Evolutionary Lossless Compression with GP-ZIP*

Ahmad Kattan
University of Essex
Dep. Computing and Electronic Systems
Colchester, CO4 3SQ
United Kingdom
akatta@essex.ac.uk

Riccardo Poli
University of Essex
Dep. Computing and Electronic Systems
Colchester, CO4 3SQ
United Kingdom
rpoli@essex.ac.uk

ABSTRACT
In recent research we proposed *GP-zip*, a system which uses evolution to find optimal ways to combine standard compression algorithms for the purpose of maximally losslessly compressing files and archives. The system divides files into blocks of predefined length. It then uses a linear, fixed-length representation where each primitive indicates what compression algorithm to use for a specific data block. GP-zip worked well with heterogonous data sets, providing significant improvements in compression ratio compared to some of the best standard compression algorithms. In this paper we propose a substantial improvement, called *GP-zip**, which uses a new representation and intelligent crossover and mutation operators such that blocks of different sizes can be evolved. Like GP-zip, GP-zip* finds what the best compression technique to use for each block is. The compression algorithms available in the primitive set of GP-zip* 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: the Burrows-Wheeler Transformation (BWT) and Move to Front (MTF). Results show that GP-zip* provides improvements in compression ratio ranging from a fraction to several tens of percent over its predecessor.

Categories and Subject Descriptors
I.2.2 [**ARTIFICIAL INTELLIGENCE**]: Automatic Programming

General Terms
Algorithms, Performance, Reliability.

Keywords
Lossless data compression, GP-zip, AC, LZW, PPMD, RLE, Boolean Minimization. BWT, MTF, GP-zip*.

1. INTRODUCTION
Over the past decade or two many techniques have been developed for data compression, each of which has their own particular advantages and disadvantages.

One of the factors that help in developing an efficient data compression model is the availability of prior knowledge on the domain of the data to be compressed. In these circumstances, compression researchers can develop specialised compression techniques which perform well on a specific type of data. However, it is difficult to find a universal compression algorithm that performs well on any data type [1]. Two principles are common among compression algorithms; *a)* There is no algorithm that is able to compress all the files even by 1 byte; *b)* There are less than 1% of all files that can be compressed losslessly by 1 byte [1]. Also there is a sort of no-free lunch result for lossless compression, by which, unfortunately, no single compression algorithm is guaranteed never to increase the size of a file.

For these reasons, the development of generic compression algorithms has lost momentum in favour of specialised ones. Nevertheless the importance of the former is considerable, as they are valuable when information on the data to be compressed is not available or when the data is composed of fragments of heterogeneous types. For example, there is today more and more use of archive files (e.g., ZIP files or TAR files), where users store large quantities of files often including diverse combinations of text, music, pictures, video, executables, and so forth. These files still require generic compression algorithms.

An ideal compression system would be one that is able to identify incompatible data fragments (both the file level and within each file) in an archive, and to allocate the best possible compression model for each, in such a way to minimise the total size of the compressed version of the archive. For large and varied datasets, for example, web sites, this would provide enormous advantages in terms of compression ratios. Solving the problem optimally, however, would be enormously complex.

In recent research [2] we started to make some progress on turning this idea into practice. In particular, we proposed a general purpose compression method, *GP-zip*, which uses a form of linear Genetic Programming (GP) to find optimal ways to combine standard compression algorithms for maximally and losslessly compressing files and archives. The system divides files into blocks of predefined length. It then uses a linear, fixed-length representation where each primitive indicates what compression algorithm to use for a specific data block. GP-zip worked well with heterogonous data sets, providing significant improvements in compression ratio over some of the best known standard compression algorithms. GP-zip had two main limitations: a) the fixed block size restricted the ways in which compression algorithms could be combined, and b) GP-zip entails a considerable computational load. In this paper we propose a substantial improvement of GP-zip, called *GP-zip**, which uses a new representation where blocks of different sizes can be evolved and intelligent operators which identify and target which elements of the representation to change to increase fitness with high probability. As we will see, the proposed improvements provide superior performance over the previous method with respect to both execution time and compression ratios.

The structure of this paper is as follows. In the next section we review previous attempts to use GP in the area of lossy and lossless compression. Section 3 provides a description of GP-zip, the starting point for GP-zip*, and a general evaluation for the algorithm's performance. Section 4 discusses GP-zip* in details. This is followed by experimental results in Section 5. Finally, conclusive remarks are given in Section 6.

2. RELATED WORK

The problem of heterogeneous file compression has been tackled by Hsu in [3]. The proposed system segmented the data into blocks of a fixed length (5 KB) and then compressed each block individually. The system passed the blocks to the appropriate compression method by using a file type detector that was able to classify ten different types of data. The approach also used a statistical method to measure the compressibility of the data. However, due to various restrictions, the results reported were not impressive.

Koza [4] was the first to use GP to perform compression. He considered, in particular, the lossy compression of images. The idea was to treat an image as a function of two variables (the row and column of each pixel) and to use GP to evolve a function that matches as closely as possible the original. One can then take the evolved GP tree as a lossy compressed version of the image. The technique, which was termed *programmatic compression*, was tested on one small synthetic image with good success. Programmatic compression was further developed and applied to realistic data (images and sounds) by Nordin and Banzhaf [5].

Iterated Functions System (IFS) are important in the domain of fractals and the fractal compression algorithm. [6] and [7] used genetic programming to solve the inverse problem of identifying a mixed IFS whose attractor is a specific binary (B/W) image of interest. The evolved program can then be taken to represent the original image. In principle this can then be further compressed. The technique is lossy, since rarely the inverse problem can be solved exactly. No practical application or compression ratio results were reported in [6], [7]. Using similar principles, Sarafoulous [8] used GP to evolve affine IFSs whose attractors represent a binary image containing a square (which was compressed exactly) and one containing fern (which was achieved with some error in finer details).

Wavelets are frequently used in lossy image and signal compression. Klappenecker [9] used GP to evolve wavelet compression algorithms, where internal nodes represented conjugate quadrate filters and leaves represented quantisers. Results on a small set of real world images were impressive, with the GP compression outperforming JPEG at all compression ratios.

A first *lossless* compression technique was reported in [10], where GP was used to evolve non-linear predictors for images. These are used to predict the gray level of a pixel will take based on the gray values of a subset of its neighbours (those that have already been computed in a row-by-row and column-by-column scan of the image array). The prediction errors together with the model's description represent a compressed version of the image. These were compressed using the Huffman encoding. Results on five images from the NASA Galileo Mission database were very promising with GP compression outperforming some of the best human-designed lossless compression algorithms.

In many compression algorithms some form of pre-processing or transformation of the original data is performed before compression. This often improves compression rates. In [11], Parent and Nowe evolved pre-processors for image compression using GP. The objective of the pre-processor was to reduce losslessly the entropy in the original image. In tests with five images from the Canterbury Corpus [12] GP was successful in significantly reducing the image entropy. As verified via the application of bzip2, the resulting images were markedly easier to compress.

In [13] the use of programmatic compression was extended from images to natural videos. A program was evolved that generates intermediate frames of video sequence, where each frame is composed by a series of transformed regions from the adjacent frames. The results were encouraging in the sense that a good approximation to frames was achieved. Naturally, although, a significant improvement in compression was achieved, programmatic compression was very slow in comparison with the other known methods, the time needed for compression being measured in hours or even days.
Acceleration in GP image compression was achieved in [14], where an optimal linear predictive technique was proposed, thanks to the use of a less complex fitness function.

3. GP-ZIP

As previously mentioned, the basic idea of GP-zip [2] was to divide the target data file into blocks of a predefined length and ask GP to identify the best possible compression technique for each block. The function set of GP-zip was composed of primitives two categories. The first category contains the following five compression algorithms: Arithmetic Coding (AC) [15], Lempel-Ziv-Welch LZW [16], unbounded Prediction by Partial Matching (PPMD) [17], Run Length Encoding (RLE) [18], and Boolean Minimization [19]. In the second category two transformation techniques are included: the Burrows-Wheeler Transformation (BWT) [20] and Move to Front (MTF) [21]. Given that these are mostly well-known techniques, we will not provide a detailed explanation of each of these compression and transformation algorithms. 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. Consequently, this does not directly produce a compression. However, often the transformed data are assumed to be more compressible, and hence, when passed to a compression algorithm in the function set, a better compression ratio is achieved.

GP-zip uses a form of "divide and conquer" strategy. Each member of the function set performs well when it works in the circumstances that it has been designed for. Dividing the given data into smaller blocks makes the creation and identification of such circumstances easier.

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

As the function set is composed of five compression algorithms and two transformation algorithms, it is clear that GP-zip has 15 different choices to compress each block. Namely, it can apply one of the five compression functions without any transformation of the data in a block, or it can precede the application of the compression function by one of two transformation functions.

The decision as to what block length to use to compress the data proceeds in stages. Initially, GP-zip randomly selects a block length from the set of possible lengths. The system starts by initializing a population randomly. As illustrated 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.

After the system finds the best possible combination of compression algorithms for the selected block-length, it restarts its search again using a different block-length. This is repeated multiple times. Of course, since testing all of the possible block lengths is considerably time-consuming, GP-zip selects the new lengths by performing a form of binary search over a set of possible lengths. This is applied for the first time after the system has tested two block lengths.

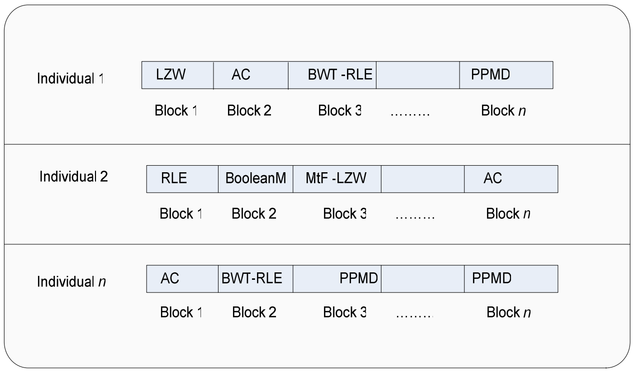

Figure 1: GP-zip individuals within a population

Since the proposed technique divides the data into blocks of predefined lengths and finds the best possible compression model for them, it is necessary for the decompression process to know which block was compressed by which compression/ transformation function. A header for the compressed files provides this information for the decompression process. The size of this header is not fixed, but it is an insignificant overhead in comparison to the file size.

After the process of searching for the best possible length for dividing the file and the best possible combination of compression/transformation functions, GP-zip merges the blocks with identical labels. For example, the chromosome `[PPMD][PPMD][LZW][LZW][LZW]` is interpreted as the application of PPMD to the first two blocks (considered as one) and LZW to the third, fourth and fifth block of a file (again considered as one). The concept being that through this "gluing" process, GP-zip obtains fewer blocks, which, on average, are bigger, leading to better compression. The assumption is that sequences of identical primitives will tend to indicate that the data in the corresponding blocks are of similar (or at least compatible) type. In experiments with a few data sets, the gluing process was shown to be beneficial. However, it is unlikely that the assumption above will apply in all cases. In those cases, it might in fact turn out that neighbouring blocks are best compressed using the same algorithm, but that there are statistical differences in the data in each block such that they should be treated (compressed) independently. Thus, while to some extent, gluing improves the flexibility of the representation, it also adds a bias which might make the algorithm less general. As we will see in the next section, with GP-zip* we will completely remove this problem while at the same time giving complete freedom to evolution to choose block sizes.

GP-zip has been tested with several collections of data (homogenous and heterogeneous). When the given data was homogenous (e.g., sets of text files), the system always converged to solutions where the data is treated as a single big contiguous block. In other words, the file is compressed normally with the best possible compression method from GP-zip's function set. Hence, GP-zip did not outperform existing algorithms on homogeneous data (it was not outperformed either, since it always chose a very good algorithm to do the compression). Alternatively, when the given data was composed of heterogeneous fragments (e.g., archive files), GP-zip did very well. We provide more information on GP-zip's performance in Section 5.

Although GP-zip has achieved good results in comparison with other well-known compression algorithms, it suffers from one major disadvantage. The described staged process of GP-zip is very time consuming (of the order of a day per megabyte). Of course, decompression is, instead, very fast. While there are many applications for which an asymmetric compression algorithm is useful–any compress-once/decompress-many scenario can accept some asymmetry–it is also clear that this level of asymmetry makes the system practically unviable even with today's CPUs.

4. GP-ZIP*

An alternative to GP-zip's scheme of imposing the use of a fixed length for the blocks is to allow the block length to freely vary (across and within individuals) to better adapt to the data. Here we propose a new method for determining the length of the blocks, which completely removes the need of a staged search for an acceptable fixed length typical of GP-zip. Furthermore, we provide new intelligent operators which are specialised to handle the new representation. As already mentioned, we call the resulting system *GP-zip**.

We have already pointed out that GP-zip is a very time consuming process. The reason is that a lot more effort is spent searching for the best possible length for the blocks, than for choosing how to compress the data. This search is performed using a type of binary search, which, in itself, is efficient. However, since each search query in fact involves the execution of a GP run, the whole process appears rather inefficient, particularly considering that the gluing method eventually will undo certain decisions regarding block lengths. This motivated us to find a way to eliminate the need of imposing a fixed length of blocks.

One initial idea to achieve more flexibility was to divide the given data into very small blocks (e.g. 100 byte per block), compress each block individually with the best compression model that fits into it, and then glue all the identical subsequent blocks. We eventually discarded this idea for two reasons. Firstly, it relies heavily on the gluing process, which, as we have seen introduces a bias in the compression (it assumes that identical neighbouring primitives indicate homogeneous data). Secondly, most compression techniques require some header information to be stored. Therefore, when applying a compression model to a very small set of data, the generated header information becomes bigger than the compressed data itself, which completely defeats the purpose.

We opted for a cleaner (and, as we will see, effective) strategy: we ask GP to solve the problem for us. That is each individual in the population represents in how many blocks to divide each file and the size of those blocks in addition to the particular algorithm used to compress each block. In other words, in GP-zip* we evolve the length of the blocks within each run, rather than use the staged evolutionary search, possibly involving many GP runs, of GP-zip. This significantly reduces the computational effort required to run the system. The difficulty of this method resides in the inapplicability of the standard genetic operators and the corresponding need to design new ones.

We describe the new representation and operators used in GP-zip* in the next sub-sections.

4.1 Representation

In the work reported in this paper we used for GP-zip* the same primitive set (i.e., the same compression and transformation models) as for GP-zip. We did this for two reasons: a) these primitives are amongst the best know compression methods and worked really well in GP-zip, and b) using the same primitives we can perform a more conclusive comparison between the two systems.

GP-zip* starts by initializing the population randomly. Similarly, to the previous method, all initial individuals contain blocks of a given length. However, differently to GP-zip, in GP-zip* the block length for each individual is chosen randomly. As shown in **Figure 2**, the resulting population includes individuals with a variety of block sizes. Individuals represent sequences of compression functions with or without transformation functions. High fitness individuals are selected probabilistically and are manipulated by crossover, mutation and reproduction operations (see below). So, although we start with individuals with equal-size blocks, during these processes the size of one or more blocks within an individual may change. This makes it possible to evolve any partition of the file into blocks. This gives GP-zip* the freedom of exploring many more possible solutions in the search space than GP-zip could. The hope is that some of these new possibilities will prove superior.

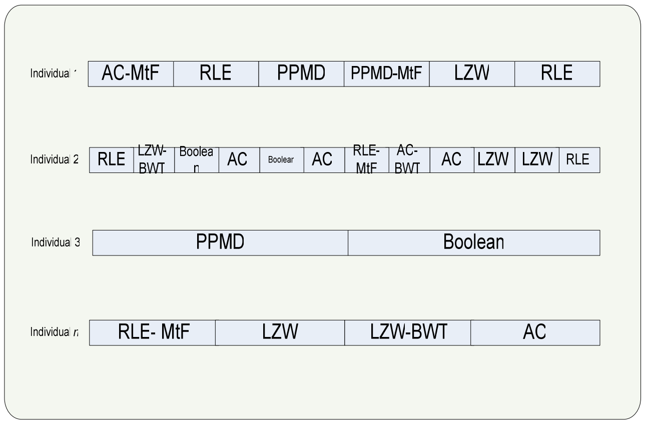

Figure 2: GP-zip* Individuals within a population

The size of the blocks for each of the generated individuals in the initialized population is a random number uniformly distributed between 200 bytes to the length of the file to be compressed. All integer values in the range are allowed.

Similarly to the old method, it is necessary for the decompression process to know which block was compressed by which compression/transformation function. A header for the compressed files provides this information. Since the new representation has eliminated the need of imposing a fixed length for the blocks, it is necessary for the decompression process to know the length for each block. This information increases the amount of information stored in the header file. However, since the size of the header is included in the calculation of the fitness (we simply use the compression ratio), evolution always stays clear of solutions that involve too many small blocks.

Also, similar to the old method, GP-zip* presents several advantages as a result of dividing the data into smaller blocks. For example, when required, the decompression process could easily process only a section of the data without processing the entire file. As another example, the decompression process could easily be parallelised (e.g., using multi- and hyper-threading, multiple CPU cores or use of GPUs) making files compressed with GP-zip* faster to decompress than those produced with most traditional methods.

GP-zip*'s individuals have a linear representation which may give the impression that the proposed system is a Genetic Algorithm rather than Genetic Programming method. However, this is arguably not true for following reason. GAs are known to have a fixed representation (binary strings) for individuals while in GP-zip* the system receive an input (block of data) of a variable length and return another block of data (typically shorter). Also, it should be noted that each member in the function set is a compression/transformation algorithm by itself which together form GP-zip* compression system. So, each element of the representation acts more like an instruction of a GP computer program than as a parameter being optimised by a GA.

4.2 Crossover

Crossover is one of the essential genetic search operators. The aim of crossover is to exchange genetic material between individuals, in order to generate offspring that hold features from both parents.

Since GP-zip used fixed length representations, GA-type standard genetic operators were used as variation operators. In particular, in the crossover, the GP-zip system selected two individuals with tournament selection, a common crossover point was then randomly chosen, and, finally, all the blocks before the crossover point in the first parent were concatenated with the blocks after the crossover point in the second parent to produce the offspring. So, this was a form of one-point crossover. Since in GP-zip* individuals are divided into blocks of different and heterogeneous lengths, we cannot use the same approach. Instead, we use an intelligent crossover.

One of the advantages of the subdivision into blocks of the individuals is that it is possible to evaluate to which degree the compression ratio of each individual contributes to the compression ratio for a file. This information can be used to identify and implement useful crossover hotspots. In our intelligent crossover operator we use this idea in conjunction with a greedy approach. The operator works by choosing one block in one parent and swapping it with one or more corresponding blocks in the other (we will see later what we mean by "corresponding"). The intelligence in the operator comes from the fact that instead of selecting a random block as a crossover point, GP-zip* selects the block with the lowest compression ratio in the first parent, which arguably is the most promising hotspot.

Naturally, the boundaries of the block chosen in the first parent will often not correspond to the boundaries of a block or of a sequence of blocks in the second parent. So, before performing crossover, GP-zip* resizes the block chosen in the first parent in such a way that its boundaries match the boundaries of blocks in the second parent. It is then possible to move all the corresponding blocks from the second parent to replace the (extended) original block in the first parent. Resizing is the process of extending the selected block size in the first parent with the intention to fit it within the boundaries of the corresponding block or the sequence of blocks in the second parent. The crossover operator is illustrated in **figure 3.**

This crossover operator (in conjunction with the new representation) is the key element for the improvements in speed and compression ratios provided by GP-zip* over its predecessor. In GP-zip the search was only guided by the fitness function. In GP-zip* it is also guided by the search operators. (As we will see in the next section also GP-zip*'s mutation uses the hotspot idea).

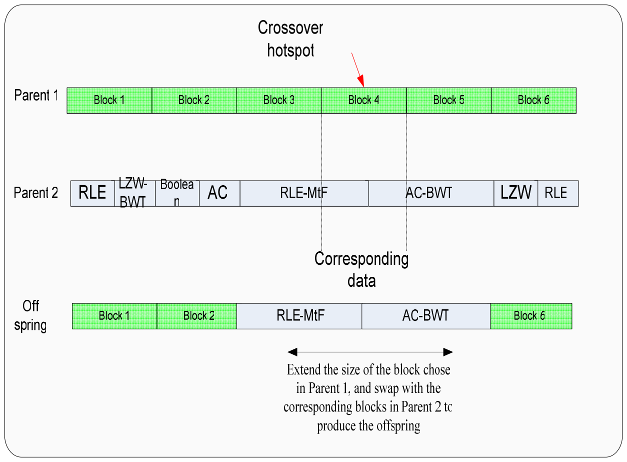

Figure 3: GP-zip* crossover operator

4.3 Mutation

Since GP-zip*'s intelligent crossover maintains homology and has the property of purity (crossing over identical parents produces offspring indistinguishable from the parents), GP-zip* populations can and do converge, unlike many other forms of linear GP. It is then important to use some form of mutation to ensure some diversity (and, so, search intensity) is maintained.

In GP-zip mutation worked as follows. One parent is selected via tournament selection, a random mutation point is chosen and then all subsequent blocks after the selected point are mutated into new randomly selected compression/transformations functions.

GP-zip* mutates individuals differently. Once again we took advantages of the block-wise nature of the individuals. GP-zip* chooses the block with the worst compression ratio in the individual. Then it randomly selects a new block size in addition to a new compression/transformation function for the block. The new size is a random number from 200 bytes to the length of the file. Once the system allocates a new size for the selected block, it resizes it. Depending on whether the new size is bigger or smaller than the previous size, the resizing process will extend or shrink the block. In either case, changing a length of one block will affect all the adjacent blocks. The changes may include: extending one or two neighbouring blocks, shrinking one or two neighbouring blocks, or even entirely removing some blocks. **Figure 4** illustrates two mutation cases.

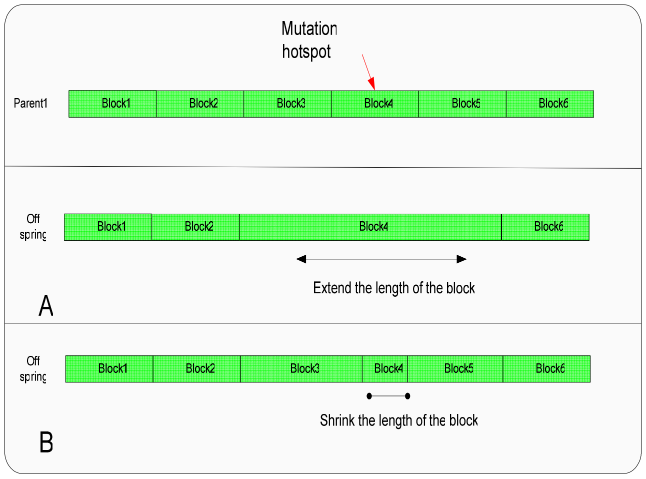

Figure 4: GP-zip* mutation operator

A: Extend the block length, B: shrink the block length

Similar to crossover, GP-zip* mutation is able to identify and target weak genetic material based on its internal credit assignment mechanism, which is a key element in GP-zip*'s improved quality of evolved solutions.

5. EXPERIMENTS

Experiments have been conducted in order to investigate the performance of GP-zip*. The aim of these experiments is to assess the benefits of its new representation and intelligent operators against the old method, as well as other widely used compression techniques. GP-zip* has been compared against the compression algorithms in its function set. In addition, bzip2 and WinRar, which are amongst the most popular compression algorithms in regular use, were included in the comparison.

In the field of compression, researchers mainly focus on measuring compression ratios. However, the time taken to perform compression and decompression is also often considered important. Unfortunately, several factors, for example, the size of the compressed file, the power of the CPU used, influence compression times, and it is, therefore, difficult to obtain a precise time comparison among different compressions models.

The evaluation of compression algorithms can be either analytical or empirical [12]. Generally, the analytical evaluation is expressed as compression ratio versus entropy of the source, which is assumed to belong to a specific class [12]. However, such evaluation is not accurate and may not be broadly applicable. In the empirical evaluation, the algorithm is tested with a collection of real files. The difficulty of this method resides in the ability of generalising its results because it is strictly limited to the domain of the experimentation. One might imagine that to establish the perfect evaluation of an algorithm one should test it with all the possible files and find the average compression ratio over those files. This approach, however, is not only computationally expensive, but also theoretically flawed [12]. The reason for this is already explained in the compression principle discussed in Section 1: *"There is no algorithm that is able to compress all the files even by 1 byte"*. That is, the average compression ratio across all files would be greater than or equal to the average of the original files!

Therefore, the empirical evaluation would really need to focus on the files that are likely to occur for each specific compression model. For example, if a compression model is designed to work for email attachments then normally all the experiments would concentrate on this set of files. In practice, even after focusing the experiments on relatively small set of files, it would still be impossible to collect all of them. Therefore, a random selection of test files must always be used.

Good criteria for selecting excremental files are detailed in [12]. Here we tried to satisfy some of them. The selected files should be representative. In other words, they should be likely to be compressed by other users. Their size should be within the normal range, not too big or too small. Of course, the more files are included in the experiments, the more accurate the characterisation of the behaviour of the algorithm is. Naturally, since each fitness evaluation in GP-zip* is very expensive, we had to balance this criterion with ensuring the experiments remained computationally feasible with the hardware available.

In order to compare the performance of GP-zip* against GP-zip, we used the same data sets (Text, Exe and Archive1) that were used to test previous method in [2]. Furthermore, new files have been included such as the Canterbury corpus [12], which is among the most popular benchmarks for data compression algorithms. **Table 1** presents a list of the files that have been included in the experiments. As one can see, the experiments covered both heterogeneous and homogenous sets of data. The text archive and the executable archive are available in [22].

The experiments presented here were performed using the following parameter settings:

- Population of size 100.
- Maximum number of generations 100.
- Crossover with probability of 75%.
- Mutation with probability 20%.
- Reproduction with probability of 5%.
- Tournament selection with tournament size 2.

Archive	Files	Size in bytes
Text	English translation of The Three Musketeers by Alexandre Dumas	1,344,739
	Anne of Green Gables by Lucy Maud Montgomery	586,960
	1995 CIA World Fact Book	2,988,578
Exe	DOS Chemical Analysis program	438,144
	Windows95/98NetscapeNavigat	2,934,336
	Linux 2.x PINE e-mail program	1,566,200
Archive1	Mp3Music	764,361
	Excel sheet	64,000
	Certificate card replacement form PDF www.padi.com	92,932
	Anne of Green Gables by Lucy Maud Montgomery (text file)	586,960
Archive2	PowerPoint slides	60,928
	JPEG file	2,171,285
	C++ source code	24,432
	Mp4 Video (5 seconds)	259,599
Archive3	GIF file	322,513
	Unicode text file (Arabic language)	379,148
	GP-zip* executable file	520,293
	Xml file	193,794
Canterbury corpus	English text, fax image, C code, Excel sheet, Technical writing, SPARC exe, English poetry, HTML, lisp code, GUN Manual Page, play text.	2,790,708

Table 1: Test files for GP-zip*

There is no terminating condition for GP-zip*. Hence, GP-zip* runs until the maximum number of generations is reached. The results of the experiments are illustrated in **tables 2** and **3**.

Compression\Files	Exe	Text	Archive1
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%	49.49%
GP-zip*	62.03%	80.23%	63.43%

Table 2: Performance comparison against GP-zip

Compression\Files	Archive2	Archive3	Canterbury
WinZip-bzip2	3.90%	64.49%	80.48%
WinRar- Best	3.19%	65.99%	85.15%
PPMD	3.90%	64.36%	81.19%
BooleanM	2.82%	23.28%	38.01%
LZW	-43.98%	43.62%	15.61%
RLE	-11.49%	9.16%	6.55%
AC	0.70%	27.62%	41.41%
GP-zip*	58.70%	75.05%	81.58%

Table 3: Performance comparison

As can be seen by looking at the compression ratios obtained on the two homogenous sets of data (first and second columns in **table 2**), GP-zip* does very well outperforming seven of its eight competitors, with only WinRar doing marginally better. This is an excellent result. (Remember bzip2 and WinRar were not available to either GP-zip or GP-zip* as primitives.) It is also particularly interesting that GP-zip* was able to compress such files slightly better than GP-zip. Although the margin is very small, the fact is noteworthy, because on such data GP-zip was only as good as the best compression algorithm in its primitive set. GP-zip* does better thanks to its increased ability to detect heterogeneous data blocks (of different sizes) even within one set of homogenous data. By then compressing each block separately (but not necessarily with a different algorithm), a better compression ratio was achieved.

While these results are very encouraging, where GP-zip* really shines is the compression of data files composed of highly heterogeneous data fragments, such as in Archive1, Archive2 and Archive3, where GP-zip* outperforms all other algorithms by very significant margins. With Archive1 (last column of **table 2**) one can appreciate the effect of the changes in representation and operators introduced in GP-zip* with respect to GP-zip. On heterogeneous data GP-zip* comes second only in the Canterbury dataset, marginally losing against WinRar. We should note, however, that this dataset is often used as a reference for comparison of compression algorithms, and so parameters (in highly optimised compression software such as WinRar) are often tuned to maximise compression on such a dataset. Furthermore, the high compressibility of the dataset indicates that, despite it being heterogeneous, effectively the entropy of the binary data it contains, may be atypically low (making it similar to a text archive).

Naturally, GP-zip* is a stochastic search algorithm. Consequently, it is not always guaranteed to obtain the best possible compression ratio. The results presented above are the best obtained when running GP-zip* 15 times. However, as shown in **table 4**, GP-zip* is very reliable with almost every run producing highly competitive results. The table reports also the average run times for the algorithm. Although, thanks to the new improvements, GP-zip* is considerably faster and it produces much better results than the old method, it is fair to say that the algorithm is still very slow (we timed the system on a 2.21GHz AMD PC). We believe that computational times can be reduced by one to 1.5 orders of magnitude by making use of multiple CPU cores and/or GPUs.

	Exe (4.07MB)	Text (4.07MB)	Canterbury (2.66MB)
Compression Average	61.68%	79.85%	79.16%
Standard deviation	0.23	0.26	4.88
Best Compression	62.03%	80.23%	81.58%
Worst Compression	61.25%	79.36%	69.53%
Compression Time in hours	12 hours	12 hours	10 hours
Compression time Hours/Megabyte	2.95	2.95	3.76
	Archive1 (1.43MB)	Archive2 (2.39MB)	Archive3 (1.35MB)
Compression Average	42.02%	16.97%	66.25%
Standard deviation	11.26	22.41	2.74
Best Compression	63.43%	58.70%	75.05%
Worst Compression	33.29%	3.70%	64.20%
Compression Time in hours	7 hours	8 hours	6 hours
Compression time Hours/Megabyte	4.90	3.35	4.44

Average of Compression time	9.17 hours
Average of compression time/Mega	3.73 hours

Table 4: Summarization of 15 GP-zip* runs for each data set

6. CONCLUSION AND FUTURE WORK

The aim of this research is to understand the benefits and limitations of the concept of identifying and using the best possible lossless compression algorithm for different parts of a data file in such a way to ensure the best possible overall compression ratio.

The GP-zip system we proposed in earlier research [2] was a good starting point. However, it suffered from several limitations. In this paper we have proposed a new system, GP-zip*, where such limitations are removed and the search efficiency is further improved thanks to the use of intelligent genetic operators. The proposed improvements have produced a system that significantly outperforms its predecessor as well as most other compression algorithms, being best of all compression algorithms tested on heterogeneous files and never being too far behind the best with other types of data.

In addition, to providing better compression, the division of data files into blocks 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 compression, as GP-zip* can send each decompressed block into the operating system pipeline sequentially.

Although the proposed technique has achieved substantial compression ratio of heterogeneous files in comparison with the other techniques, it suffers from one major disadvantage. The process of GP-zip* is very computationally intensive. In future research we will concentrate on this particular aspect of GP-zip* as further substantial improvements can be expected. Also, the experiments have demonstrated that GP-zip* always outperforms the compression models in its function set. However, the overall algorithm performance is somehow limited by the power of the used models within the function set. Increasing the number compression models within GP-zip* is expected to further improve its performance.

Also, currently, we treat each compression/ transformation model in the function set as a black box, so as an extension for this research we can decompose each of the functions in the function set and try to combine their internal features. Also, a 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.

7. REFERENCES

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

[2] Ahmad Kattan and Riccardo Poli, *Evolutionary Lossless Compression with GP-ZIP,* Proceedings of the IEEE World Congress on Computational Intelligence, IEEE 2008.

[3] William H. Hsu and Emy E. Zwarico, *Automatic Synthesis of Compression Techniques for Heterogeneous Files* SOFTPREX: Software–Practice and Experience, Vol. 25, 1995.

[4] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, ambridge, MA, USA, 1992.

[5] Peter Nordin and Wolfgang Banzhaf. *Programmatic compression of images and sound*. In John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, editors, Genetic Programming 1996: Proceedings of the First Annual Conference, pages 345–350, Stanford University, CA, USA, 28–31 July 1996. MIT Press.

[6] Evelyne Lutton, Jacques Levy-Vehel, Guillaume Cretin, Philippe Glevarec, and Cidric Roll. *Mixed IFS: Resolution of the inverse problem using genetic programming*. Complex Systems, 9:375–398, 1995.

[7] Evelyne Lutton, Jacques Levy-Vehel, Guillaume Cretin, Philippe Glevarec, and Cidric Roll. *Mixed IFS: Resolution of the inverse problem using genetic programming*. Research Report No 2631, Inria, 1995.

[8] Anargyros Sarafopoulos. *Automatic generation of affine IFS and strongly typed genetic programming*. In Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, editors, Genetic Programming, Proceedings of EuroGP'99, volume 1598 of LNCS, pages 149–160, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.

[9] Andreas Klappenecker and Frank U. May. *Evolving better wavelet compression schemes*. In Andrew F. Laine, Michael A. Unser, and Mladen V. Wickerhauser, editors, Wavelet Applications in Signal and Image Processing III, volume 2569, San Diego, CA, USA, 9-14 July 1995. SPIE.

[10] Alex Fukunaga and Andre Stechert. *Evolving nonlinear predictive models for lossless image compression with genetic programming*. In John R. Koza, Wolfgang Banzhaf, Kumar Chellapilla, Kalyanmoy Deb, Marco Dorigo, David B. Fogel, Max H. Garzon, David E. Goldberg, Hitoshi Iba, and Rick Riolo, editors, Genetic Programming 1998: Proceedings of the Third Annual Conference, pages 95–102, University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998. Morgan Kaufmann.

[11] Johan Parent and Ann Nowe. *Evolving compression preprocessors with genetic programming*. In W. B. Langdon, E. Cant´u-Paz, K. Mathias, R. Roy, D. Davis, R. Poli, K. Balakrishnan, V. Honavar, G. Rudolph, J. Wegener, L. Bull, M. A. Potter, A. C. Schultz, J. F. Miller, E. Burke, and N. Jonoska, editors, GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference, pages 861–867, New York, 9-13 July 2002. Morgan Kaufmann Publishers.

[12] Bell, R.A.a.T.C. *A corpus for the evaluation of lossless compression algorithms*. in IEEE Data Compression Conference (DCC'97). March 25 1997. Los Alamitos, California.: IEEE Computer Society.

[13] Thomas Krantz, Oscar Lindberg, Gunnar Thorburn, and Peter Nordin. *Programmatic compression of natural video*. In Erick Cant´u-Paz, editor, Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002), pages 301–307, New York, NY, July 2002. AAAI.

[14] Jingsong He, Xufa Wang, Min Zhang, Jiying Wang, and Qiansheng Fang. *New research on scalability of lossless image compression by GP engine*. In Jason Lohn, David Gwaltney, Gregory Hornby, Ricardo Zebulum, Didier Keymeulen, and Adrian Stoica, editors, Proceedings of the 2005 NASA/DoD Conference on Evolvable Hardware, pages 160–164, Washington, DC, USA, 29 June-1 July 2005. IEEE Press.

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

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

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

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

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

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

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

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