Real-Time and Distributed Applications for Dictionary-Based Data Compression

Sergio De Agostino
Computer Science Department
Sapienza University
Rome, Italy
Email: deagostino@di.uniroma1.it

***Abstract*—The greedy approach to dictionary-based static text compression can be executed by a finite state machine. When it is applied in parallel to different blocks of data independently, there is no lack of robustness even on standard large scale distributed systems with input files of arbitrary size. Beyond standard large scale, a negative effect on the compression effectiveness is caused by the very small size of the data blocks. A robust approach for extreme distributed systems is presented in this paper, where this problem is fixed by overlapping adjacent blocks and preprocessing the neighborhoods of the boundaries. Moreover, we introduce the notion of pseudo-prefix dictionary, which allows optimal compression by means of a real-time semi-greedy procedure and a slight improvement on the compression ratio obtained by the distributed implementations.**

Keywords-data compression; decoding; real time application; distributed system; scalability; robustness

I. INTRODUCTION

Real time algorithms are very important in the field of data compression, expecially during the decoding phase, and further speed-up can be obtained by means of distributed implementations. We studied these topics in the context of lossless compression applied to one-dimensional data and preliminary results were presented in [1] and [2].

Static data compression implies the knowledge of the input type. With text, dictionary-based techniques are particularly efficient and employ string factorization. The dictionary comprises typical factors plus the alphabet characters in order to guarantee feasible factorizations for every string. Factors in the input string are substituted by pointers to dictionary copies and such pointers could be either variable or fixed length codewords. The optimal factorization is the one providing the best compression, that is, the one minimizing the sum of the codeword lengths. Efficient sequential algorithms for computing optimal solutions were provided by means of dynamic programming techniques [3] or by reducing the problem to the one of finding a shortest path in a directed acyclic graph [4]. From the point of view of sequential computing, such algorithms have the limitation of using an off-line approach. However, decompression is still on-line and a very fast and simple real time decoder outputs the original string with no loss of information. Therefore, optimal solutions are practically acceptable for read-only memory files where compression is executed only once. Differently, simpler versions of dictionary-based static techniques were proposed, which achieve nearly optimal compression in practice (that is, less than ten percent loss). An important simplification is to use a fixed length code for the pointers, so that the optimal decodable compression for this coding scheme is obtained by minimizing the number of factors. Such variable to fixed length approach is robust since the dictionary factors are typical patterns of the input specifically considered. The problem of minimizing the number of factors gains a relevant computational advantage by assuming that the dictionary is *prefix-closed* (*suffix-closed*), that is, all the prefixes (suffixes) of a dictionary element are dictionary elements [5], [6], [7]. The left to right greedy approach is optimal only with suffix-closed dictionaries. An optimal factorization with prefix-closed dictionaries can be computed on-line by using a semi-greedy procedure [6], [7]. On the other hand, prefix-closed dictionaries are easier to build by standard adaptive heuristics [8], [9]. These heuristics are based on an "incremental" string factorization procedure [10], [11]. The most popular for prefix-closed dictionaries is the one presented in [12]. However, the prefix and suffix properties force the dictionary to include many useless elements, which increase the pointer size and slightly reduce the compression effectiveness. A more natural dictionary with no prefix and no suffix property is the one built by the heuristic in [13] or by means of separator characters as, for example, space, new line and punctuation characters with natural language.

Theoretical work was done, mostly in the nineties, to design efficient parallel algorithms on a random access parallel machine (PRAM) for dictionary-based static text compression [14], [15], [16], [17], [18], [19], [20], [21], [22]. Although the PRAM model is out of fashion today, shared memory parallel machines offer a good computational model for a first approach to parallelization. When we address the practical goal of designing distributed algorithms we have to consider two types of complexity, the interprocessor communication and the input-output mechanism. While the input/output issue is inherent to any parallel algorithm and has standard solutions, the communication cost of the computational phase after the distribution of

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

the data among the processors and before the output of the final result is obviously algorithm-dependent. So, we need to limit the interprocessor communication and involve more local computation to design a practical algorithm. The simplest model for this phase is, of course, a simple array of processors with no interconnections and, therefore, no communication cost. Parallel decompression is, obviously, possible on this model [17]. With parallel compression, the main issue is the one concerning scalability and robustness. Traditionally, the scale of a system is considered large when the number of nodes has the order of magnitude of a thousand. Modern distributed systems may nowadays consist of hundreds of thousands of nodes, pushing scalability well beyond traditional scenarios (extreme distributed systems).

In [23], an approximation scheme of optimal compression with static prefix-closed dictionaries was presented for massively parallel architectures, using no interprocessor communication during the computational phase since it is applied in parallel to different blocks of data independently. The scheme is algorithmically related to the semi-greedy approach previously mentioned and implementable on extreme distributed systems because adjacent blocks overlap and the neighborhoods of the boundaries are preprocessed. However, with standard large scale the overlapping of the blocks and the preprocessing of the boundaries are not necessary to achieve nearly optimal compression in practice. Furthermore, the greedy approach to dictionary-based static text compression is nearly optimal on realistic data for any kind of dictionary even if the theoretical worst-case analysis shows that the multiplicative approximation factor with respect to optimal compression achieves the maximum length of a dictionary element [9]. If the dictionary is well-constructed by relaxing the prefix property, the loss of greedy compression can go down to one percent with respent to the optimal one. In this paper, we relax the prefix property of the dictionary and present two implementations of the greedy approach to static text compression with an arbitrary dictionary on a large scale and an extreme distributed system, respectively. Moreover, we present a finite-state machine implementation of greedy static dictionary-based compression with an arbitrary dictionary that can be relevant to achieve high speed with standard scale distributed systems. We wish to point out that scalability cannot be guaranteed with adaptive dictionary approaches to data compression, as the sliding window method [24] or the dynamic one [11]. Indeed, the size of the data blocks over the distributed memory of a parallel system must be at least a few hundreds kylobytes in both cases, that is, robustness is guaranteed with scalability only with very large files [14], [26]. This is still true with improved variants employing either fixed-lenght codewords [27], [28] or variable-length ones [29], [30], [31], [32], [33].

Finally, we introduce pseudo-prefix and pseudo-suffix dictionaries and show that the algorithms computing optimal factorizations with suffix-closed and prefix-closed dictionaries still work. The advantage of using pseudo-prefix and pseudo-suffix dictionaries is that we add to an arbitrary dictionary only those prefixes or suffixes needed to guarantee the correctness of the optimal solution. This implies a slight improvement on the compression ratio obtained by the distributed implementations. Moreover, we show the impossibility of real-time optimal factorizations if the dictionary is arbitrary.

In Section II, we describe the different approaches to dictionary-based static text compression. The previous work on parallel approximations of optimal compression with prefix-closed dictionaries is given in Section III. Section IV shows the finite-state machine and the two implementations of the greedy approach for arbitrary dictionaries. Experiments are discussed in Section V. Section VI presents the notion of pseudo-prefix and pseudo-suffix dictionaries, where theoretical and further experimental results are discussed. Conclusions and future work are given in Section VII.

II. Dictionary-Based Static Text Compression

As mentioned in the introduction, the dictionary comprises typical factors (including the alphabet characters) associated with fixed or variable length codewords. The optimal factorization is the one minimizing the sum of the codeword lengths and sequential algorithms for computing optimal solutions were provided by means of dynamic programming techniques [3] or by reducing the problem to the one of finding a shortest path in a directed acyclic graph [4]. When the codewords are fixed-length, with suffix-closed dictionaries we obtain optimality by means of a simple left to right greedy approach, that is, advancing with the on-line reading of the input string by selecting the longest matching factor with a dictionary element. Such a procedure can be computed in real time by storing the dictionary in a trie data structure. If the dictionary is prefix-closed, there is an optimal semi-greedy factorization which is computed by the procedure of Figure 1 [6], [7]. At each step, we select a factor such that the longest match in the next position with a dictionary element ends to the rightest. Since the dictionary is prefix-closed, the factorization is optimal. The algorithm can even be implemented in real time with a modified trie data structure [7].

j:=0; i:=0
repeat forever
  for $k = j+1$ **to** $i+1$ **compute**
    $h(k)$: $x_k ... x_{h(k)}$ is the longest match in the kth position
  let k' be such that $h(k')$ is maximum
  $x_j ... x_{k'-1}$ is a factor of the parsing; $j := k'$; $i := h(k')$

Figure 1. The semi-greedy factorization procedure.

The semi-greedy factorization can be generalized to any dictionary by considering only those positions, among the

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

ones covered by the current factor, next to a prefix that is a dictionary element [6]. The generalized semi-greedy factorization procedure is not optimal while the greedy one is not optimal even when the dictionary is prefix-closed. The maximum length of a dictionary element is an obvious upper bound to the multiplicative approximation factor of any string factorization procedure with respect to the optimal solution. We show that this upper bound is tight for the greedy and semi-greedy procedures when the dictionary is arbitrary and that such tightness is kept by the greedy procedure even for prefix-closed dictionaries. Let $baba^n$ be the input string and let $\{a, b, bab, ba^n\}$ be the dictionary. Then, the optimal factorization is b, a, ba^n while $bab, a, a, ..., a, ...a$ is the factorization obtained whether the greedy or the semi-greedy procedure is applied. On the other hand, with the prefix-closed dictionary $\{a, b, ba, bab, ba^k : 2 \leq k \leq n\}$, the optimal factorization ba, ba^n is computed by the semi-greedy approach while the greedy factorization remains the same. These examples, obviously, prove our statement on the tightness of the upper bound.

III. Previous Work

Given an arbitrary dictionary, for every integer k greater than 1 there is an $O(km)$ time, $O(n/km)$ processors distributed algorithm factorizing an input string S with a cost which approximates the cost of the optimal factorization within the multiplicative factor $(k+m-1)/k$, where n and m are the lengths of the input string and the longest factor respectively [14]. However, with prefix-closed dictionaries a better approximation scheme was presented in [23], producing a factorization of S with a cost approximating the cost of the optimal factorization within the multiplicative factor $(k+1)/k$ in $O(km)$ time with $O(n/km)$ processors. This second approach was designed for massively parallel architecture and is suitable for extreme distributed systems, when the scale is beyond standard large values. On the other hand, the first approach applies to standard small, medium and large scale systems. Both approaches provide approximation schemes for the corresponding factorization problems since the multiplicative approximation factors converge to 1 when km converge to n. Indeed, in both cases compression is applied in parallel to different blocks of data independently. Beyond standard large scale, adjacent blocks overlap and the neighborhoods of the boundaries are preprocessed.

To decode the compressed files on a distributed system, it is enough to use a special mark occurring in the sequence of pointers each time the coding of a block ends. The input phase distributes the subsequences of pointers coding each block among the processors. Since a copy of the dictionary is stored in every processor, the decoding of the blocks is straightforward. In the following two subsections, we describe the two approaches. Then, how to speed up the preprocessing phase of the second approach is described in the last subsection. Next section will argue that we can relax the requirement of computing a theoretical approximation of optimal compression since, in practice, the greedy approach is nearly optimal on data blocks sufficiently long. On the other hand, when the blocks are too short because the scale of the distributed system is beyond standard values, the overlapping of the adjacent blocks and the preprocessing of the neighborhoods of the boundaries are necessary to garantee the robustness of the greedy approach.

A. Standard Scale Distributed Systems

We simply apply in parallel the optimal compression to blocks of length km. Every processor stores a copy of the dictionary. For an arbitrary dictionary, we execute the dynamic programming procedure computing the optimal factorization of a string in linear time [3] (the procedure in [4] is pseudo-linear for fixed-length coding and, even, super-linear for variable length). Obviously, this works for prefix- and suffix-closed dictionaries as well and, in any case, we know the semi-greedy and greedy approaches are implementable in linear time. It follows that the algorithm requires $O(km)$ time with n/km processors and the multiplicative approximation factor is $(k+m-1)/k$ with respect to any factorization. Indeed, when the boundary cuts a factor the suffix starting the block and its substrings might not be in the dictionary. Therefore, the multiplicative approximation factor follows from the fact that $m-1$ is the maximum length for a proper suffix as shown in Figure 2 (sequence of plus signs in parentheses). If the dictionary is suffix-closed, the multiplicative approximation factor is $(k+1)/k$ since each suffix of a factor is a factor.

```
            +(+++++++)
-----------/----------------
```

Figure 2. The making of the surplus factors.

The approximation scheme is suitable only for standard scale systems unless the file size is very large. In effect, the block size must be in the order of kilobytes to guarantee robustness. Beyond standard large scale, overlapping of adjacent blocks and a preprocessing of the boundaries are required as we will see in the next subsection.

B. Beyond Standard Large Scale

With prefix-closed dictionaries a better approximation scheme was presented in [23]. During the input phase blocks of length $m(k+2)$, except for the first one and the last one that are $m(k+1)$ long, are broadcasted to the processors. Each block overlaps on m characters with the adjacent block to the left and to the right, respectively (obviously, the first one overlaps only to the right and the last one only to the left). We call a *boundary match* a factor

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

covering positions in the first and second half of the $2m$ characters shared by two adjacent blocks. The processors execute the following algorithm to compress each block:

- for each block, every corresponding processor but the one associated with the last block computes the boundary match between its block and the next one ending furthest to the right, if any;
- each processor computes the optimal factorization from the beginning of its block to the beginning of the boundary match on the right boundary of its block (or the end of its block if there is no boundary match).

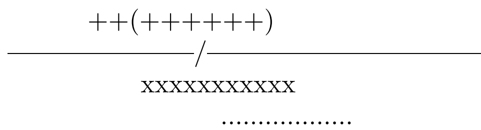

Figure 3. The making of a surplus factor.

Stopping the factorization of each block at the beginning of the right boundary match might cause the making of a surplus factor, which determines the multiplicative approximation factor $(k+1)/k$ with respect to any other factorization. Indeed, as it is shown in Figure 3, the factor in front of the right boundary match (sequence of x's) might be extended to be a boundary match itself (sequence of plus signs) and to cover the first position of the factor after the boundary (dotted line). Then, the approximation scheme produces a factorization of S with a cost approximating the cost of the optimal factorization within the multiplicative factor $(k+1)/k$ in $\mathrm{O}(km)$ time with $\mathrm{O}(n/km)$ processors (we will see in the next subsection how the preprocessing can be executed in $\mathrm{O}(m)$ time).

In [23], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorizarion is about the same as the sequential one and, consequently, the approach is suitable for extreme distributed systems, as we will explain in the next section.

C. Speeding up the Preprocessing

The parallel running time of the preprocessing phase computing the boundary matches is $\mathrm{O}(m^2)$ by brute force. To lower the complexity to $\mathrm{O}(m)$, an augmented trie data structure is needed [1]. For each node v of the trie, let f be the dictionary element corresponding to v and a an alphabet character not represented by an edge outgoing from v. Then, we add an edge from v to w with label a, where w represents the longest proper suffix of fa in the dictionary. Each processor has a copy of this augmented trie data structure and first preprocess the $2m$ characters overlapped by the adjacent block on the left boundary and, secondly, the ones on the right boundary. In each of these two sub-phases, the processors advance with the reading of the $2m$ characters from left to right, starting from the first one while visiting the trie starting from the root and using the corresponding edges. A temporary variable t_2 stores the position of the current character during the preprocessing while another temporary variable t_1 is, initially, equal to t_2. When an added edge of the augmented structure is visited, the value $t = t_2 - d + 1$ is computed where d is the depth of the node reached by such edge. If t is a position in the first half of the $2m$ characters, then t_1 is updated by changing its value to t. Else, the procedure stops and t_2 is decreased by 1. If t_2 is a position in the second half of the $2m$ characters then t_1 and t_2 are the first and last position of a boundary match, else there is no boundary match.

IV. The Greedy Approach

We provide a finite-state machine implementation of the greedy approach with an arbitrary dictionary. Then, we show the two implementations on standard large scale and extreme distributed systems.

A. The Finite-State Machine Implementation

We show the finite-state machine implementation producing the on-line greedy factorization of a string with an arbitrary dictionary. The most general formulation for a finite-state machine M is to define it as a six-tuple $M = (A, B, Q, \delta, q_0, F)$ with an input alphabet A, an output alphabet B, a set of states Q, a transition function $\delta : Q\mathrm{x}A \rightarrow Q\mathrm{x}B^*$, an initial state q_0 and a set of accepting states $F \subseteq Q$. The trie storing the dictionary is a subgraph of the finite-state machine diagram. It is well-known that each dictionary element is represented as a path from the root to a node of the trie where edges are labeled with an alphabet character (the root representing the empty string). The edges are directed from the parent to the child and the set of nodes represent the set of states of the machine. The output alphabet is binary and the factorization is represented by a binary string having the same length as the input string. The bits of the output string equal to 1 are those corresponding to the positions where the factors start. Since every string can be factorized, every state is accepting. The root represents the initial state. We need only to complete the function δ, by adding the missing edges of the diagram. The empty string is associated as output to the edges in the trie. For each node, the outgoing edges represent a subset of the input alphabet. Let f be the string (or dictionary element) corresponding to the node v in the trie and a an alphabet character not represented by an edge outgoing from v. Let $fa = f_1 \cdots f_k$ be the on-line greedy factorization of fa and i the smallest index such that $f_{i+1} \cdots f_k$ is represented by a node w in the trie. Then, we add to the trie a directed edge from v to w with label a. The output associated with the

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

edge is the binary string representing the sequence of factors $f_1 \cdots f_i$. By adding such edges, the machine is entirely defined. Redefining the machine to produce the compressed form of the string is straightforward.

B. The Distributed Implementations

In practice, greedy factorization is nearly optimal. As a first approach, we simply apply in parallel left to right greedy compression to blocks of length km. With standard scale systems, the block size must be the order of kilobytes to guarantee robustness. Each of the $O(n/km)$ processors could apply the finite-state machine implementation to its block. Beyond standard large scale, overlapping of adjacent blocks and a preprocessing of the boundaries are required as for the optimal case. Again, during the input phase overlapping blocks of length $m(k+2)$ are broadcasted to the processors as in the previous section. On the other hand, the definition of boundary match is extended to those factors, which are suffixes of the first half of the $2m$ characters shared by two adjacent blocks. The following procedure, even if it is not an approximation scheme from a theoretical point of view, performs in a nearly optimal way:

- for each block, every corresponding processor but the one associated with the last block computes the longest boundary match between its block and the next one;
- each processor computes the greedy factorization from the end of the boundary match on the left boundary of its block to the beginning of the boundary match on the right boundary.

The approach is nearly optimal for $k = 10$, as the approximation scheme of the previous section. The compression ratio achieved by such factorizarion is about the same as the sequential one. Considering that typically the average match length is 10, one processor can compress down to 100 bytes independently. This is why the approximation scheme was presented for massively parallel architecture and the approach, presented in this section, is suitable for extreme distributed systems, when the scale is beyond standard large values. Indeed, with a file size of several megabytes or more, the system scale has a greater order of magnitude than the standard large scale parameter. We wish to point out that the computation of the boundary matches is very relevant for the compression effectiveness, when an extreme distributed system is employed, since the sub-block length becomes much less than 1K. With standard large scale systems the block length is several kilobytes with just a few megabytes to compress and the approach using boundary matches is too conservative. After preprocessing, each of the $O(n/km)$ processors could apply the finite-state machine implementation to its block. However, blocks are so short that it becomes irrelevant. On the other hand, with standard scale systems and very large size files the application of the finite-state machine in parallel to the distributed blocks plays an important role to achieve high speed.

C. Speeding up the Preprocessing

To lower the time of the preprocessing phase to $O(m)$, the same augmented trie data structure, described in the previous section, is needed but, in this case, the boundary matches are the longest ones rather than the ones ending furthest to the right. Then, besides the temporary variables t_1 and t_2, employed by the preprocessing phase described in the previous section, two more variables τ_1 and τ_2 are required and, initially, equal to t_1 and t_2. Each time t_1 must be updated by such preprocessing phase, the value $t_2 - t_1 + 1$ is compared with $\tau_2 - \tau_1$ before updating. If it is greater or τ_2 is smaller than the last position of the first half of the $2m$ characters, τ_1 and τ_2 are set equal to t_1 and $t_2 - 1$. Then, t_1 is updated. At the end of the procedure, τ_1 and τ_2 are the first and last positions of the longest boundary match. We wish to point out that there is always a boundary match that is computed, since the final value of τ_2 always corresponds to a position equal either to one in the second half of the $2m$ characters or to the last position of the first half.

V. Experimental Results

Suffix-closed and prefix-closed dictionaries have been considered in static data compression because they are constructed by the LZ77 [24] and LZ78 [11] adaptive compression methods, when reading a typical string of a given source of data. When the input string to compress matches the characteristics of a dictionary given in advance and already filled with typical factors, the advantage in terms of compression efficiency is obvious. However, the bounded size of the dictionary (typically, 2^{16} factors) and its static nature imply a lack of robustness and the adaptive methods might result more effective in some cases, even if the type of data is known and the dictionary is very well constructed. We experimented this with the "compress" command line on the Unix and Linux platforms, which is the implementation of a variant of the LZ78 method, called the LZC method. LZC builds a prefix-closed dictionary of 2^{16} factors while compressing the data. When the dictionary is full, it applies static dictionary greedy compression monitoring at the same time the compression ratio. When the compression ratio starts deteriorating, it clears the dictionary and restarts dynamic compression alternating, in this way, adaptive and non-adaptive compression. We experimented that, when compressing megabytes of english text with a static prefix-closed dictionary optimally, there might be up to a ten percent loss in comparison with the compression ratio of the LZC method [23]. However, as we pointed out earlier, there is no scalable and robust implementation of the LZC method on a distributed memory system (except for the static phase of the method as shown in [26]), while a

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

nearly optimal compression distributed algorithm is possible with no scalability and robustness issues if we accept a ten percent compression ratio loss as a reasonable upper bound to the price to pay for it [23].

A prefix-closed dictionary D in [23] was filled up with 2^{16} elements, starting from the alphabet (each of the 256 bytes). Then, for each of the most common subbstrings listed in [9], every prefix of length less or equal to ten was added to D. On the other hand, for each string with no capital letters and less than eleven characters in the Unix dictionary of words, we added every prefix of length less or equal to six. For every word in the Unix dictionary inserted in D, a space was concatenated at the end of the copy in D. Another copy ending with the new line character was inserted if the word length is less than six. Finally, it was enough to add a portion of the words with six characters plus a new line character to fill up D. The average optimal compression ratio we obtained with this dictionary is 0.51, while the greedy one is even 0.57. On the other hand, the LZC average compression ratio is 0.42. It turned out that both gaps are consistently reduced when the prefix property of the dictionary is relaxed. A not prefix-closed dictionary D' was filled up with 2^{16} elements, starting from the alphabet and the 477 most common subbstrings listed in [9]. Then, we added each string with no capital letters and less than ten characters from the Unix dictionary of words. Again, for every word in the Unix dictionary inserted in D', a space was concatenated at the end of the copy in D'. Finally, it was enough to add a portion of short words with a new line character at the end to fill up D'. With such dictionary, the loss on the compression ratio goes down from ten to five percent with respect to the adaptive LZC compression. Moreover, the greedy approach has just a one percent loss with respect to optimal, as shown in Figure 4. This is because the dictionary is better constructed. In Figure 4, we also show the compression effectiveness results for the two approaches with or without boundaries preprocessing (that is, for an extreme or a standard distibuted system). The two approaches perform similarly and have a one percent loss with respect to sequential greedy, whether the dictionary is prefix-closed or not.

Dictionary	Optimal	Greedy	Standard	Extreme
Prefix-Closed	.51	.57	.58	.58
Not Prefix-Closed	.47	.48	.49	.49

Figure 4. Compression ratios with english text.

We observed in the introduction that for read-only memory files, speeding up decompression is what really matters in practice. In this context, the results presented in this paper suggest a dynamic approach (that is, working for any type of input), where the dictionary is not given in advance but learned from the input string and, then, used staticly to compress the string. This models a scheme where compression is performed only once with an off-line sequential procedure reading the string twice from left to right in such a way that decompression can be parallelized with no scalability issues. The first left-to-right reading is to learn the dictionary and better ways than the LZC algorithm exist since the dictionary provided by LZC, after reading the entire string, is constructed from a relatively short suffix of the input. A much more sophisticated approach employs the LRU (least recently used) strategy [9]. With such strategy, after the dictionay is filled up elements are removed in a continuous way by deleting at each step of the factorization the least recently used factor which is not a proper prefix of another one. A relaxed version of this approach was presented in [34], that is easier to implement, and experimental results show that the compression ratio with this type of dictionary goes down to 0.32 for english text [35]. This performance is kept if the greedy approach is applied staticly during the second reading of the string, using the dictionary obtained from the first reading. Moreover, if the compression is applied independently to different blocks of data of 1Kb or to smaller blocks after the boundaries preprocessing, there is still just a one percent loss on the compression ratio.

VI. Pseudo-Prefix and Pseudo-Suffix Dictionaries

We partially relax the suffix and prefix properties to keep respectively optimal the greedy and semi-greedy approaches by introducing pseudo-suffix and pseudo-prefix dictionaries. Then, we give an insight of why optimal factorizations can be computed in real time with pseudo-prefix and pseudo-suffix dictionaries while this is not possible if the dictionary is arbitrary. Finally, we present experimental results using pseudo-prefix dictionaries.

A. Introducing Pseudo-Prefix and -Suffix Dictionaries

Given a finite alphabet A, let p and s be a prefix and a suffix of a string $x \in A^*$ such that $x = ps$. Then, we call p the *complementary* prefix of s with respect to x. Accordingly, we call s the *complementary* suffix of p with respect to x. We say a dictionary D is *pseudo-prefix* if:

- let p be a prefix of $x \in D$ such that the complementary suffix s with respect to x is a prefix of an element in D. Then, $p \in D$.

Accordingly, we say a dictionary is *pseudo-suffix* if:

- let s be a suffix of $x \in D$ such that the complementary prefix p with respect to x is a suffix of an element in D. Then, $s \in D$.

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

We prove, now, the optimality of the on-line greedy factorization approach with pseudo-suffix dictionaries.

Theorem 1. Given a finite alphabet A, let $D \subseteq A^*$ be a pseudo-suffix dictionary. For every $x \in A^*$, the on-line greedy factorization of x is optimal.

Proof. The pseudo-suffix property implies, as the suffix property, that the on-line greedy approach selects, at each step, the factor ending furthest to the right. Indeed, assume that the factor selected by the greedy choice at the i-th step of the process ends to the right of the i-th factor of the optimal solution (which is always true at the first step). Then, there is a suffix s of the $i+1$-th factor of the optimal solution with a complementary prefix that is a suffix of the factor selected by the greedy choice at the i-th step. It follows that s is a dictionary element. Therefore, the on-line greedy approach selects, at each step, the factor ending furthest to the right and its optimality follows. q. e. d.

With the next theorem, we prove the optimality of the semi-greedy factorization process with pseudo-prefix dictionaries.

Theorem 2. Given a finite alphabet A, let $D \subseteq A^*$ be a pseudo-prefix dictionary. For every $x \in A^*$, the semi-greedy factorization of x is optimal.

Proof. The pseudo-prefix property implies, as the prefix property, that the semi-greedy approach selects, at each step, a factor such that the longest match in the next position with a dictionary element ends to the rightest. This is true at the first step, since for each suffix of the greedy factor that is a prefix of a dictionary element the complementary prefix is a dictionary element. Then, inductively, it is true for every step and the optimality follows. q. e. d.

It follows from the two theorems above and the results shown in the previous section that, as for prefix-closed and suffix-closed dictionaries, real-time optimal factorizations are possible with pseudo-prefix and pseudo-suffix dictionaries. Moreover, an optimal factorization using a pseudo-suffix dictionary is implementable with a finite state machine. However, the making of a pseudo-prefix dictionary is much simpler than the making of a pseudo-suffix one. Indeed, let D be an arbitrary dictionary stored in a trie and add prefixes of its elements to make it pseudo-prefix. The most natural way to do this is to visit the trie with a depth-first search. For each path from the root to a node representing a string not in D, such string is added to D if a descendant of the node is in D. The running time for such procedure is about the dictionary size times the depth of the trie.

B. *Canonical Factors*

We prove a property concerning optimal factorizations with pseudo-prefix and pseudo-suffix dictionaries. This property was previously proved for prefix-closed dictionaries in [18] and it gives an insight of why optimal factorizations can be computed in real time with this type of dictionary. First, we prove the property for pseudo-prefix dictionaries.

Theorem 3. Given a finite alphabet A, let $D \subseteq A^*$ be a pseudo-prefix dictionary. Let k be the number of factors of an optimal factorization of a string $s \in A^*$. Then, for $1 \leq i \leq k$, there is an optimal factorization such that its i-th factor is a substring of the i-th factor of every other optimal factorization of s.

Proof. First of all, given two optimal factorizations $s = f_1^1 \cdots f_k^1 = f_1^2 \cdots f_k^2$ we prove that f_i^1 and f_i^2 overlap for $1 \leq i \leq k$. Given any substring f of s, denote with $first(f)$ and $last(f)$ the first and the last position of s covered by f. Then, suppose $last(f_i^2) < first(f_i^1)$ for some i with $1 < 1 < k$. Let j be such that $first(f_j^1) \leq last(f_i^2) \leq last(f_j^1)$. It follows from the optimality of the two factorizations and the pseudo-prefix property that $last(f_j^1) < last(f_{i+1}^2)$. Denote with $pref(f_j^1)$ the prefix of f_j^1 such that $last(pref(f_j^1)) = last(f_i^2)$. Then, $pref(f_j^1) \in D$ since D is pseudo-prefix. It follows that the factorization $f_1^1 \cdots f_{j-1}^1 pref(f_j^1) f_{i+1}^2 \cdots f_k^2$ comprises less than k phrases since $j < i$. Therefore, f_i^1 and f_i^2 must have a not empty intersection. Suppose now that, for every optimal factorization $s = f_1 \cdots f_k$, $first(f_i) \leq first(f_i^1)$ and $last(f_i) \geq last(f_i^2)$. Denote with $pref(f_i^1)$ the prefix of f_i^1 such that $last(pref(f_i^1)) = last(f_i^2)$. Then, $pref(f_i^1) \in D$ since D is pseudo-prefix. It follows that $f_1^1 \cdots f_{i-1}^1 pref(f_i^1) f_{i+1}^2 \cdots f_k^2$ is an optimal factorization of s, with $pref(f_i^1)$ substring of the i-th factor of every other optimal factorization. q. e. d.

In the next theorem, we prove the property for pseudo-suffix dictionaries with similar arguments.

Theorem 4. Given a finite alphabet A, let $D \subseteq A^*$ be a pseudo-suffix dictionary. Let k be the number of factors of an optimal factorization of a string $s \in A^*$. Then, for $1 \leq i \leq k$, there is an optimal factorization such that its i-th factor is a substring of the i-th factor of every other optimal factorization of s.

Proof. First of all, given two optimal factorizations $s = f_1^1 \cdots f_k^1 = f_1^2 \cdots f_k^2$ we prove that f_i^1 and f_i^2 overlap for $1 \leq i \leq k$. Given any substring f of s, denote with $first(f)$ and $last(f)$ the first and the last position of s covered by f, as in Theorem 3. Then, suppose $last(f_i^2) < first(f_i^1)$ for some i with $1 < 1 < k$.

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Let j be such that $first(f_j^1) \leq last(f_i^2) \leq last(f_j^1)$. It follows from the optimality of the two factorizations and the pseudo-suffix property that $first(f_j^1) > first(f_{i-1}^2)$. Denote with $suff(f_i^2)$ the suffix of f_i^2 such that $first(suff(f_i^2)) = first(f_j^1)$. Then, $suff(f_i^2) \in D$ since D is pseudo-suffix. It follows that the factorization $f_1^1 \cdots f_{j-1}^1 suff(f_i^2) f_{i+1}^2 \cdots f_k^2$ comprises less than k phrases since $j < i$. Therefore, f_i^1 and f_i^2 must have a not empty intersection. Suppose now that, for every optimal factorization $s = f_1 \cdots f_k$, $first(f_i \leq first(f_i^1)$ and $last(f_i) \geq last(f_i^2)$. Denote with $suff(f_i^2)$ the suffix of f_i^2 such that $first(suff(f_i^2)) = first(f_i^1)$. Then, $suff(f_i^2) \in D$ since D is pseudo-suffix. It follows that $f_1^1 \cdots f_{i-1}^1 suff(f_i^2) f_{i+1}^2 \cdots f_k^2$ is an optimal factorization of s, with $suff(f_i^2)$ substring of the i-th factor of every other optimal factorization. q. e. d.

Given a finite alphabet A, a dictionary $D \subseteq A^*$ and a string $s \in A^*$, let a string $f \in A^*$ be the i-th factor of an optimal factorization of s with respect to D for some positive integer i less or equal to the optimal cost k. Then, we call f *canonical* if it is the substring of the i-th of every other optimal factorization of s. We proved, in the two theorems above, that if the dictionary is pseudo-prefix or pseudo-suffix then, given any input string, for every positive integer between 1 and the optimal factorization cost there is a canonical factor. The presence of these cannical factors gives an insight of why a real-time factorization is possible for this type of dictionaries since it proves that in order to determine the next factor of an optimal factorization we need to process only the current one.

Now, we show the impossibility of a real-time optimal factorization for every input string if the dictionary is arbitrary, by presenting an example where the dictionary is $\{a, b, a^i, a^j, a^{k(j-i)}b^2\}$ with $ki < j$. Then, we consider two strings s_1 and s_2 sharing the same prefix a^{kj} but with two different complementary suffixes equal, respectively, to $a^{k(j-i)}b^2$ and b^2. Then, the optimal factorization is $a^j, ..., a^j, a^{k(j-i)}b^2$ for s_1 and $a^i, ..., a^i, a^{k(j-i)}b^2$ for s_2. This proves that any approach to produce an optimal factorization is not independent from the maximum factor length L of the dictionary and that the complexity of the optimal factorization problem is $\Omega(nL)$, where n is the input string length.

C. Experimental Results

The results presented in Figure 4 for a not prefix-closed dictionary D' are reported again in Figure 5 as results for a not pseudo-prefix dictionary, since the dictionary D' described in the previous section was not pseudo-prefix as well. If we add prefixes to D' to make it pseudo-prefix, optimal compression is the same as before and greedy is basically optimal (less than one percent loss), as shown in Figure 5. We also show that the compression effectiveness results for the pseudo-prefix dictionary with or without boundaries preprocessing (that is, for an extreme or a standard distibuted system) have a one percent loss with respect to sequential greedy, so the pseudo-prefix dictionary has a better performance.

Dictionary	Optimal	Greedy	Standard	Extreme
Pseudo-Prefix	.47	.47	.48	.48
Not Pseudo-Prefix	.47	.48	.49	.49

Figure 5. Compression ratios with english text.

Now, we consider the off-line dynamic approach reading twice the input, which can be applied in the case of read-only memory files. The dictionary, bounded by the LRU strategy, is not pseudo-prefix if it is learned by the heuristic in [13]. We mentioned in the previous section the average compression ratio for english text is 0.32 and if the compression is applied independently to different blocks of data of 1Kb or to smaller blocks after the boundaries preprocessing, there is still just a one percent loss on the compression ratio. This loss disappears if we make the dictionary pseudo-prefix and apply the approximation scheme in [23] to optimal compression.

VII. Conclusion

We presented parallel implementations of the greedy approach to dictionary-based static text compression suitable for standard and non-standard large scale distributed systems. In order to push scalability beyond what is traditionally considered a large scale system, a more involved approach distributes overlapping blocks to compute boundary matches. These boundary matches are relevant to maintain the compression effectiveness on a so-called extreme distributed system. If we have a standard small, medium or large scale system available, the approach with no boundary matches can be used. The absence of a communication cost during the computation guarantees a linear speed-up. Moreover, the finite state machine implementation speeds up the execution of the distributed algorithm in a relevant way when the data blocks are large, that is, when the size of the input file is large and the size of the distributed system is relatively small. Finally, we introduced the notion of pseudo-prefix dictionary, which allows optimal compression by means of a real-time semi-greedy procedure and a slight improvement on the compression ratio obtained by the distributed implementations. As future work, experiments on parallel running times should be done to see how the preprocessing phase effects on the linear speed-up when the system is scaled up beyond the standard size and how relevant the employment of the finite state machine implementation is when the data blocks are small.

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

References

[1] S. DeAgostino, "The Greedy Approach to Dictionary-Based Static Text Compression on a Distributed System," Proceedings International Conference on Advances Engeneering Computing with Applications to Sciences, 2014, pp. 1-6.

[2] S. DeAgostino, "Approximating Dictionary-Based Optimal Data Compression on a Distributed System," to appear in Proceedings ACM International Conference on Computing Frontiers, 2015.

[3] R. A. Wagner, "Common Phrases and Minimum Text Storage," Communications of the ACM, vol. 16, 1973, pp. 148-152.

[4] E. J. Shoegraf and H. S. Heaps, "A Comparison of Algorithms for Data Base Compression by Use of Fragments as Language Elements," Information Storage and Retrieval, vol. 10, 1974, pp. 309-319.

[5] M. Cohn and R. Khazan, "Parsing with Suffix and Prefix Dictionaries," Proceedings IEEE Data Compression Conference, 1996, pp. 180-189.

[6] M. Crochemore and W. Rytter, Jewels of Stringology, World Scientific, 2003.

[7] A Hartman and M. Rodeh, "Optimal Parsing of Strings," Combinatorial Algorithms on Words (eds. Apostolico, A., Galil, Z.), Springer, 1985, pp. 155-167.

[8] T. C. Bell, J. G. Cleary, and I. H. Witten, Text Compression, Prentice Hall, 1990.

[9] J. A. Storer, Data Compression: Methods and Theory, Computer Science Press, 1988.

[10] A. Lempel and J. Ziv, "On the Complexity of Finite Sequences," IEEE Transactions on Information Theory, vol. 22, 1976, pp. 75-81.

[11] J. Ziv and A. Lempel, "Compression of Individual Sequences via Variable-Rate Coding," IEEE Transactions on Information Theory, vol. 24, 1978, pp. 530-536.

[12] T. A. Welch, "A Technique for High-Performance Data Compression," IEEE Computer, vol. 17, 1984, pp. 8-19.

[13] V. S. Miller and M. N. Wegman, "Variations on Theme by Ziv - Lempel," Combinatorial Algorithms on Words (eds. Apostolico, A., Galil, Z.), Springer, 1985, pp. 131-140.

[14] L. Cinque, S. De Agostino, and L. Lombardi, "Scalability and Communication in Parallel Low-Complexity Lossless Compression," Mathematics in Computer Science, vol. 3, 2010, pp. 391-406.

[15] S. De Agostino, Sub-Linear Algorithms and Complexity Issues for Lossless Data Compression, Master's Thesis, Brandeis University, 1994.

[16] S. De Agostino, Parallelism and Data Compression via Textual Substitution, Ph. D. Dissertation, Sapienza University of Rome, 1995.

[17] S. De Agostino, "Parallelism and Dictionary-Based Data Compression," Information Sciences, vol. 135, 2001, pp. 43-56.

[18] S. De Agostino S. and J. A. Storer, "Parallel Algorithms for Optimal Compression Using Dictionaries with the Prefix Property," Proceedings IEEE Data Compression Conference, 1992, pp. 52-61.

[19] D. S. Hirschberg and L. M. Stauffer, "Parsing Algorithms for Dictionary Compression on the PRAM," Proceedings IEEE Data Compression Conference, 1994, pp. 136-145.

[20] D. S. Hirschberg and L. M. Stauffer, "Dictionary Compression on the PRAM," Parallel Processing Letters, vol. 7, 1997, pp. 297-308.

[21] H. Nagumo, M. Lu, and K. Watson, "Parallel Algorithms for the Static Dictionary Compression," Proceedings IEEE Data Compression Conference, 1995, pp. 162-171.

[22] L. M. Stauffer and D. S. Hirschberg, "PRAM Algorithms for Static Dictionary Compression," Proceedings International Symposium on Parallel Processing, 1994, pp. 344-348.

[23] D. Belinskaya, S. De Agostino, and J. A. Storer, "Near Optimal Compression with respect to a Static Dictionary on a Practical Massively Parallel Architecture," Proceedings IEEE Data Compression Conference, 1995, pp. 172-181.

[24] A. Lempel and J. Ziv, "A Universal Algorithm for Sequential Data Compression," IEEE Transactions on Information Theory, vol. 23, 1977, pp. 337-343.

[25] S. DeAgostino, "Parallel Implementations of Dictionary Text Compression without Communication," London Stringology Days, 2009.

[26] S. DeAgostino, "LZW Data Compression on Large Scale and Extreme Distributed System," Proceedings Prague Stringology Conference, 2012, pp. 18-27.

[27] Y. Matias and C. S. Sahinalp, "On the Optimality of Parsing in Dynamic Dictionary-Based Data Compression," Proceedings SIAM-ACM Symposium on Discrete Algorithms, 1999, pp. 943-944.

[28] M. Crochemore, A. Langiu, and F. Mignosi, "Note on the Greedy Parsing Optimality for Dictionary-Based Text Compression," Theoretical Computer Science, vol. 525, 2014, pp. 55-59.

[29] M. Crochemore, L. Gianbruno, A. Langiu, F. Mignosi, and A. Restivo, "Dictionary-Simbolwise Flexible Parsing," Journal of Discrete Algorithms, vol. 14, 2012, pp. 74-90.

[30] A. Farrugia, P. Ferragina, A. Frangioni, and R. Venturini, "Bicriteria Data Compression," Proceedings SIAM-ACM Symposium on Discrete Algorithms, 2014, pp. 1582-1585.

[31] P. Ferragina, I. Nitto, and R. Venturini, "On Optimally Partitioning a Text to Improve Its Compression," Algorithmica, vol. 61, 2011, pp. 51-74.

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

[32] P. Ferragina, I. Nitto, and R. Venturini, ”On the Bit-Complexity of Lempel-Ziv Compression,” SIAM Journal on Computing, vol. 42, 2013, pp. 1521-1541.

[33] A. Langiu, ”On Parsing Optimality for Dictionary-Based Text Compression - the Zip Case”, Journal of Discrete Algorithms, vol. 20, 2013, pp. 65-70.

[34] S. DeAgostino and R. Silvestri, ”Bounded Size Dictionary Compression: SC^k-Completeness and NC Algorithms,” Information and Computation, vol. 180, 2003, pp. 101-112.

[35] S. DeAgostino, ”Bounded Size Dictionary Compression: Relaxing the LRU Deletion Heuristic,” Proceedings Prague Stringology Conference, 2005, pp. 135-142.

2015, © Copyright by authors, Published under agreement with IARIA - www.iaria.org