Evolutionary Lossless Compression with GP-ZIP

Ahmad Kattan and Riccardo Poli

Abstract— **In this paper we propose a new approach for applying Genetic Programming to lossless data compression based on combining well-known lossless compression algorithms. The file to be compressed is divided into chunks of a predefined length, and GP is asked to find the best possible compression algorithm for each chunk in such a way to minimise the total length of the compressed file. This technique is referred to as "*GP-zip*". The compression algorithms available to GP-zip (its function set) are: Arithmetic coding (AC), Lempel-Ziv-Welch (LZW), Unbounded Prediction by Partial Matching (PPMD), Run Length Encoding (RLE), and Boolean Minimization. In addition, two transformation techniques are available: Burrows-Wheeler Transformation (BWT) and Move to Front (MTF). In experimentation with this technique, we show that when the file to be compressed is composed of heterogeneous data fragments (as is the case, for example, in archive files), GP-zip is capable of achieving compression ratios that are superior to those obtained with well-known compression algorithms.**

I. Introduction

One of the paradoxes of technology evolution is that despite the development of computers and the increasing need for storing information, there is a lack of development of compression techniques. As the evolution of computer systems progresses rapidly, the amount of stored information will increase at the same rate. Researchers in the compression field tend to develop algorithms that work with specific types of data, taking advantage of any available knowledge about the data. It is difficult to find a universal compression algorithm that performs well on any data type [1]. Two principles are commonly accepted in the field of data compression: *a)* there is no algorithm that is able to compress all the files even by 1 byte, and *b)* there are less than 1% of all files that can be compressed losslessly by 1 byte [1] .

Consequently, the development of generic compression algorithms is attracting less attention. Nevertheless their importance should not be underestimated, as they are useful when the nature and regularities of a given data file is not predictable. For example, in archive systems, the users need to compress huge amounts of different data, such as text, music, pictures, video and so forth. A single universal compression model would be preferable in this case.

We have mentioned that best compression can be achieved when applying a compression algorithm specialised for the type of data that need compressing. In this paper we investigate the idea of evolving programs that attempt to identify what is the best way of applying different compression algorithms to different parts of a data file so as to best match the nature of such data.

The structure of this paper is as follows. In Section II some related work is briefly reviewed. Section III discusses our system, GP-zip, in detail. This is followed by experimental results with GP-zip in Section IV. Section V proposes a further improvement to GP-zip which significantly improves compression ratios. Finally, conclusive marks are given in Section VI.

Ahmad Kattan is with the Department of Computing and Electronic Systems, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, UK; (email: akatta@essex.ac.uk).

Riccardo Poli is with the Department of Computing and Electronic Systems, University of Essex, Wivenhoe Park, Colchester CO4 3SQ, UK; (email: rpoli@essex.ac.uk).

II. Related work

Data compression requires highly specialized procedures. Therefore, evolving a data compression algorithm is not an easy task. Few attempts have been made to use of evolutionary algorithms to evolve data compression models.

There are two main approaches for the use of evolutionary computations in data compression. The first is the use of genetic algorithms to find parameters for a compression algorithm with the aim of maximising the compression ratio. The second approach is the use of genetic programming (GP) for what is called *programmatic compression* [3]. Of the two, the latter is much more powerful, at least in principle, as was demonstrated by Nordin and Banzhaf who used GP to achieve lossy compression for images and sounds [3]. Others have also used genetic programming in developing compression models. For example, [4] used genetic programming for string compression. Fukunaga and Stechert [5] developed a nonlinear predictive model to compress grey scale images. Parent and Nowe [2] used genetic programming to reduce the entropy of the data. They evolved transformation programs in order to maximize compression ratios when applying the transformed data to lossless compression algorithms.

III. GP-zip

Over the past decade many techniques have been developed for lossless compression, each of which has their own particular advantages and disadvantages. Each performs well on the data domain that they are designed to work in. Unfortunately, no single compression algorithm exists that is reliable for all known data types. Here we investigate how to evolve a program that matches different parts of a data file with the best possible compression model for them. Our approach works as follows.

We divide the given data file into chunks of a certain length and ask GP to identify the best possible compression technique for each chunk. The function set of GP-zip is composed of primitives that naturally fall into two categories. The first category contains the following five

compression algorithms: Arithmetic Coding (AC) [6], Lempel-Ziv-Welch LZW [7], unbounded Prediction by Partial Matching (PPMD) [8], Run Length Encoding (RLE) [9], and Boolean Minimization [10]. In the second category, two transformation techniques are included: Burrows-Wheeler Transformation (BWT) [11] and Move to Front (MTF) [12]. Since these are all very well-known techniques, we will not provide a detailed explanation of each of these compression and transformation algorithms here.

We treat each member in the function set as a black box. Each compression function receives a stream of data as inputs and returns a (typically) smaller stream of compressed data as an output. Each transformation function receives a stream of data as input and returns a transformed stream of data as an output. So, this does not directly produce a compression. However often the transformed data are more compressible, and so, when passed to a compression algorithm in the function set, a better compression ratio is achieved

In the following two subsections we provide further details on the system.

A. Chunking

The basic idea behind dividing files into chunks is the concept of "divide and conquer". Each member function in the function set performs well when it works in the circumstances that it has been designed for. Dividing the given data into smaller chunks makes the creation and identification of such circumstances easier.

The length of the possible chunks starts from 1600 bytes and increases up to 1 Mega byte in increments of 1600 bytes. Hence, the set of possible lengths for the chunks is {1600, 3200, 4800, 6400….1MB}. The number of chunks is calculated, by dividing the file size by the chunk length. Note that the chunks are not allowed to be bigger than the size of the file itself. Moreover, the size of the file is added to the set of possible chunk lengths. This is to give GP-zip the freedom to choose whether to divide the file into smaller chunks as opposed to compressing the whole file as one single block.

B. The approach

As explained above, the function set is composed of five compression algorithms and two transformation algorithms. It is clear that we have 15 different possible ways of compressing each chunk. Namely, we can apply one of the five compression functions without any transformation of the data in a block, or we can precede the application of the compression function by one of two transformation functions.

GP-zip randomly selects a chunk length from the set of possible lengths, and then applies relatively simple evolutionary operations. The system starts by initializing a population randomly. As exemplified in Figure 1, individuals represent a sequence of compression functions with or without transformation functions. High-fitness individuals are selected with a specific probability and are manipulated by crossover, mutation and reproduction operations.

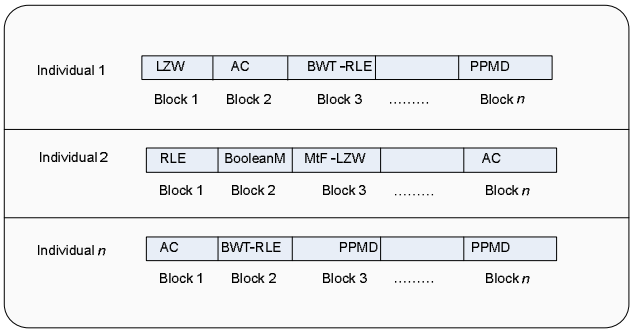

Fig 1: Individuals within a population

After the system finds the best possible compression for the selected chunk-length, the system suggests another length for the chunks and the same process is iterated.

Since testing all of the possible chunk lengths is very time consuming, GP-zip selects the new lengths by performing a form of binary search over the set of possible lengths. This is applied for the first time after the system has tested two chunk lengths. The third chunk length is chosen where, based on the results obtained with the first two chunk lengths, we would expect to obtain more promising results.

Since the proposed system divides the data into chunks and finds the best possible compression model for them, it is necessary for the decompression process to know which chunk was compressed with which compression and transformation function. A header for the compressed files has been designed which provides this information for the decompression process. The size of this header is not fixed. However, there is an insignificant overhead in comparison with size of the original (uncompressed) file. It should be noted that the advantages of dividing the data into smaller chunks manifest themselves in the decompression stage. Firstly, the decompression process can easily decompress a section of the data without processing the entire file. Furthermore, the decompression process is faster than the compression since, in principle, GP-zip can send each chunk that needs decompressing to the operating system pipeline sequentially.

IV. Experiments

Experiments have been conducted in order to investigate the performance of the proposed technique. The aim of these experiments is to measure the performance of GP-zip with different data types. The experiments covered three sets of data: i) an archive of English text files, ii) an archive of executable files, and iii) an archive file that include PDF, MP3, Excel sheet, and text files. The total sizes of the data sets are 4.07MB, 4.07MB and 1.43MB respectively. Both text files and executable files are available in [13]. There is no terminating condition for GP-zip. Therefore, GP-zip runs until it reaches the maximum number of generations.

The experiments that are presented here were done using:

- Population of size 500.
- Maximum number of generations 1000.
- One point crossover with probability of 75%.
- Mutation with probability 5%.

- Reproduction with probability of 20%
- Tournament selection with tournament size 2.

After applying GP-zip to the set of English text files and the set of executable files, the system always converged to solutions where each file is treated as a single contiguous chunk. Figures 2 and 3 show how the compression ratios increase as the number of chunks in these files decrease. This is not too surprising since both executable files and plain text, statistically, have a very regular structure.

However, when applying GP-zip to our third data set (an archive containing files of different types), the system found that dividing the file into a certain number of blocks provided best performance. (Figure 4 shows how the compression ratio varies with the number of chunks.) Naturally, this had to be expected as it is difficult to process the entire data set effectively using only one algorithm.

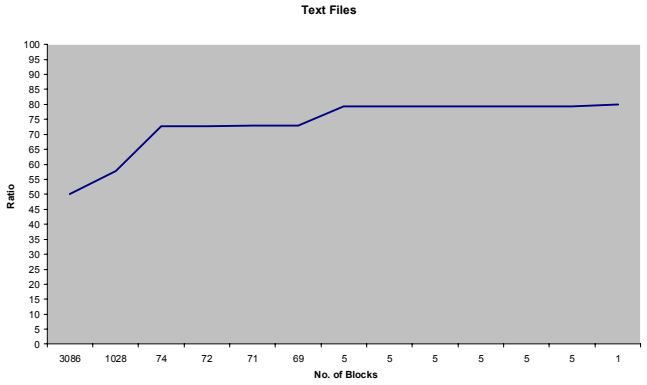

Fig 2: Number of block vs. compression Ratio for the text files.

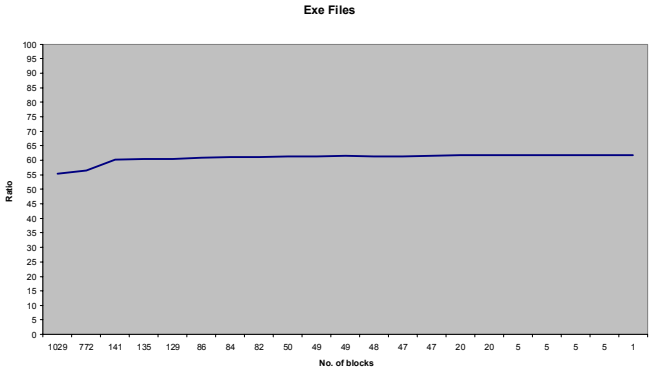

Fig 3: Number of block vs. compression Ratio for the executable files.

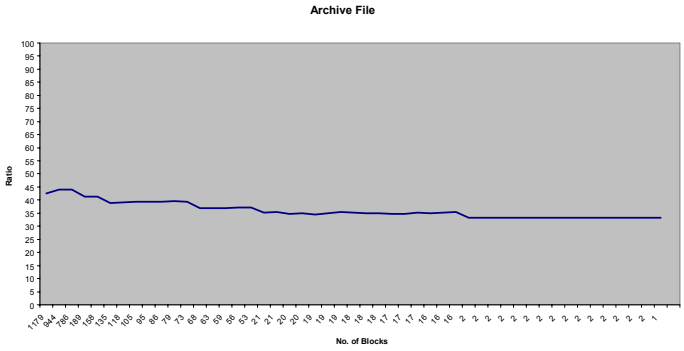

Fig 4: Number of block vs. compression Ratio for the archive file

To evaluate the relative benefits of GP-zip in comparison to other widely used techniques, we compared the performance of GP-zip against the compression algorithms in the function set. Furthermore, bzip2 and WinRar, which are amongst the most popular compression algorithms in regular use, were included in the comparison.

The results of the comparison are reported in Table I. In the first two data sets, the system decided to process the data as one big single block. Consequently, their compression ratio is based on the performance of one compression function in the function set (in fact, PPMD). So, here GP-zip cannot outperform existing algorithms. It did choose, however, a very good algorithm to do the compression, thereby resulting second best in both categories.

In the heterogeneous data set, however, GP-zip beats all other algorithms by a very considerable margin. More precisely GP-zip provides an improvement of compression ratio of 10% or more over all others. This shows that there is great potential in an evolutionary approach to data compression.

TABLE I
PERFORMANCE COMPARISON

Compression \ Files	Exe	Text	Archive
bzip2	57.86%	77.88%	32.9%
WinRar- Best	64.68%	81.42%	34.03%
PPMD	61.84%	79.95%	33.32%
Boolean Minimization	11.42%	24.24%	3.78%
LZW	35.75%	56.47%	1.13%
RLE	-4.66%	-11.33%	-10.20%
AC	17.46%	37.77%	9.98%
GP- zip	61.84%	79.95%	43.95%

Although the proposed technique has achieved substantial compression ratio with heterogeneous files in comparison with the other techniques, it suffers from one major disadvantage. Running GP-zip is very time consuming (of

the order of a day per megabyte). The reason for this is that, each time GP-zip suggests a new chunk length, it executes a new GP run which consumes considerably time by itself. Therefore, most of time is spent searching for the best possible length for the blocks, rather than for choosing how to compress the data.

V. GLUING

In our experiments we saw that when compressing a sequence of continuous data of the same kind, a single model is better than dividing the sequence into smaller pieces. Why? At one level the answer is intuitive: since the data are all of the same kind, GP-zip just decided to use the best compression technique for those data. However, in principle, one might argue, GP-zip could have decided to use the same algorithm to compress every chunk of a data file thereby achieving the same result even when a file is divided up into many smaller elements. The fallacy of this argument resides in the fact, that most compression techniques require some header information to be stored (for future decompression) before the compression of the actual data starts. So, applying say PPMD to the *N* chunks of a file is more expensive (in the sense that it gives a lower compression ratio) than applying PPMD on the whole file.

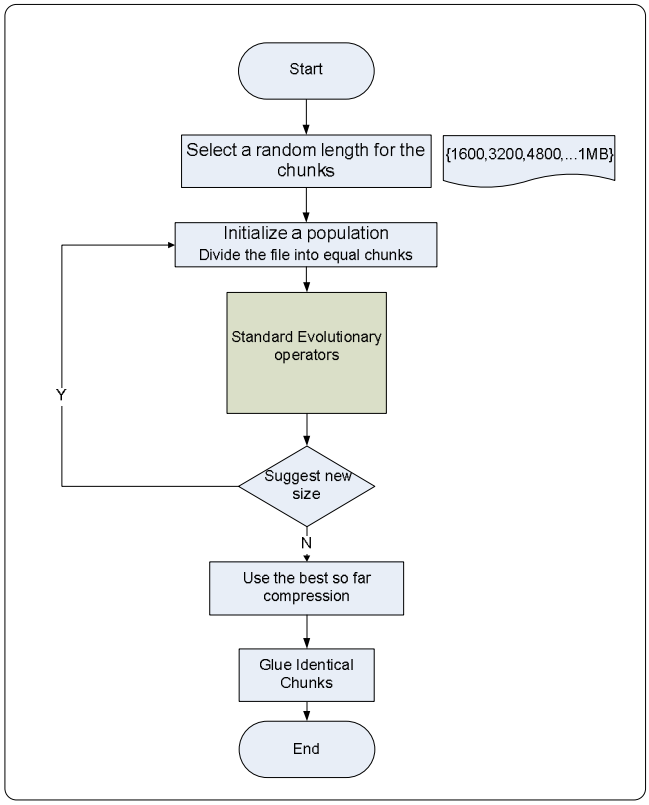

Fig 5: GP-zip flowchart

While this effect is particularly evident on homogeneous data, to a smaller scale this can manifest itself also when a data file is heterogeneous. It is possible, and in fact likely, that GP-zip chromosomes will contain repeated sequences of identical primitives, e.g.,

….[PPMD][PPMD] [LZW][LZW][LZW]…

If evolution is working well, these repeated sequences will typically indicate that the data in the corresponding chunks are of similar (or at least compatible) type. This suggests that even better compression might be achieved by avoiding repeating the header information associated to PPMD or LZW in this example.

Therefore, after GP-zip finds the best length for dividing up a given file into chunks, we try to glue together any continuous sequence of chunks that use the same functions. Gluing is the process of joining subsequence chunks to form bigger blocks, with the intention of processing them as one unit. Once we glue the chunks which are associated to the same compression and transformation functions, we have fewer chunks, which, on average, are of bigger size, leading to better compression. It should be noted that the gluing process happens only once after GP-zip has determined the length for chunks. Figure 5 illustrates GP-zip's flowchart.

Table II compares the performance of GP-zip on the heterogeneous data files with and without gluing. It is clear that gluing further improves the lead of GP-zip over other compression algorithms. Gluing the chunks not only improves the compression ratio, it also decreases the quantity of information stored within the file header.

TABLE II
GP-ZIP (GLUING)

Compression/Files	Archive
bzip2	32.9%
WinRar- Best	34.03%
PPMD	33.32%
Boolean Minimization	3.78%
LZW	1.13%
RLE	-10.20%
AC	9.98%
GP- zip	43.95%
GP- zip (Glue)	49.49%

VI. CONCLUSION AND FUTURE WORK

In the field of lossless compression it is very difficult to significantly improve on the already excellent performance provided by current algorithms. In this research we wanted to understand the benefits and limitation of combining existing lossless compression algorithms in a way that ensure the best possible match between the algorithm being used and the type of data it is applied to.

While other compression algorithms attempt to used different techniques on the file to be compressed, and then settle for the one that provides the best performance, the system we have developed, GP-zip, goes further. It divides the data file into smaller chunks and attempts to identify what combination of compression algorithms provides the best performance.

Despite the simplicity of our current realization of this idea, the results obtained by GP-zip have been remarkable, with GP-zip significantly outperforming other compression algorithms on heterogeneous files and never being too far from the best with other types of data. Furthermore, a significant improvement has been achieved when gluing sequences of identical chunks. In future research, we will concentrate on this particular aspect of GP-zip as further substantial improvements can be expected.

In addition to providing better compression (in some cases), the division of data files into chunks presents the additional advantage that, in the decompression process, one can decompress a section of the data without processing the entire file. This is particularly useful, for example, if the data are decompressed for streaming purposes (such as music and video files). Also, the decompression process is faster than the compression one, as GP-zip can send each compressed chunk to the operating system pipeline sequentially.

Although the proposed technique has achieved substantial improvements in compression ratios for heterogeneous files in comparison with other techniques, it suffers from one major disadvantage. The process of GP-zip is computationally expensive. In future research we will also concentrate on this aspect of GP-zip.

Our results are encouraging, in the sense that significant improvements have been achieved. There are many directions where we can further improve the performance of GP-zip. These range, for example, from the simple extension of the set of compression and transformation functions available in the primitive set to the open ended evolution of the compression algorithm to be performed every time a file is accessed.

We will explore these avenues in future research.

REFERENCES

[1] I. M. Pu, *Fundamental Data Compression*, HB, ISBN-13: 978-0-7506-6310-62006. Chapter 1

[2] J. Parent and A. Nowe, *Evolving Compression Preprocessors with Genetic Programming,* GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pp. 861-867, Morgan Kaufmann Publishers, 9-13 July 2002.

[3] P. Nordin and W. Banzhaf, *Programmatic Compression of Images and Sound,* Genetic Programming 1996: Proceedings of the First Annual Conference, pp. 345-350, MIT Press, 28-31 July 1996.

[4] I. De Falco and A. Iazzetta and E. Tarantino and A. Della Cioppa and G. Trautteur, *A Kolmogorov Complexity-based Genetic Programming tool for string compression,* Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2000), pp. 427-434, Morgan Kaufmann, 10-12 July 2000.

[5] A. Fukunaga and A. Stechert, *Evolving Nonlinear Predictive Models for Lossless Image Compression with Genetic Programming,* Genetic Programming 1998: Proceedings of the Third Annual Conference, pp. 95-102, Morgan Kaufmann, 22-25 July 1998.

[6] I. Witten and R. Neal and J. Cleary, *Arithmetic coding for data compression,* Communications of the ACM, Vol. 30, pp. 520-541, 1987.

[7] J. Ziv and A. Lempel, *Compression of Individual Sequences via Variable-Rate Coding*, IEEE Transactions on Information Theory, September 1978.

[8] J. G. Cleary and W. J. Teahan and Ian H. Witten, *Unbounded Length Contexts for PPM,* Data Compression Conference, pp. 52-61, 1995.

[9] S. W. Golomb, *Run-length encodings,* IEEE Trans. Inform. Theory, Vol. IT-12, pp. 399-401, 1966.

[10] A. Kattan, *Universal Lossless Data Compression with built in Encryption*. Master Thesis, University of Essex 2006.

[11] M. Burrows and D. J. Wheeler, *A block-sorting lossless data compression algorithm*, SRC, Number 124, 1994.

[12] Z. Arnavut, *Move-to-Front and Inversion Coding,* DCC: Data Compression Conference, IEEE Computer Society TCC, 2000.

[13] ACT Archive Compression Test [cited 2 December 2007]; Available from: http://compression.ca/act/act-win.html