A Grammar Compression Algorithm based on Induced Suffix Sorting

Daniel Saad Nogueira Nunes[1,2], Felipe A. Louza[3] *,
Simon Gog[4], Mauricio Ayala-Rincón[2] and Gonzalo Navarro[5]

[1]Federal Institute of Education, Science and Technology of Brasília, Brazil
daniel.nunes@ifb.edu.br
[2]Department of Computer Science, University of Brasília, Brazil
ayala@unb.br
[3]Department of Computing and Mathematics, University of São Paulo, Brazil
louza@usp.br
[4]Institute of Theoretical Informatics, Karlsruhe Institute of Technology, Germany
gog@kit.edu
[5]Center for Biotechnology and Bioengineering
Department of Computer Science, University of Chile, Chile
gnavarro@dcc.uchile.cl

Abstract

We introduce GCIS, a grammar compression algorithm based on the induced suffix sorting algorithm SAIS, presented by Nong *et al.* in 2009. Our solution builds on the factorization performed by SAIS during suffix sorting. We construct a context-free grammar on the input string which can be further reduced into a shorter string by substituting each substring by its corresponding factor. The resulting grammar is encoded by exploring some redundancies, such as common prefixes between suffix rules, which are sorted according to SAIS framework. When compared to well-known compression tools such as Re-Pair and 7-zip under repetitive sequences, our algorithm is faster at compressing and achieves compression ratio close to that of Re-Pair, at the cost of being the slowest at decompressing.

Introduction

Text compression consists in transforming an input string into another string whose bit sequence representation is smaller. Given the suffix array [1, 2] of a string, one can compute efficiently the Burrows-Wheeler transform (BWT) [3] and the Lempel-Ziv factorization (LZ77) [4, 5, 6, 7], which are at the heart of the popular data compression tools 7-ZIP and GZIP [8].

In 2009, Nong *et al.* [9] introduced a remarkable algorithm called SAIS, which runs in linear time and is fast in practice to construct the suffix array. Subsequently, SAIS was adapted to compute directly the BWT [10], the Φ-array [11, 7], the LCP array [12], and the suffix array for string collections [13].

In this article we introduce GCIS, a new grammar-based compression algorithm that builds on SAIS. We construct a context-free grammar based on the string factorization performed by SAIS recursively. The rules are encoded according to the

* FAL was supported by the grant #2017/09105-0 from the São Paulo Research Foundation (FAPESP). GN was funded by Basal Funds FB0001 and Fondecyt Grant 1-170048, Conicyt, Chile.

2375-0359/18/$31.00 ©2018 IEEE
DOI 10.1109/DCC.2018.00012

length of longest common prefixes between consecutive rules, which are sorted lexicographically by SAIS.

Our experiments show that, regarding repetitive strings and compared to Re-Pair [14] and 7-zip [15], GCIS is an interesting alternative, because it displays the fastest compression time and reaches a compression ratio close to that of Re-Pair, although it is the slowest at decoding. GCIS utilizes a novel grammar compression framework, being the first, as far as we know, based on induced suffix sorting.

Background

Let T be a string of length $|T| = n$, $T = T[1, n] = T[1] \cdot T[2] \ldots \cdot T[n]$, over a fixed ordered alphabet Σ. A constant alphabet has size $\sigma \in O(1)$ and an integer alphabet has size $\sigma \in n^{O(1)}$. We denote the concatenation of strings or symbols by the dot operator $(\cdot)$, which can be omitted. We use the symbol $<$ for the lexicographic order relation between strings.

For convenience, we assume that T always ends with a special symbol $T[n] = \$$, which is not present elsewhere in T and lexicographically precedes every symbol in Σ. Let $T[1, j]$ be the prefix of T that ends at position j, and $T[i, n]$ be the suffix of T that starts at position i also denoted as T_i by brevity. We denote the length of the longest common prefix of two strings T_1 and T_2 in Σ^* by $\texttt{lcp}(T_1, T_2)$.

The suffix array (SA) [1, 2] 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 $T_{\mathsf{SA}[1]} < T_{\mathsf{SA}[2]} < \ldots < T_{\mathsf{SA}[n]}$. The suffixes starting with the same symbol $c \in \Sigma$ form a c-bucket in the suffix array. The head and the tail of a bucket refer to the first and the last position of the bucket in SA.

Let $G = (\Sigma, \Gamma, P, X_S)$ be a reduced context-free grammar (does not contain unreachable non-terminals). Σ is the terminal alphabet of G; Γ is the set of non-terminals symbols that is disjoint from Σ; $P \subseteq \Gamma \times (\Sigma \cup \Gamma)^*$ is the set of production rules; and $X_S \in \Gamma$ is the start symbol. A production rule (X_i, α_i) is also denoted by $X_i \rightarrow \alpha_i$. We say that α_i is derived from X_i. For strings $s, t \in (\Sigma \cup \Gamma)^*$, we say that t derives from s if it is obtained by application of a production rule in P; we say that t is generated from s if t is obtained by a sequence of derivations from s. We define $|G|$ as the total length of the strings on the right side of all rules.

Given a string T, grammar compression is to find a grammar G which generates only T such that G can be represented in less space than the original T. Given that G grammar-compresses T, for $(X_i, \alpha_i) \in P$, we define $\mathcal{F}(X_i) = s$ as the single string $s \in \Sigma^*$ that is generated from α_i. The language generated by G is $L(G) = \mathcal{F}(X_S)$.

Related work

SAIS [9] builds on the induced suffix sorting technique introduced by previous algorithms [16, 17]. Induced suffix sorting consists in deducing the order of unsorted suffixes from a set of already ordered suffixes.

The next definition classifies suffixes and symbols of strings.

Definition 1 (L-type and S-type) *For any string T, $T_n = \$$ has type S. A suffix T_i is an S-suffix if $T_i < T_{i+1}$, otherwise T_i is an L-suffix. $T[i]$ has the type of T_i.*

The suffixes can be classified in linear time by scanning T once from right to left. The type of each suffix is stored in a bitmap of size n.

Note that, in a c-bucket, all L-suffixes precede to the S-suffixes.

Further, the classification of suffixes is refined as below:

Definition 2 (LMS-type) *Let T be a string. T_i is an LMS-suffix if T_i is an S-suffix and T_{i-1} is an L-suffix.*

Nong *et al.* [9] showed that the order of the LMS-suffixes is enough to induce the order of all suffixes.

SAIS works as follows:

SAIS framework:

1. Sort the LMS-suffixes. This step is explained below.

2. Insert the LMS-suffixes into their c-buckets in SA, without changing their order.

3. Induce L-suffixes by scanning SA from left to right: for each suffix $\mathsf{SA}[i]$ if $T[\mathsf{SA}[i]-1]$ is L-type, insert $\mathsf{SA}[i]-1$ into the head of its bucket.

4. Induce S-suffixes by scanning SA from right to left: for each suffix $\mathsf{SA}[i]$ if $T[\mathsf{SA}[i]-1]$ is S-type, insert $\mathsf{SA}[i]-1$ into the tail of its bucket.

We say that whenever a value is inserted in the head (or tail) of a bucket,the head (or tail) is increased (or decreased) by one.

In order to sort the LMS-suffixes in Step 1, $T[1,n]$ is divided (factorized) into LMS-substrings.

Definition 3 *$T[i,j]$ is an LMS-substring if both T_i and T_j are LMS-suffixes, but no suffix between i and j has LMS-type. The last suffix T_n is an LMS-substring.*

Let $r^1_1, r^1_2, \ldots, r^1_{n^1}$ be the n^1 LMS-substrings of T read from left-to-right. A modified version of SAIS is applied to sort the LMS-substrings. Starting from Step 2, $T[1,n]$ is scanned (right-to-left) and each unsorted LMS-suffix is inserted (bucket-sorted) regarding its first symbol at the tail of its c-bucket. Steps 3 and 4 work exactly the same. At the end, all LMS-substrings are sorted and stored in their corresponding c-buckets in SA.

Naming:

A *name* v^1_i is assigned to each LMS-substring r^1_i according to its lexicographical rank in $[1,\sigma^1]$, such that $v^1_i < v^1_j$ if $r^1_i < r^1_j$, $v^1_i = v^1_j$ if $r^1_i = r^1_j$ and σ^1 is the number of different LMS-substrings in T. In order to compute the names, each consecutive LMS-substrings in SA, say r^1_i and r^1_{i+1}, are compared to determine if either $r^1_i = r^1_{i+1}$ or $r^1_i < r^1_{i+1}$. In the former case v^1_{i+1} is named as v^1_i, whereas in the latter case v^1_{i+1} is named as $v^1_i + 1$. This procedure may be sped up by comparing the LMS-substrings first by symbol and then by type, with L-type symbols being smaller than S-type symbols in case of ties [18].

Recursive call:

A new (reduced) string $T^1 = v_1^1 \cdot v_1^1 \cdots v_{n^1}^1$ is created, whose length n^1 is at most $n/2$, and the alphabet size σ^1 is integer. If every $v_i^1 \neq v_j^1$ then all LMS-suffixes are already sorted. Otherwise, SAIS is recursively applied to sort all the suffixes of T^1. Nong *et al.* [9] showed that the relative order of the LMS-suffixes in T is the same as the order of the respective suffixes in T^1. Therefore, the order of all LMS-suffixes can be determined by the result of the recursive algorithm.

Grammar Compression by Induced Suffix Sorting

In this section we introduce the grammar compression by induced sorting (GCIS), which is based on SAIS.

First, we compute a context-free grammar $G = (\Sigma, \Gamma, P, X_S)$ that generates only $T[1, n]$. To do this we modify SAIS as follows.

Grammar construction:

Considering the j-th recursion level, in Step 1, after the input string $T^j[1, n]$ is divided into the LMS-substrings $r_1^j, r_2^j, \ldots, r_{n^j}^j$ and named into $v_1^j, v_2^j, \ldots, v_{n^j}^j$, we create a new rule $X_i \rightarrow \alpha_i$ for each different LMS-substring $r_i^j = T^j[a, b]$ in the form $r(v_i^j) \rightarrow T^j[a, b-1]$, where $r(v_i^j) = v_i^j + \sum_{k=1}^{j-1} \sigma^k$. Moreover, we create an additional rule $r(0^j) \rightarrow T^j[1, j_1 - 1]$ for the prefix of T^j that is not included in the first LMS-substring r_1^j.

The algorithm is called recursively with the reduced string $T^{j+1} = v_1^j \cdot v_2^j \cdots v_{n^j}^j$ as input while $\sigma^j < n^j$, that is, the LMS-substrings are not pairwise distinct. At the end, when $\sigma^j = n^j$, we create the start symbol of G as being X_S, such the production $X_S \rightarrow r(0^j) \cdot r(v_1^j) \cdot r(v_2^j) \cdots r(v_{n^j}^j)$ generates only the original string $T[1, n]$.

The algorithm stops after computing X_S, since we are not interested in constructing the suffix array, we do not execute Steps 2, 3 and 4 of SAIS. The recursive calls return to the top level and we have computed a grammar G that generates only $T[1, n]$.

Since for each LMS-substring a unique $r(v_i^j)$ exists, there are no cycles in any derivations, and $L(G) = T$, we have that G is a grammar that compresses T [19].

Grammar compression:

Consecutive entries in the set of productions P are likely to share a common prefix, since the LMS-substrings are given lexicographically ordered by SAIS. Therefore, each rule $X_i \rightarrow \alpha_i \in P$ is encoded using two values $(\ell_i, \mathrm{s}(\alpha_i))$, such that ℓ_i encodes the length of longest common prefix ($\texttt{lcp}$) between α_{i-1} and α_i, and the remaining symbols of α_i are given by $\mathrm{s}(\alpha_i) = \alpha_i[\ell_i + 1, |\alpha_i|]$. This technique is known as Front-coding [20].

The computation of $(\ell_i, \mathrm{s}(\alpha_i))$ is performed with no additional cost with a slight modification in the naming procedure of SAIS. Each consecutive LMS-substring in SA, say r_{i-1}^j and r_i^j are compared first by symbol and then by type to check if either $r_{i-1}^j = r_i^j$ or $r_{i-1}^j < r_i$. In order to compute $\texttt{lcp}(r_{i-1}^j, r_i^j)$ we compare them only by

Table 1: `Simple8b` possible arrangements [21].

Selector value	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Item width	0	0	1	2	3	4	5	6	7	8	10	12	15	20	30	60
Group Size	240	120	60	30	20	15	12	10	8	7	6	5	4	3	2	1
Wasted bits	60	60	0	0	0	0	0	0	4	4	0	0	0	0	0	0

symbol until finding the first mismatch. The resulting order is the same with a small slowdown in the running time.

Computational cost:

GCIS runs in $O(n)$ time, since each step of the modified SAIS is linear and the length of the reduced string T^j is at most $|T^{j-1}|/2$.

Implementation details

In this section we discuss implementation details of the GCIS encoding and decoding processes.

Encoding:

A rule X_i is derived into a pair $\alpha_i = (\ell_i, \mathrm{s}(\alpha_i))$, where ℓ_i equals $\texttt{lcp}(\alpha_{i-1}, \alpha_i)$ and $\mathrm{s}(\alpha_i)$ corresponds to the remaining $\alpha_i[\ell_i + 1, |\alpha_i|]$ symbols. The ℓ values tend to be small and, considering the j-th recursion value, the sum of such values cannot be greater than n^j, since no two LMS-substrings overlap.

One can encode all ℓ values into a sequence of computer words L by using `Simple8b` encoding [21]. This technique packs a number of small integers in a 64-bit word using the number of bits required by the largest integer. Basically it identifies a word with a 4-bit tag called *selector*, which specifies the number of integers encoded in a single word and the width of such integers. `Simple8b` also has specific selectors for a run consisting of zeroes. If a run of 240 or 120 zeros is encountered, it can be represented with a single 64-bit word. Table 1 contain all possible selector values, which reflects the possible arrangements of fixed-width integers storage in a single 64-bit word under this encoding scheme.

All $\mathrm{s}(\alpha_i)$ are encoded in a single fixed-width integer array R, consisting of width $\lfloor \lg(\sigma^j) \rfloor + 1$ bits. The length of each $\mathrm{s}(\alpha_i)$ is also encoded using `Simple8b` into a word array S. The same observation of the `lcp` sum can be done here: the sum of all $|\mathrm{s}(\alpha_i)|$ is no larger than n^j.

A greedy strategy was employed to stop the recursion when the dictionary size of the $(j+1)$-th level plus the size in bits of T^{j+1} is bigger than the size in bits of T^j. In this situation, the computation done on the j-th level is discarded and the algorithm stops. When this condition is met, $\sigma^j < n^j$, but this does not interfere on the decoding algorithm.

Decoding:

The decoding process is done level-wise, starting from the last level, by decoding the right side of each rule. In the j-th level, the values (x, y, z) from L, R and S are decoded in a sequential way. In order to compute α_{k+1} from α_k, the first x symbols of α_k are copied to α_{k+1} and the z symbols from R, which correspond to the string y, are copied to α_{k+1} as well. A bitmap D is built to contain the length of all α_i by using `Rice-coding`. With two $\texttt{select}_1$ operations it is possible to query the starting point of each α_i in this array and the length $|\alpha_i|$ in constant time using $2n^j + o(n^j)$ bits, where j corresponds to the j-th recursive step of the grammar construction.

Once all rules are expanded into a fixed-width integer array of $\lfloor \lg(\sigma^j) \rfloor + 1$ bits, T^{j-1} can be decoded from T^j. First, the right side of $r(0^j)$ is copied into T^{j-1}. Then, T^j is scanned in a left-to-right fashion and for each $T^j[i]$ the algorithm copies a substring to T^{j-1} which equals the right side of $r(T^j[i])$ and can be easily found with the bitmap D support.

Experiments

We compared GCIS with Re-Pair[1] and 7-zip[2] regarding Pizza&Chili Repetitive *Corpus*[3] under the subjects of compression ratio, compression and decompression running time. In particular, we used a space-efficient implementation of Re-Pair by Wan [22], which encodes each rule with one integer plus few bits. GCIS was implemented in `C++11` using the Succinct Data Structure Library (SDSL) [23].

All experiments were conducted on machine with `2x Intel(R) Xeon(R) CPU E5-2407 v2 @ 2.40GHz` CPUs and 256GB of RAM memory. The operating system used was based on the Debian GNU/Linux O.S. The input size of each experiment is given in the second column of Tables 2, 3 and 4. The source code of GCIS is publicly available at `https://github.com/danielsaad/gcis`.

Experimental results show that our algorithm is effective at handling repetitive strings. Regarding 7-zip and Re-Pair, it shows to be faster at compressing, having a slightly worse compression ratio (but comparable) and slower at decoding.

Compression and decompression:

Table 2 comprises the compression Ratio (%), corresponding to the size of the compressed text over the original input size. 7-zip presents the best compression ratio, except for `coreutils`, `fib41`, `rs` and `tm29`, where Re-Pair outperforms it. Note that GCIS obtains slightly worse compression ratio than Re-Pair.

Table 3 shows the compression time of each algorithm. GCIS is the fastest algorithm, except for `einstein.de`, `einstein.en` and `proteins`, where 7-zip was the fastest. GCIS outperforms Re-Pair and 7-zip by a large margin in most cases, being up to 6.5 times faster than Re-Pair (`tm29`) and up to 6.9 times faster than 7-zip (`cere`).

[1] `https://github.com/rwanwork/Re-Pair`

[2] `http://p7zip.sourceforge.net/`

[3] `http://pizzachili.dcc.uchile.cl/repcorpus.html`

Table 2: Compression ratio regarding Pizza&Chili repetitive *corpus*.

Pizza&Chili Repetitive *Corpus*		Compression Ratio (%)		
Experiment	Input Size (MB)	GCIS	Re-Pair	7-zip
`cere`	461.29	3.76	1.86	**1.82**
`coreutils`	205.28	5.39	**2.54**	11.63
`dblp.xml.00001.1`	104.86	0.43	0.19	**0.16**
`dblp.xml.00001.2`	104.86	0.43	0.18	**0.16**
`dblp.xml.0001.1`	104.86	0.84	0.46	**0.20**
`dblp.xml.0001.2`	104.86	0.77	0.39	**0.19**
`dna.001.1`	104.86	3.55	2.43	**0.51**
`einstein.de.txt`	92.76	0.31	0.16	**0.11**
`einstein.en.txt`	467.63	0.20	0.10	**0.07**
`english.001.2`	104.86	4.17	2.41	**0.55**
`escherichiacoli`	112.69	14.14	9.60	**6.56**
`fib41`	267.91	0.03	**0.00**	0.36
`influenza`	154.81	4.76	3.26	**1.65**
`kernel`	257.96	2.37	1.10	**0.82**
`para`	429.27	4.98	2.74	**2.39**
`proteins.001.1`	104.86	4.13	2.64	**0.59**
`rs.13`	216.75	0.02	**0.00**	0.16
`sources.001.2`	104.86	4.10	2.34	**0.45**
`tm29`	268.44	0.02	**0.00**	0.72
`world_leaders`	46.97	3.38	1.79	**1.39**

Table 4 presents the decompression time of each algorithm. 7-zip outperforms Re-Pair and GCIS, except for `fib41`, `rs` and `tm29`, where Re-Pair was the fastest. GCIS is up to 20 times slower than Re-Pair and 7-zip (`einstein.en`), whereas Re-Pair is up to 6.6 times slower than 7-zip (`cere`).

Peak memory:

We evaluated the peak memory consumption of Re-Pair and GCIS in compression and decompression procedures. 7-zip and was not evaluated since it require negligible amount of space when compressing or decompressing.

Figure 1a shows that GCIS requires five times less the space needed by Re-Pair during compression. Since GCIS is based on SAIS, it requires $\approx 5 \times n$ bytes, for inputs with $n <$ 4GB, whereas Re-Pair requires $\approx 30 \times n$ bytes, becoming prohibitive when the input is large. In decompression, illustrated by 1b, Re-Pair has a lower peak memory usage than GCIS, making the former more appealing when memory is limited.

Table 3: Compression time regarding Pizza&Chili repetitive *corpus*.

Pizza&Chili Repetitive *Corpus*		Compression Time (s)		
Experiment	Input Size (MB)	GCIS	Re-Pair	7-zip
cere	461.29	**100.61**	464.62	693.10
coreutils	205.28	**44.48**	210.21	85.19
dblp.xml.00001.1	104.86	**21.34**	71.85	25.63
dblp.xml.00001.2	104.86	**21.59**	72.31	25.60
dblp.xml.0001.1	104.86	**21.21**	72.35	25.79
dblp.xml.0001.2	104.86	**21.76**	73.70	27.16
dna.001.1	104.86	**19.48**	73.83	63.56
einstein.de.txt	92.76	22.48	62.17	**16.26**
einstein.en.txt	467.63	135.19	338.30	**85.02**
english.001.2	104.86	**27.79**	93.61	41.36
escherichiacoli	112.69	**22.42**	138.06	143.05
fib41	267.91	**15.58**	77.35	29.36
influenza	154.81	**26.64**	108.98	46.14
kernel	257.96	**60.26**	223.52	120.18
para	429.27	**95.93**	512.93	583.92
proteins.001.1	104.86	29.05	82.86	**21.27**
rs.13	216.75	**12.04**	69.58	22.88
sources.001.2	104.86	**23.56**	85.69	31.16
tm29	268.44	**14.33**	92.70	39.11
world_leaders	46.97	**5.98**	23.57	9.26

Conclusions

In the article we introduced a new grammar-based compression algorithm, called GCIS, which is based on the induced suffix sorting framework of SAIS [9]. Our experiments show that GCIS is faster than Re-Pair and 7-zip at compressing, while obtaining a compression ratio close to that of Re-Pair. In exchange, Re-Pair is faster at decompressing.

Future work: As a future work, one can think of a GCIS/Re-Pair hybrid approach The key idea is to encode the first recursive levels using GCIS and then shift to Re-Pair. While making the compression a little slower, this approach can make decompression faster while preserving a good compression ratio.

We remark that GCIS, as well as Re-Pair, can support extract random substrings $T[l, r]$ without decompressing the complete string $T[1, n]$, by storing additional data structures [24], whereas such operation is not possible for LZ77 based compressors [25]. We intend to implement this operation aiming at reducing its memory footprint. Also, an efficient way to search for a pattern in the compressed text is desirable.

Table 4: Decompression time regarding Pizza&Chili repetitive *corpus*.

Pizza&Chili Repetitive *Corpus*		Decompression Time (s)		
Experiment	Input Size (MB)	GCIS	Re-Pair	7-zip
cere	461.29	18.88	13.31	**2.01**
coreutils	205.28	13.53	3.95	**2.37**
dblp.xml.00001.1	104.86	5.61	0.82	**0.34**
dblp.xml.00001.2	104.86	5.62	0.85	**0.34**
dblp.xml.0001.1	104.86	5.58	0.85	**0.34**
dblp.xml.0001.2	104.86	5.65	1.04	**0.34**
dna.001.1	104.86	6.31	1.75	**0.37**
einstein.de.txt	92.76	5.57	0.45	**0.29**
einstein.en.txt	467.63	29.40	2.70	**1.43**
english.001.2	104.86	7.48	3.76	**0.37**
escherichiacoli	112.69	7.49	3.36	**0.87**
fib41	267.91	11.55	**0.53**	1.09
influenza	154.81	9.16	1.09	**0.67**
kernel	257.96	16.50	5.96	**0.94**
para	429.27	19.18	12.88	**2.16**
proteins.001.1	104.86	7.69	2.45	**0.38**
rs.13	216.75	9.19	**0.43**	0.71
sources.001.2	104.86	6.93	3.21	**0.36**
tm29	268.44	10.26	**0.53**	1.16
world_leaders	46.97	1.66	0.45	**0.20**

References

[1] U. Manber and E. W. Myers, "Suffix arrays: A new method for on-line string searches," *SIAM J. Comput.*, vol. 22, no. 5, pp. 935–948, 1993.

[2] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider, "New indices for text: Pat trees and pat arrays," in *Information Retrieval.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1992, pp. 66–82.

[3] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital SRC Research Report, Tech. Rep., 1994.

[4] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.

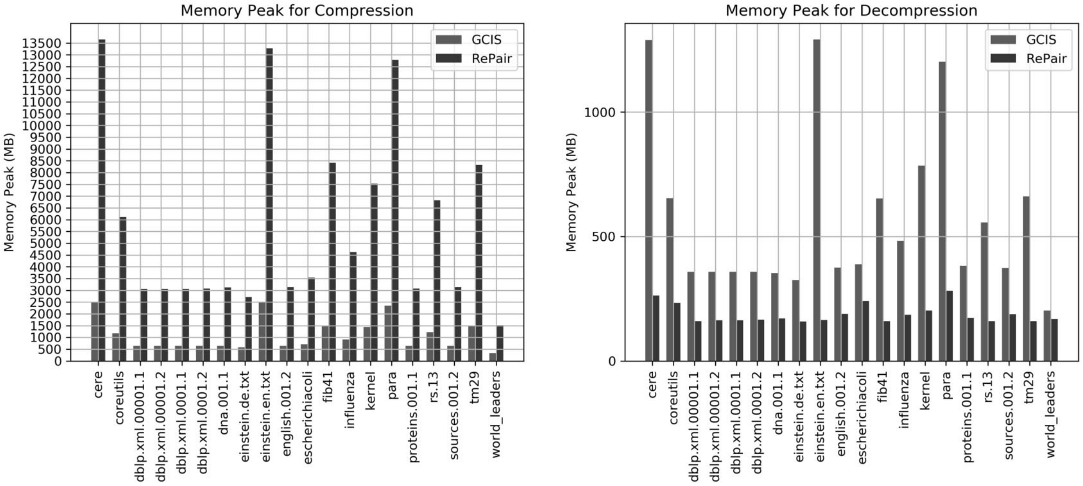
[5] E. Ohlebusch and S. Gog, "Lempel-ziv factorization revisited," in *Proc. CPM*, 2011, pp. 15–26.

[6] J. Kärkkäinen, D. Kempa, and S. J. Puglisi, "Linear time lempel-ziv factorization: Simple, fast, small," in *Proc. CPM*, 2013, pp. 189–200.

[7] K. Goto and H. Bannai, "Space efficient linear time Lempel-Ziv factorization for small alphabets," in *Proc. DCC*, 2014, pp. 163–172.

[8] G. Navarro, *Compact Data Structures – A practical approach.* Cambridge University Press, 2016.

[9] G. Nong, S. Zhang, and W. H. Chan, "Linear suffix array construction by almost pure induced-sorting," in *Proc. DCC*, 2009, pp. 193–202.



(a) Memory Peak during compression. (b) Memory Peak during decompression.

Figure 1: Peak memory (in MB) of GCIS and Re-Pair.

[10] D. Okanohara and K. Sadakane, "A linear-time Burrows-Wheeler transform using induced sorting," in *Proc. SPIRE*, 2009, pp. 90–101.

[11] J. Kärkkäinen, G. Manzini, and S. J. Puglisi, "Permuted longest-common-prefix array," in *Proc. CPM*, 2009, pp. 181–192.

[12] J. Fischer, "Inducing the LCP-Array," in *Proc. WADS*, 2011, pp. 374–385.

[13] F. A. Louza, S. Gog, and G. P. Telles, "Inducing enhanced suffix arrays for string collections," *Theor. Comput. Sci.*, vol. 678, pp. 22–39, 2017.

[14] N. J. Larsson and A. Moffat, "Offline dictionary-based compression," in *Proc. DCC*, 1999, pp. 296–305.

[15] I. Pavlov, "The 7zip home page," http://www.7-zip.org/, accessed: 10/2017.

[16] H. Itoh and H. Tanaka, "An efficient method for in memory construction of suffix arrays," in *Proc. SPIRE*, 1999, pp. 81–88.

[17] P. Ko and S. Aluru, "Space efficient linear time construction of suffix arrays," in *Proc. CPM*, 2003, pp. 200–210.

[18] G. Nong, S. Zhang, and W. H. Chan, "Two efficient algorithms for linear time suffix array construction," *IEEE Trans. Comput.*, vol. 60, no. 10, pp. 1471–1484, 2011.

[19] J. Arpe and R. Reischuk, "On the complexity of optimal grammar-based compression," in *Proc. DCC*, 2006, pp. 173–182.

[20] I. H. Witten, A. Moffat, and T. C. Bell, *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition*. Morgan Kaufmann, 1999.

[21] V. N. Anh and A. Moffat, "Index compression using 64-bit words," *Softw., Pract. Exper.*, vol. 40, no. 2, pp. 131–147, 2010.

[22] R. Wan, "Browsing and searching compressed documents," Ph.D. dissertation, University of Melbourne, Australia, Dec. 2003.

[23] S. Gog, T. Beller, A. Moffat, and M. Petri, "From theory to practice: Plug and play with succinct data structures," in *Proc. SEA*, 2014, pp. 326–337.

[24] F. Claude and G. Navarro, "Improved grammar-based compressed indexes," in *Proc. SPIRE*, 2012, pp. 180–192.

[25] S. Kreft and G. Navarro, "LZ77-like compression with fast random access," in *Proc. DCC*, 2010, pp. 239–248.