i-ETC: ISEL Academic Journal of Electronics, Telecommunications and Computers
CETC2011 Issue, Vol. 2, n. 1 (2013) ID-4

LEMPEL-ZIV SLIDING WINDOW UPDATE WITH SUFFIX ARRAYS

Artur Ferreira[1,3,4] Arlindo Oliveira[2,4] Mário Figueiredo[3,4]
[1]*Instituto Superior de Engenharia de Lisboa (ISEL)*
[2]*Instituto de Engenharia de Sistemas e Computadores – Investigação e Desenvolvimento (INESC-ID)*
[3]*Instituto de Telecomunicações (IT)*
[4]*Instituto Superior Técnico (IST), Lisboa, PORTUGAL*
arturj@isel.pt *aml@inesc-id.pt* *mtf@lx.it.pt*

Keywords: Lempel-Ziv compression, suffix arrays, sliding window update, substring search.

Abstract: The sliding window dictionary-based algorithms of the Lempel-Ziv (LZ) 77 family are widely used for universal lossless data compression. The encoding component of these algorithms performs repeated substring search. Data structures, such as hash tables, binary search trees, and suffix trees have been used to speedup these searches, at the expense of memory usage. Previous work has shown how *suffix arrays* (SA) can be used for dictionary representation and LZ77 decomposition. In this paper, we improve over that work by proposing a new efficient algorithm to update the sliding window each time a token is produced at the output. The proposed algorithm toggles between two SA on consecutive tokens. The resulting SA-based encoder requires less memory than the conventional tree-based encoders. In comparing our SA-based technique against tree-based encoders, on a large set of benchmark files, we find that, in some compression settings, our encoder is also faster than tree-based encoders.

1 INTRODUCTION

The Lempel-Ziv 77 (LZ77) [14, 19], and its variant Lempel-Ziv-Storer-Szymanski (LZSS) [14, 16], are lossless compression algorithms that are the basis of a wide variety of universal data compression applications, such as GZip, WinZip, PkZip, WinRar, and 7-Zip. Those algorithms are asymmetric in terms of time and memory requirements, with encoding being much more demanding than decoding. The main reason for this difference is that the encoder part requires substring search over a dictionary, whereas decoding involves no search.

Most LZ-based encoders use efficient data structures, such as *binary trees* (BT) [6, 11], *suffix trees* (ST) [5, 7, 9, 13, 17], and hash tables, thus allowing fast search at the expense of higher memory requirement. The use of a Bayer-tree, along with special binary searches on a sorted sliding window, has been proposed to speedup the encoding procedure [6]. *Suffix arrays* (SA) [7, 10, 15], due to their simplicity, space efficiency, and linear time construction algorithms [8, 12, 18], have been a focus of research; *e.g.*, SA have been used in encoding data with anti-dictionaries [4] and to find repeating sub-sequences for data deduplication [1], among other applications.

Recently, algorithms for computing the LZ77 factorization of a string, based on SA and auxiliary arrays, have been proposed to replace trees [2, 3]. These SA-based encoders have the two following memory-related advantages over tree-based encoders: they require less memory; the amount of allocated memory is constant and *a priori* known, being independent of the dictionary contents. In contrast, encoders based on hash tables or trees encoders, usually require allocating a maximum amount of memory. The main disadvantage of SA-based encoders is their encoding time, which is typically above that of tree-based encoders, attaining roughly the same compression ratio.

Regarding previous work on SA for LZ decomposition, it has been found that the main drawback of the method in [2] is the absence of a strategy to update the SA as encoding proceeds: the entire SA is repeatedly rebuilt. The proposals in [3] for LZ decomposition with SA are memory efficient, but the encoding time is above that of tree-based encoders.

1.1 Our Contribution

In this paper, we improve on previous approaches [2, 3] by proposing an algorithm for sliding window update using SA and a fast technique for finding the tokens. The application of these techniques to LZ77/LZSS encoding does not involve any changes on the decoder side. Our SA-based encoder uses a small amount of memory and can be faster than the tree-based ones, like 7-Zip, being close to GZip in encoding time on several standard benchmark files, for some compression settings.

The rest of the paper is organized as follows. Section 2 reviews basic concepts of LZ77/LZSS encoding and decoding as well as the use of SA for this purpose. Section 3 describes the proposed algorithms. The experimental results are presented and discussed in Section 4, while Section 5 contains some concluding remarks.

2 LEMPEL-ZIV BASICS

The LZ77 and LZSS [14, 16, 19] lossless compression techniques use a sliding window over the sequence of symbols to be encoded with two subwindows:

- the *dictionary* which holds the symbols already encoded;
- the *look-ahead-buffer* (LAB), containing the next symbols to be encoded.

As the string in the LAB is encoded, the window slides to include it in the dictionary (this string is said to *slide in*); consequently, the symbols at the far end of the dictionary are dropped (they *slide out*).

At each step of the LZ77/LZSS encoding algorithm, the longest prefix of the LAB which can be found anywhere in the dictionary is determined and its position stored. For these two algorithms, encoding of a string consists in describing it by a token. The LZ77 token is a triplet of fields, (*pos, len, sym*), with the following meanings:

- *pos* - location of the longest prefix of the LAB found in the current dictionary;
- *len* - length of the matched string;
- *sym* - the first symbol in the LAB that does not belong to the matched string (*i.e.*, that breaks the match).

In the absence of a match, the LZ77 token is (*0,0,sym*). Each LZ77 token uses $\log_2(|\text{dictionary}|) + \log_2(|\text{LAB}|) + 8$ bits, where $|.|$ denotes length (number of bytes); usually, $|\text{dictionary}| \gg |\text{LAB}|$. In LZSS, the token has the format (*bit,code*), with the structure of *code* depending on value *bit* as follows:

$$\begin{cases} bit = 0 \;\Rightarrow\; code = (sym), \\ bit = 1 \;\Rightarrow\; code = (pos, len). \end{cases} \quad (1)$$

In the absence of a match, LZSS produces (0, *sym*), otherwise (1, *pos, len*). The idea is that, if a match exists, there is no need to explicitly encode the next symbol. Besides this modification, Storer and Szymanski [16] also proposed keeping the LAB in a circular queue and the dictionary in a binary search tree, to optimize the search. LZSS is widely used in practice since it typically achieves higher compression ratios than LZ77 [14]. The fundamental and most expensive component of LZ77/LZSS encoding is the search for the longest match between LAB prefixes and the dictionary.

In LZSS, the token uses either 9 bits, when it has the form (0,*sym*), or $1 + \log_2(|\text{dictionary}|) + \log_2(|\text{LAB}|)$ bits, when it has the form (1,(*pos,len*)). Figure 1 shows an example of LZ77 encoding for a dictionary of length 16 and LAB with 8 symbols.

2.1 Decoding Procedures

Assuming that the decoder and encoder are initialized with equal dictionaries, the decoding of each LZ77 token (*pos,len,sym*) proceeds as follows:

1) *len* symbols are copied from the dictionary to the output, starting at position *pos* of the dictionary;
2) the symbol *sym* is appended to the output;
3) the string just produced at the output is slid into the dictionary.

For LZSS decoding, we have:

1) if the bit field is 1, *len* symbols, starting at position *pos* of the dictionary, are copied to the output; otherwise *sym* is copied to the output;
2) the string just produced at the output is slid into the dictionary.

Both LZ77 and LZSS decoding are low complexity procedures which do not involve any search, thus decoding is much faster than encoding.

2.2 Using a Suffix Array

A *suffix array* (SA) represents the lexicographically sorted array of the suffixes of a string [7, 10]. For a string *D* of length *m* (with *m* suffixes), the suffix array *P* is the set of integers from 1 to *m*, sorted by the lexicographic order of the suffixes of *D*. For instance, if we consider dictionary *D*=`business-machine` (with *m*=16), we get

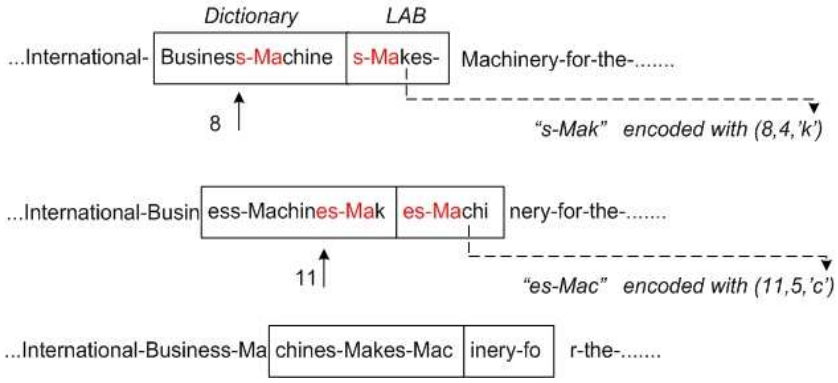

Figure 1: Illustration of LZ77 encoding with dictionary "business-machine". We show the corresponding outputted LZ77 tokens for the encoding of the prefix of the LAB.

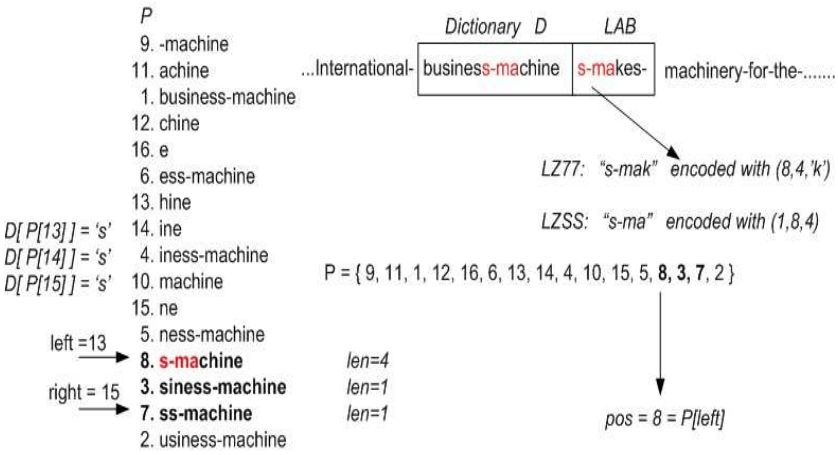

Figure 2: LZ77 and LZSS with dictionary "business-machine" and its representation with SA P. The encoding of the prefix "s-mak" of the LAB can be done with substrings ranging from $D[P[left]]$ to $D[P[right]]$. We choose the 4-symbol match at $P[13]$ producing the depicted LZ77/LZSS tokens.

$P = \{9,11,1,12,16,6,13,14,4,10,15,5,8,3,7,2\}$, as shown in Figure 2.

Each integer in P is the suffix number corresponding to its position in D. Finding a substring of D, as required by LZ77/LZSS, can be done by searching array P; for instance, the set of substrings of D that start with 's', can be found at indexes 3, 7, and 8 of D, ranging from index 13 to 15 on P. In this work, we use the *suffix array induced sorting* (SA-IS) algorithm to build the SA [12].

3 SLIDING WINDOW UPDATE ALGORITHM

In this section we present the proposed algorithm for sliding window update as well as an accelerated technique to obtain the tokens over a dictionary. This work addresses only the encoder side data structures and algorithms, with no effect in the decoder. Decoding does not need any special data structure and follows standard LZ77/LZSS decoding, as described in subsection 2.1.

3.1 Accelerated Encoder

The LZ77/LZSS tokens can be found faster if we use an auxiliary array of 256 integers (named LI – *Left-Index*). This array holds, for each ASCII symbol, the first index of the suffix array where we can find the first suffix that starts with that symbol (the *left* index for each symbol, as shown in Figure 3). For symbols that are not the start of any suffix, the corresponding entry is labeled with -1, meaning that we have an empty match for those symbols. Figure 3 shows the LI array for the dictionary of Figure 2. The *left* value, as depicted in Figure 2, is computed by

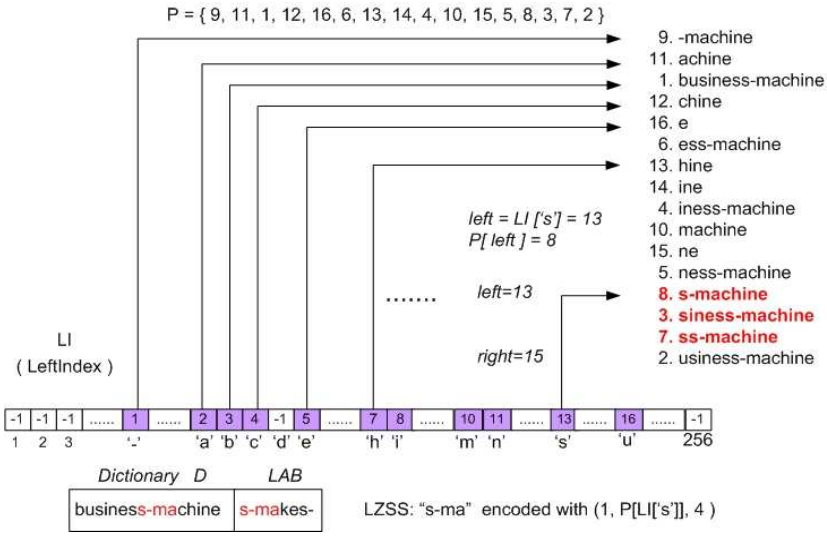

Figure 3: The LI (*LeftIndex*) auxiliary array: for each symbol that starts a suffix it holds the index of the SA *P* in which that suffix starts. For the symbols that are not the start of any suffix, the corresponding entry is marked with -1, meaning that we have an empty match for substrings that start with that symbol.

$left \leftarrow P[LI[LAB[1]]]$. The *right* indicator is found by iterating *P* starting on index *left* and performing a single symbol comparison. If we want LZ77/LZSS "fast" compression, we choose *pos* = *left*; for "best" compression, we choose $left \leq pos \leq right$, such that we have the longest match between substrings starting at $D[pos]$ and $LAB[1]$.

3.2 Proposed Algorithm

The key idea of our sliding window technique is to use two SA of length |dictionary|, named P_A and P_B, and a pointer *P* (to P_A or P_B) to represent the dictionary. At each substring match, that is, each time we produce a token, we toggle pointer *P* between the two SA and we also update the LI array. If the previous token was encoded with P_A, the steps next described are carried out using P_B, and vice-versa. This idea somewhat resembles the double buffering technique, since we are switching from one SA to the other, every time a token is produced. If we used a single SA, we would have a slow encoder, because we would have to perform several displacements of the integers on the unique large SA. These integer displacements would leads to us to a situation in which the encoder would be slow. For both LZ77/LZSS encoding, each time we output a token encoding *L* symbols, the dictionary is updated as follows:

R. Remove suffixes $\{1,\ldots,L\}$ (they *slide out*);

I. Insert in a lexicographic order the suffixes ranging from $|\text{dictionary}|-L+1$ to |dictionary| (they *slide in*);

U. Update suffixes $\{L+1,\ldots,|\text{dictionary}|\}$; these are subtracted by *L*.

Figure 4 shows these R, I, and U actions, for the dictionary in Figs. 2 and 3, after encoding `s-mak` with *L*=5 symbols. The removal action (*slid out*) is implicit by toggling from SA P_A to P_B; the updated suffixes keep the order between them; the inserted (*slid in*) suffixes are placed in the destination SA, in lexicographic order. Algorithm 1 details the set of actions taken by our proposed algorithm. Algorithm 1 runs each time we produce a LZ77/LZSS token; in the case of LZ77, we set $L = len+1$; for LZSS $L = len$. We can also update the SA with the entire LAB contents using $L = |LAB|$, after we produce the set of tokens encoding the entire LAB. This algorithm also works in the "no match" case of LZSS, in which the token is (0, *sym*), with *L*=1. Notice that we use two auxiliary arrays with length up to $|LAB|$; we thus have a memory-efficient sliding window algorithm. The amount of memory for the encoder data structures is

$$M_{SA} = 2|\text{P}| + |\text{LI}| + |P_S| + |I| \qquad (2)$$

bytes. Figure 5 illustrates Algorithm 1 using the dictionary shown in Figs. 3 and 4, after encoding `s-mak` with $L = 5$. We see the SA P_A as origin and P_B as destination; we also show the contents of P_S and *I* with 5 positions each.

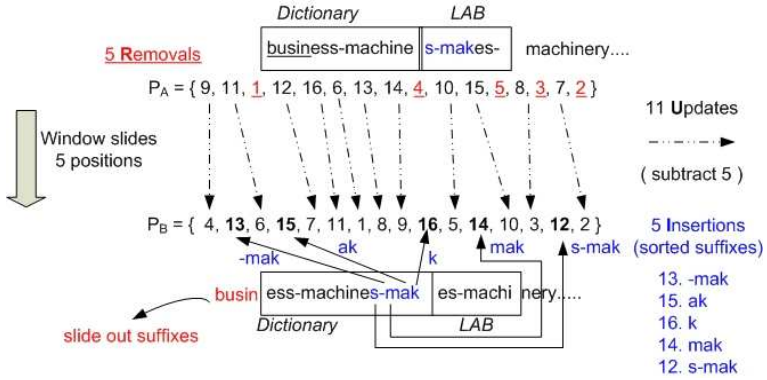

Figure 4: Illustration of our sliding window update algorithm with the R I U actions, after encoding "s-mak" with L=5: R) suffixes 1 to 5 slide out of P_A; I) suffixes 12 to 16 are inserted in lexicographic order into P_B; U) suffixes 6 to 16 are updated being subtracted by 5.

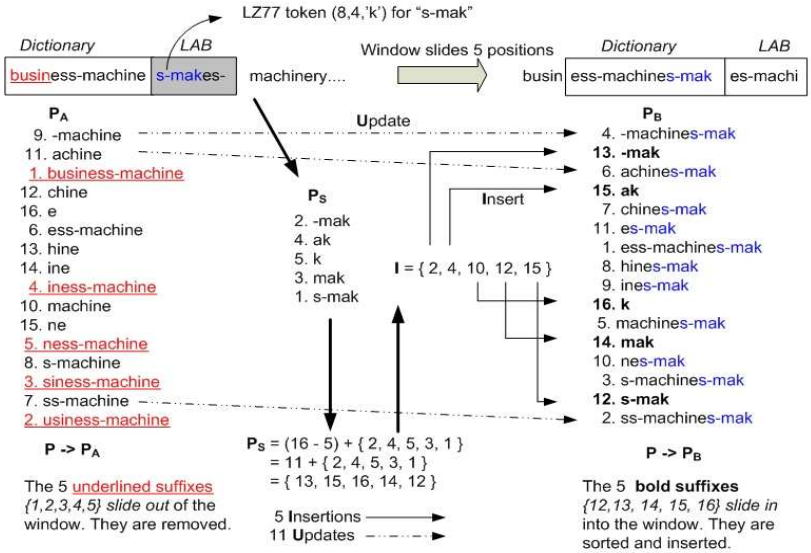

Figure 5: Sliding window update with pointer P set to P_A initially; the first update is done using P_B as destination. Array I holds the indexes where to insert the new suffixes into P_B. The update of the indexes is done over P_B and there is no modification on P_A or integer displacement on P_B.

4 EXPERIMENTAL RESULTS

Our experimental tests were carried out on a laptop with a 2 GHz Intel Core2Duo T7300 CPU and 2 GB of RAM, using a single core. The code was written in *C*, using Microsoft Visual Studio 2008. The linear time SA construction algorithm SA-IS [12] (available at `yuta.256.googlepages.com/sais`) was used. For comparison purposes, we also present the results of Nelson's *binary tree* (BT) encoder [11], GZip[1], and the *LZ Markov chain algorithm* (LZMA[2]). These three encoders were chosen as benchmark, since they represent the typical usage of tree and hash tables data structures for LZ77 compression. The BT-encoder represents the dictionary with a binary tree data structure. The well-known GZip encoder uses trees and hash tables. LZMA is the default compression technique employed by the 7z format in the 7-ZIP program. Both the GZip and LZMA encoders perform entropy encoding of the tokens produced by the LZMA algorithm. This allows for these algorithms to attain a higher compression ratio than our algorithms and the BT-encoder (it does not perform entropy encoding of the tokens).

The test files are from the standard corpora Calgary (18 files, 3 MB) and Silesia (12 files, 211 MB),

[1] `www.gzip.org/`
[2] `www.7-zip.org`

available at `www.data-compression.info`. We use the "best" compression option, by choosing the longest match as discussed in section 3.1 and depicted in Figure 2.

4.1 Experimental Setup

In our tests, we assess the following measures: encoding time (in seconds, measured by the C function `clock`); compression ratio (in bits per byte, bpb); amount of memory for encoder data structures (in bytes).

Nelson's BT-encoder [11] uses 3 integers per tree node with $|\text{dictionary}|+1$ nodes, occupying

$$M_{BT} = 13 \times |\text{dictionary}| + 12 \qquad (3)$$

bytes, using 4-byte integers. Larsson's suffix tree encoder (available at `www.larsson.dogma.net/research.html`) uses 3 integers and a symbol for each node, occupying 16 bytes, placed in a hash table [9], using the maximum amount of memory

$$M_{ST} = 25 \times |\text{dictionary}| + 4 \times \text{hashsz} + 16 \qquad (4)$$

bytes, where *hashsz* is the hash table size. The GZip encoder occupies M_{GZIP}=313408 bytes, as measured by `sizeof` C operator. The LZMA encoder data structures occupy

$$M_{LZMA} = 4194304 + \begin{cases} 9.5|\text{dict.}|, & \text{if MF = BT2} \\ 11.5|\text{dict.}|, & \text{if MF = BT3} \\ 11.5|\text{dict.}|, & \text{if MF = BT4} \\ 7.5|\text{dict.}|, & \text{if MF = HC4} \end{cases}, \qquad (5)$$

bytes, depending on the *match finder* (MF) used as well as on |dictionary| with BT# denoting binary tree with # bytes hashing and HC4 denoting hash chain with 4 bytes hashing. For instance, with $(|\text{dictionary}|, |\text{LAB}|) = (65536, 4096)$, we have in increasing order M_{SA} =627712, M_{BT} =851980, M_{ST} =1900560, and M_{LZMA}=4816896 bytes. If we consider an application in which we only have a low fixed amount of memory, such as the internal memory of an embedded device, it may not be possible to instantiate a tree or a hash table based encoder.

The GZip and LZMA[3] encoders are built upon the deflate algorithm, and perform entropy encoding of the tokens achieving better compression ratio than our LZSS encoding algorithms. These encoders are useful as a benchmark comparison, regarding encoding time and amount of memory. For both compression techniques, we have compiled their C/C++ sources using the same compiler settings as for our encoders.

The compression ratio of our encoders as well as that of the BT-encoder can be easily improved by entropy-encoding the tokens. Our purpose is to focus only on the construction and update of the dictionary and searching over it, using less memory than the conventional solutions with trees and hash tables.

Algorithm 1 SA Sliding Window Algorithm

Input: P_A, P_B, m-length SA;
P, pointer to P_A or P_B;
P_{dst}, pointer to P_B or P_A;
LAB, look-ahead buffer;
LI, 256-position length LeftIndex array;
$L \leq |\text{LAB}|$, number of symbols in the previously produced token(s).
Output: P_A or P_B updated;
P pointing to the recently updated SA.

```
1: if P points to P_A then
2:     Set P_dst to P_B.      {/*R action. Implicit removal.*/}
3: else
4:     Set P_dst to P_A.
5: end if
6: Compute the SA P_S for the encoded substring (with L positions).
7: Using LI and P_S, fill the L-length array I with the insertion indexes (slide in suffixes).
8: for i = 1 to L do
9:     P_dst[ I[i] ] = P_S[i] + |dictionary| - L.  {/*I action.*/}
10: end for
11: Do nUpdate = |dictionary| - L;
12: Do j=1.          {/*Perform |dictionary| - L updates.*/}
13: for i = 1 to |dictionary| do
14:     if (P[i] - L) > 0 then
15:         while (j ∈ I) do
16:             j = j + 1.      {/*Make sure that j is an update position.*/}
17:         end while
18:         P_dst[j] = P[i] - L.                {/*U action.*/}
19:         j = j + 1.
20:         nUpdate = nUpdate - 1.
21:         if (nUpdate==0) then
22:             break.      {/*Destination SA is complete.*/}
23:         end if
24:     end if
25: end for
26: Set P to P_dst.    {/*P points to recently updated SA.*/}
```

4.2 Comparison with other encoders

We encode each file of the two corpora using LZSS and compute the total encoding time as well as the average compression ratio, for different configurations of $(|\text{dictionary}|, |\text{LAB}|)$, with "best" compression option and $L = |LAB|$ for Algorithm 1. Table 1 shows the results of these tests on the Calgary Corpus. Our SA-encoder is faster than BT, except on tests 5 to 7; on test 6 (the GZip-like scenario), BT-encoder is about

[3] LZMA SDK, version 4.65, 3 Feb. 2009, available at `www.7-zip.org/sdk.html`

Table 1: Amount of memory, total encoding time (in seconds), and average compression ratio (in bpb), for several lengths of (|dictionary|, |LAB|) on the Calgary Corpus, using "best" compression. GZip "fast" leads to Time=0.5 and bpb=3.20 while GZip "best" yields Time=1.2 and bpb=2.79. The best encoding time is underlined.

Calgary Corpus			**SA (proposed)**			**BT**			**LZMA**		
#	**|Dict.|**	**|LAB|**	$\mathbf{M_{SA}}$	**Time**	**bpb**	$\mathbf{M_{BT}}$	**Time**	**bpb**	$\mathbf{M_{LZMA}}$	**Time**	**bpb**
1	2048	1024	25600	**2.2**	5 77	26636	3 92	5 65	4217856	4 7	2 99
2	4096	1024	41984	**2.5**	5 40	53260	4 3	4 98	4241408	4 8	2 82
3	4096	2048	50176	**2.4**	5 75	53260	11 1	5 48	4241408	4 8	2 82
4	8192	2048	82944	**3.8**	5 49	106508	11 7	4 88	4288512	5 1	2 69
5	16384	256	134144	9 1	4 36	213004	**4 5**	4 12	4382720	5 2	2 61
6	32768	256	265216	18 4	4 31	425996	**5 5**	4 08	4571136	4 9	2 54
7	32768	1024	271360	11 1	4 86	425996	**7 5**	4 40	4571136	4 9	2 54
8	32768	2048	279552	**9.5**	5 16	425996	15 8	4 57	4571136	4 9	2 54

Table 2: Amount of memory, total encoding time (in seconds) and average compression ratio (in bpb), for several lengths of (|dictionary|, |LAB|) on the Silesia Corpus, using "best" compression. GZip "fast" obtains Time=19.5 and bpb=3.32 while GZip "best" does Time=74.4 and bpb=2.98. The best encoding time is underlined.

Silesia Corpus			**SA (proposed)**			**BT**			**LZMA**		
#	**|Dict.|**	**|LAB|**	$\mathbf{M_{SA}}$	**Time**	**bpb**	$\mathbf{M_{BT}}$	**Time**	**bpb**	$\mathbf{M_{LZMA}}$	**Time**	**bpb**
1	2048	1024	25600	**118.7**	5 66	26636	249 5	5 65	4217856	333 53	3 05
2	4096	1024	41984	**116.9**	5 41	53260	303 4	5 25	4241408	349 05	2 90
3	4096	2048	50176	**112.9**	5 68	53260	694 9	5 63	4241408	349 05	2 90
4	8192	2048	82944	**143.4**	5 44	106508	668 9	5 27	4288512	356 77	2 76
5	16384	256	134144	319 1	4 55	213004	**254.6**	4 44	4382720	366 47	2 62
6	32768	256	265216	542 7	4 41	425996	**318.1**	4 31	4571136	356 34	2 52
7	32768	1024	271360	**322.2**	4 80	425996	382 6	4 64	4571136	356 34	2 52
8	32768	2048	279552	**302.3**	5 02	425996	979 8	4 81	4571136	356 34	2 52

3.5 times faster than SA. Table 2 shows the results for the Silesia Corpus. In these tests, the SA-encoder is the fastest except on tests 5 and 6. On test 3, the SA-encoder is about 5 times faster than the BT-encoder, achieving about the same compression ratio. Notice that that for the BT and LZMA encoders, the amount of memory only depends on the length of the dictionary. For our SA-encoder the amount of memory for the encoder data structures also depends on the length of the LAB, due to the use of the P_S and I arrays, as given by (2).

Figure 6 shows the performance measure $Time \times Memory$ on the encoding of the Calgary and Silesia corpora, on the tests shown on Tables 1 and 2, for SA and BT-encoders, including GZip "best" test results for comparison. Regarding the Calgary Corpus test results, the SA-encoder has better performance than BT-encoder on tests 1 to 4 and 8; on Silesia Corpus, this happens on all tests except on test 6.

For all these encoders searching and updating the dictionary are the most time-consuming tasks. A high compression ratio like those of LZMA and GZip can be attained only when we use entropy encoding with appropriate models for the tokens. The SA encoder is faster than the BT encoder, when the LAB is not too small. Our algorithms (without entropy encoding) are thus positioned in a trade-off between time and memory, that can make them suitable to replace binary trees on LZMA or in substring search. The usage of the LI array over the SA allows to quickly find the set of substrings that start with a given symbol acting as an accelerator of the encoding process.

5 CONCLUSIONS

In this paper, we have proposed a sliding window update algorithm for Lempel-Ziv compression based on suffix arrays, improving on earlier work in terms of encoding time, with similar low memory requirements. The proposed algorithm uses an auxiliary array as an accelerator to the encoding procedure, as well as a fast update of the dictionary based on two suffix arrays. It allows *a priori* computing the exact amount of memory necessary for the encoder data structures without any waste of memory; usually this may not be the case when using (binary/suffix) trees.

We have compared our algorithm on standard corpora against tree-based encoders, including GZip and LZMA. The experimental tests showed that our en-

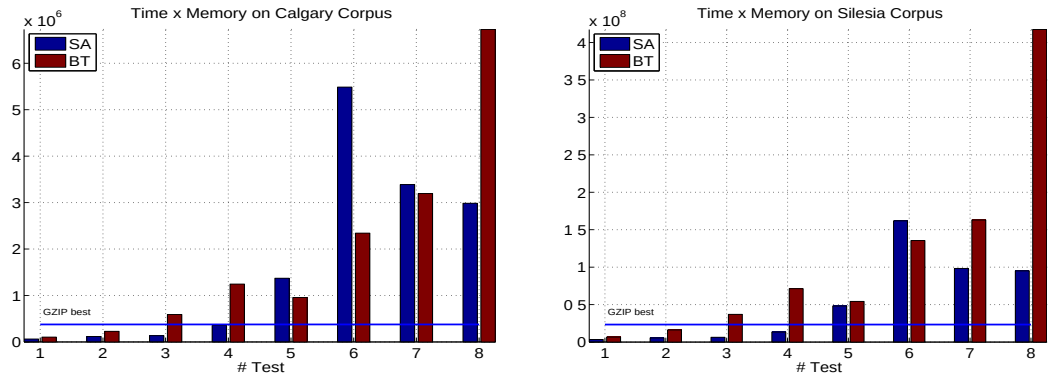

Figure 6: Time $\times$ Memory performance measure for SA and BT on the Calgary and Silesia Corpus, on the 8 encoding tests of Tables 1 and 2. We include GZip "best" performance for comparison.

coders *always* occupy less memory than tree-based encoders; moreover, in some (typical) compression settings the SA-encoders are also faster than tree-based encoders. The position of the proposed algorithm in the time-memory tradeoff makes it suitable as a replacement of trees and hash tables, for some compression settings. These compression settings include all the situations in which the length of the look-ahead-buffer window is not too small, as compared to the length of the dictionary.

REFERENCES

[1] C. Constantinescu, J. Pieper, and T. Li. Block size optimization in deduplication systems. In *DCC '09: Proc. of the IEEE Conference on Data Compression*, page 442, Washington, DC, USA, 2009.

[2] M. Crochemore, L. Ilie, and W. Smyth. A simple algorithm for computing the Lempel-Ziv factorization. In *DCC '08: Proc. of the IEEE Conference on Data Compression*, pages 482–488, Washington, DC, USA, March 2008. IEEE Computer Society.

[3] A. Ferreira, A. Oliveira, and M. Figueiredo. On the use of suffix arrays for memory-efficient Lempel-Ziv data compression. In *DCC '09: Proc. of the IEEE Conference on Data Compression*, page 444, Washington, DC, USA, March 2009. IEEE Computer Society.

[4] M. Fiala and J. Holub. DCA using suffix arrays. In *DCC '08: Proc. of the IEEE Conference on Data Compression*, page 516, Washington, DC, USA, March 2008. IEEE Computer Society.

[5] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. New indices for text: PAT trees and PAT arrays. *Information retrieval: data structures and algorithms*, pages 66–82, 1992.

[6] Ulrich Gräf. Sorted sliding window compression. In *DCC '99: Proc. of the IEEE Conference on Data Compression*, page 527, Washington, DC, USA, 1999.

[7] D. Gusfield. *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[8] J. Karkainen, P. Sanders, and S. Burkhardt. Linear work suffix array construction. *Journal of the ACM*, 53(6):918–936, 2006.

[9] N. Larsson. *Structures of String Matching and Data Compression*. PhD thesis, Department of Computer Science, Lund University, Sweden, September 1999.

[10] U. Manber and G. Myers. Suffix Arrays: a new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, October 1993.

[11] M. Nelson and J. Gailly. *The Data Compression Book.* M & T Books, New York, 2nd edition, 1995.

[12] G. Nong, S. Zhang, and W. Chan. Linear suffix array construction by almost pure induced-sorting. In *DCC '09: Proc. of the IEEE Conference on Data Compression*, pages 193–202, March 2009.

[13] Michael Rodeh, Vaughan Pratt, and Shimon Even. Linear algorithm for data compression via string matching. *J. ACM*, 28(1):16–24, 1981.

[14] D. Salomon. *Data Compression - The complete reference*. Springer-Verlag London Ltd, London, fourth edition, Januray 2007.

[15] M. Salson, T. Lecroq, M. Léonard, and L. Mouchard. Dynamic extended suffix arrays. *Journal of Discrete Algorithms*, In Press, Corrected Proof, 2009.

[16] J. Storer and T. Szymanski. Data compression via textual substitution. *Journal of ACM*, 29(4):928–951, October 1982.

[17] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[18] S. Zhang and G. Nong. Fast and space efficient linear suffix array construction. In *DCC '08: Proc. of the IEEE Conference on Data Compression*, page 553, Washington, DC, USA, March 2008. IEEE Computer Society.

[19] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, IT-23(3):337–343, May 1977.