Greedy versus Optimal Analysis of Bounded Size Dictionary Compression and On-the-Fly Distributed Computing

Sergio De Agostino

Computer Science Department
Sapienza University of Rome
Via Salaria 113, 00198 Rome, Italy
deagostino@di.uniroma1.it

Abstract. Scalability and robustness are not an issue when compression is applied for massive data storage, in the context of distributed computing. Speeding up on-the-fly compression for data transmission is more controversial. In such case, a compression technique merging together an adaptive and a non-adaptive approach has to be considered. A practical implementation of LZW (Lempel, Ziv and Welch) compression, called LZC (C indicates the Unix command 'compress'), has this characteristic. The non-adaptive phases work with bounded size prefix dictionaries built by LZW factorizations during the adaptive ones. In order to improve the compression effectiveness, we suggest to apply LZMW (Lempel, Ziv, Miller and Wegman) factorization to LZC compression (LZCMW) during the adaptive phases since it builds better dictionaries than LZW. The LZMW heuristic was originally thought with a dictionary bounded by the least recently used strategy. We introduce the LZCMW heuristic in order to have non-adaptive phases. All the heuristics mentioned above employ the greedy approach. We show, finally, a worst case analysis of the greedy approach with respect to the optimal solution decodable by the LZC decompressor. Such analysis suggests parallelization of on the fly compression is not suitable for highly disseminated data since the non-adaptive phases are too far from optimal.

Keywords: on-the-fly compression, factorization, distributed system, scalability

1 Introduction

Massive data are, usually, defined as big when the order of magnitude is greater than a terabyte. Compression is considered advantageous and actionable by the big data community since it is possible to evaluate many predicates without having to decompress. However, the locality principle stated by Zipf's law [26] for arbitrarily large datasets provides another good reason (perhaps an even better reason in the near future) for massive data compression, that is, compression and, more importantly, decompression that are highly parallelizable. In fact, using the computational resources to speed up compression and decompression could be more practical than employing sophisticated techniques to query compressed data (as, for example, compressed pattern matching). Moreover, the locality principle is so general that parallelism could be applied to any kind of data and with any compression technique, even with smaller order of magnitudes than a petabyte. Indeed, any sequential compression technique could be applied in parallel and independently to relatively large pieces of data on standard small, medium and large scale distributed systems.

While speeding up massive data compression for storage on a distributed system is not an issue for the reasons mentioned above, speeding up on-the-fly compression

Sergio De Agostino: Greedy versus Optimal Analysis of Bounded Size Dictionary Compression and On-the-Fly Distributed Computing, pp. 74–83.
Proceedings of PSC 2020, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06749-9

for data transmission is more controversial. In such case, it seems that the only way to exploit the full computational power of a distributed system is to employ a technique comprising an adaptive phase followed by a non-adaptive one which can process the data until it becomes obsolete. Then, iteratively, the two phases can be repeated. This iteration of the sequential procedure can be parallelized if we divide the input into blocks of data long enough to run one step of the iteration on each block. Each block is, obviously, split in a first sub-block where the adaptive phase is run and a second one to be compressed in a non-adaptive way. A general parallel approach for this kind of compression technique is to compress the first sub-block with only one processor and, afterwards, to parallelize massively the compression of the second sub-block. We will see that this is possible if we employ a practical implementation of LZW compression [24], called LZC [1], since it has this characteristic. The non-adaptive phases work with bounded size prefix dictionaries built by LZW factorizations during the adaptive ones. In order to improve the compression effectiveness, we suggest to apply LZMW (Lempel, Ziv, Miller and Wegman) factorization to LZC compression (LZCMW) during the adaptive phases since it builds better dictionaries than LZW. The LZMW heuristic was originally thought with a dictionary bounded by the least recently used strategy [21]. We introduce the LZCMW heuristic in order to have non-adaptive phases.

All the heuristics mentioned above employ the greedy approach. We show, finally, a worst case analysis of the greedy approach with respect to the optimal solution decodable by the LZC decompressor. Such analysis suggests parallelization of on-the-fly compression is not suitable for highly disseminated data since the non-adaptive phases are too far from optimal. Massive data compression for storage and the locality principle are discussed in Section 2, where implementations of Zip and LZW compressors are described. How to speed-up on-the-fly LZW compression with distributed computing is faced in Section 3, where the improvement by means of the LZMW heuristic is discussed. The greedy versus optimal analysis is given in Section 5. Conclusion and future work of this pure theory paper are given in Section 6.

2 Distributed Massive Data Compression

Distributed systems have two types of complexities, 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 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. The arguments of next subsection imply that compressing massive data for storage is not an issue for any technique on such a distributed system and the most popular compressors belonging to the Zip family are described. On the other hand, improving the speed-up of on-the-fly compression for massive data transmission is more controversial and requires characteristics that, to our knowledge, only LZW compression has. We describe the unbounded and bounded memory versions of this technique in the second and third subsection so that the issue of massive data transmission and distributed computing can be faced in the next section.

2.1 The Locality Principle

Zipf's law states that the frequency of any word in a collection is inversely proportional to its rank in the frequency table. The most frequent word occurs twice as often as the second most frequent, and so on. The effect of this uneven distribution of byte patterns is evident in the effectiveness of common compression programs, as for example, the family of Zip compressors based on the Lempel-Ziv sliding window factorization technique [18], [19], [23]. Such factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where f_i is the longest match with a substring occurring previously in the prefix $f_1 f_2 \cdots f_i$ if $f_i \neq \lambda$ (empty string), otherwise f_i is the alphabet character next to $f_1 f_2 \cdots f_{i-1}$. f_i is encoded by the pointer $q_i = (d_i, \ell_i)$, where d_i is the displacement back to the copy of the factor and ℓ_i is the length of the factor. If $d_i = 0$, ℓ_i is the alphabet character. In other words, there is a window sliding its right end over the input string and all the substrings of the prefix read so far in the computation are potential reference copies for the current factor. In practice, the work space must be bounded and this is done by sliding a fixed length window and by bounding the match length. Simple real time implementations are realized by means of hashing techniques providing a specific position in the window where a good approximation of the longest match is found on realistic data. In [25], the two current characters are hashed and collisions are chained via an offset array. The Unix gzip compressor chains collisions too but hashes three characters [16].

While gzip stores 64 kB of history, it averages approximately 64 percent compression. Instead, bzip2 stores between 100 kB and 900 kB of history and averages 66 percent compression, a very small gain in comparison with the increase of memory. Therefore, 64 kB of history are enough to compress data and this can be generalized to any compression technique we want to use. Such locality principle is critical if we want to speed up compression and, more importantly, decompression on a large scale network. Indeed, we can apply compression in parallel to data blocks relatively small (a few hundreds kilobytes) since the locality principle holds for arbitrarily large datasets on the basis of Zipf's law and scalability and robustness are guaranteed. This follows from the fact that the loss of compression effectiveness due to the lack of history at the beginning of each block is amortized if the block length is one order of magnitude greater than the amount of past data to which we need to refer for compression [8]. Then, distributing data blocks of size approximately half a megabyte among the nodes of a network allows an efficient application of any adaptive compression method and guarantees scalability and robustness for massive data (for the interested reader, improved variants of the sliding window factorization technique exist employing either fixed-length codewords [2], [20] or variable-length ones [3], [13], [14], [15], [17]). Parallel decompression is symmetrical.

2.2 LZW Compression

Ziv-Lempel compression is a dictionary-based technique [18], [27], using a string factorization process where the factors of the string are substituted by *pointers* to copies stored in a dictionary which are called *targets*. The *standard* LZW (Lempel-Ziv-Welch) factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where each factor f_i is the longest match with the concatenation of a previous factor and the next character [24]. f_i is encoded by a pointer q_i whose target is such concatenation (LZW compression). LZW compression can be implemented in real time by storing the dictionary with a trie data structure. When the string length goes to infinity, also the dictionary size

does. Such unbounded version was proved to be P-complete [4], [5], [6], [7], [8], meaning that such factorization is hard to parallelize even on a shared memory random access machine or on a highly interconnected network. Therefore, we need to consider bounded memory versions of LZW compression in order to make such technique suitable for distributed computing.

2.3 Bounded Size Dictionaries

In practical implementations the dictionary size is bounded by a constant and the pointers have equal size. Let $d + \alpha$ be the cardinality of the fixed size dictionary where α is the cardinality of the alphabet. With the most naive approach, there is a first phase of the factorization process where the dictionary is filled up and "frozen". Afterwards, the factorization continues in a non-adaptive way using the factors of the frozen dictionary. In other words, the factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where f_i is the longest match with the concatenation of a previous factorf_j, with $j \leq d$, and the next character. The shortcoming of this heuristic is that after processing the string for a while the dictionary often becomes obsolete. If the dictionary elements were removed with a heuristic, new elements could be added.

The best deletion heuristic is the LRU (last recently used) strategy [22]. The LRU deletion heuristic removes elements from the dictionary by deleting at each step of the factorization the least recently used factor, which is not a proper prefix of another one. If the size of the dictionary is $\mathrm{O}(\log^k n)$, the LRU strategy is log-space hard for SC^k [7],[8],[12]. SC is the class of problems solvable simultaneously in polynomial time and polylogarithmic space and SC^k is the class of problems solvable simultaneously in polynomial time and $\mathrm{O}(\log^k n)$ space for a fixed k. LZW compression using a dictionary of size $\mathrm{O}(\log^k n)$ and the LRU deletion heuristic belongs to SC^{k+1}. Indeed, the LZW algorithm with LRU deletion heuristic on a dictionary of size $\mathrm{O}(\log^k n)$ can be performed in polynomial time and $\mathrm{O}(\log^k n \log(\log n))$ space, where n is the length of the input string. The trie requires $\mathrm{O}(\log^k n)$ space by using an array implementation since the number of children for each node is bounded by the alphabet cardinality. The $\log(\log n)$ factor is required to store the information needed for the LRU deletion heuristic since each node must have a different age, which is an integer value between 0 and the dictionary size.

The hardness result is not so relevant for the space complexity analysis since $\Omega(\log^k n)$ is an obvious lower bound to the work space needed for the computation. Much more interesting is what can be said about the parallel complexity analysis. In [12] it was shown that LZW compression using the LRU deletion heuristic with a dictionary of size c can be performed in parallel either in $\mathrm{O}(\log n)$ time with $2^{\mathrm{O}(c \log c)} n$ processors or in $2^{\mathrm{O}(c \log c)} \log n$ time with $\mathrm{O}(n)$ processors on a shared memory random access machine or a highly connected network. This means that if the dictionary size is constant the compression problem belongs to NC, the class of problems solvable in polylogarithmic parallel time with a polynomial number of processors on a random access shared memory machine or a highly connected network. NC and SC are classes that can be viewed in some sense symmetric and are believed to be incomparable. Since log-space reductions are in NC, the compression problem cannot belong to NC when the dictionary size is polylogarithmic if NC and SC are incomparable. We want to point out that the dictionary size c figures as an exponent in the parallel complexity of the problem. This is not by accident. If we believe that SC is not included in NC, then the SC^k-hardness of the problem when c is $\mathrm{O}(\log^k n)$ implies the exponentiation

of some increasing and diverging function of c. Indeed, without such exponentiation either in the number of processors or in the parallel running time, the problem would be SC^k-hard and in NC when c is $\mathrm{O}(\log^k n)$. Observe that the P-completeness of the problem, which requires a superpolylogarithmic value for c, does not suffice to infer this exponentiation since c can figure as a multiplicative factor of the time function. Moreover, this is a unique case so far where somehow we use hardness results to argue that practical algorithms of a certain kind (NC in this case) do not exist because of huge multiplicative constant factors occurring in their analysis. If in theory LZW compression with the LRU deletion heuristic cannot be parallelized on any kind of parallel or distributed system for the reasons explained above, in practice the locality principle allows an effective distributed implementation. Considering the fact that a practical bound to the dictionary size is 2^{16} and that about 300 kilobytes are enough to fill up a dictionary of this size on realistic data, we can argue that compressing independently blocks of at least 600 kilobytes is a sufficiently robust approach. However, the process of adding and removing dictionary elements at each step never works in a non-adaptive way. The deletion heuristic providing such method is RESTART. After the dictionary is filled up, the RESTART deletion heuristic starts a non-adaptive phase and monitors the compression ratio. When the ratio deteriorates, the heuristic deletes all the elements from the dictionary but the alphabet characters and restarts a new adaptive phase. Let $S = f_1 f_2 \cdots f_j \cdots f_i \cdots f_m$ be the factorization of the input string S computed by the LZW compression algorithm using the RESTART deletion heuristic. Let j be the highest index less than i where a restart operation happens. Then, f_j is an alphabet character and f_i is the longest match with the concatenation of a previous factor f_h, with $h \geq j$, and the next character (1 and $k+1$ are considered restart positions by default). This heuristic, called LZC [1], is used by the Unix command Compress since it has a good compression effectiveness and it is easy to implement. Since the dictionary size is 2^{16} the number of different concatenations of a factor with the next character between f_h and f_t is equal to 2^{16} decreased by the alphabet size, with h and t two consecutive positions where the restart operation happens (f_t is not counted). Usually, the dictionary performs well in a non-adaptive way on a block long enough to learn another dictionary of the same size. This is what is done by the SWAP deletion heuristic. When the other dictionary is filled, they swap their roles on the successive block. The improvement introduced by the SWAP heuristic cannot be utilized with distributed computing since the processing of the successive block cannot start until the dictionary is learned from the previous block. In conclusion, LZC compression is the version we will adopt in the next section.

3 Speeding up On-the-Fly Data Compression

We have seen in the previous section that LZC compression is a technique comprising an adaptive phase followed by a non-adaptive one which can process the data until it becomes obsolete. Since the dictionary size is 2^{16} in practical implementations and about 300 kB are enough to fill up a dictionary of this size on realistic data, another 300 kB can be considered a robust lower bound to the amount of data for which the non-adaptive way works properly [9]. Then, we divide conceptually the input into blocks of about 600 kB with each block split in a first half of about 300 kB, where the adaptive phase is run sequentially, and a second one to be compressed in a non-adaptive way with the dictionary just learned, using the computational power of the distributed system. The second half is, therefore, partitioned into sub-blocks. The

architecture of the distributed system could be modeled as a star network where the central node runs the sequential adaptive phase until the dictionary is filled. At each factorization step, the central node sends the current factor concatenated with the next character to the adjacent nodes to update their own copy of the dictionary. To speed up the broadcasting of such concatenation its longest proper prefix (that is, the current factor) can be compressed with the pointer having such prefix as target. This means that the processors receiving the factors store the dictionary in a trie using an auxiliary perfect hashing table where the pointers are the keys and the targets are the values. Then sub-blocks of the next 300 kilobytes are broadcasted to the adjacent nodes. We show how to implement the non-adaptive phase on small and medium scale systems in the next subsection. Then, we scale up the system and modify the approach to keep its robustness in the second subsection. The third subsection considers decompression. Improvements by the LZMW approach are discussed in the last subsection.

3.1 Small and Medium Scale

After having just learned the dictionary, 300 kB have to be compressed in a non-adaptive way using such dictionary. In [10], it is shown that if we distribute sub-blocks among different processors to compress them independently then the sub-block length must be the order of a few kilobytes to guarantee robustness. Ten processors and one hundred processors are the orders of magnitudes for small and medium scale distributed systems, respectively. Therefore, robustness is guaranteed. On the other hand, the size of a large scale system is the order of magnitude of a thousand implying that the sub-block length has the order of magnitude of 100 bytes. In such case, overlapping of adjacent sub-blocks and a preprocessing of the boundaries are necessary. We experimented in [10] that, when compressing megabytes of English text, the LZC average compression ratio is 0.42 while the distributed approach has a one percent loss in both cases.

3.2 Large Scale

Both overlapping of adjacent sub-blocks and a preprocessing of the boundaries are necessary since the boundary positions are arbitrary and, therefore, likely not to be at the beginning of natural factors. Consequently, a sub-block factorization starts with factors much smaller than the average factor length. This initial disadvantage is amortized if the sub-block length has the order of magnitude of a kilobyte, which is not the case of large scale distributed systems as explained in the previous subsection. Again, after having just learned the dictionary, 300 kB have to be compressed in a non-adaptive way using such dictionary. Generally speaking, sub-blocks of length $M(k+2)$, except for the first one and the last one which are $M(k+1)$ long, are broadcasted to the processors, with k a positive integer and M the maximum factor length. Each sub-block overlaps on $2M$ characters with the adjacent ones 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 either covering positions in the first and second half of the $2M$ characters shared by two adjacent sub-blocks or being a suffix of the first half. The processors execute the following algorithm to compress each sub-block:

- for each sub-block, every corresponding processor computes the longest boundary matches to the left and to the right (only to the right (left) if the sub-block is the first (last) one).
- each processor computes the greedy factorization from the end of the boundary match on the left boundary of its sub-block to the beginning of the boundary match on the right boundary.

The parallel running time of the preprocessing phase computing the boundary matches is $O(M^2)$ by brute force. In [10], it is shown experimentally that for $k = 10$ the compression ratio achieved by such factorization is about the same as the sequential one. Considering that typically the average match length is about 10, one processor can compress 100 bytes independently and this is why this approach is suitable for a large scale distributed system.

3.3 On-the-Fly Decompression

To decode on-the-fly the compressed data on a distributed system, after the sequential decompression of the first half of a block, it is enough to use during the compressing phase a special mark occurring in the sequence of pointers each time the coding of a sub-block ends in the second half of the block. A copy of the dictionary is stored in every processor since the sequential decompression of the first half is run again by the central node of the star network, sending a new dictionary element to the adjacent nodes at each step. Differently from the coding phase, a perfect hashing table rather than a trie is used to store the dictionary. Moreover, the special marks allow the broadcasting of the subsequences of pointers coding each sub-block of the second half to the adjacent nodes. Therefore, the decoding of the sub-blocks is straightforward.

3.4 Improving On-the-Fly Compression of Massive Data

The prefix dictionaries built by LZW have the disadvantage to include useless elements. To ameliorate this, the LZMW factorization builds better dictionaries by updating it at each step with the concatenation of the last two factors. Therefore, the LZMW factorization of a string S is $S = f_1 f_2 \cdots f_i \cdots f_m$ where each factor f_i is the longest match with the concatenation of two consecutive previous factors. In practical implementations, the LZMW heuristic was originally thought with a dictionary bounded by the least recently used strategy and f_i was encoded by a pointer q_i whose target is in such dictionary [21], improving even ten per cent on LZW in some cases. We propose the LZCMW heuristic in order to have non-adaptive phases, where the dictionary is bounded to 2^{16} elements by the RESTART deletion heuristic. This makes LZMW compression suitable for parallel implementations of on-the-fly compression on small and large scale distributed systems in a similar fashion as for LZC. Therefore, the amelioration provided by LZMW compression in the sequential case is kept on every scale distributed system since the loss of compression effectiveness of the RESTART deletion heuristic with respect to LRU is quite limited [1]. Running the LZCMW heuristic on different types of data could be an interesting future experimental work.

4 The Greedy versus Optimal Analysis

A *feasible* d-restarted LZW factorization $S = f_1 \cdots f_m$ is such that the number of different concatenations of a factor with the next character between f_h and f_t (f_t is

not counted) is less or equal than d decreased by the alphabet size, with h and t two consecutive positions where the restart operation happens, and each factor f_i with $h < i < t$ is equal to $f_j c$, where c is the first character of f_{j+1} and $h \leq j < i$ (1 and $k+1$ are considered restart positions by default). We define *optimal* the feasible d-restarted LZW factorization with the smallest number of factors. A practical algorithm to compute the optimal solution is not known.

A trivial upper bound to the approximation multiplicative factor of a feasible factorization with respect to the optimal one is the maximum factor length of the optimal solution, that is, the height of the trie storing the dictionary. Such upper bound is $\Theta(d)$, where d is the dictionary size ($O(d)$ follows from the feasibility of the factorization and $\Omega(d)$ from the factorization of the unary string). We give worst case examples for the procedures of section 3 using only two characters a and b. It follows that the worst case analysis is valid for any given alphabet of cardinality greater than 1. If any of the procedures described in subsections 3.1 and 3.2 is applied to the input block of length d^2

$$b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)(\prod_{i=1}^{d} a^{d/2})$$

where the dictionary is learned in the first half and employed to compress in a non-adaptive way the second one, then the factorization of the first half of the block, that is, $b^{d^2/4-d/2}(\prod_{i=0}^{d/2-1} ab^iba^i)$ is

$$b, bb, \ldots, b^\ell, b^{\ell'}, a, b, ab, ba, abb, baa, \ldots, ab^i, ba^i, \ldots, ab^{d/2-1}, ba^{d/2-1}$$

where $\ell' \leq \ell + 1$ and the dictionary is filled if we assume its size is $d + \ell + 3$. The non-adaptive factorization of the second half is $a, a, \cdots a, a$ and the total cost of the factorization of the block is $\ell + 1 + d + d^2/2$, which is $\Theta(d^2)$. On the other hand, the cost of the optimal solution on the block is $\ell + 5d/2$ which is $\Theta(d)$ since the factorization of the first half is

$$b, bb, \ldots, b^\ell, b^{\ell'}, a, b, ab, b, a, abb, b, aa, \ldots, ab^i, b, a^i, \ldots, ab^{d/2-1}, b, a^{d/2-1}$$

Observe that the $O(d)$ approximation multiplicative factor depends on the non-adaptive phase and this happens when the dictionary learned on the first half of the block performs badly on the second half, that is in practice, when the data are highly disseminated. In such case, the *totally adaptive* feasible d-restarted LZW factorization (restarting as soon as the dictionary is filled with a greedy choice at each factorization step) is much more appropriate, as shown by the following theorem (the proof employs techniques similar to the ones for the unbounded dictionary case of [11]), but unfortunately does not seem to be parallelizable.

Theorem 1. *The totally adaptive d-restarted LZW factorization is an $O(\sqrt{d})$ approximation of the optimal one, where d is the dictionary size.*

Proof. Let S be the input string and T be the trie storing the dictionary of factors of the optimal d-restarted LZW factorization Φ of S between two consecutive positions where the restart operation happens. Each dictionary element (but the alphabet characters) corresponds to the concatenation of a factor f of the optimal factorization with the first character of the next factor, that we call an *occurrence* of the dictionary

element (node of the trie) in Φ. We call an element of the dictionary built by the greedy process *internal* if its occurrence is contained in the occurrence of a node of T and denote with M_T the number of internal occurrences. The number of non-internal occurrences is less than the number of factors of Φ. Therefore, we can consider only the internal ones. An occurrence f' of the greedy factorization internal to an factor f of Φ is represented by a subpath of the path representing f in T. Let u be the endpoint at the lower level in T of this subpath (which, obviously, represents a prefix of f). Let $d(u)$ be the number of subpaths representing internal phrases with endpoint u and let $c(u)$ be the total sum of their lengths. All the occurrences of the greedy factorization are different from each other between two consecutive positions where the restart operation happens. Since two subpaths with the same endpoint and equal length represent the same factor, we have $c(u) \geq d(u)(d(u)+1)/2$. Therefore

$$1/2 \sum_{u \in T} d(u)(d(u)+1) \leq \sum_{u \in T} c(u) \leq 2n \leq 2|\Phi| H_T$$

where H_T is the height of T, $|\Phi|$ is the number of phrases of Φ and the multiplicative factor 2 is due to the fact that occurrences of dictionary elements may overlap. We denote with $|T|$ the number of nodes in T; since $M_T = \sum_{u \in T} d(u)$, we have

$$M_T^2 \leq |T| \sum_{u \in T} d(u)^2 \leq |T| \sum_{u \in T} d(u)(d(u)+1) \leq 4|T||\Phi| H_T$$

where the first inequality follows from the fact that the arithmetic mean is less than the quadratic mean. Then

$$M_T \leq \sqrt{4|T||\Phi| H_T} = |\Phi| \sqrt{\frac{4|T| H_T}{|\Pi|}} \leq 2|\Phi| \sqrt{H_T}$$

Since the trie height is $\Theta(d)$ at worst, the theorem statement follows. □

We wish to point out that the proof works for any alphabet cardinality, so the comparative analysis of the procedures of section 3 with the totally adaptive d-restarted LZW factorization is the same for every non-unary alphabet on the highly disseminated data illustrated above.

5 Conclusion

We discussed massive data compression and decompression in the context of distributed computing. The locality principle implies that scalability and robustness are not an issue when compression is applied for massive data storage. Speeding up on-the-fly compression for data transmission is more controversial and, in such case, it seems we need compression techniques merging together an adaptive and a non-adaptive approach. We proposed a practical implementation of LZW compression, called LZC, as the most suitable to our knowledge since it has this characteriistic and introduced a new version of LZC compression, employing LZMW factorization in order to improve the compression effectiveness. However, a greedy versus optimal analysis shows such approach is not suitable for highly disseminated data. As future work, different techniques could be developed or existing ones could be adapted to work in this fashion with the purpose of improving compression effectiveness further.

References

1. T. C. Bell and I. H. Witten: *Text Compression*, Prentice Hall, 1990.
2. M. Crochemore, A. Langiu, and F. Mignosii: *Note on the greedy parsing optimality for dictionary-based text compression.* Theoretical Computer Science, 525 2014, pp. 55–59.
3. M. Crochemore, G. M., A. Langiu, F. Mignosi, and A. Restivo: *Dictionary-simbolwise flexible parsing.* Journal of Discrete Algorithms, 14 2012, pp. 74–90.
4. S. DeAgostino: *P-complete problems in data compression.* Theoretical Computer Science, 127 1994, pp. 181–186.
5. S. DeAgostino: *Sub-linear algorithms and complexity issues for lossless data compression*, 1994.
6. S. DeAgostino: *Parallelism and data compression via textual substitution*, 1995.
7. S. DeAgostino: *Parallelism and dictionary-based data compression.* Information Sciences, 135 2001, pp. 43–56.
8. S. DeAgostino: *Lempel-ziv data compression on parallel and distributed systems.* Algorithms, 4 2011, pp. 183–199.
9. S. DeAgostino: *Lzw data compression on large scale and extreme distributed systems*, in Proceedings Prague Stringology Conference, 2012, pp. 18–27.
10. S. DeAgostino: *The greedy approach to dictionary-based static text compression on a distributed system.* Journal of Discrete Algorithms, 34 2015, pp. 54–61.
11. S. DeAgostino and R. Silvestri: *A worst case analisys of the lz2 compression algorithm*, 1997.
12. S. DeAgostino and R. Silvestri: *Bounded size dictionary compression:* SC^k*-completeness and* nc *algorithms.* Information and Computation, 180 2003, pp. 101–112.
13. A. Farrugia, P. Ferragina, A. Frangioni, and R. Venturini: *Bicriteria data compression*, in Proceedings SIAM-ACM Symposium on Discrete Algorithms (SODA 14), 2014, pp. 1582–1585.
14. P. Ferragina, I. Nittoi, and R. Venturini: *On optimally partitioning a text to improve its compression.* Algorithmica, 61 2011.
15. P. Ferragina, I. Nittoi, and R. Venturini: *On the bit-complexity of lempel-ziv compression.* SIAM Journal on Computing, 42 2013.
16. J. Gailly and M. Adler: *http://www.gzip.org*, 1991.
17. A. Langiu: *On parsing optimality for dictionary-based text compression - the zip case.* Journal of Discrete Algorithms, 20 2013, pp. 65–70.
18. A. Lempel and J. Ziv: *On the complexity of finite sequences.* IEEE Transactions on Information Theory, 22 1976, pp. 75–81.
19. A. Lempel and J. Ziv: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23 1977, pp. 337–343.
20. Y. Matias and C. S. Sahinalp: *On the optimality of parsing in dynamic dictionary-based data compression*, in Proceedings SIAM-ACM Symposium on Discrete Algorithms (SODA 99), 1999, pp. 943–944.
21. V. S. Miller and M. N. Wegman: *Variations on theme by ziv-lempel*, 1985.
22. J. A. Storer: *Data Compression: Methods and Theory*, Computer Science Press, 1988.
23. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* Journal of ACM, 29 1982, pp. 928–951.
24. T. A. Welch: *A technique for high-performance data compression.* IEEE Computer, 17 1984, pp. 8–19.
25. D. Whiting, G. A. George, and G. E. Ivey: *Data compression apparatus and method*, 1991.
26. G. K. Zipf: *The Ppsycology of Language*, Houghton-Mifflin, 1935.
27. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Transactions on Information Theory, 24 1978, pp. 530–536.