, , 1–28 ()
© Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Evolution of Human-competitive Lossless Compression Algorithms with GP-zip2

AHMED KATTAN akatta@essex.ac.uk
School of Computer Science and Electronic Engineering, University of Essex, Colchester, CO4 3SQ, UK

RICCARDO POLI rpoli@essex.ac.uk
School of Computer Science and Electronic Engineering, University of Essex, Colchester, CO4 3SQ, UK

Abstract. We propose GP-zip2, a new approach to lossless data compression based on Genetic Programming (GP). GP is used to optimally combine well-known lossless compression algorithms to maximise data compression. GP-zip2 evolves programs with multiple components. One component analyses statistical features extracted by sequentially scanning the data to be compressed and divides the data into blocks. These blocks are projected onto a two-dimensional Euclidean space via two further (evolved) program components. K-means clustering is then applied to group similar data blocks. Each cluster is labelled with the optimal compression algorithm for its member blocks. After evolution, evolved programs can be used to compress unseen data. The compression algorithms available to GP-zip2 are: Arithmetic coding, Lempel-Ziv-Welch, Unbounded Prediction by Partial Matching, Run Length Encoding, and Bzip2. Experimentation shows that the results produced by GP-zip2 are human-competitive, being typically superior to well-established human-designed compression algorithms in terms of the compression ratios achieved in heterogeneous archive files.

Keywords: Genetic Programming, Lossless Compression, Human-competitive Results, Feature Extraction, K-means.

1. Introduction

Today we live in a digital era where information is created at an increasingly explosive rate. For decades the demand for data storage and transfer has constantly stretched both the hardware infrastructure of the Internet and available space on storage media to their limits. Of course, in response to this, there has been a stable improvement of storage and transmission technologies (e.g., Blue Ray CDs and fibre-optics networks). However, the development and take up of these technologies is never fast enough to cope with user needs [36].

Data compression [36, 34, 33] is one of many technologies that help overcome these limitations. High-quality digital TV would not be possible without data compression: *one second* of video transmitted or stored without compression using the traditional CCIR 601 standard (625 lines per frame, 720 samples per line) would need over 20 MB, i.e., 70 GB for a two-hour movie! Also, the clarity we experience in our phone calls and in the music produced by our MP3 players would not be possible without data compression: 2 minutes of a uncompressed CD-quality (16 bit per sample) music would require more about 80 MBs. Without compression uploading or downloading online videos and music would consume prohibitive amounts of bandwidth and/or time. Even faxing documents over ordinary phone lines would have not been possible without data compression.

So, what exactly is data compression? Data compression [36, 34, 33] is the process of observing regularities in data streams and exploiting them to minimise the total length of the compressed file. Naturally, the process of compressing and decompressing files takes time and costs CPU cycles and, so, one always needs to balance these costs against the extra time and costs involved in the storage and transfer (to and from disk and/or via a network) of uncompressed files. While in some domains it may be more attractive to spend money on storage rather than on CPU cycles (for compression and decompression), for most users of computers and other digital devices the balance is in favour of the use of compressed files, particularly considering how many CPU cycles are wasted every day in computers idling or running screen savers and the cost and difficulties in upgrading disks. For example, upgrading the hard disk of a laptop requires technical competence and a lengthy backup/restore procedure, which may even involve the complete re-installation of all software packages, including the operating system, from the original disks.

A key question when deciding whether to use compression is how much can compression save. The answer depends on the compression model used and the type and amount of redundancy in the given data [33]. This is particularly true for lossless compression, the focus of this paper, where simple counting arguments show that a sort of no-free lunch principle exists by which, unfortunately, no single compression algorithm is guaranteed never to increase the size of a file. For example, if all possible files of length n could be compressed to length $m < n$, how could the compression be lossless given that there are 2^n files to reconstruct but only 2^m ($< 2^n$) possible compressed representations? Naturally, in practice we never need to compress all possible files and ample regularities and redundancies exist in the data to be compressed. These, however, are not all of the same nature and in practice no single compression model works effectively for all data types. So, while effective lossless compressions is definitely possible, it is very challenging because one needs to ensure that there is a good match between the adopted compression model and the regularities and redundancies in the data.

When a great deal is known about the regularities in the files to be compressed, e.g., in the compression of audio, photo and video data, it is possible to find compression algorithms that match such regularities almost optimally, thereby providing significant reductions in file sizes. These algorithms are difficult to derive and are often the result of many person-years of effort (e.g., the JPEG and MPEG standards have been developed for approximately two decades). Nonetheless, these are so successful that effectively some types of data are regularly saved, kept and transferred, only in compressed form, to be decompressed automatically and transparently to the user only when loaded into applications that make use of them (e.g., an MP3 player).

When the nature and regularities of the data to be compressed is less predictable or when the user has to deal with heterogeneous sets of data, then a general purpose compression system, while unavoidably less effective than the highly specialised ones mentioned above, is the only option. Heterogeneous data sets occur in a variety of situations. For example, a user may decide to use a compressed file system for his disk drive (a standard option in many operating systems) to prolong its life, which implies that a mixture of file types (possibly some of which, such as music and videos, are already in compressed form) need to be kept in losslessly compressed form. Perhaps even more frequently, a computer user may want to store multiple files in an archive file and then compress it to save space. Also, compressing archives is often necessary to make it possible to email them to other people given that many mail servers limit the size of each email to 10 or 15MB (even the generous Gmail limits emails to 25MB). Compression is essential also in large software distributions, particularly of operating systems, where the size of the media (CD or DVD) is fixed and compression is used to cram as much material as possible on one disk. Also, there are nowadays many files with a complex internal structure that store data of different types simultaneously (e.g., Microsoft Office documents, PDFs, computer games, HTML pages with online images, etc.). All of these situations (and many more) require a general-purpose lossless compression algorithm. These are important but also very difficult to derive and a lot of research effort has been devoted to producing such algorithms. Several such algorithms have been in the past (or are still today) covered by patents.

Of course, as we noted earlier, a truly general compression algorithm is impossible. So, what is typically meant by "general purpose" in relation to a compression algorithm is that the algorithm works well, on average, with the type of data typically handled by an average computer user. This is both a strength and a weakness, though. It is a strength in that using knowledge of typical use patterns allows the designers to "beat" the theoretical limitations of lossless compression. It is a weakness because not all computer users store and transfer the same proportion of files of each type. Also, proportions of files of each type vary from computer to computer even for the same user (think, for example, of the differences between the types of data stored in the computers used at work and in those at home). It is clear that significant benefits could be accrued if a compression system, while still remaining general purpose, could adapt to individual users and use patterns.

Also, the best current compression algorithms do to capture regularities in heterogeneous data sets is to ensure files with each known extension are compressed with an algorithm which on average works well on files with that extension. Some systems (but not all) may even ensure each file in an archive is compressed with a supposedly good algorithm for that type of file. However, no algorithm attempts to capture the differences in the internal regularities within single files to any significant degree (more on this later). Yet, clearly, exploiting these differences could markedly improve the compression achievable.

In this paper we investigate the idea of *evolving general-purpose compression programs that can improve over current compression systems by addressing the aforementioned limitations in relation to heterogeneous data*, particularly archive files. The objective is to be able to distinguish different fragments in the data and optimally compress them using the strengths of existing compression models to achieve higher compression.

Data compression is a highly sophisticated procedure. Therefore, evolving a data compression algorithm is not an easy task. Nonetheless, few attempts have been made to use Evolutionary Algorithms (EAs) to evolve data compression models obtaining good results in comparison to other compression algorithms. So, after briefly reviewing some key classical compression algorithms, in section 2 we will review EA-based systems in detail. In the same section we will also review two further techniques, GP-zip and GP-zip*, that are predecessors of the system described in this paper, which we call GP-zip2. Although the results obtained with GP-zip and GP-zip* were very encouraging, they suffered from one major disadvantage: the compression process was very slow and impractical for large files. This is common to other prior work where GP systems have been reported to be able to achieve good results at evolving compression models but also to take several orders of magnitude longer than standard approaches. Also, GP was used as a metaheuristic in GP-zip and GP-zip*, i.e., we did not evolve compression algorithms but compressed versions of files. In Section 3 a detailed description of the new system, GP-zip2, which overcomes these problems, is presented. This is followed by Section 4 which provides experimental results. Finally, Section 5 concludes this paper and suggests some possible future research.

2. Related work

In this section we some important classical and evolutionary compression algorithms. Throughout the section and in the rest of the paper we use the notion of *compression ratio* to quantify the reduction in the data size produced by a compression algorithm. This is defined as $1 - (\textit{Compressed File Size})/(\textit{Uncompressed File Size})$.

2.1. Conventional Data Compression

Data compression is a huge area and it is impossible to review all important work in this article. We will, however, briefly mention some key algorithms which are relevant to this paper.

Typically data compression algorithms have been implemented through a two-pass approach. In the first pass, the algorithm collects information regarding the data to be compressed. In the second pass, the actual encoding takes place. The famous Huffman code [14], which is still frequently used nowadays, falls in this category. Arithmetic Coding (AC), the first improvement over Huffman's code after a gap of 25 years [25], is also in this category, the extra power over its predecessor deriving from its ability to exploit longer-distance regularities in the data. AC will be used in GP-zip2.

This standard two-pass approach has been improved through so the called adaptive compression, where the algorithm only needs one pass to compress the data [8]. The main idea of adaptive compression algorithms is that the encoding scheme changes as the data are being compressed. A key advantage of adaptive compression is that it does not require the entire message to be loaded into the memory before the compression process can start. The disadvantage, however, is that it involves intensive computational efforts. This idea has been applied to Huffman coding [39] and AC [33]. Another adaptive compression is Prediction by Partial Matching (PPM) [6] which tries to predict the probability of a particular character being in a specific location from the n symbols previously occurred. The actual encoding of the symbols is done with AC compression. An improved version of the algorithm, called PPMD [7], will be used in GP-zip2.

In dictionary-based compression the main idea is to replace frequent substrings in the data with references that match the data that has already been passed to the encoder. The encoder creates a dictionary for the common sub-strings in the file and uses it as reference for the data. Algorithms in this category include the well-known Lempel-Ziv-Welch (LZW) algorithm [40] which was effectively a refinement of an earlier compression systems known as LZ77 [44]. We use also LZW within GP-zip2.

In many real world applications, it is ineffective to use a single compression model to exploit the regularities of the data. Therefore, a practical approach is to apply a composite model. One way of composing compression algorithms is to run a number of compression algorithms successively. A very good composite compression system of this type is Bzip2 [37]. It has been widely used in many commercial compression applications, such as WinZip [41]. In this algorithm, the data are compressed through several compression and transformation techniques, including the Huffman coding. Bzip2 is generally considered among the best general-purpose compression algorithms, but it is relatively slow, particularly in the compression phase. Also this algorithm is used within GP-zip2.

Before we move on to discuss compression algorithms based on EAs, we would like to mention one non-evolutionary approach that attempted to capture the differences in the internal regularities within single files, with some degree of success. This idea was explored in [13] where files were segmented into blocks of a fixed length (5 KB). If a block was deemed to be compressible based on certain numerical quantities, the block was passed to a modified Unix `file` command which attempted to determine the data's type. Then each compressible block was individually compressed with a good compression algorithm for that type of data. In experimentation with 20 heterogeneous small test files (3MB in total) the proposed system was able to outperform the best commercial compression system of the time by a 5.6%, suggesting that the basic ideas of the approach were right. We are not aware of any further exploration of this technique.[1]

2.2. Evolutionary Compression

Koza [20] was the first to use GP to perform compression. He considered 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 function 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) in [30].

In [23] the use of programmatic compression was extended from images to videos. A program was evolved that generates intermediate frames of a video sequence. Each frame is composed by a series of transformed regions from adjacent frames. Although a significant improvement in compression was achieved, programmatic compression was very slow in comparison with other known methods, the time needed for compression being measured in hours or even days. In [12] an optimal linear predictive technique was proposed. The use of a simpler fitness function led to an acceleration in GP image compression.

Iterated Functions Systems (IFS) are important in the domain of fractals and the fractal compression algorithms. [27] and [28] used GP to solve the inverse problem of identifying a mixed IFS whose attractor is a specific binary image of interest. The evolved program can be taken to represent the original image. In principle this can be further compressed. The technique is lossy, since the inverse problem can rarely be solved exactly. No practical application or compression ratio results were reported. Using similar principles, [35] used GP to evolve affine IFSs whose attractors represent a binary image containing a square (compressed exactly) and one containing fern (compressed with some errors).

Wavelets are frequently used in lossy image and signal compression. [19] used GP to evolve wavelet compression algorithms. 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 [9], where GP was used to evolve non-linear predictors for images. These were used to predict the gray level 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). The prediction errors together with the model's description represent a compressed version of the image. In [9] these were further 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.

Compression algorithms often use dictionaries to perform compression. By choosing suitable syllables for a dictionary it is possible to achieve a higher compression ratio than with single characters. However, the space of possible syllable dictionaries is typically immense. [38, 24] used Genetic Algorithms (GAs) to evolve dictionaries of syllables for the Huffman coding and LZW. In many conditions the achieved compression ratios with evolved dictionaries were superior to those of standard methods. Tests were performed with Turkish [38], Czech and English [24].

Zaki and Sayed [43] used GP to improve the standard Huffman coding. The proposed system utilised GP to find the most repetitive substrings within the text to be compressed. The search population was a collection of nodes that describe different substrings. A Huffman tree was generated to encode high frequency substrings with shorter references. Reported results demonstrated that the Huffman tree generated with this system was able to achieve higher compression than the standard Huffman coding algorithm by small margins in some of the cases.

In [31] Oroumchian *et al.* proposed an online text compression system for web application based on a GA. The system utilises a GA to find the most repetitive N-grams within the text and replaces them with shorter references. This allows the system to achieve high compression ratios while minimising the total number of N-grams used. The proposed method was tested on Persian text obtaining good results but no comparisons with other compression techniques was reported.

In many compression algorithms some form of pre-processing or transformation of the original data is performed before compression. This can improve compression ratios. [32] 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 GP was successful in significantly reducing image entropy. As verified via the application of bzip2, the resulting images were easier to compress.

Recently we presented a lossless GP data compression system called GP-zip and a further improvement called GP-zip* [17, 18]. We describe these predecessors of GP-zip2 below.

*2.3. GP-zip and GP-zip**

GP-zip [17] automatically evolved data compression algorithms which were specific for each data file. The system took advantage of existing compression techniques. The basic idea 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 primitive set of GP-zip included five compression algorithms: Arithmetic Coding [42], Lempel-Ziv-Welch [45], unbounded Prediction by Partial Matching [7], Run Length Encoding (RLE) [10] and

Boolean Minimisation [15] — an algorithm developed by the first author as part of his MSc project. Each algorithm could be used alone or combined with one of two transformation techniques, the Burrows-Wheeler Transformation (BWT) [5] and Move to Front (MTF) [1], which can make data more compressible.

GP-zip individuals were linear structures that represent sequences of compression functions with or without transformation functions. Each primitive was treated as a black box which receives a block of data as input and returns a (typically) smaller block of compressed data as an output. The decision as to which block length to use for the compression of a file was made by running GP multiple times (each time with a different block length, based on a bisection strategy) and choosing the one that provided best results. The length of the file before compression was one of the available block lengths. So, GP-zip could choose not to divide the file into smaller blocks if this provided better compression. After evolution, the system glued together sequences of blocks that had been marked to be compressed with the same compression algorithm since compressing them as one block was more effective.

GP-zip performed well in terms of compression ratios (more on this below). However, it suffered from a major disadvantage: the long time needed for the execution of the compression which ranged from several hours to a day per megabyte, making the system orders of magnitude slower than other compression algorithms. A reason for this heavy computational load was the staged search process which GP-zip used to identify the best block length.

In further research [18], we proposed a system called *GP-zip**, which explored the idea of allowing the block length to freely vary (across and within individuals) to better adapt to the data. In GP-zip*, each individual in the population encoded the number of blocks in which to divide a file and the size of those blocks in addition to the particular algorithms to be used to compress each block. To manipulate this representation we designed intelligent crossover and mutation operators which targeted “hot spots” in the parent individuals. For example, the crossover operator worked by choosing the block with the lowest compression ratio in the first parent and swapping it with one or more corresponding blocks in the other.

The new representation and operators used in GP-zip* resulted in better compression ratios. However, GP-zip* was only about 50% faster than GP-zip, thus remaining still significantly slower than the other compression algorithms, the time needed for compressing one megabyte ranging from 6 to 12 hours. This is because, although GP-zip* required only one GP run instead of the multiple runs used in GP-zip, runs had to be bigger.

Processing times of this magnitude imply that GP-zip and GP-zip* have a relatively small niche in the compression world. Because decompression is fast but compression is very slow, they can only be applied in situations where a file needs to be read/transferred and decompressed many times, such as in the production of movies (e.g., DVDs) or large scale software distributions. For other applications, however, a better solution is needed.

Before we present GP-zip2 in the next section, we would like to come back to the performance of GP-zip and GP-zip*. These algorithms were initially reported as being able of producing compression ratios significantly superior to all other algorithms against which they were compared [17, 17, 18, 18]. However, during the review process of this paper (many thanks to the reviewers for forcing us to look carefully at our data) we found that a bug affected the performance of such systems. The problem was in the Boolean Minimisation algorithm which we originally adopted also in this work. The bug meant that in certain conditions Boolean Minimisation produced compressed versions of blocks which are impossible to decompress. The files or blocks compressed via the faulty algorithm were typically smaller than those produced by other algorithms, often making evolution prefer such an algorithm over other primitives. Because of this, previous published results regarding the performance of GP-zip and GP-zip* were incorrect (corrections will be published in due course).

Having found this problem, we have re-run all experiments with GP-zip and GP-zip*, omitting the buggy Boolean Minimisation algorithm (see [16]). Naturally, results are inferior to those previously reported. However, they still indicate the promise of the ideas behind these two systems. For example, when using the same test sets as in [17, 18], GP-zip is 1.28% better than its closest competitor (WinRar-Best) and 2.39% better than Bzip2, while GP-zip* is on average 2.60% better than (WinRar-Best) and 3.24% better than Bzip2 on heterogeneous data.

3. GP-zip2

The interpretation of the sequence of bytes contained in a data file entirely depends on what we know about that file and what our expectations regarding the content of the file are. In most cases these are determined by the file name and extension (although further information may also be available). When present, one can use such knowledge about a file to decide how to compress it, and this is what many off-the-shelf compression algorithms do. However, when facing unknown data (e.g., an archive in a format unknown to the operating system) one cannot exploit this information. Also, in order to do really well in compressing archives we need to go deeper than the level of component files: we need to be able to identify incompatible data fragments *within* files and to allocate the best possible compression model to each. However, no source of information is typically available on what regularities there are and where they are within each file. Thus, we are left with a stream of data units but without an interpretation.

To counter this, GP-zip2 tries to spot regularities within the data and associate them with the best compression technique for such regularities. The exact nature of the regularities to be exploited and the association with compression algorithms is performed by evolution during a training phase.

GP-zip2 uses the following five compression algorithms: Arithmetic Coding, Lempel-Ziv-Welch (LZW), unbounded Prediction by Partial Matching (PPMD), Run-Length Encoding, and Bzip2. These five compression algorithms were selected because they belong to different compression categories and are known to commonly perform well with data of one or more types. For example, PPMD performs best when dealing with natural language text, run-Length encoding is very powerful with black and white pictures while LZW is widely used with GIF and TIFF images. The parameters setting of all five algorithms were selected based on their designers' recommendations. In addition, GP-zip2 has a "no compression" option, which gives the system the freedom to not compress some parts of the data.

Unlike GP-zip and GP-zip* which used a linear representation, GP-zip2 uses a *multi-tree representation* [21, 11, 26] and a type system to ensure primitives are used correctly.

Table 1. The primitive set used for the splitter and feature-extractions trees in GP-zip2 individuals.

Primitive	*Arity*	*Input type(s)*	*Output type*
Median, Mean, Average deviation, Standard deviation, Variance, Signal size, Skew, Kurtosis, Entropy	1	Array of integers (0–255)	Real number
Plus, Minus, Div, Mul	2	Real numbers	Real number
Sin, Cos, Sqrt	1	Real number	Real number
List	0	NA	Array of integers (0–255)

Each individual includes one *splitter tree*, and two *feature-extraction trees* (more on their function in the next subsections). These trees use the primitive set shown in Table 1 which allows extracting statistical features out of the byte series associated to a file. Note that, while these trees use the same primitives, the contents and size of the *List* terminal are context-sensitive: in the splitter tree *List* contains a fragment of data of fixed size which falls within a specific window (more on this later), while in the feature-extraction trees *List* contains a variable-size block of data segmented by the splitter.

3.1. *File segmentation*

The main job of splitter trees is to split a raw byte-series into meaningful segments, where by "meaningful" we mean that each segment can be compressed well with one of the available compression algorithms.

The system moves a sliding window of size L over the given byte-series with steps of S bytes. At each step the splitter tree is evaluated. This corresponds to applying a function, f_{splitter}, to the data in the window. The output of f_{splitter} is a number which is an abstract representation of the features of the signal in the window. The system then splits the byte-series at a particular position if the difference between the output of f_{splitter} in two consecutive windows is more than a predefined threshold θ. The operation of the splitter tree is illustrated in the lower part of Figure 1.

3.2. *Classification of Segments*

The job of each feature-extraction tree in our GP representation is to analyse statistical information on the segments identified by the splitter tree using the primitives in Table 1 and condense it into a higher-level numerical feature.

As illustrated in upper part of Figure 1, by using the outputs produced by the two trees as coordinates, we map each segment into a point in a two-dimensional Euclidean space. We then use an unsupervised pattern classification approach on the points resulting from processing a set of training files to discover regularities in them. In particular, we use K-means clustering to organise blocks (as represented by their two composite features) into groups. With this algorithm objects within a cluster are similar to each other but dissimilar

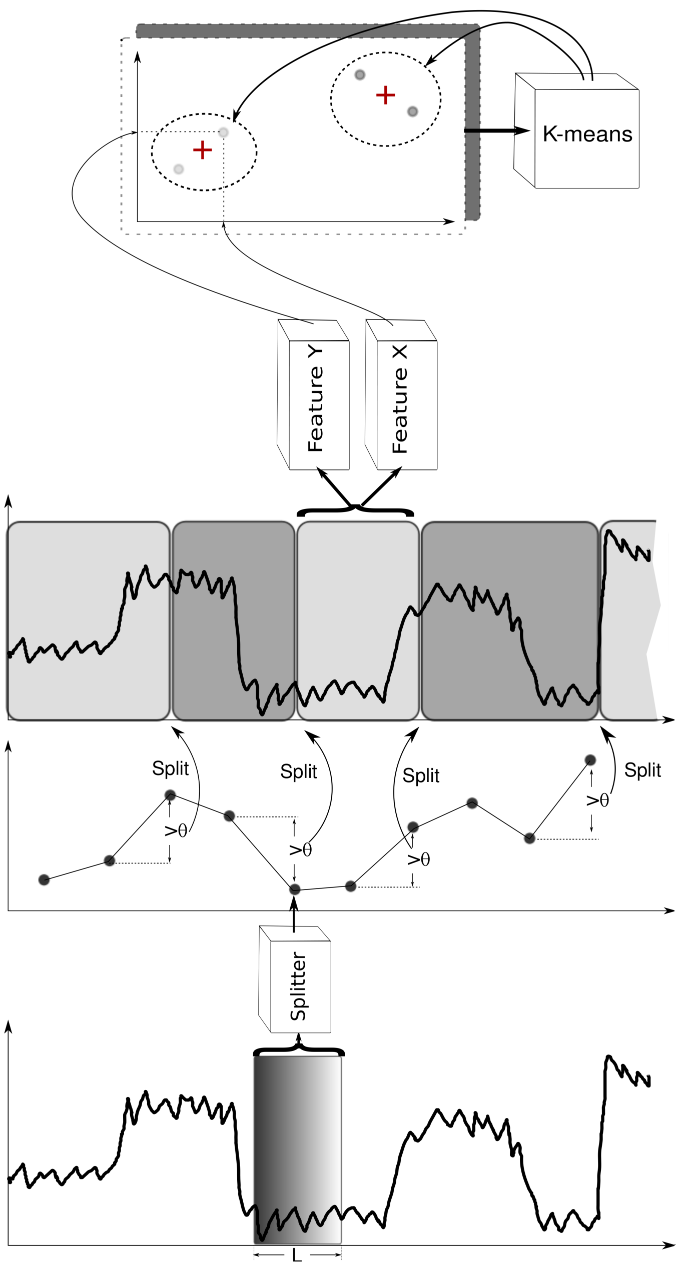

Figure 1. File segmentation in GP-zip2. The splitter tree is repeatedly applied to the data (seen as a 1–D signal) in a sliding window (bottom). The output in consecutive windows is compared: if the difference is higher than a threshold θ the data file is split at the window's position (middle). The blocks identified by the splitter tree are transformed into points in a 2–D space by the feature-extraction trees (top). During training, K-means then groups the points (blocks) into clusters (the dashed ellipses in the topmost plot) and the centroids of the clusters (crosses) are stored for later classification. After training, the same operations are performed, except that K-means is *not* invoked. Instead, the distance between a point and the nearest centroid is used to decide which compression algorithm to apply to a block (see text).

from objects in other clusters. The advantage is that the experimenter doesn't need to label the training set. Also, the approach does not impose constraints on the shape of the clusters.

Our objective is to first cluster the blocks found by the splitter tree in the training set and then to label each cluster with the compression algorithm that provides the best compression for the segments in the cluster. This allows us to later use the clusters found by K-means to perform classification of unseen data.

As we will explain in the next section, as part of the fitness evaluation we establish, by brute force, what is the best algorithm to compress each segment identified by the splitter tree in the training set. In order to choose the value of K for K-means in the classification of segments, we count the compression algorithms (out of the available ones) which were optimal for at least one of the segments.

Naturally, while we can tell K-means to group items in exactly K clusters, being unsupervised, K-means has no way of knowing what each cluster is meant to represent. So, it might produce clusters that are not suitable to decide which algorithm to use to compress unseen blocks of data. For example, at least in principle, K-means might find that text files naturally form two separate groups (judging from their two composite features), while perhaps there is a single best algorithm (say PPMD) for compressing them.

To circumvent this problem we don't act on the K-means algorithm, but we modify the composite features. That is, we ask GP to come up with two feature-extraction trees that lead K-means to cluster the segments in the training set in such a way that all segments in a cluster are optimally compressed by a different compression algorithm. If GP is successful, K-means is able to group blocks based on their compressibility with a specific algorithm.

A fitness function that guides GP to evolve composite features with the desired properties is described in the following section.

3.3. Fitness Evaluation

The fitness function has two equally-weighted components which evaluate the quality of the splitter tree and the quality of the feature-extraction trees, respectively.

After the splitter tree has segmented the training data, its fitness contribution can be computed. This is measured by calculating the maximum achievable compression ratio for the segments identified by the splitter, which in turn is computed by compressing each segment with all available compression algorithms. If each compression algorithm is represented by a function C_x and each segment is represented by S_i, we denote with $|C_x(S_i)|$ the length of the i-th segment when compressed with C_x. Then the fitness of the splitter tree can be expressed as:

$$f_{\text{splitter}} = 100 - \left(\frac{\sum_{i=1}^{n} \min_x |C_x(S_i)|}{\textit{File Size}} \times 100 \right) \tag{1}$$

where x ranges over the algorithms in our compression pool.

After file segmentation, the segments identified by the splitter tree are passed to the feature extraction trees, transformed into 2–D points and clustered by K-means as already explained. The information about optimal segment compression gathered in the evaluation of f_{splitter} is now re-used to label the clusters found by K-means. Labels are chosen according to the dominant algorithm in each cluster (we break ties randomly).

At this point the fitness contribution of the feature-extraction trees can be evaluated. This is based on the quality of the clusters produced by K-means, which is assessed by their *homogeneity* and *separation*.

The homogeneity of a cluster, $H(CL_i)$, is the proportion of data points – segments – that are optimally compressed with the algorithm that labels the cluster. Using the information that we obtained about the segments and their optimal compressions we can easily find the total number of segments that should be compressed with a particular compression model. Any deviations from this optimal value due to clusters containing extra members should be discouraged. Thus, we use a penalty term λ to deter clusters from acquiring too many extra members. The penalty is computed as follows:

$$\lambda = \sum_{i=1}^{K} \frac{|Seg(C_i)|}{100} \times \big[|mem(CL_i)| - |mem\_true\_label(CL_i)|\big], \tag{2}$$

where K is the total number of clusters, CL_i is the i-th cluster, $|mem(CL_i)|$ is the total number of members (segments) that belong to CL_i, $|mem\_true\_label(CL_i)|$ is the total number of members that belong to CL_i and are optimally compressed with the same compression algorithm that labels the cluster, and $|Seg(C_i)|$ is the total number of segments that have been optimally compressed with C_i (the compression algorithm that labels CL_i).

The contribution to the fitness of the feature-extraction trees deriving from cluster homogeneity is

$$f_{\text{Homogeneity}} = \frac{\sum_{i=1}^{K} H(CL_i) - \lambda}{K}. \tag{3}$$

Homogeneous clusters with objects far apart within a cluster extend the clusters' boundary and may lead to inaccurate classification of unseen objects. Also, clusters that overlap are not suitable. Ideal clusters are separated from each other and densely grouped near their centroids. This is why we also measure and reward the separation of the clusters. The Davis Bouldin Index (DBI) [3] is used for this purpose.

DBI is a measure of the nearness of the clusters' members to their centroids in relation to the distance between clusters' centroids. A small DBI index indicates well separated and grouped clusters. Therefore, if we subtract the DBI index from the fitness contribution associated to the homogeneity of clusters (Equation (3)), we push evolution to separate clusters (i.e., minimise the DBI). In other words, we treat the DBI as a penalty value.

In summary, we define the fitness of the feature extraction trees as follows:

$$f_{\text{Feature}-\text{Extraction}} = f_{\text{Homogeneity}} - DBI. \tag{4}$$

The total fitness of a program can then be expressed as:

$$f = \frac{f_{\text{Feature}-\text{Extraction}} + f_{\text{Splitter}}}{2}. \tag{5}$$

The performance of the feature-extraction trees depends on the output of the splitter tree. In certain conditions, the splitter tree may force them to produce classifications which are unlikely to be suitable to classify unseen data. Let us consider a specific example. Let us imagine that an individual's splitter tree finds only two blocks in a training set and that each block is best compressed by a different compression algorithm. With only two blocks, the feature extraction trees within that individual are likely to form two clusters. Consequently, the system will rate the homogeneity of the clusters as 100% and DBI index as minimum, resulting in a very high $f_{\text{Feature-Extraction}}$. However, each cluster has only one data member and, so, clusters are unlikely to be representative of unseen data.

To avoid pathologies of this kind, the system verifies whether the splitter tree has found a sufficiently large number of blocks. In particular, it penalises trees that obtain less than a minimum number of blocks for each cluster.

3.4. Training and Testing

The ability of GP-zip2 to find solutions that work well on unseen data depends crucially on the data used in training. Two main factors have been considered while designing the training set. Firstly, the training set has to contain enough data and enough diversity of data types to ensure the generality of the evolved compression algorithms. In deciding which data to use one should also consider what data types are likely to be compressed by the end users of the system. Secondly, we should keep in mind that the system will process the training set many times for each individual in each generation. Thus, the size of the training data should be small enough to keep run-time under control.

Table 2 summarises key properties of our training archive. This contains 11 different data types for a total size of 332KB. As we will see, this ensures the generality of the evolved solutions.

The output at the end of a GP-zip2 run consists of the clusters' prototypes, a splitter tree that detects different fragments within a stream of data and the two feature-extraction trees. Each cluster prototype is defined as the centeroid of the cluster's members. This information can be used to compress new data (independently of the GP system).

Since our objective is to evolve compression algorithms which are general-purpose and can be used on their own many times after evolution, the user is expected to run the system multiple times until it succeeds in evolving a solution which achieves adequate performance on the training set.

Testing/using the system involves compressing unseen files using the trees and clusters identified during the training phase. It should be noticed that our test set is completely independent of the training set. When a test file is processed, the file is divided into segments by the splitter tree as shown in Figure 1. These are then fed into the feature-extraction trees and projected into a two-dimensional space as was done during training and is illustrated in Figure 1. However, unlike during training, after this stage we do not invoke K-means. Instead, each block is assigned to the cluster whose centeroid is closest (in terms of Euclidean distance) to the block's two-dimensional feature vector. The block is then compressed with the compression algorithm associated to the closest centeroid.

Files can be compressed in two different ways: in *gluing mode* consecutive segments that have been assigned to the same compression algorithm are compressed as one (bigger) block; in *non-gluing mode* each segment is compressed independently. In our experiments the system tried both methods and used the one that provided the highest compression for each file. In most cases, the winning approach was the non-gluing one. (In our tests, the

Table 2. Summary of types and sizes of the files used for the training and testing of GP-zip2.

Phase	File types	Total size of file set
Training	pdf, exe, CPP code, gif, jpg, xls, ppt, mp3, mp4, txt, xml	332 KB
Testing	pdf, exe, CPP code, gif, jpg, xls, ppt, mp3, mp4, txt, xml, Unicode txt, accdb, tif, wmv, mov, ps, docx, Flv, HTML, lisp code, GUN	133.5 MB

difference in compression ratios between gluing and non-gluing was always $\leq 1\%$. Since the splitter tree does a good job at finding the boundaries between different data types, the system rarely uses the same compression method for many consecutive chunks.)

For comparison Table 2 also includes key properties of the test set we used to check the generality of evolved compression algorithms. This contains 22 different data types within 12 archives, for a total of *over 130MB of test data*. The test set contains both homogeneous and heterogeneous files. Furthermore, it contains file types similar to those used in the training set as well as new data types to which the algorithm has not been exposed during training. (More details on the test set are provided in Table 3.)

3.5. Search Operators

In GP-zip2, we used the standard genetic operators: sub-tree crossover, sub-tree mutation and reproduction. Naturally, the genetic operators take the multi-tree representation of individuals as well as the types of primitives into account.

There are several options for applying genetic operators to a multi-tree representation, but it is unclear which technique is best. In [29] it was argued that crossing over trees at different positions might result in swapping useless genetic material resulting in weaker offspring. On the contrary, [4] suggested that restricting the crossover positions is misleading for evolution. What's best appears to depend on the semantics of the representation.

In preliminary experiments we tried a variety of approaches and found that the following works well in GP-zip2. Let T_c^i be the c^{th} tree of individual i, where $c \in \{splitter, feature{-}extractor_x, feature{-}extractor_y\}$. The system selects an operator with a predefined probability for each T_c^i. In crossover, a restriction is applied so that splitter trees can only be crossed over with splitter trees. However, the $feature{-}extractor_x$ tree of one parent can be crossed over with either the $feature{-}extractor_x$ or the $feature{-}extractor_y$ of the other. The $feature{-}extractor_y$ trees are treated symmetrically.

3.6. Decompression Process

Since GP-zip2 divides the data into segments and compresses them with different models, it is necessary for the decompression process to know which segment was compressed with which compression algorithm. Thus, GP-zip2 files start with a header which provides this information.

The header consists of a sequence of 2-bytes words corresponding to successive blocks of input data. Each of these words is divided into two parts. The first 4 bits encode the algorithm used to compress a segment; the remaining 12 bits encode the length of the compressed segment. A special character marks the end of the header. Note that the size of the header depends on the number of identified segments. However, generally, the header's size is insignificant in comparison with the size of the original (uncompressed) file.

4. Experiments

4.1. Testing strategy and data

The evaluation of compression algorithms can be either analytical or empirical [2]. Generally, the analytical evaluation is expressed as the compression ratio versus the entropy of the source, which is assumed to belong to a specific class [2]. However, such an evaluation is not accurate and may not be broadly applicable. In an empirical evaluation, instead, the algorithm is tested with a collection of real files. The difficulty of this method resides in the ability of generalising compression results beyond the domain of the experimentation.

One might imagine that for the perfect evaluation of an algorithm one should test it with all possible files and find the average compression ratio over them. This approach, however, is not only computationally unfeasible, but also theoretically flawed: as indicated in section 1 there is no compression algorithm that can compress all files types effectively [33]. Therefore, the empirical evaluation really needs to focus on the files that are likely to be compressed by users. Other criteria for selecting experimental files are detailed in [2]: here we tried to satisfy them all. Of course, the more files are included in the experiments, the more accurate the characterisation of the behaviour of the algorithm is, but also the more computational intensive the testing is.

Table 3 presents a list of the 12 archives that have been used in the experiments. Together these are over 130MB of test data, which we felt is reasonably representative of typical computer use, while still remaining computationally feasible with the hardware available to us. Archives contain both homogeneous and heterogeneous sets of data. The files within the archives were chosen in such a way to ensure that each archive contains a unique combination of 22 file distinct types (see Table 2). The main aim of the tests was to assess the system's performance under a variety of circumstances and determine whether the compression algorithms evolved by the system can be used as general-purpose file compressors.

GP-zip2 was compared against Arithmetic Coding, Lempel-Ziv-Welch, unbounded Prediction by Partial Matching, Run Length Encoding, Winzip, and WinRar.

In addition, we included in the comparison a simple *baseline method* suggested by one of the reviewers of this article which requires no machine learning techniques. The baseline method applies all compression algorithms available to GP-zip2 on each individual file in an archive. For each file, it saves the shortest compressed version. Finally, it places all resulting compressed files into an archive, which is then taken to be the compressed version of the original archive.

For the reasons explained in Section 2.3 Boolean Minimisation was omitted both as a comparison algorithm and as a primitive available to the system.

Table 3. Details of the types and sizes of the files present in the archives used for *testing* GP-zip2.

Archive	Files	Size (KB)
Text	English translation of The Three Musketeers by Alexandre Dumas, Anne of Green Gables by Lucy Maud Montgomery, 1995 CIA World Fact Book	4,822
Exe	DOW Chemical Analysis program,Windows95/98NetscapeNavigat,Linux 2.x, PINE e-mail program	4,824
Archive1	Mp3 Music, Excel sheet, Certificate card replacement form PDF `http://www.padi.com/`, Anne of Green Gables by Lucy Maud Montgomery (text file)	1,474
Archive2	PowerPoint slides, JPEG file, C++ source code, Mp4 Video (5 seconds)	2,458
Archive3	GIF file ,Unicode text file (Arabic language), GP-zip* executable file, Xml file	1,384
Archive4*	GP-symbolic regression system, MS Access database file, Text file.	34,069
Archive5*	JPG (picture of faces), Word file (résumé), Tif (Fax cover), WMV movie 6:25 minutes	19,309
Archive6	Windows95/98 Application, JPG picture of sea and sky, Text book, Tif picture Lena, Resume.xml	2,518
Archive7	Exe application (file splitter program), JPG picture, Text file (book), XML database	694
Archive8*	Java code (Tiny_GP), Mov (high definition file movie), PS file (Journal paper)	56,883
Archive9*	PDF file, Docx Word 2007, FLV video 3:09 minutes	2,793
Canterbury corpus*	English text, fax image, C code, Excel sheet, Technical writing, SPARC exe, English poetry, HTML, lisp code, GUN Manual Page, play text.	2,276
		Total size 133.5 MB

* Contains data types to which the algorithm was not exposed during training.

Table 4. Tableau of the GP-zip2's parameter settings used in our experiments.

Parameter	*Value*
Population size	100
Maximum number of generations	30
Crossover probability	90%
Mutation probability	5%
Reproduction probability	5%
Tournament selection with tournaments of size	5
Initialisation	ramped half-and-half
Window size for splitter tree (L)	100
Window step for splitter tree (S)	50
Splitter tree threshold (θ)	10
Number of feature-extraction trees	2

The experiments presented here were performed using the parameter values shown in Table 4. Parameters were set partly by using values commonly found in the GP literature and by performing a variety of preliminary experiments and selecting the values that gave us good results while keeping the processing time under control. More information on our choice of parameter values is provided in Section 4.5.

It is practically impossible to determine the best fitness level that could be reached with realistic training sets in GP-zip2. So, we decided not to set any termination condition for the system. However, based on the experience gained in preliminary runs, we set the maximum number of generations for runs to a value that ensured improvements to the best-of-run fitness had almost stopped by the last generation.

4.2. GP-zip2 behaviour across runs and test files

GP-zip2's performance has been measured through 15 independent runs, each of which evolved a compression system. Below we present and discuss the results.

Let us start with the traditional fitness plots. Plots of the overall fitness of the best-of-generation program averaged across runs are shown in Figure 2 together with the averages of the fitness components associated with the splitter tree and with the feature-extraction trees. As the "overall fitness" plot illustrates, generation after generation GP-zip2 successfully and reliably (note the small standard error of the mean) produces a sequence of ever improving best-of-generation individuals.

Looking at the two components of the fitness, we see that the fitness of the splitter tree starts relatively high and improves at a lower rate than the other component. This is because the splitter's fitness represents the compression ratio (for the *training set*) obtained by testing all compression algorithms on each block and picking the best result. So, even with random blocks (such as those produced by the splitter tree at generation 1), we can still achieve good compression. However, generation after generation this fitness component improves, as better and better splitter trees are found. The feature-extraction fitness shows more dynamics. Part of this is, of course, due to the fact that feature extractors are initially random and evolution needs time to improve them. However, the dynamics is also determined by the fact that the performance of the feature-extraction trees depends on the quality of the segments extracted by the splitter tree. So, improvements to the latter tend to produce subsequent positive effects on the former.

While this is encouraging, the acid test for GP-zip2 is clearly its performance on unseen data. Table 5 reports the results obtained by applying each of the 15 best-of-run programs obtained in our runs to compress the 12 different archives in the *test set*. More specifically, the second column lists the average compression achieved for each archive by our best-of-run programs. The third column shows the corresponding standard deviations. The fourth column reports the compression ratios obtained by the best overall best-of-run program on each archive. The fifth column illustrates the performance of the worst overall best-of-run algorithm on each archive. Finally, the last two columns show the best and worst compression ratios achieved by any of the 15 best-of-run programs evolved by GP-zip2.

There are several things worth noticing in this table. For example, we find that, as illustrated by the second column, runs of GP-zip2 were able to evolve compression algorithms which generally do a very good job at compressing multi-file archives. Also, as shown by the relatively small standard deviations in column 3 and the worst performance figures listed in columns 5 and 7, in these tests GP-zip2 was very reliable, producing effective compression algorithms *in all runs*.

We should note that the standard strategy used in GP practice to identify solutions to a problem is to perform multiple runs with that problem and then pick the best best-of-run program as its solution. In the case of GP-zip2, the compression algorithm a user would likely choose after our set of 15 runs is the best-of-run program that showed the best average performance over the 12 archives in the test set. If we look at the performance of such a program (reported in column 4 of the table), we find that this strategy is certainly a very good one, the designated solution being 3.75% better than average while also staying very general (the program produces above-average compression ratios in 11 out of 12 archives).

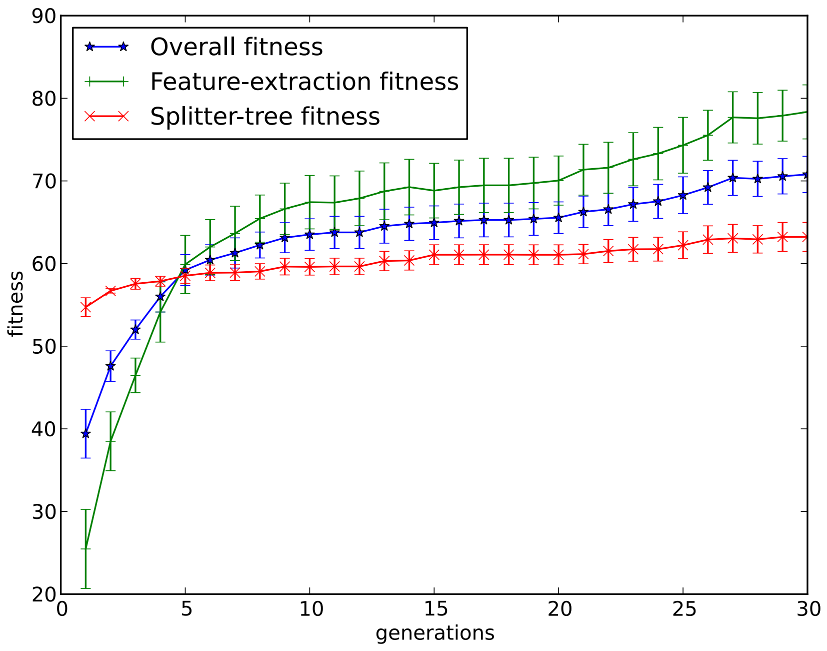

Figure 2. Evolution of best-of-generation fitness during our experiments. The two components of the fitness (the average fitness associate with the splitter tree and the one associated with the feature-extraction trees) are also plotted. Bars represent the standard errors of the means.

It is clear that the strategy is effective but relatively wasteful since it completely discards all other best-of-run results. In the case of GP-zip2, there is also an alternative, albeit more expensive, strategy which avoids wasting all other best-of-run algorithms: when given a file to compress, one can execute each of the 15 best-of-run programs evolved by GP-zip2 on the data and pick the result with the best compression ratio. As illustrated in column 6 of table 5, this approach can further improve results. Naturally, this requires a compression effort 15 times bigger than for a single program. However, when the data needs to be compressed only once (as, for example, in large scale software distributions) and decompressed many times, the extra compression overhead associated with this technique might be acceptable.

4.3. GP-zip2 vs standard compression algorithms

Table 6 shows the compression ratios obtained on each of our 12 test archives by standard compression algorithms, the baseline method described in Section 4.1, the best best-of-run program evolved by GP-zip2 and the strategy highlighted above of testing all 15 best-of-run programs sequentially. As one can see from the mean compression ratios at the bottom of the table, both GP-zip2 compression systems outperform the other seven algorithms. In fact, the sequential GP-zip2 strategy is better than them on 8 out of 12 archives and second

Table 5. Archive-by-archive analysis of the compression ratios obtained by the best-of-run programs evolved in 15 independent runs of GP-zip2.

File	Compression Average for all experiments	Standard Deviation	Best Run overall	Worst Run overall	Best Compression in all experiments	Worst Compression in all experiments
Archive1	32.22	5.58	35.36	20.42	35.98	17.13
Archive2	3.50	0.30	3.34	3.71	3.93	2.91
Archive3	60.27	13.78	66.66	20.67	66.69	20.67
Archive4	90.20	2.75	91.46	91.08	91.65	80.42
Archive5	8.96	0.68	9.73	8.44	9.90	7.42
Archive6	53.68	12.26	59.89	57.32	59.90	13.15
Archive7	70.88	3.35	72.76	70.52	72.76	58.98
Archive8	18.94	0.77	19.65	17.26	19.95	17.26
Archive9	3.01	0.41	3.71	3.24	3.71	2.25
Canterbury	68.35	13.65	77.94	46.18	80.76	35.37
Exe	59.33	9.03	67.86	40.20	67.86	38.77
Text	74.25	11.49	80.29	52.79	80.44	44.47
Mean	45.30	6.17	49.05	35.99	49.46	28.23
StdDev	30.53	5.51	32.61	28.27	32.74	24.07

in a further 3 while the best-overall GP-zip2 strategy is better in 7 archives and second in a further 3. Only WinRar-Best can get anywhere near our evolved algorithms, beating them only on 4 archives. Note that WinRar is a proprietary compression algorithm whose internal details are not publicly available. This is done so as to maintain its competitive edge over other commercial and open-source techniques. So, the fact that GP-zip2 can improve on WinRar is a significant achievement, suggesting, in fact, the human-competitiveness (in the sense of [22]) of the results produced by the system. The reason why GP-zip2 can achieve better compression ratios is that it makes use of the available compression algorithms (i.e., AC, PPMD, LZW, RLE, Bzip2) in such a way to best match the nature of the data. Also, GP-zip2 can choose to not compress some parts of the data (which may be necessary when fragments of data are not compressible).

The sequential GP-zip2 strategy was not overall best on the *Text, Archive4, Archive5* and *Canterbury* data sets. Let us briefly discuss the reasons for this. Firstly, we should note that the *Text* archive is composed of homogeneous sets of files. Thus, it is best compressed with a single model. Secondly, the Canterbury corpus is often used as a reference for comparison of compression algorithms. Thus, highly optimised compression software is often tuned to maximise compression on such a dataset (we did not do this in GP-zip2), with potentially deleterious consequences for other data types. Also, 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). So, it is not surprising to see that GP-zip2 was not best on the Canterbury corpus. Finally, we should note that a big part of Archive4 consists of an Access database file — a type of data which was not part of the training set — and that Archive5 included 6 minutes and 25 seconds of WMV video to which GP-zip2 had not been exposed during training.

Table 6. Performance comparison (compression ratio %) between GP-zip2 and other techniques.

File	Winzip - Bzip2	WinRar - Best	PPMD	LZW	RLE	AC	Baseline Method	GP-zip2 Best	GP-zip2 Sequential
Archive1	32.90	34.04	33.32	1.14	-15.47	9.98	33.90	35.36	35.98
Archive2	3.90	3.19	3.90	-43.98	-13.75	0.70	4.16	3.34	3.93
Archive3	64.49	65.99	64.36	43.62	9.16	27.62	65.78	66.66	66.69
Archive4	90.96	93.13	90.73	-24.74	-2.37	58.25	91.00	91.46	91.65
Archive5	7.98	11.17	8.64	-36.17	-7.05	2.73	8.75	9.73	9.90
Archive6	50.61	56.33	49.07	-3.66	-7.00	11.89	51.83	59.89	59.90
Archive7	68.64	64.21	70.76	33.62	-6.99	33.98	71.19	72.76	72.76
Archive8	15.68	14.41	18.74	-47.74	-7.46	5.40	18.73	19.65	19.95
Archive9	2.94	3.31	2.28	-41.54	-11.97	0.30	3.30	3.71	3.71
Canterbury	80.48	85.15	81.19	15.61	6.55	41.41	81.83	77.94	80.76
Exe	57.86	64.68	61.84	35.75	-4.66	17.47	62.19	67.86	67.86
Text	77.88	81.42	79.95	56.47	-11.34	37.77	80.17	80.29	80.44
Mean	46.20	48.09	47.07	-0.97	-6.03	20.62	47.73	49.05	49.46
StdDev	32.20	33.24	32.43	37.67	7.20	18.90	32.50	32.60	32.74

Table 7. Details on an additional test set including archives which contained file types not in the training set.

Archive	Files	Size
Archive10	PDF, doc, xlsx	229KB
Archive11	Ram, bmp, ppt	6.35MB
Archive12	Gif, ppt, ps, ram	2.57MB
Archive13	Mp3, hh, gif	3.27MB
Archive14	pdf, tif, dat	339KB
Calgary	Standard benchmark	2.66MB
Worms2Demo10OctA	Free game Demo - Standard benchmark	10.1MB
Chess MAC	Free game Demo	4.95MB

The performance of GP-zip2's solutions was very close to that of its best competitors in the few cases where it did not beat them. However, the slight reduction in performance on Archive4 and Archive5 prompted us to further verify the generality of GP-zip2's evolved solutions with files which presented more marked differences from the training set. To do so we created a second test set containing 8 new files. Five of these files were archives containing a combination of seen and unseen file types. New file types included: *doc, xlsx, ram* and *dat*. Also, we included two popular benchmarks in the compression field — *Calgary* and *Worms2Demo* — as well as *ChessMAC* which is a chess game for the Macintosh platform. We should note that Calgary is an old benchmark containing file types that are not in common use anymore, and that the training set did not include any Macintosh executables.

Table 7 lists the file types and archive sizes for the new test set and Table 8 reports the performance of the best evolved program and the sequential strategy on it. These further

Table 8. Performance comparison (compression ratio %) between GP-zip2 and other techniques on the test set in Table 7.

File	Winzip - Bzip2	WinRar - Best	PPMD	LZW	RLE	AC	Baseline Method	GP-zip2 Best	GP-zip2 Sequential
Archive10	26.89	28.81	27.96	-8.47	2.34	8.30	28.54	29.29	30.96
Archive11	24.59	21.73	23.58	-19.37	-10.92	2.68	24.74	27.57	27.63
Archive12	42.94	45.47	43.59	-7.06	-0.20	7.39	44.48	47.25	47.90
Archive13	4.77	4.57	3.65	-38.54	-11.18	0.92	4.77	5.70	5.70
Archive14	5.33	8.88	4.72	-37.17	10.52	1.18	5.98	9.20	10.88
ChessMAC	68.88	71.37	71.28	48.14	26.39	32.73	75.89	72.35	75.32
Worms2Demo	2.32	2.71	2.85	-41.74	-11.46	0.16	2.01	2.41	2.46
Calgary	73.28	70.57	75.40	48.20	3.88	32.91	75.85	69.84	70.73
Mean	31.12	31.76	31.63	-7.00	1.17	10.78	32.78	32.95	33.95
StdDev	28.27	27.98	29.38	36.55	12.21	13.92	30.19	27.77	28.36

tests confirm that GP-zip2's evolved solutions generalise well. Both GP-zip2's approaches were best in 5 out of 8 archives, second best in one case (ChessMAC), and not too far from the best in the remaining two cases (Calgary and Worms2Demo). This is a remarkable result given that the algorithms had to deal with a variety of new file types.

Let us now turn our attention to the computational effort involved in the use of GP-zip2 and its evolved solutions. GP-zip2 runs lasted between 300 and 420 minutes on an AWS cloud computing virtual PC with 2 EC2 Compute Units.[2] On average GP-zip2 training required 388 minutes to evolve a competitive solution in our 15 independent runs (the standard deviation of the training time is approximately 37 minutes). Also, the best-of-run individuals produced by GP-zip2 took only *between 5 and 10 minutes to perform the compression of the 133.5MB of data in our larger test set* (mean 472 seconds, standard deviation 109 seconds). In other words, on average, *GP-zip2 evolved programs can compress a reasonable 0.28MB per second.*

For comparison, in table 9, we show the processing times required by standard compression algorithms when compressing our test set using the same machine. It is clear that GP-zip2 data compressors are substantially slower than most other algorithms, the closest being PPMD which can compress 0.42MB per second. However, we should note that most of the algorithms in the table have highly optimised implementations, while the optimisation of the implementation of GP-zip2 was a secondary objective in our research agenda. Also, while GP-zip2 is slower than most, it is not impractical. As is, it could compress the contents of a 4.7GB data DVD in less than 8 hours, e.g., overnight. Also, it is likely that an optimised multi-core implementation of GP-zip2 could bring its processing times on par with its commercial rivals, while delivering superior compression.

4.4. Analysis of Evolved Solutions

It is interesting to look at the behaviour of the best-of-run solutions evolved by GP-zip2. One way to do this is to study how they split archives of files and what algorithms they

Table 9. Processing times required by different compression algorithms when compressing our 133.5MB test set.

Algorithm	Time in seconds	Megabyte/second
WinZip-bzip2	103	1.30
WinRar-Best	116	1.15
PPMD	320	0.42
LZW	54	2.49
RLE	1.33	100.6
AC	154	0.87
GP-zip2	472 ± 109	0.28

allocate to the data segments. Figure 3 illustrates the typical behaviour of the best-of-run algorithms on a small test archive (Archive10, see table 7) containing a PDF, a Word document and an Excel spreadsheet.

As shown in the figure, there is significant consistency in the behaviour of solutions. For example, as illustrated in Figure 3(d) most compressors treat the Excel file and to a lesser extent also the PDF file as a single block, while attempting to fragment the DOC file into multiple block, which is reasonable given the variability in its byte-values (see Figure 3(b)). Also, note the twin peaks in Figure 3(d) corresponding to the transition from PDF to DOC and from DOC to Excel, respectively, and the secondary peak in the middle of the PDF file which is precisely aligned with a change in its byte frequency distribution.

Clearly, GP-zip2 detects significant transitions in byte statistics and correctly attempts to exploit them.

4.5. GP-zip2's Parameter Settings

Like other GP systems, GP-zip2 is controlled by a certain number of parameters (see table 2). Naturally these parameters have a significant influence on the system's performance. The population was kept very small (100 individuals) and runs short (30 generations) because of the computational load involved in each fitness evaluation. The other GP parameters in the upper part of Table 4 were initially set to values commonly found in the literature and then adjusted in preliminary runs so as provide a reasonable trade-off between run time and quality of the evolved solutions. For example, we used relatively large tournaments in relation to the population size because of the small number of generations adopted.

During these experiments we also identified reasonable regions in parameter space for the GP-zip2-specific variables in the bottom part of Table 4. In relation to the parameter θ associated with the splitter tree we found that the value 10 was a reasonable assignment but also that small changes in θ did not significantly affect performance, which isn't unexpected since evolution can easily change the magnitude of the outputs of the splitter tree.

To avoid boundary effects, in the experiments we always set $S = 50$ so that consecutive windows overlapped, and we did not need to revisit this choice. However, we found

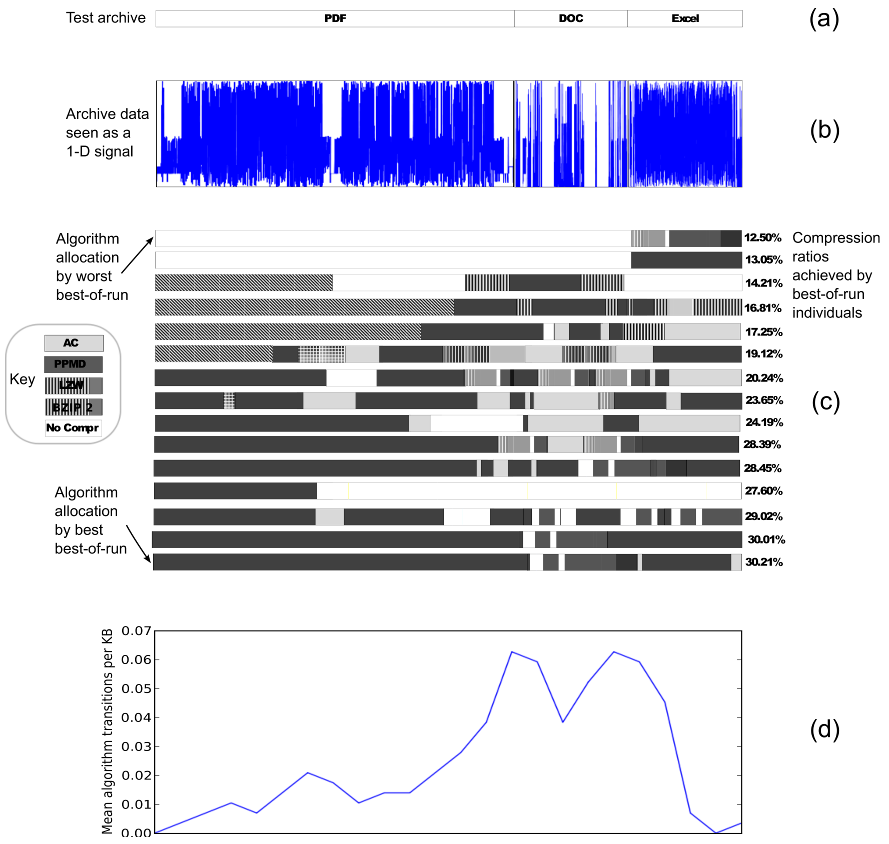

Figure 3. Analysis of GP-zip2's allocation of compression algorithms to a test archive. The test archive, represented in (a) as a strip, contains three files: a PDF (filling approximately 60% of the archive), a Word document (approx 20% of the archive) and an Excel document (approx 20% of the archive). As illustrated in Figure 1, archives are processed as 1–D signals by the splitter and feature extraction trees in GP-zip2 individuals. The signal corresponding to the test archive in (a) is plotted in (b). Data in (b) were sub-sampled by factor of 10 for clearer visualisation. The allocation of compression algorithms to different parts of the test archive by the different best-of-run individuals produced in our runs is shown in (c). Algorithm allocations in (c) have been sorted by compression ratio to highlight similarities and differences in behaviour. In (d) we plot the number of compression-algorithm transitions along the length of the test archive averaged across the algorithm allocations shown in (c). The transitions plot in (d) has been smoothed for a clearer visualisation.

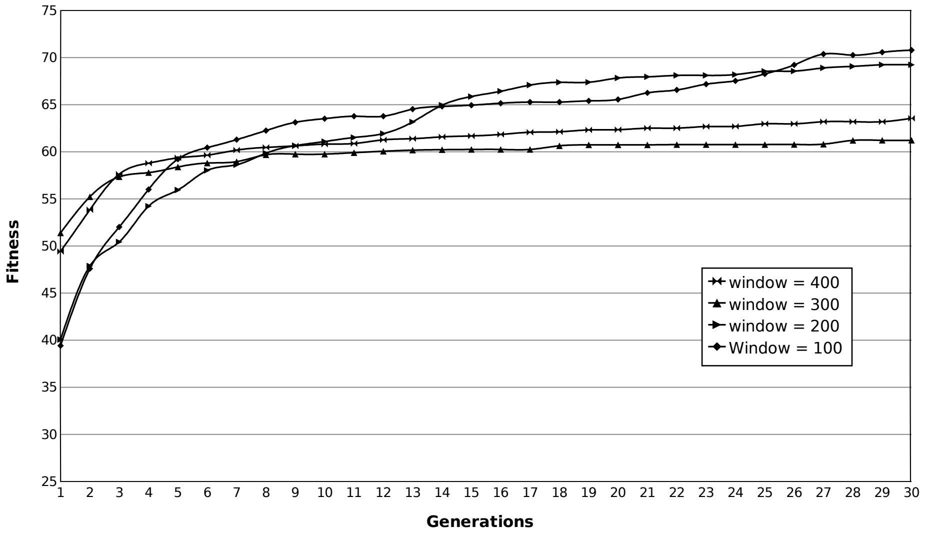

Figure 4. Mean best fitness vs. generation for different splitter tree's window sizes (L). Data-points represents averages over in 15 independent runs.

that the value of L was quite important. An excessively large window may conceal useful statistical variations within the data, resulting in too few blocks that can be compressed less effectively. An excessively small window, instead, will make the statistics less reliable (being based on fewer data-points), which, in turn, leads to an excessive fragmentation of the data, with potentially poor compression results. Also, since each fragment is independently processed by the feature extraction trees, excessive fragmentation may lead to significantly longer evolution times.

The preliminary experiments suggested that a value of L of a few hundred bytes was appropriate. Within this region we performed a more systematic exploration of the performance as a function of L. Figure 4 show GP-zip2 performance in 60 different runs with four different sliding window sizes (15 runs for each window size). The plots suggest that higher end-of-run fitness is associated with smaller windows. Indeed, both the Kolmogorov-Smirnov test and the Wilcoxon signed-rank test indicate that $L = 100$ and $L = 200$ are statistically significantly superior to $L = 300$ and $L = 400$. Based on the average fitness of $L = 100$ being slightly superior to that of $L = 200$, we, therefore, decided to go for $L = 100$.

The last parameter we need to look at is the number of feature extraction trees used by GP-zip2 (see last row of Table 4). It is reasonable to question whether the choice of using two feature-extraction trees rather than some other number is optimal. On the one hand, one would expect that using more feature-extraction trees would provide more information to GP-zip2's classification algorithm (K-means) which, consequently, would be able to provide a more accurate classification. On the other hand, increasing the dimensionality of the feature space makes fitness evaluation slower. Furthermore, the size of the search space increases exponentially with the number of feature trees. So, one should expect that the optimal number of feature-extraction trees for GP-zip2 will be the result of a compromise between these antagonistic forces.

To study this trade-off, we varied the number of feature-extraction trees from 1 to 3. Performance was measured via 15 independent runs (5 runs for each dimension). For each run we measured the quality of the evolved program by calculating average compression ratio on the test set (results not reported). We found that the system's performance improved significantly as the number of dimensions increases from 1 to 2. However, there was almost not further improvement when we used three feature extraction trees. Since, training time was, instead, significantly increased, we decided to set the number of feature-extraction trees to two.

5. Conclusions

An ideal compression system should be able to identify incompatible data fragments (within and across files) and allocate the best possible compression model for each fragment quickly and in such a way to minimise the total size of the compressed version of the file. Here we have proposed a system, GP-zip2, which we feel is one step closer to this ideal general-purpose compression model.

The GP-zip and GP-zip* systems we developed in earlier research [17, 18] influenced the development of GP-zip2. For example, we adopted the primitives, the block splitting and block gluing strategies developed earlier. However, in order to overcome the severe limitations of such systems in terms of computational load during the compression phase, we completely re-designed the decision making and learning strategies for GP-zip2.

Our guiding idea was to divide the task into three simpler ones: splitting files into blocks (using a function especially evolved for that task), extracting features from the blocks (using feature-extractors especially evolved so as to best relate the characteristics of the blocks to their compressibility with the algorithms in the primitive set), and finally classifying the segments into different families based on their compressibility features.

This approach has produced a system that outperforms seven state-of-the-art compression algorithms, including the well know WinZip/biz2 and WinRar. It comes top on heterogeneous files and is never too far behind the best with other types of data. In the highly competitive world of lossless compression, where differences in compression ratios of fractions of a percent can make the difference between widespread use and obsolescence, we feel that the results obtained by GP-zip2 are remarkable and satisfy several of the criteria for human-competitiveness proposed by Koza [22].

In addition to providing excellent compression, GP-zip2 has other advantages. The division of data files into blocks, for example, allows one to 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 or to create efficient compressed file systems. Also, most operations in GP-zip2 can be parallelised thereby allowing the full exploitation of the power of modern multi-core CPUs (e.g., different cores can decompress separate blocks).

Although GP-zip2 has achieved good results, its performance depends on the knowledge it acquires during evolution. Thus, users need to carefully select the training set according to their needs. A significant advantage with our method, from that point of view, is that the user is not required to manually split data files optimally and then label them with a compression algorithm to create a training set (all of which would be extremely difficult to do). Instead, everything is delegated to evolution and the K-means algorithm.

Naturally, on an ordinary PC, GP-zip2 runs require several hours to complete and this is certainly a disadvantage over traditional systems, although not a long lasting one since, once an acceptable compressor is evolved, further evolution is not required. Another disadvantage of GP-zip2 is that evolved solutions are still significantly slower than standard compression algorithms.

In future research we will concentrate on this last aspect of GP-zip2 as further substantial improvements can be expected. Furthermore, in the future we will investigate the use of different interpretations for the raw data in files (i.e., looking at them as sequences of 2-byte or 4-byte integers). Also, clustering signals using more sophisticated classification techniques is likely to provide further improvements to the system's performance. In addition, since GP-zip2 tends to exploit and outperform the compression models available to it, it is reasonable to expect that by providing a larger number of compression models to GP-zip2 further improvements in performance might be obtained. Another interesting research avenue is the idea of storing the state of GP runs in such a way to make it possible to restart evolution if it becomes apparent that the current best compression model is unsuitable for new data files.

Acknowledgements

The authors would like to thank the editor-in-chief and the anonymous reviewers for their thoughtful, constructive and supportive comments. These have helped us enormously in improving this paper, including uncovering and correcting an important mistake.

Notes

1. To be fair, in the bzip2 algorithm some regularities may be captured in particularly large files as a byproduct of its splitting files into large blocks for the purpose of keeping its memory footprint under control. For example, at the highest compression setting bzip2 splits files into 900KB blocks. However, each block is processed via exactly the same stages of analysis and compression.
2. One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

References

1. Z. Arnavut. Move-to-front and inversion coding. In *Data Compression Conference, 2000. Proceedings. DCC 2000*, pages 193–202, 2000.
2. R. Arnold and T. Bell. A corpus for the evaluation of lossless compression algorithms. In *Data Compression Conference, 1997. DCC'97. Proceedings*, pages 201–210, 1997.
3. J. Bezdek and N. Pal. Some new indexes of cluster validity. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 28(3):301–315, 1998.

4. N. Boric and P. A. Estevez. Genetic programming-based clustering using an information theoretic fitness measure. In D. Srinivasan and L. Wang, editors, *2007 IEEE Congress on Evolutionary Computation*, pages 31–38, Singapore, 25-28 Sept. 2007. IEEE Computational Intelligence Society, IEEE Press.
5. M. Burrows and D. Wheeler. A block-sorting lossless data compression algorithm. *Digital SRC Research Report*, 1994.
6. J. Cleary and I. Witten. Data Compression Using Adaptive Coding and Partial String Matching. *IEEE Transactions on Communications*, 32(4):396–402, April 1984.
7. J. G. Cleary, W. J. Teahan, and I. H. Witten. Unbounded length contexts for PPM. In *Data Compression Conference*, pages 52–61, 1995.
8. G. V. Cormack and R. N. Horspool. Data compression using dynamic markov modelling. *Comput. J.*, 30(6):541–550, 1987.
9. A. Fukunaga and A. Stechert. Evolving nonlinear predictive models for lossless image compression with genetic programming. In J. R. Koza, W. Banzhaf, K. Chellapilla, K. Deb, M. Dorigo, D. B. Fogel, M. H. Garzon, D. E. Goldberg, H. Iba, and R. 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.
10. S. W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theory*, IT-12:399–401, 1966.
11. T. Haynes, S. Sen, D. Schoenefeld, and R. Wainwright. Evolving a team. In E. V. Siegel and J. R. Koza, editors, *Working Notes for the AAAI Symposium on Genetic Programming*, pages 23–30, MIT, Cambridge, MA, USA, 10–12 Nov. 1995. AAAI.
12. J. He, X. Wang, M. Zhang, J. Wang, and Q. Fang. New research on scalability of lossless image compression by GP engine. In J. Lohn, D. Gwaltney, G. Hornby, R. Zebulum, D. Keymeulen, and A. 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.
13. W. H. Hsu and A. E. Zwarico. Automatic synthesis of compression techniques for heterogeneous files. *Softw. Pract. Exper.*, 25(10):1097–1116, 1995.
14. D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.
15. A. Kattan. Universal lossless data compression with built in encryption. Master's thesis, School of Computer Science and Electronic Engineering, University of Essex, 2006.
16. A. Kattan. *Evolutionary Synthesis of Lossless Compression Algorithms: the GP-zip Family*. PhD thesis, School of Computer Science and Electronic Engineering, University of Essex, October 2010.
17. A. Kattan and R. Poli. Evolutionary lossless compression with GP-ZIP. In J. Wang, editor, *2008 IEEE World Congress on Computational Intelligence*, Hong Kong, 1-6 June 2008. IEEE Computational Intelligence Society, IEEE Press.
18. A. Kattan and R. Poli. Evolutionary lossless compression with GP-ZIP*. In M. Keijzer, G. Antoniol, C. B. Congdon, K. Deb, B. Doerr, N. Hansen, J. H. Holmes, G. S. Hornby, D. Howard, J. Kennedy, S. Kumar, F. G. Lobo, J. F. Miller, J. Moore, F. Neumann, M. Pelikan, J. Pollack, K. Sastry, K. Stanley, A. Stoica, E.-G. Talbi, and I. Wegener, editors, *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, pages 1211–1218, Atlanta, GA, USA, 12-16 July 2008. ACM.
19. A. Klappenecker and F. U. May. Evolving better wavelet compression schemes. In A. F. Laine, M. A. Unser, and M. V. Wickerhauser, editors, *Wavelet Applications in Signal and Image Processing III*, volume 2569, San Diego, CA, USA, 9-14 July 1995. SPIE.
20. J. Koza. *Genetic programming: on the programming of computers by means of natural selection*. The MIT press, 1992.
21. J. R. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs*. MIT Press, Cambridge Massachusetts, May 1994. ISBN 0-262-11189-6,746 pages.
22. J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, J. Yu, and G. Lanza. *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*. Kluwer Academic Publishers, 2003.
23. T. Krantz, O. Lindberg, G. Thorburn, and P. Nordin. Programmatic compression of natural video. In E. Cantú-Paz, editor, *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 301–307, New York, NY, July 2002. AAAI.
24. T. Kuthan and J. Lansky. Genetic algorithms in syllable-based text compression. In J. Pokorný, V. Snásel, and K. Richta, editors, *DATESO*, volume 235 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.
25. G. G. Langdon. Arithmetic coding. *IBM J. Res. Develop*, 23:149–162, 1979.

26. S. Luke and L. Spector. Evolving teamwork and coordination with genetic programming. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. L. Riolo, editors, *Genetic Programming 1996: Proceedings of the First Annual Conference*, pages 150–156, Stanford University, CA, USA, 28–31 July 1996. MIT Press.
27. E. Lutton, J. Levy-Vehel, G. Cretin, P. Glevarec, and C. Roll. Mixed IFS: Resolution of the inverse problem using genetic programming. *Complex Systems*, 9:375–398, 1995.
28. E. Lutton, J. Levy-Vehel, G. Cretin, P. Glevarec, and C. Roll. Mixed IFS: Resolution of the inverse problem using genetic programming. Research Report No 2631, Inria, 1995.
29. D. P. Muni, N. R. Pal, and J. Das. A novel approach to design classifier using genetic programming. *IEEE Transactions on Evolutionary Computation*, 8(2):183–196, Apr. 2004.
30. P. Nordin and W. Banzhaf. Programmatic compression of images and sound. In J. R. Koza, D. E. Goldberg, D. B. Fogel, and R. 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.
31. F. Oroumchian, E. Darrudi, F. Taghiyareh, and N. Angoshtari. Experiments with persian text compression for web. In *WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers & posters*, pages 478–479, New York, NY, USA, 2004. ACM.
32. J. Parent and A. Nowe. Evolving compression preprocessors with genetic programming. In W. B. Langdon, E. Cantú-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.
33. I. M. Pu. *Fundamental Data Compression*. Butterworth-Heinemann, Newton, MA, USA, 2005.
34. D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York Inc, second edition, 2004.
35. A. Sarafopoulos. Automatic generation of affine IFS and strongly typed genetic programming. In R. Poli, P. Nordin, W. B. Langdon, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'99*, volume 1598 of *LNCS*, pages 149–160, Goteborg, Sweden, 26-27 May 1999. Springer-Verlag.
36. K. Sayood. *Introduction to Data Compression*. Morgan Kaufmann Publishers, San Francisco, CA, USA, second edition, 2000.
37. J. Seward. Bzip2. Website, 2009, Nov. `http://www.bzip.org/`.
38. G. Ucoluk and I. H. Toroslu. A genetic algorithm approach for verification of the syllable based text compression technique. *Journal of Information Science*, 23(5):365–372, 1997.
39. J. S. Vitter. Design and analysis of dynamic huffman codes. *J. ACM*, 34(4):825–845, 1987.
40. T. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, June 1984.
41. WinZip. The compression utility for windows. Website, 2009, Nov. `http://www.winzip.com/`.
42. I. Witten, R. Neal, and J. Cleary. Arithmetic coding for data compression. 1987.
43. M. J. Zaki and M. Sayed. The use of genetic programming for adaptive text compression. *Int. J. Inf. Coding Theory*, 1(1):88–108, 2009.
44. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.
45. J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.