DA.

**Implementation**

gsufsort is implemented in ANSI C and requires a single Make command to be compiled. It may receive a collection of strings in fasta, fastq or raw ASCII text formats and computes SA and related data structures, according to input parameters. gsufsort optionally supports gzipped input data using zlib\(^1\) and kseq\(^2\) libraries. Setting command-line arguments allows selecting which data structures are computed and written on disk, and which construction algorithm is used (see below). Additionally, a function for loading pre-constructed data structures from disk is also provided.

Given a collection of $d$ strings $T^1, T^2, \ldots, T^d$ from an alphabet $\Sigma = \{1, \sigma\}$ of ASCII symbols, having lengths $n_1, n_2, \ldots, n_d$, the strings are concatenated into a single string $T[0, N - 1] = T^1T^2S \cdots T^dS\#$ using the same separator $\$ and an end-marker $\#$, such that $\$ and $\#$ do not occur in any string $T^i$, and $\# < \$ < \alpha$ for any other symbol $\alpha \in \Sigma$. The total length of $T$ is $\sum_{i=1}^{d}(n_i + 1) + 1 = N$.

Before giving details on gsufsort implementation, we briefly recall some data structures definitions. For a string $S$ of length $n$ let the suffix starting at position $i$ be denoted $S_i$, $0 \leq i \leq n - 1$. The suffix array SA of a string $S$ of length $n$ is an array with a permutation of $[0, n - 1]$ that gives the suffixes of $S$ in lexicographic order. The length of the longest common prefix of strings $R$ and $S$ is denoted by $lcp(R, S)$. The LCP array for $S$ gives the $lcp$ between consecutive suffixes in the order of SA, that is $LCP[0] = 0$ and $LCP[i] = lcp(S_{SA[i]}, S_{SA[i-1]})$, $0 < i \leq n - 1$. For a suffix array of a collection of strings, the position $i$ of the document array DA gives the string to which suffix $T_{SA[i]}$ belongs. For the last suffix $T_{N-1} = \#$ we have $DA[0] = d + 1$. The generalized suffix array gives the order of the suffixes of every string in a collection, that is, the GSA is as an array of $N$ pairs of integers $(a, b)$ where each entry $(a, b)$ represents the suffix $T_a^b$, with $1 \leq a \leq d$ and $0 \leq b \leq n_a - 1$.

gsufsort uses algorithm gSACA-K [10] to construct SA for the concatenated string $T[0, N - 1]$, which breaks ties between equal suffixes from different strings $T^i$ and $T^j$ by their ranks, namely $i$ and $j$. gSACA-K can also compute LCP and DA during SA construction, such that LCP values do not exceed separator symbols. gSACA-K runs in $O(N)$ time using $O(\sigma)$ working space.

The BWT is calculated during the output to disk according to its well-known relation to SA [3]

$$BWT[i] = T[(SA[i] - 1) \mod N].$$

The generalized suffix array (GSA) can be computed by gsufsort from SA and DA during the output to disk, using the identity

$$GSA[i] = \begin{cases} (DA[i], SA[i] - \#[DA[i] - 1]) & \text{if } DA[i] > 1 \\ (DA[i], SA[i]) & \text{otherwise} \end{cases}$$

We also provide a lightweight version (gsufsort-light) for the computation of DA. It uses less memory at the price of being slightly slower. It computes a bit-vector $B[0, N - 1]$ with $O(1)$ rank support [4] such that $B[i] = 1$ if $T[i] = \#$, and $B[i] = 0$ otherwise. The values in DA are obtained on-the-fly while DA (or GSA) is written to disk, through the identity

$$DA[i] = \text{rank}_1(SA[i]) + 1.$$

**Results**

We compared our tool and mkESA. mkESA [17] is a fast suffix array construction software designed for bioinformatics applications.

We ran both versions of our tool, gsufsort and gsufsort-light, to build arrays GSA and LCP, while mkESA\(^3\) was run to build arrays SA and LCP for the concatenation of all strings (using the same symbol as separators). The experiments were conducted on a single core of a machine with GNU/Linux (Debian 8, kernel 3.16.0-4, 64 bits) with an Intel Xeon E5-2630 2.40-GHz, 384 GB RAM and 13 TB SATA storage. The sources were compiled by GNU GCC version 4.8.4 with option -O3.

The collections we used in our experiments are described in Table 1. They comprise real DNAs, real proteins, documents, random DNA and random protein, and differ by their alphabet size and also by the maximum

---

\(^1\)https://zlib.net
\(^2\)http://lh3lh3.users.sourceforge.net/kseq.shtml
\(^3\)http://www.obiserv.cebitec.uni-bielefeld.de/mkesa
and average \( lcp \), which offer an approximation for suffix sorting difficulty.

The results are shown in Table 2. The data shows a clear time/memory tradeoff for DNA sequences, \texttt{gsufsort} being faster while using approximately 1.25 more memory, \texttt{gsufsort-light} using slightly less memory than \texttt{mkESA} but taking more time. On proteins, \texttt{gsufsort-light} is only marginally slower than \texttt{gsufsort} but faster than \texttt{mkESA}. The authors of \texttt{mkESA} reported a 32% gain on a large protein dataset using 16 threads [17], but larger \( lcp \) values seem not to favor \texttt{mkESA} when compared to \texttt{gsufsort-light}, which is 47.9% faster on proteins and 12.9% faster on DNA.

The memory ratio (bytes/N) of \texttt{gsufsort} and \texttt{gsufsort-light} is constant, 21 and 17 bytes per input symbol respectively, corresponding to the space of the input string \( T \) (N bytes) plus the space for arrays \( SA \) and \( LCP \) (8N bytes each) and, only for \texttt{gsufsort}, the space for DA (4N bytes).

We have also evaluated the performance of \texttt{gsufsort}, \texttt{gsufsort-light} and \texttt{mkESA} on collections of random DNA and random protein sequences. The collections have a growing number of 1MB sequences. The running time in seconds and the peak memory usage in GB are shown in Fig. 1 (logarithmic scale). Using random sequences reduces the variation due to \( lcp \) among

Table 1 Collections

| Collection     | size  | \( \sigma \) | N. of strings | Max. len. | Avg. len | Max. lcp | Avg. lcp |
|----------------|-------|-------------|--------------|-----------|----------|----------|----------|
| shortreads     | 16.00 | 5           | 171.8        | 100       | 100      | 100      | 32.87    |
| reads          | 16.00 | 6           | 57.3         | 300       | 300      | 300      | 91.29    |
| pacbio         | 16.00 | 5           | 1.9          | 71,561    | 9117     | 3084     | 19.08    |
| pacbio.1000    | 16.00 | 5           | 17.2         | 1,000     | 1000     | 876      | 18.67    |
| uniprot        | 16.04 | 25          | 46.1         | 74,488    | 374      | 74,293   | 99.24    |
| gutenberg      | 15.88 | 255         | 334.3        | 757,936   | 50       | 9060     | 18.97    |
| random.dna     | 16.00 | 4           | 16.1         | 1,048,576 | 33       | 16.18    |          |
| random.protein | 16.00 | 25          | 16.1         | 1,048,576 | 13       | 6.89     |          |

Columns 2 and 3 show the collection size (in GB) and the alphabet size. Column 4 shows the number of strings (in millions). Columns 5 and 6 show the maximum and average lengths of strings in a collection. Columns 7 and 8 show the maximum and average \( lcp \) of strings in a collection.

Collections:
- \texttt{shortreads} are Illumina reads from human genome trimmed to 100 nucleotides (http://ftp.sra.ebi.ac.uk/vol1/ERA015/ERA015743/srf);
- \texttt{reads} are Illumina HiSeq 4000 paired-end RNA-seq reads from plant \textit{Setaria viridis} trimmed to 300 nucleotides (http://www.trace.ncbi.nlm.nih.gov/Traces/sra/?run=ERR1942989);
- \texttt{pacbio} are PacBio RS II reads from \textit{Triticum aestivum} (wheat) genome (http://www.trace.ncbi.nlm.nih.gov/Traces/sra/?run=SRR5816116);
- \texttt{pacbio.1000} are strings from pacbio trimmed to length 1,000;
- \texttt{uniprot} are protein sequences from TrEMBl downloaded on May 28, 2019 (http://www.ebi.ac.uk/uniprot/download-center);
- \texttt{gutenberg} are ASCII books in English from Project Gutenberg (http://www.gutenberg.org);
- \texttt{random.dna} was generated with even sampling probability on the standard 4 letter alphabet;
- \texttt{random.protein} was generated with even sampling probability on the IUPAC 25 letter alphabet.

Table 2 Algorithms’ running times and memory usage on different datasets collections

| Collection     | \texttt{gsufsort} | \texttt{gsufsort-light} | \texttt{mkESA} |
|----------------|------------------|------------------------|---------------|
|                | Time    | RAM     | Bytes/N | Time    | RAM     | Bytes/N | Time    | RAM     | Bytes/N |
| shortreads     | 4:25:52 | 336.00  | 21.00   | 4:51:48 | 274.73  | 17.17   |
| reads          | 5:00:27 | 336.00  | 21.00   | 4:10:04 | 272.00  | 17.00   |
| pacbio         | 4:19:37 | 336.04  | 21.00   | 4:44:50 | 272.32  | 17.02   |
| pacbio.1000    | 4:28:22 | 336.00  | 21.00   | 4:26:39 | 272.58  | 17.03   |
| uniprot        | 5:13:13 | 336.90  | 21.00   | 4:44:50 | 272.32  | 17.02   |
| gutenberg      | 4:17:52 | 334.40  | 21.00   | 9:58:03 | 294.86  | 18.38   |
| random.dna     | 4:23:56 | 331.08  | 21.00   | 4:28:34 | 269.02  | 17.00   |
| random.protein | 5:00:06 | 331.08  | 21.00   | 4:37:16 | 268.33  | 17.02   |

Columns RAM and bytes/N show the peak memory in GB and the bytes per input symbol ratio. Each symbol of \( T \) uses 1 byte. Results for \texttt{gutenberg} are reported for \texttt{gsufsort} and \texttt{gsufsort-light} only, as \texttt{mkESA} is restricted to DNA and amino-acid alphabets. The best results are indicated in italics.
We can see a perfectly steady behavior of \textit{mkESA}. While still $O(N)$, \textit{gsufsort} displays a deviation due to larger constants.

**Conclusions**

We have introduced \textit{gsufsort}, a fast, portable, and lightweight tool for constructing the suffix array and additional data structures for string collections. \textit{gsufsort} may be used to pre-compute indexing structures and write them to disk, or may be included as a component in different applications. As an additional advantage, \textit{gsufsort} is not restricted to biological sequences, as it can process collections of strings over ASCII alphabets.

**Availability and requirements**

- Project name: \textit{gsufsort}
- Project home page: http://www.github.com/felipelouza/gsufsort
- Operating system(s): Platform independent
- Programming language: ANSI C
- Other requirements: make, zlib (optional)
- License: GNU GPL v-3.0.

**Acknowledgements**

The authors thank Prof. Nalvo Almeida (UFMS, Brazil) for granting access to the machine used for the experiments.

**Authors’ contributions**

FAL and GR devised the main algorithmic idea. FAL, GPT, SG, NP and GR contributed to improve the algorithms and participated in their implementations. NP designed and performed the experiments. All authors read and approved the final manuscript.

**Funding**

FAL and GPT acknowledge the financial support of Brazilian Agencies CNPq and CAPES. GR is partially and NP is supported by the project MIUR-SIR CMACBioSeq ("Combinatorial methods for analysis and compression of biological sequences") grant n. RBS1146R5L.

**Availability**

The source code of the proposed algorithm is available at https://www.github.com/felipelouza/gsufsort.
Ethics approval and consent to participate
Not applicable.

Consent for publication
Not applicable.

Competing interests
The authors declare that they have no competing interests.

Author details
1 Faculdade de Engenharia Elétrica, Universidade Federal de Uberlândia, Uberlândia, Brazil. 2 Instituto de Computação, Universidade Estadual de Campinas, Campinas, Brazil. 3 eBay Inc., San Jose, USA. 4 LUISS Guido Carli, University, Rome, Italy. 5 Dipartimento di Informatica, Università di Pisa, Pisa, Italy.

Received: 24 April 2020 Accepted: 8 September 2020
Published online: 22 September 2020

References
1. Manber U, Myers EW. Suffix arrays: a new method for on-line string searches. SIAM J Comput. 1993;22(5):935–48.
2. Mäkinen V, Belazzougui D, Cunial F, Tomescu AI. Genome-scale algorithm design. Cambridge: Cambridge University Press; 2015.
3. Ohlebusch E. Bioinformatics algorithms: sequence analysis, genome rearrangements, and phylogenetic reconstruction. Bremen: Oldenbusch; 2013.
4. Navarro G. Compact data structures: a practical approach. Cambridge: Cambridge University Press; 2016.
5. Burrows M, Wheeler DJ. A block-sorting lossless data compression algorithm. Technical report, Digital SRC Research Report; 1994.
6. Fischer J, Wee LCP. Inf Process Lett. 2010;110(8–9):317–20.
7. Muthukrishnan S. Efficient algorithms for document retrieval problems. In: Proceedings of the ACM-SIAM symposium on discrete algorithms (SODA). ACM/SIAM, San Francisco-CA, USA; 2002. p. 657–66.
8. Puglisi SJ, Smyth WF, Turpin AH. A taxonomy of suffix array construction algorithms. ACM Comput Surv. 2007;39(2):1–31.
9. Dhahbi J. Faster semi-external suffix sorting. Inf Process Lett. 2014;114(4):174–8.
10. Louza FA, Gog S, Telles GP. Inducing enhanced suffix arrays for string collections. Theor Comput Sci. 2017;678:22–39.
11. Mantaro S, Restivo A, Rosone G, Sciortino M. An extension of the Burrows–Wheeler transform. Theor Comput Sci. 2007;387(3):298–312.
12. Bauer MJ, Cox AJ, Rosone G. Lightweight algorithms for constructing and inverting the BWT of string collections. Theor Comput Sci. 2013;483:134–48.
13. Simpson JT, Durbin R. Efficient construction of an assembly string graph using the FM-index. Bioinformatics. 2010;26(12):367–73.
14. Hazelhurst S, Lipták Z. Kaboom! A new suffix array based algorithm for clustering expression data. Bioinformatics. 2011;27(24):3348–55.
15. Askitis N, Sinha R. Repmaestro: scalable repeat detection on disk-based genome sequences. Bioinformatics. 2010;26(19):2368–74.
16. Vyverman M, De Baets B, Fack V, Dawyndt P, essaMEM: finding maximal exact matches using enhanced sparse suffix arrays. Bioinformatics. 2013;29:802–4.
17. Homann R, Fleer D, Giegerich R, Rehmsmeier M, mkESA: enhanced suffix array construction tool. Bioinformatics. 2009;25:1084–5.

Publisher’s Note
Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Ready to submit your research? Choose BMC and benefit from:
• fast, convenient online submission
• thorough peer review by experienced researchers in your field
• rapid publication on acceptance
• support for research data, including large and complex data types
• gold Open Access which fosters wider collaboration and increased citations
• maximum visibility for your research: over 100M website views per year

At BMC, research is always in progress.
Learn more biomedcentral.com/submissions