A Review: Lempel-Ziv (LZ'78) Coding

Harman Jot, Mrs. Rupinder Kaur

Abstract— **Hashing algorithm named two-level hashing that enables fast longest match searching from a sliding dictionary, and the other uses suffix sorting. The former was suitable for small dictionaries and it significantly improves the speed of gzip, which uses a naïve hashing algorithm.**

A similar notion of "finite-state encryptability" of an individual plain-text sequence was defined, as the minimum asymptotic key rate that must be consumed by finite-state encrypters so as to guarantee perfect secrecy in a well-defined sense. Our main basic result was that the finite-state encryptability is equal to the finite-state compressibility for every individual sequence. This is in parallelism to Shannon's classical probabilistic counterpart result, asserting that the minimum required key rate is equal to the entropy rate of the source.

Gipfeli a high-speed compression algorithm that uses backward references with a 16-bit sliding window, based on 1977 paper by Lempel and Ziv, enriched with an ad-hoc entropy coding for both literals and backward references.

I. INTRODUCTION

Over recent years improvements in memory bandwidth have lagged behind advancesin CPU performance. This puts other parts of the system under pressure and often makes I/O operations a bottleneck. Gipfeli1 is a new high-speed compression algorithm which tries to address this issue by trading CPU performance used in data compression for improvements in I/O throughput by reducing the amount of data transferred. Its goal is to decrease both running-time and memory usage. An overview of throughput of different I/O components of a computer.

There are several other high-speed compression algorithms. The main competitors are Snappy [1] (currently being the key part of the Google infrastructure and sometimes referred to also as Zippy), QuickLZ [2] and FastLZ [3]. The goal of this work was to have the algorithm with the best compression ratio in this category. Gipfeli uses backward references, as introduced by Lempel and Ziv in their famous 1977 paper [4], with very light-weight hashing and a 16-bit sliding window. This has already been suggested in the previous work of Williams [5], Fiala and Greene. The algorithm consists of two main parts. The first part, LZ77, builds on the implementation in Snappy with several improvements. The main difference was the second part, which was rare in the area of high-speed compression algorithms: it is an application of entropy coding. Gipfeli uses a static entropy code for backward references and an ad-hoc entropy code for literals based on sampling the input. Sampling is necessary, because there is not enough time and memory to read the whole input in order to gather all the statistics and build a conversion table. They also could not use Huffman codes or arithmetic coding, because of their slow performance.

The paradigm of individual sequences and finite-state machines (FSMs), as an alternative to the traditional probabilistic modeling of sources and channels, has been studied and explored quite extensively in several information-theoretic problem areas, including data compression, source/channel simulation [6], classification [7], [8], prediction denoising, and even channel coding [8], [9], just to name very few representative references out of many more. On the other hand, it is fairly safe to say that the entire literature on information-theoretic security, starting from Shannon's classical work [8] and ending with some of the most recent work in this problem area for surveys as well as references therein), is based exclusively on the probabilistic setting

To the best of knowledge, the only exception to this rule is an unpublished memorandum by Ziv [5]. In that work, the plain-text source to be encrypted, using a secret key, is an individual sequence, the encrypter is a general block encoder, and the eavesdropper employs an FSM as a message discriminator. Specifically, it is postulated in[1] that the eavesdropper may have some prior knowledge about the plain text that can be expressed in terms of the existence of some set of "acceptable messages" that constitutes the *a priori* level of uncertainty (or equivocation) that the eavesdropper has concerning the plaintext message: The larger the acceptance set, the larger the uncertainty. Next, it was assumed that there exists an FSM that can test whether a given candidate plain-text message is acceptable or not: If and only if the FSM produces the all-zero sequence in response to that message, then this message is acceptable.

Perfect security was then defined as a situation where the size of the acceptance set is not reduced (and hence neither is the uncertainty) in the presence of the cryptogram. The main result in [10] is that the asymptotic key rate needed for perfectly secure encryption in that sense cannot be smaller (up to asymptoticallyvanishing terms) than the Lempel–Ziv (LZ) complexity of the plain-text source [11]. The lower bound was obviously asymptotically achieved by one-time pad encryption of the bit stream obtained by LZ data compression of the plain-text source. This was in parallelism to Shannon's classical probabilistic counterpart result, asserting that the minimum required key rate is equal to the entropy rate of the source.

Many data compression schemes have been developed, and they were selected according to their compression speed, decompression speed, compression performance, memory requirements, etc. The LZ77 compression scheme [12] is a lossless compression scheme. Now it becomes a basis of

Manuscript received July 12, 2015

Harman Jot, M.Tech, Department of Electronics and Communication, Punjabi University, Patiala, Punjab, India

Mrs. Rupinder Kaur, Department of Electronics and Communication, Punjabi University,Patiala, Punjab, India

many compression schemes. Its de-compression speed is very fast and the memory required is small.

The LZ77 scheme compresses a string from left to right. It first finds a prefix of a string to be encoded from the string already encoded called dictionary. Then the prefix is encoded by its length and the distance between it and the string in the dictionary. The size of the dictionary is usually limited because of memory and compression time limitations, and therefore the dictionary stores only the newer part of the string. This type of dictionary is called a sliding dictionary. To compress a string well, we have to find the longest match string in the dictionary. It is also important to find the nearest one among the longest match strings because the nearest one is encoded in fewer bits. The most time-consuming task in the LZ77 compression is to find the longest match strings. Hence the main topic of this paper is to find them quickly. Though the LZ77 has significant features described above, it is difficult to implement a fast encoder in practice. The LZ77 compression using the sliding dictionary can be done in linear time [13][14]. However, the algorithm requires huge memory and it is not fast in practice. Another problem is that it cannot find the nearest string in the dictionary. This causes compression loss.

The problems can be solved in part by using hashing algorithms. Almost all programs using the LZ77 scheme, for example gzip, Info-ZIP, PKZIP, lha and arj, use hashing data structures because of practical speed and memory efficiency. Among them, gzip [15] is a typical and commonly used implementation of the LZ77 scheme. Though the hashing algorithms are fast enough for many strings, they become extremely slow for some strings. This is a reason to consider new algorithms for the LZ77.

II. WORK DONE

The benchmarks in Gipfeli can achieve compression ratios that are 30% better than Snappy with slow-down being only around 30%. Gipfeli achieves even higher speed for html content and remote procedure calls than for text content. they argue, using the I/O data from the introduction, that once the compression algorithm is fast enough, i.e. computations are bound by external I/O costs, they needed to compress as densely as possible. At that point, improvements in the compression ratio are more important than running time. The encouraging outcome of the benchmarks led us to test Gipfeli in a real-world setting to support our theoretical assumptions with practical experiment. Our case study was MapReduce technology [16] used inside Google to run distributed computations. A typical computation processes terabytes of data and runs on thousands of machines. In short, MapReduce consists of two phases: the first phase, Map, applies some computation in parallel to all the input items to produce the intermediate items, which are then merged in the second phase, Reduce. Currently, Snappy and Zlib are both options used inside Map Reduce. They replaced Snappy by Gipfeli and our experiment confirmed their expectations: the computation was faster (up to 10%) and led to lower RAM usage. Since Gipfeli can compress better, it was now a candidate replacement for both Snappy and Zlib (which is currently used in the situations where a better compression ratio is needed) in MapReduce. Other plausible applications are: the replacement of Snappy in Big table technology [17], which is used to store large amounts of data; and in Google's internal remote procedure calls. Currently, Snappy and Zlib are both options used inside MapReduce. We replaced Snappy by Gipfeli and our experiment confirmed their expectations: the commutationwas faster (up to 10%) and led to lower RAM usage. Since Gipfeli can compress better, it is now a candidate replacement for both Snappy and Zlib (which is currently used in the situations where a better compression ratio is needed) in Map Reduce. Other plausible applications are: the replacement of Snappy in Big table technology, which is used to store large amounts of data; and in Google's internal remote procedure calls.

Encryption of individual sequences, but our modeling approach and the definition of perfect secrecy are substantially different. Rather than assuming that the encrypter and decrypter have unlimited resources, and that it was the eavesdropper which has limited resources, modeled in terms of FSMs, in our setting, the converse is true. They adopt a model of a finite-state encrypter, which receives as inputs the plain-text stream and the secret key bit stream, and it produces a cipher text, while the internal state variable of the FSM, that designates limited memory of the past plain-text, is evolving in response to the plain-text input. Based on this model, they defined a notion of *finite-state encryptability* (in analogy to the notions of finite- state compressibility and the finite-state predictability), as the minimum achievable rate at which key bits must be consumed by any finite-state encrypter in order to guarantee perfect security against an unauthorized party, while keeping the cryptogram decipherable at the legitimate receiver, which has access to the key. Their main result is that the finite-state encryptability is equal to the finite-state compressibility,

A technique called lazy evaluation to improve compression ratio, encoded a prefix of suffix $S_p$as a literal x_p if $l1 < l2$. This technique is used in gzipand many other programs, and its effect was analyzed in. It improves compression ratio by about 0.05 bits/character for both optlz77 and lz77 with various dictionary sizes in our experiments. The adaptive arithmetic code encodes characters and match lengths according to their frequencies which were updated each time a character or a length is en- coded. We call this program optlz77 where opt means that this program finds the closest longest matches. Its performance will be close to the best program using the LZ77 scheme. They also use a program named lz77. Their difference is that lz77 may not find the closest one among the longest match strings in the longest matchfunction. The function returns the first matched string among strings that match p. Therefore it may not find the closest longest match string and the compression ratio will decrease. Because the optlz77 and the lz77 use the lazy evaluation, their compression speed is a little slower than the lz77 (sort+longest match) and the lz77 (sort only) in respectively. The compression ratio of optlz77 is better than that of lz77, which implies that finding the nearest longest match is important even if we use large dictionaries. The compression ratio of gzip is better than optlz77 with a dictionary of 32Kbytes because encoding of distances in gzip is optimized for the 32Kbytes window.

The compression ratio of bzip2 decreases faster than that of optlz77 and lz77 although the sliding window LZ77 is asymptotically optimal for all finite-alphabet stationary ergodic sources [16]. Therefore the block sorting has better compression ratio than the LZ77 for blocks of moderate sizes.

The compression ratio of optlz77 is better than bzip2 if dictionary size is less than 128Kbytes. In this case, optlz77 is better than bzip2 in both compression speed and ratio and the two-level hashing algorithm significantly improves compression speed.

CONCLUSION

Gipfeli is currently in the alpha phase. The algorithm has not gone through complete testing, but a few terabytes of data have been successfully pushed through it. Compared to Snappy, Quick[1] LZ and Fast LZ, which have been tuned for several years, Gipfeli is still young. They have open sourced it to allow additional improvements and optimizations by the community.

Asymptotically achieved performance by block codes as follows. Given any vector x^n, divide it into blocks and calculate the corresponding [10]empirical distribution. Let X^m be a random variable governed by this empirical distribution.

By increasing the speed of an LZ77-type data compression scheme. In the LZ77 compression, finding the longest match string was the most time consuming process. Though the widely used gzip also uses the LZ77, its compression algorithm was not optimized. Thereforethe compression speed of gzip needs to be improved, not only for practical purposes but also as it is the touchstone of compression algorithms. They proposed two algorithms for finding the longest match string. One uses two-level hash and the other suffix sorting [7].

REFERENCES

[1] R. Lenhardt and J. Alakuijala, "Gipfeli-high speed compression algorithm," in Data Compression Conference (DCC), 2012, 2012.

[2] S. Borkar, P. Dubey, K. Kahn, D. Kuck, H. Mulder, S. Pawlowski and J. Rattner, "Platform 2015: Intel processor and platform evolution for the next decade," Technology, p. 1, 2005.

[3] A. Lempel, "Cryptology in transition," ACM Computing Surveys (CSUR), vol. 11, no. 4, pp. 285-303, 1979.

[4] J. C. Kieffer and E.-h. Yang, "Sequential codes, lossless compression of individual sequences, and Kolmogorov complexity," Information Theory, IEEE Transactions on, vol. 42, no. 1, pp. 29-39, 1996.

[5] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," IEEE Transactions on information theory, vol. 23, no. 3, pp. 337-343, 1977.

[6] T. Bell and D. Kulp, "Longest-match string searching for ziv-lempel compression," Software: Practice and Experience, vol. 23, no. 7, pp. 757-771, 1993.

[7] Y. Liang, H. V. Poor and others, "Information theoretic security," Foundations and Trends in Communications and Information Theory, vol. 5, no. 4--5, pp. 355-580, 2009.

[8] Y. Lomnitz and M. Feder, "Universal communication over modulo-additive channels with an individual noise sequence," 2010.

[9] A. Martin, N. Merhav, G. Seroussi and M. J. Weinberger, "Twice-universal simulation of Markov sources and individual sequences," Information Theory, IEEE Transactions on, vol. 56, no. 9, pp. 4245-4255, 2010.

[10] N. Merhav, "Perfectly secure encryption of individual sequences," Information Theory, IEEE Transactions on, vol. 59, no. 3, pp. 1302-1310, 2013.

[11] Y. Lomnitz and M. Feder, "Communication over Individual Channels--a general framework," arXiv preprint arXiv:1203.1406, 2012.

[12] P. K. Pearson, "Fast hashing of variable-length text strings," Communications of the ACM, vol. 33, no. 6, pp. 677-680, 1990.

[13] A. D. Wyner and J. Ziv, "The sliding-window Lempel-Ziv algorithm is asymptotically optimal," Proceedings of the IEEE, vol. 82, no. 6, pp. 872-877, 1994.

[14] U. Manber and G. Myers, "Suffix arrays: a new method for on-line string searches," siam Journal on Computing, vol. 22, no. 5, pp. 935-948, 1993.

[15] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," 1994.

[16] N. J. Larsson, "Extended application of suffix trees to data compression," in Data Compression Conference, 1996. DCC'96. Proceedings, 1996.

[17] N. J. Larsson and K. Sadakane, Faster suffix sorting, Citeseer, 1999.