UNIVERSITY OF ESSEX

Evolutionary Synthesis of Lossless Compression Algorithms: the GP-zip Family

by

Ahmed Kattan

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

School of Computer Science and Electronic Engineering

October 2010

“We can’t solve problems by using the same kind of thinking we used when we created them”

–Albert Einstein

Abstract

Data Compression algorithms have existed from almost forty years. Many algorithms have been developed. Each of which has their own strengths and weaknesses. Each works best with the data types they were designed to work for. No Compression algorithm can compress all data types effectively. Nowadays files with a complex internal structure that stores data of different types simultaneously are in common use (e.g., Microsoft Office documents, PDFs, computer games, HTML pages with online images, etc.). All of these situations (and many more) make lossless data compression a difficult, but increasingly significant, problem.

The main motivation for this thesis was the realisation that the development of data compression algorithms capable to deal with heterogeneous data has significantly slowed down in the last few years. Furthermore, there is relatively little research on using Computational Intelligence paradigms to develop reliable universal compression systems. The primary aim of the work presented in this thesis is to make some progress towards turning the idea of using artificial evolution to evolve human-competitive general-purpose compression system into practice. We aim to improve over current compression systems by addressing their limitations in relation to heterogeneous data, particularly archive files.

Our guiding idea is to combine existing, well-known data compression schemes in order to develop an intelligent universal data compression system that can deal with different types of data effectively. The system learns when to switch from one compression algorithm to another as required by the particular regularities in a file. Genetic Programming (GP) has been used to automate this process.

This thesis contributes to the applications of GP in the lossless data compression domain. In particular we proposed a series of intelligent universal compression systems: the GP-zip family. We presented four members of this family, namely, *GP-zip, GP-zip*, GP-zip2* and *GP-zip3.* Each new version addresses the limitations of previous systems and improves upon them. In addition, this thesis presents a new learning technique that specialised on analysing continues stream of data, detect different patterns within them and associate these patterns with different classes according to the user need. Hence, we extended this work and explored our learning technique applications to the problem of the analysing human muscles EMG signals to predict fatigue onset and the identification of file types. This thesis includes an extensive empirical evaluation of the systems developed in a variety of real world situations. Results have revealed the effectiveness of the systems.

Acknowledgements

The completion of this thesis was made possible through the support and kind assistance of many people. To all of these, and many others, who contributed directly or indirectly, I owe a debt of gratitude.

First, I must record my immense gratitude to my supervisor Professor Riccardo Poli who patiently listened to many fragments of data and arguments and was able to make the discussions very stimulating. His guidance and conclusive remarks had a remarkable impact on my thesis. Further, I would like to thank all people who I collaborated with, Edgar Galván-López, Michael O'Neill, Alexandros Agapitos and Francisco Sepulveda and Mohammed AL-Mulla.

I am also indebted to all my friends who supported me during my PhD. Their cheerfulness and sense of humour would always brighten a bad day; I would have never made it without their unfailing support. Special thank to, Ahmed Basrawi, Almonther Harshan, Atif Alhejali, Mohammed Al-Yazidi, Loay Balkiar, Abdullah Shata, Nasser Tairan and Abdullah Alsheddy, and also Mohammed AL-Mulla.

Last but not least, as always, I owe more than I can say to my exceptionally loving parents whose blessings and nurture pave every step of my way . . .

Ahmed Jamil Kattan
October 2010

Contents

Abstract ii

Acknowledgements iii

List of Figures viii

List of Tables x

1 Preliminaries 1
- 1.1 Problem Domain . . . 1
- 1.2 Problem Solving as a Search Task . . . 5
- 1.3 Why Genetic Programming ? . . . 7
- 1.4 Motivation and Objectives . . . 8
- 1.5 Thesis Contributions . . . 9
- 1.6 List of Publications . . . 10
- 1.7 Summary of the Thesis . . . 12

2 Background 14
- 2.1 General Genetic Programming Concepts . . . 14
  - 2.1.1 Population Initialisation . . . 17
  - 2.1.2 Selection . . . 18
  - 2.1.3 Genetic Operators . . . 20
    - 2.1.3.1 Crossover . . . 20
    - 2.1.3.2 Mutation . . . 21
    - 2.1.3.3 Reproduction . . . 23
  - 2.1.4 Linear Genetic Programming . . . 23
  - 2.1.5 Limitations of Genetic Programming . . . 23
- 2.2 Data Compression . . . 25
  - 2.2.1 Modelling and Coding . . . 26
  - 2.2.2 Compression Categories: Lossy and Lossless . . . 26
  - 2.2.3 Universal Compression . . . 27
  - 2.2.4 Measure of Performance . . . 29
  - 2.2.5 Pre-processing data . . . 31

3 Intelligent Universal Data Compression Systems 32

3.1 Conventional Universal Data Compression 33
3.1.1 Adaptive Compression Algorithms 33
3.1.2 Composite Compression Algorithms 37
3.2 Data Compression with Computational Intelligence Paradigms 40
3.2.1 Neural Networks Applications in Data Compression 41
3.2.1.1 Code Transformation 41
3.2.1.2 Vector Quantisation 43
3.2.1.3 Predictive Coding 43
3.2.2 Genetic Algorithms Applications in Data Compression 44
3.2.2.1 Lossy Image and Video Compression 45
3.2.2.2 Lossless Compression 48
3.2.2.3 Optimise Existing Compression Algorithms 49
3.2.2.4 Evolve Data Transformation Algorithms 51
3.2.2.5 Wavelet Compression 51
3.2.2.6 Vector Quantization 52
3.2.3 Limitation of the current systems 52

4 The GP- zip approach: Compression during Evolution 54
4.1 GP-zip 55
4.1.1 Data Segmentation 56
4.1.2 The Approach 56
4.1.3 Search Operators 59
4.2 GP-zip* 60
4.2.1 Representation 62
4.2.2 Crossover 64
4.2.3 Mutation 65
4.3 Experimental Results 67
4.3.1 GP-zip Experimental Results 67
4.3.2 GP-zip* Experimental Results 69
4.4 Summary 72

5 The GP-zip2 approach: Compression after Evolution 75
5.1 GP-zip2 76
5.1.1 The Basic Idea 76
5.1.2 System's Operation 78
5.1.3 File Segmentation 80
5.1.4 Classification of Segments 81
5.1.5 Search Operators 83
5.1.6 Decompression Process 84
5.1.7 Fitness Evaluation 85
5.1.8 Training and Testing 89
5.2 Experiments 91
5.2.1 Experimental Results 91
5.2.1.1 GP-zip2's Behaviour Across Runs 93
5.2.1.2 GP-zip2 vs. other Compression Algorithms 95
5.2.1.3 GP-zip2 Generalisation 98

5.2.1.4 Compression Algorithms Allocation 100
5.3 Summary 100

6 Speeding up Evolution in GP-zip2 by Compression Prediction 103
6.1 Compression Ratio Prediction 104
6.1.1 Analysing the Byte-Series and Byte-Frequency Distribution 106
6.1.2 Decision Tree 108
6.1.3 Search Operators 109
6.1.4 Fitness Evaluation 109
6.1.5 Training and Testing 111
6.2 GP-zip3 112
6.2.1 GP-zip3 Search Operators 113
6.2.2 GP-zip3 Fitness Evaluation 113
6.2.3 GP-zip3 Training and Testing 114
6.3 Experiments 115
6.3.1 GP-zip3 Experimental Results 115
6.3.1.1 GP-zip3's Behaviour Across Runs 116
6.3.1.2 GP-zip3 vs. GP-zip2 118
6.3.1.3 GP-zip3 Generalisation 119
6.3.2 Compression Prediction System Experimental Results 120
6.4 Summary 124

7 Extensions of GP-zip2 127
7.1 Detecting Localised Muscle Fatigue during
Isometric Contraction 128
7.1.1 Introduction 128
7.1.2 Applying GP-zip2 Model 130
7.1.3 Splitter Tree 131
7.1.4 Feature-Extraction Trees 132
7.1.5 Labelling the Training Set 134
7.1.6 EMG Filtering 135
7.1.7 Fitness Evaluation 136
7.1.8 Search Operators 139
7.1.9 Experiments 139
7.1.9.1 EMG Recording 139
7.1.9.2 GP Setup 140
7.1.9.3 Results and Analysis 141
7.1.10 Muscle Fatigue Prediction Summary and Future Work 143
7.2 GP-Fileprints: File Types Detection 145
7.2.1 Introduction 145
7.2.2 Applying GP-zip2 Model 148
7.2.3 Splitter Tree 149
7.2.4 Fileprint Tree 150
7.2.5 Feature-Extraction Trees 151
7.2.6 Fitness Evaluation 152
7.2.7 Search Operators 153
7.2.8 Experiments 154

7.2.9 File Types Detection Summary and Future Work 157
7.3 Summary 158

8 Conclusions and Future Work 160
8.1 Conclusions 160
8.1.1 Motivations and Objectives 160
8.1.2 Contributions of the Thesis 161
8.1.2.1 The GP-zip Family 161
8.1.2.2 Generalisation of GP-zip2 Beyond Data Compression 164
8.2 Future Work 165

Bibliography 168

List of Figures

2.1 Program representation in a standard GP system. 15
2.2 GP search flowchart. 17
2.3 Sub-tree crossover operator. 21
2.4 Sub-tree mutation operator. 22
2.5 Stages of compression process. 27

3.1 AC encoding process. The region is divide into sub-regions proportional to symbol frequencies, then the sub-region containing the point is successively subdivided in the same way. Each division map all possible messages at the particular point. 35

4.1 GP-zip individuals are linear representation, with a compression and/or transformation function in each position. 57
4.2 GP-zip flowchart. 59
4.3 Typical archive file being compressed with GP-zip: A: represents data segmentation, B: represents the gluing process of similar segments. 61
4.4 GP-zip* individuals representation. 63
4.5 GP-zip* crossover operator. 66
4.6 GP-zip* mutation operator. 67

5.1 Plotting the signals associated with different file types. Different data types often correspond to signals with very different characteristics 77
5.2 Outline of GP-zip2 training process. 78
5.3 File segmentation in GP-zip2. The splitter tree is repeatedly applied to the data in a sliding window (bottom). The output in consecutive windows is compared: if it is higher than a threshold θ the data file is split at the window's position (top). 81
5.4 Illustration of the operation of the feature extraction trees. 82
5.5 Homogeneity of clusters. (The cluster on the left has 60% homogeneity while the cluster on the right has 80% homogeneity) 86
5.6 Evolution of best-of-generation fitness during our experiments. The two components of the fitness (the fitness associate with the splitter tree and the fitness associated with the feature-extraction trees are also plotted). Bars represent the standard errors of the means. 94
5.7 Allocation of compression algorithms to a test archive file by different best-of-run individuals (a), average number of block transitions along the length of the archive (b) and plot of the byte values in the archive (c). Algorithm allocations in (a) have been sorted by compression ratio to highlight similarities and differences in behaviour. The transitions plot in (b) has been smoothed for a clearer visualisation. Data in (c) were sub-sampled by factor of 10 for the same reason. 102

6.1 Individuals representation within the prediction system's population. 105
6.2 GP-zip2 vs. GP-zip3. 117
6.3 Summary of 10 GP runs to evolve predictors for the AC compression. 122
6.4 Summary of 10 GP runs to evolve predictors for the PPMD compression. . . . 122

7.1 Labelled EMG signal using the zero crossing approach. Fatigue = blue, Non-Fatigue = red, Transition-to-Fatigue = green. 135
7.2 EMG analysis - Homogeneity of the clusters 138
7.3 Visualised illustration of GP performance in one of the runs for test signal B4. GP prediction (bottom) is compared against actual muscle's status (top). Fatigue = blue, Non-Fatigue = red, Transition-to-Fatigue = green. 144
7.4 Outline of the file-type detection process. 149
7.5 File-type detection system - Individuals' representation. 150
7.6 The fileprint tree processes the segments identified by splitter tree (top). Its output produces a GP-fingerprint for the file (bottom). 151
7.7 GP-fileprint - Homogeneity measure of the clusters. 153

List of Tables

4.1 GP-zip - Parameters settings. 68
4.2 GP-zip performance comparisons. (Bold numbers are the highest). 69
4.3 Test files for GP-zip*. 70
4.4 GP-zip* - Parameters settings. 70
4.5 GP-zip* performance comparisons. (Bold numbers are the highest, N/T: not tested) 71
4.6 Summarisation of 15 GP-zip* runs. 72

5.1 GP-zip2 primitive set 79
5.2 Training and testing files for GP-zip2 90
5.3 Test files for GP-zip2 92
5.4 Tableau of the GP-zip2's parameter settings used in our experiments. 93
5.5 GP-zip2 summary of results in 15 independent GP runs 95
5.6 Performance comparison (compression ratio %) between GP-zip2 and other techniques. 97
5.7 GP-zip2 training and compression times. 97
5.8 Processing times required by standard compression algorithms when compressing our 133.5MB test set. 98
5.9 GP-zip2 - Archives containing file types not in the training set. 99
5.10 GP-zip2 performance generalisation 99

6.1 Compression prediction system - Primitive set of the Byte-Series tree and Byte-Frequency Distribution tree. 105
6.2 Compression prediction system - Primitive set of the decision tree. 105
6.3 Training file types for the compression prediction system. 111
6.4 Testing files types for the compression prediction system. 112
6.5 GP-zip3 parameters settings. 115
6.6 GP-zip3 summary of results in 15 independent GP runs 116
6.7 Performance comparison (compression ratio %) between GP-zip3 and GP-zip2 119
6.8 GP-zip3 generalisation performance 120
6.9 Compression prediction system parameter settings. 121
6.10 Compression prediction system's performance Comparison (Seen file types vs. Unseen file types) 123
6.11 Compression prediction system - Decision tree performance 123
6.12 Compression prediction system - Decision tree- Right decisions statistics . . . 124
6.13 Compression prediction system - Compression time vs. Prediction time. 124

7.1 Fatigue prediction system - Primitive set 131
7.2 Muscle fatigue prediction system - Parameter settings. 141

7.3 Muscle fatigue prediction system - Summary of performance of 18 different GP runs. 142
7.4 Muscle fatigue prediction system - Summary of 18 runs. 142
7.5 Muscle fatigue prediction system - Best GP test run vs. worst GP test run. . . . 143
7.6 File-type detection system - Primitive set. 149
7.7 GP-fileprint - Parameter settings. 154
7.8 GP-fileprint - Training and test sets for the experiments. 155
7.9 GP-fileprint - Test-set performance results. Numbers in **boldface** represent the best performance achieved. 156
7.10 GP-fileprint - Summary of results of 40 runs. 157

Chapter 1

Preliminaries

1.1 Problem Domain

One of the paradoxes of technology evolution is that despite the development of the computer and the increased need for storing information, there is a lack of development in the area of data compression. As the evolution of computer systems progresses rapidly, the amount of stored information will increase at an unrelentingly fast rate. Hence, over the past two decades, several techniques have been developed for data compression and each of which has its own particular advantages and disadvantages. Each technique works best with the circumstances that it is designed to work for.

Data compression [1–3] is one of many technologies that enables today's information revolution. 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 file 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 is the process of observing regularities in the data and trying to eliminate them with the aim of reduce the total size of the data. Different types of data have different structures and different forms of regularities. For instance, a common pattern in English text files is *"th"* or *"qu"*, while a common pattern in images is the similarity of adjacent pixel values. In videos, a common pattern is the similarity of subsequent frames. Therefore, it is no surprise to observe poor performance in relation to the compression ratio when applying a compression algorithm in a different environment than the one it was designed to work for. This is because different compression algorithms are designed to capture different forms of redundancy.

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 to compress and decompress, 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.

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 match different compression algorithms with different parts of the data in such away to achieve the maximum compression ratio (i.e., compression ratio equal to the entropy of the data), 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 (which is offered as a standard option on 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 in the past 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, of 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 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.[1] Clearly, exploiting these differences could improve the compression ratios achievable very significantly.

We noticed that it is very difficult to design a single generic —universal— compression algorithm that works effectively with any data type without having some knowledge or assumption of the data to be compressed. In this work we attempt to tackle the problem using an evolutionary approach. The next section will introduce a well-known approach to formalise and solve problems. This will be used as the main method in this thesis in order to solve problems related to universal compression algorithms.

[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.

1.2 Problem Solving as a Search Task

"I have not failed. I have just found 10,000 ways that won't work", so said Thomas Edison speaking of his attempts to perfect the incandescent light bulb. One of the classical approaches to solve problems is to investigate several candidate solutions and try each of them until we find one that solves the given problem. Naturally, we learn new knowledge about the problem from each mistake and try to avoid it in the next step hoping to find the right solution.

Problems can be defined as collections of information, from which an answer will be extracted or guessed [4]. The process of taking steps to find the right solution for a specific problem can be seen as a form of search. From the abstract point of view of a computer program, a search algorithm will take the problem as input and return the solution as output, often having evaluated some numbers of candidate solutions. Generally, the first step that should be taken, in order to tackle a specific problem, is called problem formulation [4]. Michalewicz and Fogel [5] have defined three main concepts to solve problems:

- *Choice of a representation: the coding of the candidate solutions.*
- *Specification of the objective: a description of the aim to be achieved.*
- *Definition of an evaluation function: the fitness function.*

Solving problems depends on prior knowledge on the domain. If there is sufficient information available about the problem domain, then there may be no need to search for a solution. However, if there is insufficient knowledge available about the problem domain (as is the case with most real world problems) then a search algorithm should be allocated to the targeted problem in order to find a solution.

Over the past few decades, computer scientists have begun to solve problems through search. Many algorithms have been invented to do this. In these techniques, accurate and timely search are key objectives. However, researchers must face a necessary trade-off between flexibility,

where the algorithm can work with multiple domains, and speed of execution [6]. Many search algorithms favour speed over flexibility. In addition, current search algorithms are domain knowledge dependent [6].

Effective search algorithms have to balance between two main objectives: *Exploiting* and *Exploring* [4]. Exploiting means that the algorithm searches a particular area in the search space thoroughly. On the contrary, exploration refers to the searching of different areas in the search space. It is clear there is a conflict between these objectives. Consider exploitation; here the search focuses on a specific area, which is necessarily small, and consequently it may not contain the best solution. In contrast, excessive exploration can result in the wasting of a great deal of time. Often the size of the search space as so big, that a human life is short in comparison to the time needed by the fastest supercomputers to look at just a fraction of the space [6]. Thus, random search is infeasible as it explores too much.

A good approach for solving problems through search is Evolutionary Computation (EC). EC is a field of research that is inspired by the theory of evolution and our contemporary understanding of biology and natural evolution. Hence, EC algorithms are based on the idea of searching for a solution for a given problem by evolving a population of candidate solutions. EC techniques have been used as a search and optimisation technique for non-trivial problems with remarkable success. There are several branches of Evolutionary Computation: Genetic Algorithm, Genetic Programming, Evolutionary Programming, Evolutionary Strategies and often some researchers count the Particle Swarm Optimisation as a fifth branch of Evolutionary Computation [7].

Another approach for searching problems search space is Heuristic Search. The basic idea of heuristic search algorithms is to try all neighbour solutions in particular point at the search space and select the best solution in the neighbourhood. This process iterates until the algorithm terminate. [2] Heuristic Search algorithms include *Hill Climbing, Best-First-Search* and *A** [8].

[2]The algorithm terminate if it meets its termination condition e.g., finish the maximum number of iterations or it found the desired solution

The work presented in this thesis involves the applications of Genetic Programming techniques in the lossless data compression domain. Therefore, details of the other search techniques will not be explained here. More details on the other EC algorithms are available on [4, 8, 9].

1.3 Why Genetic Programming ?

Genetic Programming (GP) is a domain independent framework for automatically solving problems via a form of Darwinian evolution, originating from high level statements of what needs to be done [10]. GP often ends up with solutions that a human may never think about. To that end, GP has attracted the attention of many researchers to investigate its capabilities, limitations and applications. In this thesis we mainly explore the applications of GP in the lossless data compression domain. However, some of the techniques developed have proven to be useful also for other domains (details are given in section 1.5). GP has been selected as the main learning paradigm in this research for the following reasons:

- The flexibility and expressiveness in GP representation allows the characterisation of complex programs, as is the case with data compression algorithms.
- In recent years, GP has been reported to achieve success in relation to the solving of complex problems in several domains such as: image processing [11], signal analysis [12] and data compression [13].
- It may be possible to learn new knowledge from the process of evolution. GP can discover new solutions that human never thought about.

For these reasons GP was selected as the basis of research for this thesis to explore the possibility of evolving intelligent compression algorithms.

1.4 Motivation and Objectives

Today we live in a digital era where information is increasing in an explosive rate. The demand for data transfer today is stretching the original hardware infrastructure of the internet to its limits. Yet, there is a lack of development of reliable compression techniques that can deal effectively with multiple data types. Compression researchers have always tended to develop specialised compression algorithms which perform well on a specific type of data. However, over time there has been an increase in the use of files with complex structures that store data of different types simultaneously (e.g., games, single file website). Thus, an ideal compression system would be one that is able to efficiently compress all data types well and in a satisfactory time. 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.

The central aim of the work presented in this thesis is to make some progress towards turning the idea of using artificial evolution to evolve human-competitive general-purpose compression system into practice. We aim to improve over current compression systems by addressing their limitations in relation to heterogeneous data, particularly archive files. The main idea is to use GP to combine existing data compression schemes in order to develop an intelligent universal data compression system that can deal with different types of data effectively. This can be achieved by utilising existing compression algorithms where they are expected to perform best. We want to develop a system that is able to automatically generate lossless compression algorithms based on the given encoding situation by using GP to learn the structure of the given data and adapt the system's behaviour in order to provide the best possible compression level.

The main objectives of this thesis are as follows:

- To propose a new system based on GP to evolve intelligent Lossless Compression algorithms, (this will be referred to as GP-zip).

- To find adjustments to the standard GP for the purpose of improving the evolution in our application.
- To generalise the proposed model and adjustments by studying their practicality and applications in other comparable domains.

1.5 Thesis Contributions

The main contributions of this thesis are twofold. Firstly, the presented work is contributing to the applications of GP in the lossless data compression domain. A new method of combining data compression algorithms under the general paradigm of GP resulted in a series of intelligent compression systems (the GP-zip Family). GP-zip systems are based on the idea of applying a combination of basic compression algorithms to a file based on the given coding situation. The systems use GP to learn when to switch from one compression algorithm to another as required by the particular regularities in a file. These algorithms are treated as black boxes that receive input and return output, although some algorithms may use another learning mechanisms and/or multiple layer of compressions themselves.

Secondly, although the main ideas in this thesis are substantially related to lossless data compression programs, we found that the developed techniques have surprising breadth of applicability in other problem domains. The second contribution of this thesis is the production of a pattern recognition model that is able to find for previously unknown, but meaningful, regularities in a given stream of data.

The following is a list of the ingredients that form the technical and scientific contributions of the thesis.

I. Contributions related to the applications of GP in the lossless data compression domain (the GP-zip family)

- GP-zip* is a generic lossless compression system that is ideal for small/medium files in scenarios where the user needs to compress once and decompress many times.
- GP-zip2 is a training-based generic lossless compression system that is faster than GP-zip* but has slightly inferior performance in terms of achieved compression ratios.
- GP-zip3 is a training-based generic lossless compression system that has faster training time than GP-zip2 and achieved almost the same performance in terms of compression ratios. However, it has less generalisation abilities.

II. The generalisation of GP-zip to a broader range of problems

- The application of the GP-zip2 model to analyse muscle EMG signals to predict fatigue onset during isometric contractions.
- The application of the GP-zip2 model to the: identification of file types via the analysis of raw binary streams (i.e., without the use of meta data).

A substantial number of experiments have been conducted in order to examine, verify and reveal the flexibility and generalisation of GP-zip systems in compression and in the problems mentioned above. Hopefully, this thesis will inspire other researchers working in the same area or along similar lines.

1.6 List of Publications

The work presented in this thesis has been published in eight peer-reviewed conference papers and one journal article. The following list relates the chapters of the thesis with the published work.

- **Chapter 3: Intelligent Universal Data Compression Systems**
  - Ahmed Kattan, *Universal Intelligent Data Compression Systems: A Review*, Proceedings of the 2nd Computer Science and Electronic Engineering Conference, Colchester, ISBN: 978-1-4244-9030-1, United Kingdom, IEEE press, 2010.

- **Chapter 4: The GP-zip approach (Compression during Evolution)**
  - Ahmed Kattan and Riccardo Poli, *Evolutionary Lossless Compression with GP-ZIP*, IEEE World Congress on Computational Intelligence, Hong Kong, 2008. ISBN: 978-1-4244-1823-7, IEEE Press.
  - Ahmed Kattan and Riccardo Poli, *Evolutionary Lossless Compression with GP-ZIP**, Proceedings of the 10th annual conference on Genetic and Evolutionary Computation, pages 1211-1218, Atlanta, GA, USA, 2008. ACM.

- **Chapter 5: The GP-zip2 approach (Compression after Evolution)**
  - Ahmed Kattan and Riccardo Poli, *Evolutionary synthesis of lossless compression algorithm with GP-zip2* Submitted to journal of Genetic Programming and Evolvable Machines, 2010.

- **Chapter 6: Speeding up Evolution in GP-zip2 by Compression Prediction**
  - Ahmed Kattan and Riccardo Poli, *Genetic programming as a predictor of Data Compression saving*, Artificial Evolution, LNCS Springer, Pages 13-24 Strasbourg-France, 2009.
  - Ahmed Kattan and Riccardo Poli, *Evolutionary synthesis of lossless compression algorithm with GP-zip3.* IEEE World Congress on Computational Intelligence, Spain, Barcelona, 2010. **(Best paper award nominated)**

- **Chapter 7: Extensions of GP-zip2**
  - Ahmed Kattan, Mohammed Al-Mulla, Francisco Sepulveda and Riccardo Poli, *Detecting Localised Muscle Fatigue during Isometric Contraction using Genetic Programming*, International Conference on Evolutionary Computation, IJCCI Springer, Pages 292-296, Madeira - Portugal, 2009.
  - Ahmed Kattan, Edgar Galván-López, Riccardo Poli and Mike O'Nill. *GP-fileprints: File Types Detection using Genetic Programming*, Proceedings of the 13th European Conference, EuroGP 2010, LNCS, Springer-Verlag, 7-9 April 2010.
  - Ahmed Kattan, Alexandros Agapitos and Riccardo Poli. *Unsupervised Problem decomposition using Genetic Programming*, Proceedings of the 13th European Conference, EuroGP 2010, LNCS, Springer-Verlag, 7-9 April 2010.

1.7 Summary of the Thesis

This chapter has given a preliminary introduction to the problem domain and suggested the idea of treating the universal data compression problem as a search task. In addition, the goals and contributors of this thesis have been highlighted.

In chapter 2, a basic background on GP is presented. Also, we provide basic information on data compression and its categories. Furthermore, we present our definition of the ideal universal compression algorithm.

Chapter 3 presents a review of previous attempts to develop intelligent compression systems using different approaches and highlights current limitations and possible improvements.

Chapter 4 provides a description of the first two members of the GP-zip family (GP-zip and GP-zip*). GP-zip is primarily provided as proof of concept and it is not designed for practical use. However, GP-zip* is a more advanced version of its predecessor and fulfils a bigger niche in the compression world. This is verified by experimental results on a variety of test files and a detailed comparison with state of the art compression algorithms. Finally, a general evaluation of these systems has been presented. In this part of the GP-zip family, the systems try to find patterns in the data and match them with standard compression algorithms during evolution to maximally and losslessly compressing files and archives.

Chapters 5 and 6 provide a description of the second part of the GP-zip family (GP-zip2 and GP-zip3), respectively. In GP-zip2, we used a completely different approach to the one used in GP-zip*. GP-zip2 is a training-based compression system. Moreover, we present GP-zip3, which addresses some of GP-zip2's limitations and improves upon them. Also, in these chapters we present experimental results and their analysis to test and demonstrate the practicality of GP-zip2 and GP-zip3. In this second part of the GP-zip family, the systems learn to find patterns in the data and match them with standard compression algorithms for maximally and losslessly compressing files and archives. Thereafter, this knowledge is applied to unseen data. Thus, GP-zip2&3 can be seen as a pattern recognition system.

As we mentioned previously, we found that the developed model in GP-zip2 and GP-zip3 has surprising breadth of applicability in other problems that have similar characteristics. Thus, chapter 7 goes beyond data compression and studies the application of GP-zip2 as a pattern recognition model with two other problems. Firstly, we present a successful application of the model in analysing human muscles EMG signals to predict fatigue. The chapter includes experimentation that revealed potential applications domains such as ergonomics, sports physiology, and physiotherapy. Secondly, we apply the GP-zip2 model in another novel application of GP. Here, we used the model for identification of file types via the analysis of raw binary streams (i.e., without the use of meta data). Also, we presented experimentation, which revealed that the identification accuracy obtained by the GP-zip2 model was far superior to those obtained by a number of standard algorithms. The resulting programs are entirely practical, being able to process tens of megabytes of data in few seconds, which could perhaps be integrated in spam filters and anti-virus software.

Finally the thesis concludes in chapter 8, which includes some conclusive remarks about how this thesis satisfied the objectives mentioned above and proposes new interesting questions that will be addressed in future research.

Chapter 2

Background

2.1 General Genetic Programming Concepts

While a technique to evolve trees was suggested by Kramer in 1985, the field of Genetic programming (GP) was founded by John Koza in 1992 [14]. GP is a powerful learning engine inspired by biology and natural evolution, for automatically generating working computer programs.

Based on Darwin's theory of evolution, computer programs are measured against the task they are intended to do and then receive scores accordingly. The programs with the highest scores are considered the fittest. The fittest programs are selected to join the evolutionary process via three standard genetic operators: crossover, mutation and reproduction (further details are given in the next subsections). These operators aim to amend the program's structure and create new offspring, which will hopefully be better. Computer programs are treated as a collection of functions and inputs for these functions; which can be seen as their genetic material. In GP, programs are often represented using a tree-like representation, where the functions appear in the internal nodes of the tree and the inputs for the programs appear on the leaves. This is illustrated in figure 2.1 [14].

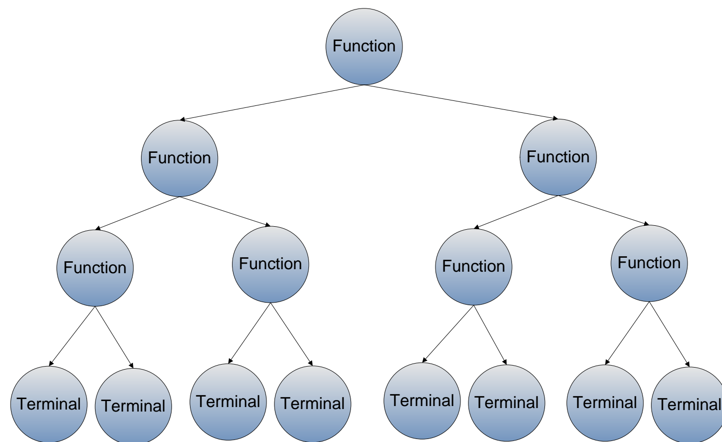

FIGURE 2.1: Program representation in a standard GP system.

The functions and the inputs are part of two major elements for any GP system: the *Function Set* and the *Terminal Set*. Function set and terminal set are problem dependent. For example, the function set for designing an electronic circuit that adds two binary numbers, an "Adder", might be $F = \{AND, OR, NOT\}$ and the terminal set might be $T = \{1, 0\}$. In curve fitting problem (also known as symbolic regression), the function set might include mathematical notations (cos, sin, tan, ...etc.), arithmetic operations (+, -, *, /, %, ...etc.), or even 'if' statements [14]. Each function needs a certain number of inputs known as the function *arity* [14]. Terminals can be considered functions with arity zero. There are two properties which should be satisfied in any GP system: *a) Closure* and *b) Sufficiency* [14]. Closure means that any function should be able to receive any input in the terminal set and return output that can be readable by the other functions in the function set. Sufficiency means that there exist some combinations of functions and terminals which produce the right solution for the targeted problem. All the possible combinations of functions and terminals are called the *Search Space*.

GP starts the search process by randomly generating a number of programs with a tree-like representation. These are also referred to as *individuals*. A group of individuals are referred to as a *population*. The construction of new population is called a *generation*. The success of a program is determined by the *fitness function*. The fitness function tests the program output against the desired behaviour. Naturally, the fitness function does vary according to the problem.

In some cases, multiple fitness functions are required to evaluate the success of a program. A generational GP system evaluates all the programs in the population and then generates new programs via genetic operators to cause the system to move to the next generation. The newly generated programs constitute the new population. Typically, the best program that appears in the last generation or the best program throughout the run is designated as the result of the run.

The system favours those programs that are closer to solving the problem. The fitness function guides the GP system to find a better program in each generation. A GP system run ends when the termination condition is fulfilled. The termination condition is usually that the GP has reached the maximum number of generations or the system has found the right solution. Mainly, there are three types of operation used by GP to generate the programs in a new generation. These are: *crossover, mutation* and *reproduction* (see sections 2.1.3.1, 2.1.3.2 and 2.1.3.3 respectively).

GP is a member of a class of algorithms called beam search [15], where the algorithm knows little about the domain, make assumptions about the search space and searches for potential solutions. Unlike blind search and hill-climbing search, which apply a single point-to-point method to explore the search space, GP maintains a population of search "beams" in the search space. The fitness function acts as a metric to select the most promising points in the search space to join further transformations via crossover and mutation [16]. Generally, Evolutionary Computation and GP do not directly construct the solution, instead they search for a solution in the given search space [17].

The flowchart presented in figure 2.2 summarises the steps involved in a GP search. The next subsections will provide a detailed description for each step and will illustrate some of the most widespread techniques used to facilitate the GP search process.

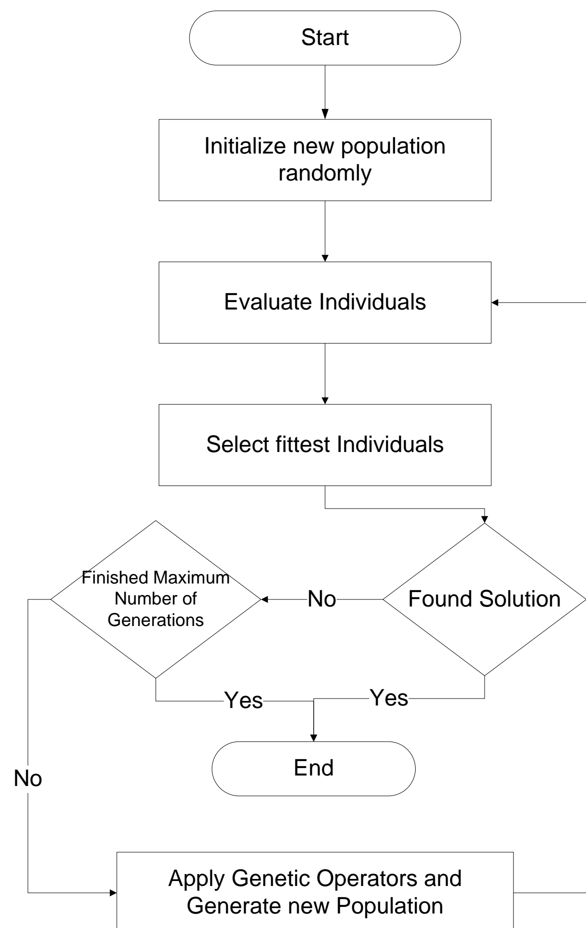

FIGURE 2.2: GP search flowchart.

2.1.1 Population Initialisation

As mentioned previously, all possible combinations of functions and terminals form the GP search space. Since their number is usually very big, we effectively can refer to the size of the search space as infinite. Most GP systems define a maximum depth or size for the trees to be generated in the initial population. This is in order to limit the search process to a relatively smaller subset of the search space.

The initial population is a seed for the GP system and is used to find a solution for the given problem. The generation of the initial population has a significant influence on GP's performance. In particular, populations with poor diversity can have potentially disastrous consequences for a run. Initial populations are typically randomly generated. Here, we will describe three of the

most common algorithms used to generate the initial population *(full, grow* and *ramped-half-and-half).*

In the full method, nodes are taken randomly from the function set until a maximum depth is reached, then only terminals are chosen randomly to complete the generated tree. The full method always guarantees the generation of tree of maximum depth. This, however, does not imply that all generated trees will have exactly the same shapes or equal sizes [10]. [1] This only happens when all functions have an equal number of arities. A major problem with the full method is that it is biased towards a particular type of trees (i.e., it generates big trees only).

In the grow method, nodes are taken randomly from both function and terminal sets until the generator selects enough terminals to end the tree or it reaches the maximum depth. The grow method is known to produce trees of irregular shapes [16]. Similar to the full tree generator, the problem with the grow method is that the shape of the trees with the grow method is directly influenced by the size of the function and the terminal sets. Thus, a big number of terminals increases the chance of generating small trees and, in contrast, a small number of terminals increases the chance of generating big trees [10].

Koza in proposed a combination between the two methods referred to as ramped-half-and-half [14]. With this method, half the population is generated with full and the other half is generated with grow.

2.1.2 Selection

In the evolutionary process embraced by GP, the fittest programs are selected to be processed by the evolutionary operators and constitute the new generations. This, however, does not imply that inferior programs should be completely ignored. In fact, it is necessary to include some low fitness programs to prevent the search from being biased. The selection algorithm plays an important role in guiding the search process. Generally, the selection decision is based on

[1] The depth of a tree is the number of edges from the root node (which is assumed to be at depth 0) to the deepest leaf node.

a competition between individuals in the population [16]. The degree to which individuals can win this competition is regulated by the selection pressure.[2] Individuals with high fitness values are more likely to be selected to produce the new offspring. In this section two of the most commonly used selection algorithms will be described *(tournament selection* and *fitness-proportionate selection)*.

In tournament selection, a number of individuals are selected randomly (independently from their fitness values) to be included in the tournament pool. These are compared with each other based on fitness and the best is selected to be a parent. The number of programs to be included in the tournament pool is referred to as the tournament size. Adjusting the tournament size allows controlling the selection pressure [16]. Small tournament sizes result in low selection pressure on the population while big tournament sizes produce high pressure. When applying crossover, tournament selection is performed twice. The blind selection of tournaments automatically rescales fitness in such a way as to make the selection pressure remain constant [10]. Thus, even the fittest program in the population cannot pass its genetic materials to the next generation without limits. With this algorithm an average program has a fair chance of being selected.

In fitness-proportionate selection, also known roulette-wheel selection, each individual is given a specific probability to pass offspring into next generation [16]. An individual x has a probability:

$$p_x = \frac{f_x}{\sum_x f_x}$$

The roulette-wheel analogy can be envisaged by presenting a roulette wheel and candidate solutions as pockets in the wheel. The size of the pockets is proportionate to the probability of the candidate solution. Consequently, the fittest individuals are more likely to be selected. On the other hand, weaker individuals still have a chance of surviving and joining the next generations. This can be considered as an advantage as an individual with low fitness may include useful

[2] Selection pressure is a key property to selection algorithm. A system with high selection pressure favours high fitness programs, while a system with low pressure does not discriminate against inferior programs.

genetic material for the recombination process. Fitness-proportionate selection has been used in the GA community for a long time while in GP tournament selection is more common.

2.1.3 Genetic Operators

As mentioned previously, most GP systems generate the initial population randomly. Naturally, this results in very low fitness. GP modifies the structure of the fittest programs via genetic operators in order to improve their performance. Genetic operators are analogous to those which occur in nature. In AI terminology, they are called search operators [16]. Search operators in any GP system are important as they guide the search though the search space to discover new solutions. Three standard search operators are in common use: crossover, mutation and reproduction. The choice of which operator to use is based on predefined probabilities called the *operator rates* [10]. Typically, in GP crossover is applied with a probability of 0.9, while mutation and reproduction are applied with much lower probabilities.

2.1.3.1 Crossover

The main aim of crossover is to combine the genetic materials of two parents by exchanging some genetic material from one parent tree with some from the other. The most commonly used type of crossover is *sub-tree crossover*. In sub-tree crossover, GP system selects two trees (using a selection algorithm). The system randomly selects two crossover points in each parent and swaps the sub-tree rooted there. Then, it creates a new offspring which consists of parts of the two selected parents [14]. The crossover operator returns one offspring. However, it is possible to allow the system to return two offspring.

The sub-tree crossover operator is problem independent. Therefore, the crossover points are selected randomly and independently without respect for the parents' context. So, crossover operators are often expected to behave destructively and produce unfit individuals from fit parents.

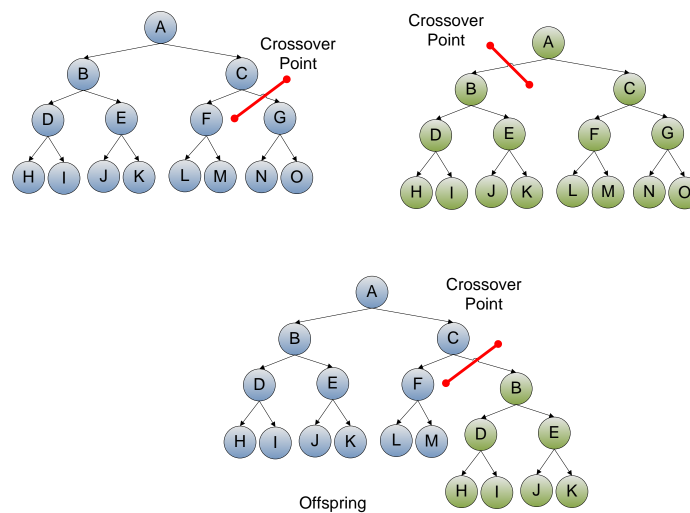

FIGURE 2.3: Sub-tree crossover operator.

In principle, the selection pressure should be able to discriminate against weak individuals; however, the destructive nature of crossover operators remains a problem. To address this problem, researchers have tended to design new operators that identify promising crossover points [18–20]. The main problem, however, with these new operators is that they are problem dependant and cannot be applied to a broader range of problems. Sub-tree crossover operator is illustrated in figure 2.3.

2.1.3.2 Mutation

The most commonly used form of mutation is called *sub-tree mutation*. Unlike crossover, mutation operates only on one parent. A random mutation point is selected, the sub-tree rooted at the selected point is deleted, and a new sub-tree inserted. The new sub-tree is generated randomly, similarly to the initial population and subject to constraints (i.e., maximum allowed depth or maximum size). Like crossover, also, mutation always guarantees the generation of individuals that are structurally valid. The application of the mutation operator in GP search prevents premature convergence as it provides the system with a way to avoid getting stuck in local optima. Sub-tree mutation is illustrated in figure 2.4.

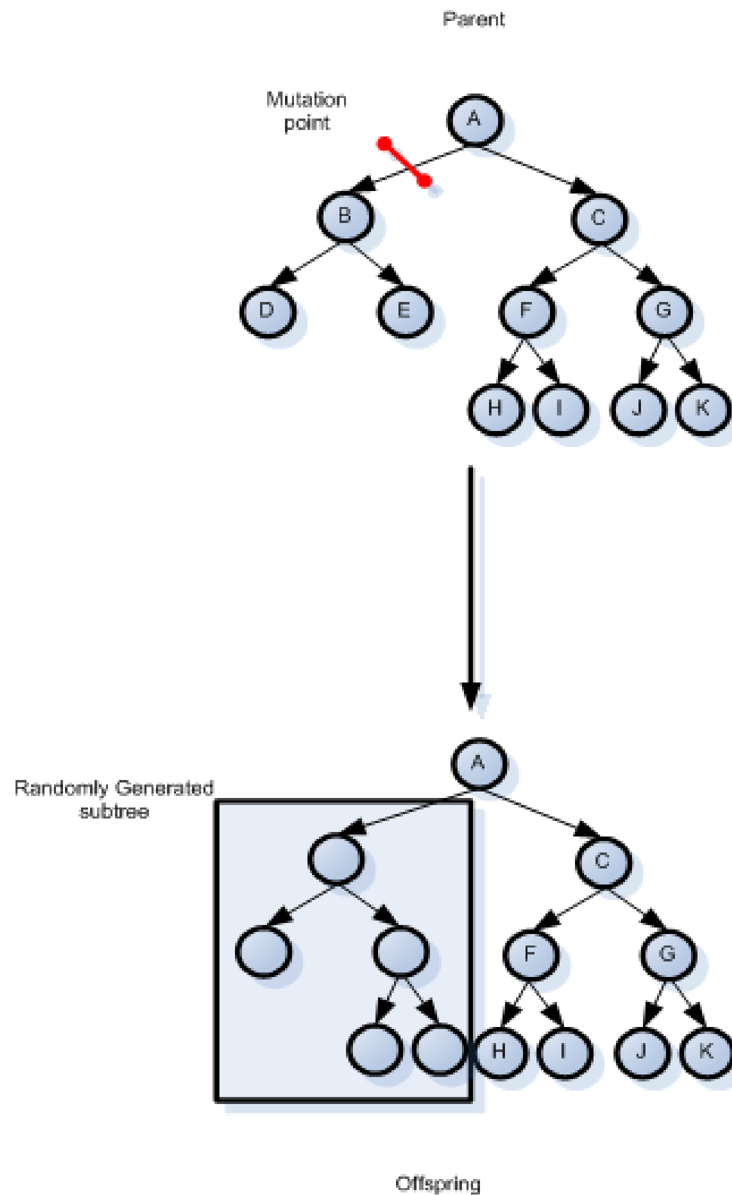

FIGURE 2.4: Sub-tree mutation operator.

Mutation is equivalent to a crossover between a selected tree from the population and a new randomly generated tree (this is also known as *headless chicken crossover* [10]). Another form of mutation is referred to as *point-mutation* where the system replaces the primitive of a selected node with another primitive (randomly selected) of the same arity [10]. If no other primitives of the same arity are available, the node remains unchanged.

2.1.3.3 Reproduction

The reproduction operator is very simple. As the name indicates, the GP system selects one parent with a predefined probability based on its fitness and copies it without any changes to the next generation. This operator is particularly useful in preventing good individuals from being lost. Also, it protects the genetic materials within good individuals from the potentially destructive nature of crossover and mutation.

2.1.4 Linear Genetic Programming

The standard and the most widespread type of GP representation is the tree-like. However, other types of GP representation have been developed as well. This section presents Linear GP (LGP) [21].

LGP was proposed by Brameier and Banzhaf [21]. In LGP programs are presented as linear sequence of instructions. The number and size of these instructions can be either fixed or variable [10]. The main motivation of developing LGP is to obey the natural computers' representation of the programs and avoid the computationally expensive interpretation of the tree shape representation. LGP can be several order of magnitude faster than standard GP. Special crossover and mutation operators have been designed for LGP. In the corssover, the system randomly choose two points in each parent and swap the fragments between them [10]. Crossover may produce offspring of different size if LGP was using variable size instructions for its individuals [10]. In mutation, a random point is selected in the parent and a new instruction is randomly generated. Other operators for LGP are also possible such as homologous crossover [10].

2.1.5 Limitations of Genetic Programming

Despite the remarkable success achieved by GP in solving a variety of complex problems with only little available knowledge about the problem domain, we have to address some fundamental

problems of GP as it is currently being practiced. This section will highlight some of the most common problems with GP systems.

The first and utmost problem is the lengthy execution time required by GP systems. The execution of computer programs on a number of test cases is a relatively slow process, without which fitness can not be evaluated. For this reason, GP is known to take long time to evolve competitive solutions. Most of the time required by a GP search is spent evaluating the population's programs.

Another common problem of GP systems is known as *bloat*. Bloat is defined as the growth in the trees sizes without significant improvement in their fitness values [10]. Researchers noticed that the average size of the population trees keep increasing rapidly after a certain point. Many theories have been proposed to explain this phenomenon, however there is still no clear explanation that justifies the different experimental results [10]. Naturally, bloat has negative effects on a GP search, as large programs are often hard for humans to comprehend and require grater computational efforts. Moreover, they often yield poor generalisation.

GP has been often used as a problem solver for a wide variety of tasks. However, one can argue that the actual limitation of GP systems is that their evolved solutions are often difficult to interpret by humans as they are complex and difficult to understand. This leaves the users with solutions but without any new knowledge regarding the problem domain. One can monitor the functions' proportions within particular solutions and associate them with a particular fitness range. This method, however, is not effective because fitness in GP depends on the order of the primitives. Thus, one node in the tree may be the reason for achieving high fitness value, while other nodes are redundant. Therefore, it is not an easy task to provide a sufficient analysis of the solutions found by GP.

GP search is based on the random generation of individuals; thus, different runs may yield different results. For this reason, GPs' results should be collected through multiple runs [10]. Often, the stability of a GP system is favoured over achieving good solutions in a single run. The stability of a GP system can be defined as the similarity of solutions' accuracy in different runs.

So, a stable GP system is the one that produces almost the same results in every run. Stability can be measured by calculating the standard deviation of the best solution's fitness in each run. Naturally, small standard deviation of solutions' fitnesses and high average indicates that the system always produce good solutions in each run. A disadvantage is that runs may require a long time to execute (depending on the complexity of the fitness function).

Another typical problem of GP is the neutrality of the mutation operator. This happens when the mutation operator has no effect on the fitness value of an individual, for example if the mutation targets redundant code (also known as *introns*) within the individual. Neutrality is often cited as a misunderstood subject in the EC community, mainly due the contradictions in opinions on its usefulness to the evolutionary search and the various interpretations of its causes [22]. For example, some researchers treat neutrality and redundancy as two separated problems, while others relate redundancy as a main cause of neutrality (e.g., [23] and [24]).

2.2 Data Compression

Data compression is the process of observing regularities within data streams and exploiting them to minimise the total length of the compressed file. The question is then, how much can this minimisation save? The answer depends on the compression model used and the amount of redundancy in the given data [3]. There are several reasons for reliance on compression algorithms. Firstly, the demand for storage increases at the same speed as storage capacity development. A second reason is to save the bandwidth of communication channels. Thus, minimising the data has significant importance in reducing costs, whether the cost of data storage or the cost of data transmission. This allows a wide range of practical applications for real world problems, some of these applications includes: data archiving systems, telecommunications, multimedia processing and data encryption. [3] Furthermore, compression has an important role

[3] Typically information goes through a compression process before being encrypted. This helps to remove redundant contents and increases the encryption efficiency [25].

in the spread of the Web, since any improvements will decrease the amount of transmitted data over the Internet.

Research in compression applications is divided into two main categories: *lossy* and *lossless*. In lossy techniques a certain loss of data is accepted in exchange for a higher compression ratio. Generally, lossy techniques are applied to images, videos and audios. On the contrary, lossless techniques are applied to information that we could not afford any lose (e.g., database records, executable files and document files). The next subsections will present a basic background on data compression and its different types.

2.2.1 Modelling and Coding

The reconstruction requirement may force the decision whether to apply loss or lossless compression on a given data. However, regardless of the compression type, all algorithms share a similar process when compressing information, namely: *modelling* and *coding* [1]. The difference between modelling and coding has often been misunderstood. In the first stage (modelling), the algorithm tries to model the redundancy within the data . Generally, this can be achieved by collecting information regarding the frequencies of the repetitive patterns. In the second stage (coding), a description of the used model is encoded [1]. Typically, this description is encoded using binary streams, which represent the compressed version of the information. Hence, the decision of how to encode a particular symbol (or set of symbols) is based on the model. Figure 2.5 illustrates the two stages of the compression process.

2.2.2 Compression Categories: Lossy and Lossless

Lossy compression involves some loss of information. Data that have been compressed using lossy techniques cannot be fully recovered. In return for accepting this loss, a higher compression ratio can be obtained in comparison to lossless techniques. Generally, the applications of lossy compression are limited to the media domain (i.e., images, videos and audio) where a lack

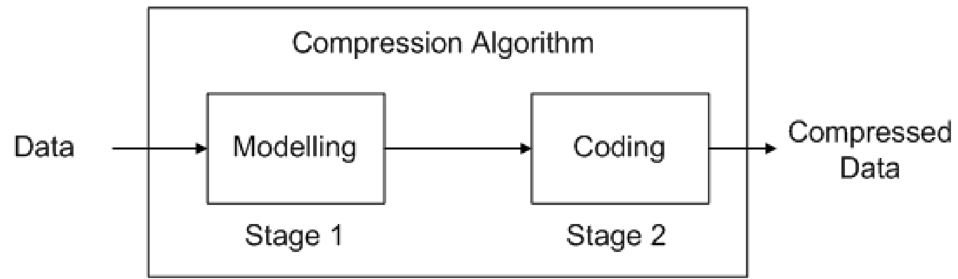

FIGURE 2.5: Stages of compression process.

of exact reconstruction is not a problem. Lossy compressions can be evaluated by measuring the difference between the source and its compressed version, which is often referred to as *distortion* [3]. Lossy compression can be formalised as follows: let S be a sequence of data where $S \in \sum\{0,1\}^+$, and C is a lossy compression algorithm. Thus, $C(S) = \bar{S}$ where $|\bar{S}| < |S|$ and this process is not reversible.

On the other hand, a compression approach is lossless if it is possible to fully recover the data from the compressed version. Lossless compression has a wider range of applications than lossy techniques. Mainly, lossless techniques are applied on any type of data where loss of information is not tolerated (e.g., text, medical images, executable files).

The work presented in this thesis is mainly concerned with lossless compression techniques. Therefore, further details on lossy compression will not be presented here. Additional information on lossy compression techniques can be found in [1, 3, 26].

2.2.3 Universal Compression

The idea of designing a universal compression algorithm that can compress all files has been considered by many researchers in the past few decades. However, there is a consensus among researchers that there is a "no-free" lunch result for lossless compression, just as there one in machine learning [27]. Therefore, it is unlikely that there exists a single compression algorithm that guarantees never to increase the size of a file. This statement can be proved as follows: Let

file.dat and let **com(file.dat)** be a lossless compression algorithm that guarantees to never increases the size of any file. Hence, the outcome of **com(file.dat)** is a compressed version of original data and the outcome of **com(com(file.dat))** is the compression of the compressed version of the original file [3]. Thus, in principle, a series of successive compressions **com(com(com(...com(file.dat))))** would result in a final compressed version of size 1 byte [3]. This, however, is not only practically unfeasible, it is also theoretically flawed as it would be impossible to losslessly reconstruct the original file [3].

The term *"Universal Data Compression"*, typically refers to a compression scheme which is asymptotically optimal.[4] Thus, any data compression algorithm necessarily leaves some input data uncompressed. Typical data types used on a regular basis in computers include (but are not limited to) texts, images, videos, binary data, and audio. Therefore, it is reasonable to call a compression algorithm that can deal with these general data types (or some of them) *universal*. Also, it is logical to associate universal compression algorithms with lossless techniques.

Naturally, because of the nature of different data types and the different information they represent, it is difficult to utilise available knowledge on the domain of the data (i.e., colour levels, video frames, repetitive text, ...etc.) and process them with a single scheme. Consequently, universal compression algorithms sacrifice the ability to utilise this information and treat the data at the binary streams level. Ideally, universal compression algorithms could calculate statistics from the source and utilise them to predict segments of the data. Compressed file stores incorrect prediction. Later, data can be reconstructed by using this prediction model, once the prediction model fail to predict a particular character correctly, the system uses the stored information in the compressed file to fully reconstruct the original data. However, in practice, such prediction is not always straightforward and it becomes difficult to provide accurate predictions when the volume of data is growing exponentially. Therefore, new ideas are needed to advance the universal compression domain.

[4]An optimal lossless compression algorithm is the one that achieves a compression rate equal to the entropy rate (i.e., the theoretical lower bound of bits needed to represent the data).

We define an *ideal universal compression* as a system that reads data from the files and rapidly identifies different incompatible fragments within the data before picking a specialised compression technique for each fragment. Typically, the final outcome of the algorithm should be smaller than the original data (or at most of the same size if the data is not compressible).

2.2.4 Measure of Performance

The performance of a compression algorithm can be measured in various ways. It is difficult to measure the performance of a particular compression algorithm in general across all files as the algorithm's behaviour greatly depends on the nature of the test files and whether the data contain the redundancy that the algorithm is designed to capture [3]. A common method used to evaluate a compression algorithm is to measure the amount of reduction in size in the compressed version of the files, which is often referred to as *compression ratio*. The compression ratio is simply the ratio of the compressed file size to the original size. Thus, a bigger compression ratio indicates higher reduction in the compressed file size. This can be calculated as follows:

$$\textbf{Compression ratio} = 100 \times [1 - \frac{compressed_file_size}{uncompressed_file_size}] \tag{2.1}$$

Swapping the numerator with denominator of the previous equation is called *compression factor* [3]. The compression factor is used to numerically quantify the amount of data reduction in numbers.

$$\textbf{Compression factor} = \frac{uncompressed_file_size}{compressed_file_size} \tag{2.2}$$

An alternative method for measuring the algorithm's performance is to report the average number of bits required to encode a single symbol. This is often called the *compression rate* [1]. In lossy compression the difference between the source and its compressed version is used to measure the compression quality. This is often referred to as *distortion* or *fidelity* [1].

Also, compression time is considered an important factor in assessing algorithms' performance. Unfortunately, due to several factors, including, for example, the size of the compressed file, disk speeds and the power of the CPU used, it is difficult to obtain precise time comparisons among different compressions models.

Other criteria are used as well. For example, the *computational complexity* of the algorithm used to perform compression and/or decompression is used to assess compression systems [3]. Also, the amount of memory required to implement compression is considered important [1]. The *overhead* is used to measure the amount of extra information stored to the compressed file to be used in decompression [3]. Furthermore, the *entropy* is used to quantify the theoretical bound of the source [3]. The difference between the average code length and the entropy of the source is referred to as *redundancy* [3].

The evaluation of compression algorithms can be either analytical or empirical [28]. 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 [28]. However, such an evaluation is not accurate and may not be broadly applicable. Instead, in an empirical evaluation, the algorithm is tested with a collection of real files. The difficulty of this method resides in the ability to generalise its 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 those files. This approach, however, is not only computationally unfeasible, but also theoretically flawed [28]. The reason for this is that there is no compression algorithm that can compress all files types with the same efficiency (see section 2.2.3). Thus, the average compression ratio across all files would be greater than or equal to the average of the original files! Therefore, empirical evaluation really needs 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 a relatively small set of files, a random selection of test files must always be used.

As we have seen above, different measurements can be used to assess the performance of an algorithm. However, as mentioned previously, it is difficult if not impossible to quantify the general performance under all possible circumstances. In this thesis, we are mainly concerned with the compression ratio and the compression time.

2.2.5 Pre-processing data

As mentioned previously, data compression models aim to identify regular patterns within the data. However, an algorithm may fail to capture these patterns if they are not obvious or isolated in separate locations within the file. Here, some form of pre-processing or transformation of the original data may provide further assistance to the compression. The preparation of data before applying a certain compression is referred to as *preprocessing* [3]. Generally, preprocessing operations can be divided into two types: i) data preprocessing for lossy compression and ii) data preprocessing for lossless compression. Typical preprocessing operations for the first type involve sampling, quantisation, prediction and transforms [3]. On the other hand, preprocessing for lossless compression usually involves reversible operations that rearrange the data in such a way as to simplify the task of the compression model. Thus, in practice, preprocessing algorithms are applied to the data before the actual compression. The disadvantage here is the extra computational efforts required to preprocess the data before the compression stage and after the decompression in order to allow for the full recovery of the original data.

Chapter 3

Intelligent Universal Data Compression Systems

Traditional compression techniques involve analysing the data and identifying regular patterns within them using a single model. Such techniques are designed to capture particular forms of regularities and, therefore, they are not practical when dealing with broad range of data types. Thus, new methods are needed to advance the data compression domain.

Data compression is a sophisticated process and a good universal compression algorithm would need to take intelligent actions to allow for the encoding of different types of data. Researchers have classically addressed the problem of universal compression using two approaches. The first approach has been to develop adaptive compression algorithms, where the system changes its behaviour during the compression to fit the encoding situation of the given data. The second approach has been to use the composition of multiple compression algorithms [29]. Recently, however, a third approach has been adopted by researchers in order to develop compression systems: the application of computational intelligence paradigms. This has shown remarkable results in the data compression domain improving the decision making process and outperforming conventional systems of data compression. However, researchers are facing difficulties that are limiting the practicality of these approaches. The following sections reviews some of the

previous attempts to address the universal compression problem within conventional and computational intelligence techniques.

3.1 Conventional Universal Data Compression

Most universal compression techniques collect statistical information to identify the highest frequency patterns in the data. In this section some of the previous attempts to address the universal compression problem by utilising statistical approaches are reviewed.

3.1.1 Adaptive Compression Algorithms

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, such as the frequencies of characters or substrings. In the second pass, the actual encoding takes place. A fixed encoding scheme is required to ensure that the decoder is able to retrieve the original message. This approach has been improved through so called *adaptive compression*, where the algorithm only needs one pass to compress the data [30]. The main idea of adaptive compression algorithms is that the encoding scheme changes as the data are being compressed. Thus, the encoding of the nth symbol is based on the characteristics of the data until position $n - 1$ [30]. 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.

Adaptive Huffman coding [31] is a statistical lossless compression where the code is represented using a binary tree structure. This algorithm gives the most frequent characters that appear in the file to be compressed using shorter codes than those characters which are less frequent. The top nodes in the tree store the most frequent characters. To produce the compressed version of a file, the algorithm simply traverses the tree from the root node to the target character. At each step the algorithm will encode 1 if it has moved to the left branch and 0 otherwise. The Huffman tree creates a unique variable length code for each character. The novelty of the Adaptive Huffman

compression is that nodes swap locations within the tree during the compression. This allows the algorithm to adapt its compression as the frequency of the symbols changes throughout the file.

Adaptive Arithmetic Coding (AC) is a statistical compression that learns the distribution of the source during the compression process [3]. AC has historic significance as it was the best alternative to Huffman coding after a gap of 25 years [3]. Adaptive AC encodes the source message to variable-length code in such a way that frequently used characters get fewer bits than infrequently used ones. Unlike other compression techniques which replace blocks of the data with smaller code words, AC encodes the entire message into a single real number. Hence, the AC is based on the mathematical fact that the cumulative probability of a particular sequence of characters has a unique and small subinterval within $[0, 1)$. The algorithm simply calculates the cumulative probability of each symbol within the message at a time. The count for each encountered symbol increases after it has been encoded. Then, the cumulative count table is updated accordingly. For example, consider a source that generates a message from the alphabet $\alpha = \{a, b, c\}$ with probability of $P(a) = 60\%$, $P(b) = 20\%$ and $P(c) = 20\%$, and the AC encoder aim to encode the message M, where $M = \{abac\}$. As illustrated in figure 3.1, each symbol leads the algorithm to a new smaller subinterval based on its probability. The compression process starts with the initial interval $[0, 1)$, and keeps iterating until it reads the entire source. The final output is a single real number selected from the smallest identified subinterval. In the decompression process, the algorithm receives the encoded real number as input and starts from the initial interval. The algorithm divides the main interval into smaller subinterval according to which section the input falls in. The new subinterval become the new main interval and output the corresponding symbol. This process keeps iterating until the entire message is decoded.

Another adaptive compression is Prediction by Partial Matching (PPM) which was proposed by Cleary and Witten in 1984 [1]. This algorithm is classified as statistical compression. The main ideas of this algorithm are context modelling and prediction. PPM tries to predict the

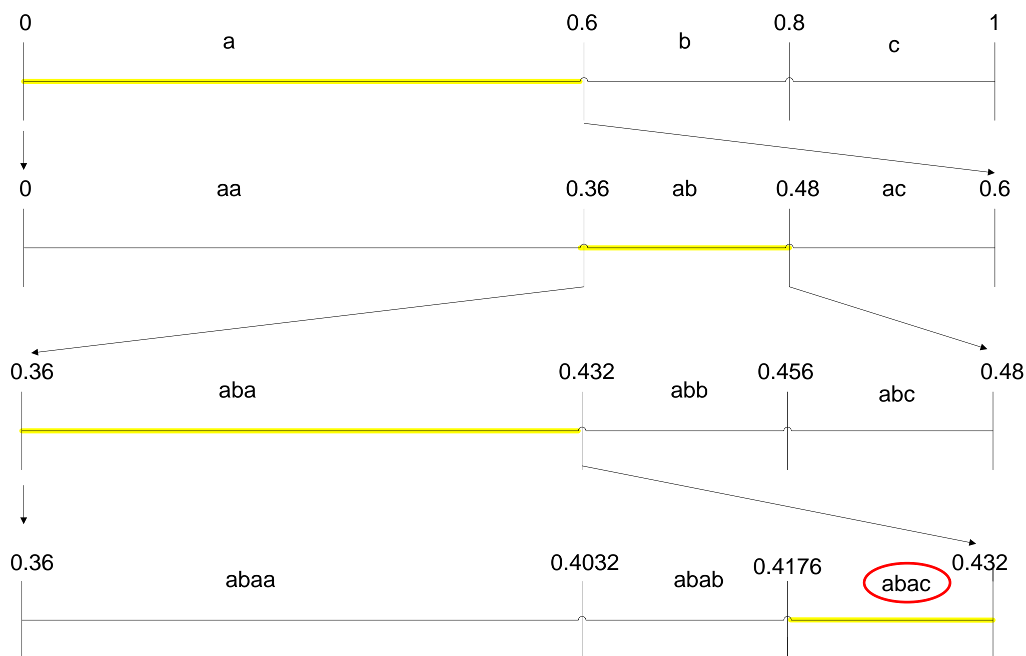

FIGURE 3.1: AC encoding process.
The region is divide into sub-regions proportional to symbol frequencies, then the sub-region containing the point is successively subdivided in the same way. Each division map all possible messages at the particular point.

probability of a particular character being in a specific location from the previous n symbols previously occurred. To this end, a table of statistical information is created, which stores strings of size o with the probabilities of their following characters. The number of previous symbols o is called the order of the PPM model. If the symbol to be encoded has not been previously encountered in the context then an escape symbol is encoded and the o is reduced. In the next iteration the algorithm uses a table of size $o - 1$. This process keep iterating until the algorithm encounters the target symbol or it concludes that the symbol has never been seen before. In this case (i.e., the symbol was never seen before), the algorithm updates the statistical tables with a new enrty (i.e., add the new symbol with its probability and its o previous symbols). The probability of the newly added symbol is $1/M$, where M is the size of the source alphabet [1]. The actual encoding of the symbols is done with Arithmetic Coding compression. A variant of the algorithm called PPMD [32] is used where it increments the probability of the escape symbol every time it is used.

Cormack and Horspool introduced Dynamic Markov Compression (DMC) in 1987 [30]. DMC is another statistical compression algorithm using on context modelling and prediction. Similarly to PPM, it uses Arithmetic Coding to encode predicted symbols. The difference, however, is that the model operates at the bit level rather than on bytes. Although this model has similar performance to PPM it is not widely used.

Lempel and Ziv developed a universal compression system in 1977 known as LZ77 [33]. Later, in 1978, an improved version was presented which is referred to as LZ78 [1]. These algorithms are classified as dictionary-based compression. The main idea is to replace frequent substrings in the data with references that match the data that has already have been passed to the encoder. The encoder creates a dictionary for the common substring in the file and uses it as reference for the data. Both encoder and decoder must share the same dictionary to perform the compression/decompression process. In LZ77 a sliding window of predefined length is used to maintain the dictionary. Data are scanned via a look-ahead buffer and compared with the previous data within the sliding window. Once a match is found the dictionary is updated. LZ78 eliminated the need for a fixed size window and builds the dictionary out of all previously seen symbols. LZ78 works very well where the frequent symbols are distributed in isolated locations in the file. Unlike LZ77, strings in LZ78 can be longer which gives flexibility and higher performance. The adaptability of the LZ algorithms comes from the fact that the algorithms update the dictionary contents during the compression process to tune the algorithm's behaviour as the frequency of the symbols change throughout a file.

In some sense adaptive compression algorithms represent a form of intelligent systems that adapt their behaviour based on the given encoding situation. The primary advantage of adaptive compression is that it uses a single pass to compress data. This accelerates the compression process and makes it appealing for use in the cases where speed is favoured over performance. The use of a single pass has the disadvantage of sacrificing the ability to see ahead in the file to determine future fluctuations which may contain valuable information for the encoding scheme. In adaptive compression once the algorithm makes the decision to encode the data in a particular

way it will not be able to change it, even if the algorithm later learns a better way of encoding this information. Another problem with adaptive compression algorithms is that they are sensitive only to changes in relation to the particular redundancy type which they are specialised to detect, hence, they are not able to exploit different forms of redundancy [29]. Also, because they learn the alphabetic distribution during the compression time, they cannot handle drastic changes in the files streams as, for example, is the case with changes from text to picture. Consequently, adaptive schemes are not recommended for heterogeneous files where multiple types of data are stored in a single file and their constituent parts have different degrees of compressibility.

3.1.2 Composite Compression Algorithms

In many real world applications, it is ineffective to use a single compression model to adapt the regularities of the data. Therefore, a practical approach is to apply a composite model, which can be defined as a combination of several models where only one model can be active at any given time [1]. This is another approach to develop universal compression algorithms and to achieve higher compression. It has been reported that this approach show superior performance to standard compression algorithms. A successful composition of compression algorithms can be achieved in two ways. Firstly, by running a number of compression algorithms successively. Secondly, by combining a number of simple compression algorithms and heuristically selecting them where they are expected to perform best.

A well-known composite compression system is Bzip2. Bzip2 is a free, open-source, composite and lossless compression algorithm developed in 1996 [34]. It has been widely used in many commercial compression applications, such as WinZip [35]. In this algorithm, the data are compressed through several compression and transformation techniques in a particular order. The reverse order is used in the decompression process. Bzip2 compresses files using the Burrows-Wheeler transform and the Huffman coding. This technique has been found to be better than LZ77 and LZ78 [34]. Bzip2 is known for being slow in the compression process and much faster in decompression.

Katz in [36] proposed an algorithm that combines LZ77 with Huffman coding as an improvement for the PKZIP archiving tool. This algorithm is referred to as *Deflate*. In the original implementation of LZ77, the algorithm tries to match strings within a sliding window with a look-ahead buffer. Matched strings are used to build a dictionary. Each repeated string is replaced each with a triplet (pointer [1], length and next symbol). The next symbol element is needed in case there is no exact match in the dictionary (e.g., William and Will). In Deflate a variant of LZ77 is used, which eliminates the third element and encodes a pair (pointer, length). Unmatched characters are written in the compressed stream [2]. In the original implementation of the Deflate algorithm, compressed data consisted of a sequence of blocks corresponding to successive blocks of input data. These blocks can be of different lengths based on the various prefix codes used and the memory available to the encoder. The algorithm has three options for each input block: *i) Apply no compression*, which is used if the data is already compressed, *ii) Compress with fixed Huffman code* and *iii) Compress with dynamic Huffman code*. Each uncompressed block is individually compressed using the previously described modification of LZ77 and then Huffman coding. Thus, each block is composed of two parts: *i) a Huffman tree* that describe the data and *ii) the compressed data itself*. Deflate has been widely implemented in many commercial compression applications, such as gzip, the HTTP protocol, the PPP compression protocol, PNG (Portable Network Graphics) and Adobe's PDF (Portable Document File) [2].

Another composite compression system is the Lempel-Ziv Markov-chain Algorithm (LZMA) [2]. LZMA uses a similar approach to Deflate. The difference is that it uses range encoding instead of Huffman coding (the range encoder is an integer-based version of Arithmetic Coding) [2]. This enhances the compression performance, but at the expense of increasing the encoder's complexity. In LZMA, the input stream is divided into blocks, each block describing either a single byte, or an LZ77 sequence with its length and distance.

The main disadvantage of standard composite compression algorithms is that they are unreliable when dealing with heterogeneous files, i.e., files that are composed of multiple data types

[1] pointer is an index in the dictionary

such as archive files (e.g., ZIP and TAR). This is because most standard composite compression schemes follow deterministic procedures that involve applying several compression models in a particular order. These procedures are selected based on the designer's experience or experimental evidence which demonstrated their superior performance under certain conditions. Whilst these methods have proven to be successful in achieving high compression ratios, the use of deterministic steps to perform compression entails the disadvantage of making the encoding decision fixed and unable to deal with unpredictable types of data.

Another approach to developing composite compression system is allowing the system to heuristically select and apply compression algorithms where they are expected to perform best. This idea has been explored by Hsu in [29]. Hsu's system segmented the data into blocks of a fixed length (5 KB) and then compressed each block individually with the best compression algorithm. Four compression algorithms were used in Hsu's system, namely, Arithmetic Coding, Run-length encoding, LZW and JPEG for image compression. The system works in two phases. In the first phase, the blocks are scanned to determine the compressibility and the contents of each block. The compressibility of the blocks is calculated by measuring three different quantitative metrics: alphabetic distribution, average run length and string repetition ratio. The system considers the blocks already compressed if these measures were under a predefined threshold. Consequently, already compressed blocks are skipped. The files contents are determined using a modified Unix ***file*** command. This command is able to classify ten different types of data. The modified file command works by examining the first, middle and last (if it exists) 512 bytes and thereafter comparing their patterns with collections of known patterns from the Unix operating system. In the second phase, the actual compression takes place, where the system passes the blocks to the appropriate compression model based on the gathered information from the first phase. Experimentation with 20 heterogeneous test files revealed that the proposed system was able to outperform other commercial compression systems with 16% saving on average. The main disadvantage of Hus's system is that the size of the blocks is fixed to 5KB, which limits the algorithm's ability to identify the true boundaries of heterogeneous fragments within the data. Moreover, the modified Unix command used detects ten file types only and is not reliable

enough to guarantee that the blocks are passed to the optimal compression algorithm. This is because the system assumes that one particular compression algorithm is suitable for all files of a particular type. This assumption, however, is flawed as the compressibility of the data depends on the forms of regularities within the data and whether the compression algorithm is designed to capture them. These regularities are not necessarily to be correlated with a particular file type. Thus, at least in principle, two text files may be better compressed with two different compression algorithms depending on their contents.

3.2 Data Compression with Computational Intelligence Paradigms

Computational Intelligence (CI) researchers attempt to understand and simulate intelligent behaviour through the modelling of natural intelligence, such as evolution, insects swarms, neural systems and immune systems. This has resulted in a variety of paradigms including Artificial Immune Systems, Neural Networks, Particle Swarm Optimisation, Ant Colony, Fuzzy systems and Genetic Algorithms [9]. The main problems solved by CI paradigms include but are not limited to optimisation, classification, prediction and pattern recognition. Researchers have achieved significant successes in solving real world problems using these techniques.

Due to the great potential of current CI paradigms in solving complex problems, researchers have tended to apply some of these techniques to develop new intelligent data compression systems. However, thus far, only a little has been done to address this problem. Current research has focused mainly on investigating the applications of neural networks and genetic algorithms in order to explore the future of data compression. Applications of neural networks and genetic algorithms illustrate that CI paradigms in this area may be ready to play a significant role in assisting and complementing traditional techniques. In this section, we will review some of the previous attempts to use neural networks and genetic algorithms techniques to address this problem.

3.2.1 Neural Networks Applications in Data Compression

Artificial Neural Networks (ANN) have the potential to extend data compression algorithms beyond the standard methods of detecting regularities within the data. Although, they have often been avoided because they considered too slow for practical use. Nevertheless, neural networks have been applied to data compression problems. Existing research on neural network applications in compression can be summarised into three categories: transform coding, vector quantization, and predictive coding [37].

3.2.1.1 Code Transformation

In the first category (code transformation), the neural network is asked to identity a mapping between input and output which is then used for compression. Neural networks fit well with image compression because they have the ability to process input patterns to produce simpler patterns [38]. The network is composed of three layers; input, hidden and output [38]. Here, the desired output is to be identical to the input itself. Compression is achieved when the number of neurons in the hidden layer is smaller than the dimensionality of the input and output layers [38]. The input images are divided into blocks of $4 \times 4, 8 \times 8$ or 16×16 pixels [38]. The size of the input layer is equal to the number of pixels in the blocks. The neural network is trained to scale an input of N dimensions into narrower outlets of M dimensions at the hidden layer. It then produces the same input at the output layer. The quality of the network is calculated by measuring the difference between input and output. The idea is in the activiation of the neurons in the hidden layer will be saved or transmitted as the compressed version of the image. The original image will be reconstructed using the output layer to achieve the decompression process. However, it should be noticed that the activities of the hidden layer are real numbers between -1 and 1 which will most likely require greater storage space than the image itself. Therefore, the outputs of the hidden layer are encoded (quantised) to reduce the size of the data. This approach has been implemented in [39], where experimentation with three black and white images was conducted to prove its practicality.

This approach has been improved by Hassoun in [40] who proposed an alternating Hebbian algorithm in order to improve the training approach for the decompression part in the network. Also, the learning rule was extended to capture nonlinear relationships among the component in the training patterns. In [38] a cumulative distribution function was estimated for the images and used to map images pixels. Experimentation showed that this improves the compression ratio with back-propagation networks and accelerates convergence.

Namphol *et al*. [41] proposed a hierarchical architecture in order to improve the basic back-propagation network. In this work, two more hidden layers were added to the overall network. The three hidden layers are referred to as the combiner layer, compressor layer and decombiner layer respectively. All hidden layers are fully connected. The aim is to exploit the correlation between pixels via the combiner layer and the correlation between blocks of pixels via the decombiner layer. Thus, compression is achieved by dividing the target image into a number of blocks and each block is further divided into smaller blocks. The proposed system used nested training algorithm to improve the training time. Experimentation with computer generated images and real world images demonstrated the stability and generlisation of the system, however, the compression rate achieved by the system was not significant (1 bit/pixel) to that of standard data compression techniques.

An adaptive back-propagation neural network has been proposed to overcome the limitations in the basic back-propagation in [42]. The basic idea is that different neural networks have different performance and only certain types of networks work best with a particular set of images. As a result, a group of neural networks with an increasing number of hidden layers in the range (h_{min}, h_{max}) are used in a single system. The aim is to match the input image blocks with the most appropriate network based on their complexity. Several training schemes have been proposed to train the new adaptive architecture, including parallel training, serial training, activity-based training and direction-based training [42].

3.2.1.2 Vector Quantisation

In the second category (vector quantisation), neural networks are used to encode the input image into a set of codewords. The input image is divided into blocks and then fed into the network as input. The input layer is K dimensional and $M < K$ output neurons are designed to calculate the vector quantisation codebook [42]. Each neuron represents one codeword. The learning algorithm aims to adjust the weights of the network in such a way as to associate the ith neuron from the input layer with the cth codeword [42]. Around this basic architecture, many learning algorithms have been developed to optimise the learning process, such as competitive learning [43, 44], fuzzy competitive learning [45], and predictive vector quantisation [46].

Laskaris and Fotopoulos in [47] proposed a modified learning scheme to improve codebook design, which describes the mapping from the input data to their compressed representation. In this work, the sequential presentation of the training patterns is controlled via an external, user-defined criterion. The proposed modification considers blocks with spatial structure (e.g., well-formed edges) more important than homogenous regions. Therefore, they require more information in the codebook in order to represent them. The *roulette-wheel* [10] sampling technique (borrowed from the genetic algorithm literature) has been applied to favour some of the training samples. A technique was also used to maintain the diversity of the training samples. The proposed training scheme was found to be specialised in the representation of edges while the standard training scheme is better tailored to luminance variations. Comparisons with the standard training procedure based on the quality of reconstructed image showed that roulette-wheel-based training achieved significant improvements in codebook design.

3.2.1.3 Predictive Coding

In the third category (predictive coding), neural networks were used to improve existing compression technology. Schmidhuber and Heil [48] combined 3-layered predictive neural networks with statistical coding schemes to compress text. In this work, neural networks trained

by back propagation were used to approximate characters probabilities when given n previous characters. The predictor input comes from a time-window of size t that scans the data with steps of size p. The outputs of the network are then fed to standard statistical coding algorithms to encode the data. Two different coding methods were introduced in conjunction with the predictor network: Huffman coding and Arithmetic Coding. In [49], the same authors have extended their work and introduced a third method. In this work, prediction networks have been used in a similar manner as in PPM (see section 3.1.1). Thus, a time-window corresponding to the predictor input scans the data sequentially. The output produced by the prediction network is compared with the actual data. Wrong predictions are stored in a separated file (Meta file) to be used in the decoding process. Then, Huffman coding is applied to the Meta file in order to reduce the total size of the compressed data. The decoding process starts from n default characters to be fed to the predictor network. The predictor will sequentially predict the next character. Information in the Meta file is used to correct wrong predictions and restore the original file without loss. Experimentation with these three methods revealed that they outperformed LZW and Huffman coding. The major disadvantage, however, is that the algorithm was too slow for practical use. Training on 10KB to 20KB consumed 3 days of computational time.

Mahoney [50] proposed a faster text compression method based on neural networks that could compress the same amount of data in about 2 seconds. In this work, a 2-layers predictive neural network that predicts one bit at a time was used. The proposed predictive network learns and predicts in a single pass. Experimentation with the proposed method showed better compression ratios, but slower than LZW. However, results were almost identical to PPM both in terms of speed and achieved compression ratios.

3.2.2 Genetic Algorithms Applications in Data Compression

Data compression algorithms often require a large search space to be explored in order to find some forms of redundancy within the data or to select an optimal encoding scheme based on the given files. Evolutionary Algorithms (EAs) can be used as a search engine for this purpose or

even as a method to evolve new compression algorithms. Nevertheless, data compression is a highly sophisticated procedure and evolving a data compression algorithm is not an easy task. Yet, few attempts have been made to use EAs to evolve data compression models. Existing research mainly focuses on investigating the applications of Genetic Algorithm (GA) and Genetic Programming (GP) in order to explore their potentials in solving data compression problems. As we will see, the good compression ratios achieved by early attempt with these techniques in comparison to other conventional compression algorithms show the importance of further exploring the applications of EAs in the field of data compression.

3.2.2.1 Lossy Image and Video Compression

Koza [14] 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 the original as closely as possible. Small, 30×30 pixel, images were treated as symbolic regression problems with just the basic arithmetic operators in the functions set. The evolved function can be then considered as a lossy compressed version of the image. The technique, which was termed *programmatic compression*, was tested just 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 [13]. In this research the authors presented the target data as a continuous sequence of numbers (fitness cases) and asked GP to evolve a program that could fit these data. A problem, however, appears with very large fitness cases (which is the case with most images) where GP may fail to find a solution of acceptable quality. As a result, an alternative approach was presented where the fitness cases were divided into equally sized subsets and solutions were evolved for each of them individually. The main disadvantage of the system is that the time required to compress a picture was up to 10 days.

In [51] the use of programmatic compression was extended from images to natural videos. A program was evolved that generates intermediate frames of a video sequence. A sequence of

gray-scale frames was considered. Each frame is composed of a series of transformed regions from adjacent frames. The possible motions within a single frame can be expressed as a combination of scaling, shearing, rotating and change in luminance. The function set of the system was composed of one function **(Transform)** that encapsulates all these operations. The **Transform's** output is controlled by several parameters that set the object to be moved, the start location, the end location, and the luminance change. Programs are evaluated by measuring the difference between the approximated frames and the target frames. If a program achieves satisfactory approximation in one frame the system uses it for the following frame, utilising the fact that subsequent frames usually share similar characteristics. The results were encouraging as a good approximation to frames was achieved. Naturally, although a significant improvement in compression was achieved, programmatic compression was very slow in comparison with other methods, the time needed for compression being measured in hours or even days. In [52] an optimal linear predictive technique was proposed. Thanks to the use of a simpler fitness function, acceleration in GP image compression was achieved.

GAs have been used for image compression. In [53] Mitra *et al.* proposed a new GA-based method for fractal image compression. The proposed method utilises the GA for finding self similarities in the given image. Then, the system divides the given images into blocks and tries to find functions that approximate the target block. Here, the squared mean error was used as fitness function to evaluate the quality of the individuals.

Iterated Functions Systems (IFS) are important in the domain of fractals and the fractal compression algorithms. [54] and [55] 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 reverse problem can rarely be solved exactly. No practical application or compression ratio results were reported in [54] and [55]. Using similar principles, Sarafoulous [56] 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 the finer

details).

The application of GAs to accelerate fractal coding time has been explored in several works. In [57, 58] a GA has been used to compress fractal images by finding Local Iterated Functions Systems that encode a single image and reducing the time needed to 30% compared with other methods. In [59] a GA implementation to speed up the compression without significant loss of the image quality was proposed. The GA searches the optimal parameters for domain block coordinates and isometric flip. The luminance and contrast parameters are computed with standard equations. In this implementation, chromosomes include 5 genes, from which only 3 genes are submitted for genetic modification and the other 2 are computed with a standard equation. This was found to improve compression time, in the reported experiments. The time needed to perform compression ranged from 9 seconds to 23 minutes depending on the settings used for the system.

Evolvable hardware (EHW) has been explored in the practical applications of data compression. In [60, 61] Salami *et al.* applied EHW to data compression applications. In this work, a new type of hardware evolution was proposed which is referred to as function-level EHW [62]. EHW was used as a predictive function for image compression. In this approach, the images to be compressed are divided into blocks and then EHW finds a function for each block. Thus, by finding a different predictive function for each region of the image EHW applies an adaptive predictive technique for the image. The system can be treated as lossy if it ignores errors, or lossless if it tracked the errors for the decompression process. Sekanina [63] extended this work and proposed a technique to allow balancing between achieved compression ratio and image quality. Four threshold values were added to the terminal set (randomly initialised). Each value corresponds to a quarter of the image. The system transfers any particular pixel (represented as position and value) to the compressed image if its prediction error is greater than the relevant threshold. Thus, transferred pixels are ignored during the decompression process. Threshold values change during evolution. The system only allows the transfer of pixels up to a maximum limit. To maintain a balance between compression and quality, the number of

transferred pixels was treated as a penalty value in the fitness calculations. Experimentation with the proposed technique revealed that the quality of the images increase with the size of the population. However, JPEG still outperformed EHW in terms of achieved compression ratio. Furthermore, the time needed to complete the evolution process was considerably larger than for standard image compression methods.

3.2.2.2 Lossless Compression

A first lossless compression technique was reported in [64], where GP was used to evolve non-linear predictors for images. These are 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 terminals used for the GP system were the values of four neighbouring pixels, constants, arithmetic operators, and Min/Max functions. The prediction errors together with the model's description represent a compressed version of the image. These were further compressed using Huffman encoding. The system required several hours to compress a single image. Results on five gray images from the NASA Galileo Mission database were very promising, with GP compression outperforming some of the best human-designed lossless compression algorithms.

Text compression via text substitution has been investigated in several applications using both GA and GP. Ucoluk and Toroslu [65] used a GA to evolve a dictionary of syllables for the Huffman coding. The GA was used to search for a combination of the alphabet in such a way as to improve the Huffman coding performance. The chromosomes of each individual represent a selected combination of a predefined set of text. Individuals were evaluated by measuring the entropy of the encoded text. Experimentation with Turkish text corpora revealed that the achieved compression ratios when using evolved dictionaries were superior to those provided by standard methods. This work has been extended in [66] with Czech and English text where the approach has been tested with the LZW algorithm. Also, in [67] the same method has

been tested, where GA was used to construct dictionaries for text substitution. Experiments demonstrated that GA always outperforms standard text dictionary-based compression methods.

Intel patented a method in [68] to compress microcode based on GAs. The proposed method utilises a GA to find common patterns in the microcode's bit strings and store them in a table with a unique ID for each pattern. To achieve this task, the system represents the microcode's bit strings in a matrix format and groups similar columns of microcode storage into clusters to minimise the total size.

3.2.2.3 Optimise Existing Compression Algorithms

Zaki and Sayed [69] 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. In this work, the search population was a collection of nodes that describe different substrings. Then, the Huffman tree is generated to encode high frequency substrings with shorter references. For the proposed representation, the authors used specialised search operators. Reported results demonstrated that the Huffman tree generated with this system was able to achieve higher compression than the standard Huffman coding algorithm in some cases by small margins. However, due to various restrictions on the system, standard Huffman coding was not significantly improved.

In [70] 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. N-gram tables are stored to be used in the decompression process. In the first step, the system calculates the frequencies of 2-, 3-, 4- and 5- grams. The aim is to allow the GA to find the best combination of N-grams replacements in such a way as to achieve a high compression ratio while minimising the total number of N-grams used. The compression process is carried out on the HTTP server side. The decompression process is performed during the loading of the web page. The proposed method was tested on Persian text. The best result reported by the proposed system was 52.26%. However, no comparisons with other compression techniques were reported.

Many conventional compression systems have a number of parameters, e.g., window size, prediction order, dictionary limits, etc. Such parameters typically influence the compression performance. The ideal setting for these parameter is depending on the structure of the file to be compressed in most cases. For this purpose, GAs and GP have been used to optimise the parameters of existing compression systems (e.g., as previously reviewed in [65, 66, 69]). Recently, Burtscher and Ratanaworabha [71] proposed a system based on a GA to tune the parameters of the FPC compression [72]. FPC divides the data to be compressed into blocks and processes each block individually. The GA was used to initialise the population of settings (i.e., each individual represents a different configuration). Each individual in the first population applies its settings to the FPC and compresses the first block. The designed fitness function favoured individuals that yield highest compression ratios. The next block is compressed with the new generation. Thus, the size of the blocks is directly related to the number of generations. Experimentation with several versions of the FPC algorithm revealed that evolution was able to find optimal settings for each block, which led the algorithm to achieve higher compression ratios in comparison with the standard versions of the algorithm. Typically, this evolutionary process slows the compression phase significantly.

A GA system for the optimisation of combined fuzzy image compression and decompression was introduced in [73]. Here, an image is divided into squares and a fuzzy system replaces the whole square with a single pixel of particular colour decided according to its neighbours. The square is a fuzzy set defined through a membership function that describes to what degree each neighbour pixel belong to the square. Thus, the content of the square is mapped into a single pixel in the compressed image. The reverse process is used in the decompression phase. The pixels of the compressed squares are restored by calculating their values with the appropriate membership function according to the value of the single pixel in the relevant square. Generally, some pixels may belong to more than one square with different degrees depending on their position. A GA has been used to optimise the parameters of the membership functions for both the compression and the decompression process. The Mean Square Error (MSE) between original and compressed images was used to guide evolution. Experimentation demonstrated

that the system has good generalisation capabilities when tested with images outside of the training set. However, one of the main problems with restored images is the blurring of contours.

3.2.2.4 Evolve Data Transformation 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 [74] Parent and Nowe evolved pre-processors for lossless compression using GP. The objective of the pre-processor was to reduce losslessly the entropy in the original image. The function set of the GP system was divided into two categories, functions that read the input data into a buffer and functions that process the data read from it. All used operators are reversible in order to guarantee applicability for lossless compression. In tests with five images from the Canterbury Corpus [28] GP was successful in significantly reducing image entropy. As verified via the application of Bzip2, the resulting images were markedly easier to compress.

3.2.2.5 Wavelet Compression

Klappenecker [75] used GP to find optimal parameters to optimise the performance of conventional wavelet compression schemes, where internal nodes represented conjugate quadrate filters and leaves represented quantisers. The aim was to minimise the rate-distortion while maintaining high compression ratios. Results on a small set of real world images were impressive, with the GP compression outperforming JPEG at all compression ratios.

Wavelets are frequently used in lossy image and signal compression. In [76] Grasemann and Miikkulainen used a co-evolutionary GA to find wavelets that are specifically adapted to fingerprint images. The evolved wavelets were compared with human designed wavelets that were used by the FBI to compress fingerprint images and standard JPEG2000. In this work, the GA initialises several populations of lifting steps [2] in parallel and then randomly combined to form

[2]The lifting scheme introduced by Sweldens in [77] offers an effective way to construct complementary filter pairs. A finite filter called a Lifting step and can be used to construct a new filter pair. Thus, new wavelets can be constructed starting from a known complementary filter pair.

new wavelets. Each sub-population is evolved in isolation. At each generation, the best individuals are selected from each population to join the new wavelet. Results showed that the GA was able to outperform its competitors in terms of compressed image quality by a factor of 15% to 20%. The authors reported that evolution required approximately 45 minutes to complete the learning process.

3.2.2.6 Vector Quantization

Feiel and Ramakrishnan [78] proposed a new vector quantisation scheme using a GA (GAVQ) to optimise the compression of coloured images. In this work, the GA was used to find optimal, or more precisely near optimal codebooks that describe the mapping from the input data to their compressed version. Experiments showed that the results obtained are better than the LBG algorithm by of 5% to 25%.

Keong *et al*. [67] used GA to optimise the a Generalised Lloyd Algorithm (GLA). [3] GLA receives the input vectors and initialises the codebook randomly, such that it maps the input sequence to a compressed digital sequence. GLA refines the codebook though an iterative process in order to reduce the average distortion. The authors introduced a Genetic GLA (GGLA), which uses a GA to find optimal codebooks. Each chromosome in the GA population represents a codebook and the set of genes corresponds to the codewords. Simulations with the three versions of the proposed system have been conducted with first-order Gaussian-Markov processes. This showed that GGLAs outperformed GLA in most cases.

3.2.3 Limitation of the current systems

The main limitations of the previous techniques are twofold: *practicality* and *generality*. The implementation of CI techniques in the data compression domain has proven to smarten compression programs by allowing them to make educated decisions with regard to how to encode

[3]GLA is a lossy image compression method based on vector quantisation.

the data. These have often consequently outperformed conventional compression schemes. Naturally, this gain in performance comes at a fairly high cost in termes of increased program complexity and intensive load of computations. This makes CI-based compression algorithms slow and inapplicable for practical use in most cases. Nevertheless, some applications in the CI literature have been reported to be suitable for practical use (e.g, [38, 50, 65, 66, 70]). Yet, all of these applications are tailored to handle a single type of data, such as wavelets, images, text, ...etc, or are designed for a few specific classes of data. Hence, the second limitation is that none of the previously reviewed CI-based methods is a reliable universal compression technique. None has straightforwardly addressed the problem of heterogeneous files compression, despite their increased use.

There is an urgent need to have smart compression systems that can deal effectively with any regular data type and can tailor different encoding techniques based on the given situation. In this thesis we address the problem of heterogeneous data compression and propose a series of intelligent universal compression systems that are hopefully steps towards a practical solution. As we will see, the end results of this research is a pattern recognition model based on GP that can identify different patterns within the data. The proposed system learns to generalise the characteristics of these different patterns and matches them with different compression models. Then, eliminates the disadvantage of slow performance by providing a time-efficient compression technique based on GP that works best on heterogeneous data, archives as well as being competitive with other compression systems on single type files.

Chapter 4

The GP- zip approach: Compression during Evolution

As stated previously, the central objective of the work presented in this thesis is to use artificial evolution to develop a system that is able to automatically generate lossless compression algorithms based on the given encoding situation by using GP to learn the structure of the given data and adapt the system's behaviour in order to provide the best possible compression level (see section 1.4). The *ideal universal compression* has been defined (see section 2.2.3) as a system that reads data from the files and rapidly identifies different incompatible fragments within the data before applying a specialised compression technique for each fragment. Typically, the final output of the algorithm should be smaller than the original data (or at least of the same size if the data is not compressible).

Here we investigate the main idea to use GP to find different patterns within the data and match them with different compression systems.

In this chapter, two steps toward developing an intelligent universal compression system are presented: GP-zip and GP-zip*. As will be explained shortly, the proposed systems take advantage of the existing compression techniques and utilise them where they are expected to perform best. The presented work on GP-zip and GP-zip* has been published in [79] and [80], respectively.

4.1 GP-zip

In this version of the system we wanted to understand the benefits and limitation of combining existing lossless compression algorithms in a way that ensure the best possible match between the algorithm being used and the type of data it is applied to. Thus, we implemented a simple decision making mechanism to ensure the effectiveness of the basic idea.

In GP-zip, the system divides the given data file into segments of a certain length and asks GP to identify the best possible compression technique for each segment. The function set of GP-zip is composed of primitives that naturally fall under two categories: *i) compression algorithms* and *ii) transformation algorithms*. The first category contains the following four compression algorithms: Arithmetic Coding (AC) [81], Lempel-Ziv-Welch (LZW) [82], Unbounded Prediction by Partial Matching (PPMD) [32] and Run Length Encoding (RLE) [83] These four compression algorithms were selected because they belong to different compression categories. Each of them is known to commonly perform well with a particular family of data types. For example, PPMD performs best when dealing with natural language text. Run-Length encoding is very powerful with black and white pictures. It is commonly used in fax machines combined with other techniques. It is also used to compress icon pictures. LZW is widely used with GIF and TIFF images. Arithmetic coding is useful with short English text. In the second category, two transformation techniques are included: Burrows-Wheeler Transformation (BWT) [84] and Move-to-Front (MTF) [85]. Thus, unlike traditional GP systems, where the function set is a collection of simple primitives, in GP-zip the function set is composed of sophisticated algorithms. The algorithms themselves may use different forms of learning mechanisms and multiple layers of compressions. However, they are treated as black boxes. Each compression function receives a stream of data as inputs and returns a (typically) smaller stream of compressed data as an output. Each transformation function receives a stream of data as input and returns a transformed stream of data as an output. So, this does not directly produce a compression. However, often the transformed data are more compressible, and so, when passed to a compression algorithm in the function set, a better compression ratio is achieved.

4.1.1 Data Segmentation

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

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

4.1.2 The Approach

As explained above, the function set is composed of four compression algorithms and two transformation algorithms. It is clear that there are 12 different ways of compressing each segment. Namely, the system can apply one of the four compression functions without any transformation of the data in a segment, or it can proceed the application of the compression function with one of two transformation functions.

GP-zip randomly selects a segment length from the set of possible lengths and then applies relatively simple evolutionary operations. The system starts by initialising a population randomly. As shown in figure 4.1, individuals are represented as a linear 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. The fitness value is simply the total achieved compression ratio (the compression ratio is the ratio of the compressed file size to the original size, see equation 2.1).

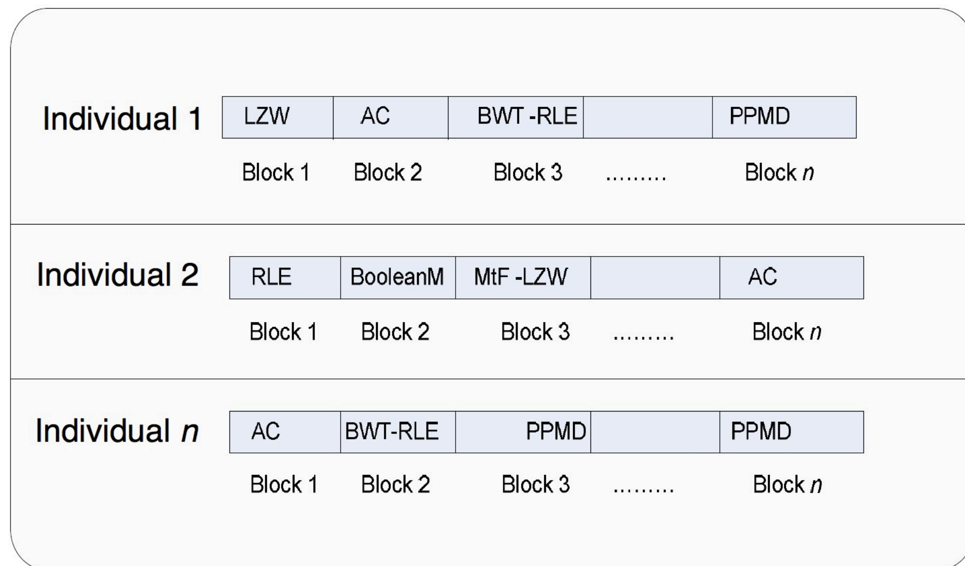

FIGURE 4.1: GP-zip individuals are linear representation, with a compression and/or transformation function in each position.

Once GP-zip finds the best possible compression for the selected segment-length, the system suggests another length for the segments and the same process is iterated.

Clearly, testing all of the possible segment lengths is very time consuming. Therefore, GP-zip selects the new lengths by performing a form of binary search over the set of possible lengths. This is applied for the first time after the system has tested two segment lengths. The third segment length is chosen where (based on the results obtained with the first two segment lengths) more promising results are expected.

Since the proposed system divides the data into segments and finds the best possible compression model for them, it is necessary for the decompression process to know which segment was compressed with which compression and transformation functions. Thus, a header for the compressed files has been designed, which provides this information for the decompression process. The size of this header is not fixed, rather it depends on the number of the identified segment. However, there is an insignificant overhead in comparison with the size of the original (uncompressed) file. Moreover, it should be noted that the advantages of dividing the data into smaller

segments manifest themselves in the decompression stage. Firstly, the decompression process can easily decompress a section of the data without processing the entire file. Furthermore, the decompression process is faster than the compression since, in principle, GP-zip can send each segment that needs decompressing to the operating system pipeline sequentially.

The header consists of a sequence of 2-byte blocks corresponding to successive blocks of input data. Each of these blocks is divided into two parts. The first part is composed of 4 bits and encodes which compression/transformation has been used for each segment in the original data. [1] The second part 12 bits encodes the length of the compressed segments. The size of the compressed segments is needed to allow the decompression process to identify which part of the final compressed file corresponds to which segment in the original file. An EOH (End Of Header) character is added the end of the header information. Thus, in the decompression process, the system starts by sequentially reading 2-byte blocks until the EOH character. Each block tells the system how to decompresses a particular segment and what is the size of this segments. GP-zip decompress each segment sequentially and appends the decompressed data into a buffer. At the end of this process, the original file will be reconstructed. In future research we can allow GP-zip to decompress different parts of the data (according to the user's need) without the need to decompress the whole file.

GP-zip was quite successful (more details will be reported in section 4.3). However, it suffered from a major disadvantage: the length of 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 is the staged search process which GP-zip uses to identify the best segment length. Figure 4.2 illustrates GP-zip's flowchart.

[1] 4 bits are used to encode the 12 different possibilities to compress each segment.

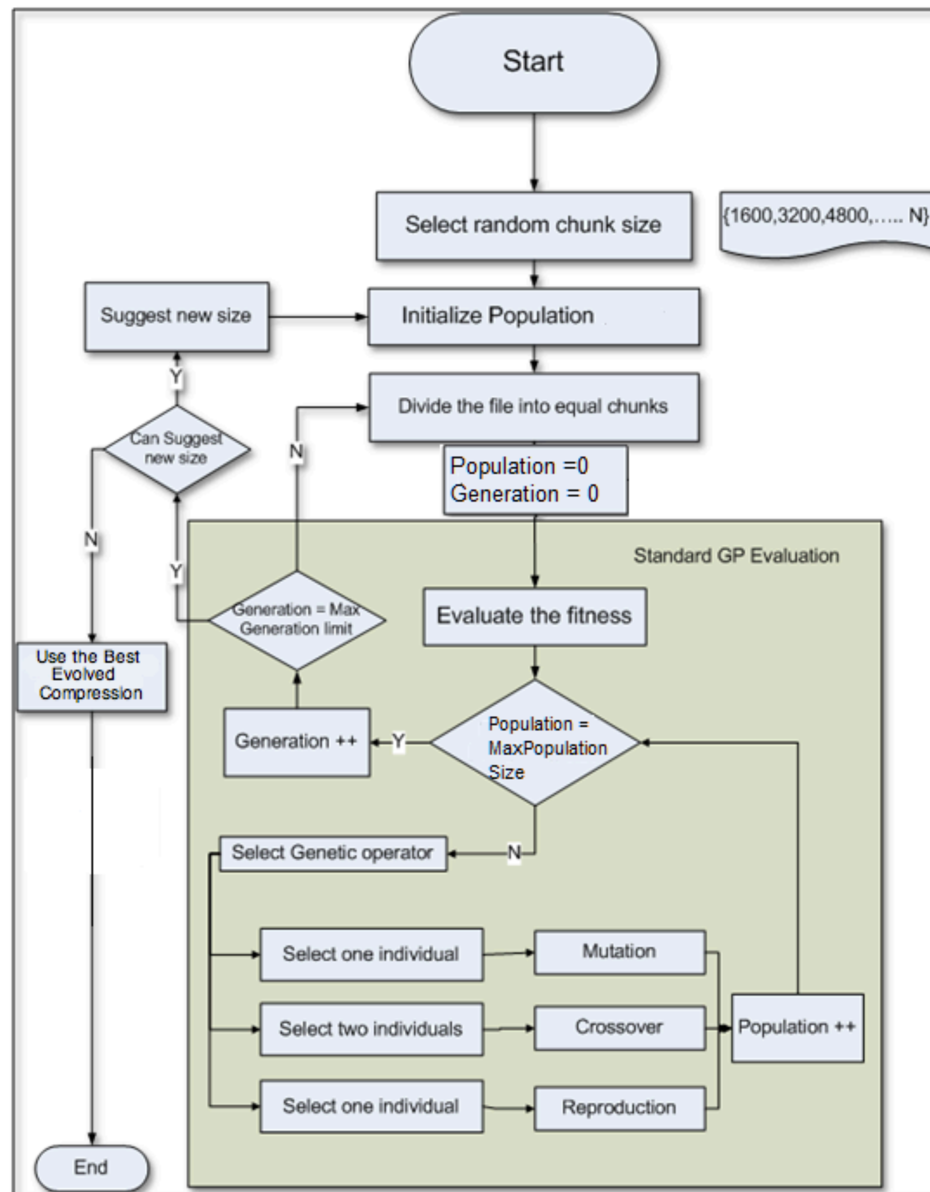

FIGURE 4.2: GP-zip flowchart.

4.1.3 Search Operators

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 selects two individuals with tournament selection. Then, a common crossover point was randomly chosen. Finally, all

the segments before the crossover point in the first parent were concatenated with the segments after the crossover point in the second parent in order to produce the offspring. Hence, this was a form of one-point crossover.

The mutation worked as follows. One parent is selected via tournament selection, a random mutation point is chosen and then the selected point are mutated into new randomly selected compression/transformations functions.

4.2 GP-zip*

In the previous version of the system we wanted to confirm the validity of the idea of combining existing lossless compression algorithms in such a way that ensure the best possible match between the algorithm being used and the type of data it is applied to. In this section, a further step toward improving the basic GP-zip is presented in GP-zip*.

GP-zip used a form of linear GP to find optimal ways to combine standard compression algorithms for maximally and losslessly compressing files and archives. As will be shown in section 4.3, GP-zip worked well with heterogonous data sets, providing improvements in compression ratios over some of the best known standard compression algorithms. However, GP-zip had two main limitations. The first limitation is the fixed segmentation of the data (i.e., dividing the data into segments of equal length), which limits the algorithm's ability to identify the true boundaries of homogeneous fragments within the data. For instance, consider the example illustrated in figure 4.3 where a typical archive file is composed of multiple inhomogeneous fragments of data being compressed. Here, GP-zip's approach decides to divide the data into segments of equal length then match each segment with the best possible functions. It is possible, and in fact most likely, that the sizes of the segments will not correspond to the edges of the different data fragments. Therefore, the assigned compression functions are applied to different types of data (possibly data types that they are not designed to work with). Overall, this decreases the algorithm's ability to correctly match different parts of the data with different

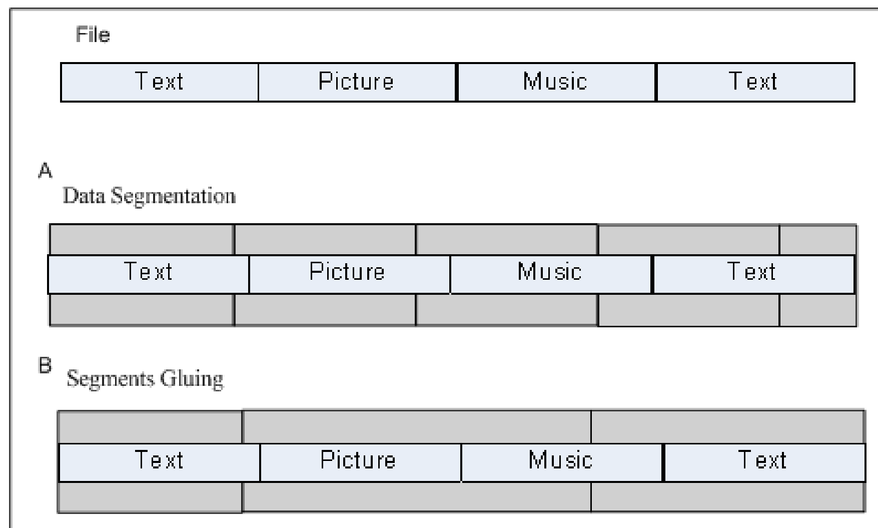

FIGURE 4.3: Typical archive file being compressed with GP-zip:
A: represents data segmentation, B: represents the gluing process of similar segments.

compression functions. The second limitation of GP-zip is the considerable computational load caused by the use of a staged search process, which GP-zip uses to identify the best segment length.

In this section we present a substantial improvement to GP-zip, called *GP-zip**, which uses a new representation where segments of different sizes can be evolved and intelligent operators are used to identify and target which elements of the representation to change in order 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.

An alternative to GP-zip's scheme of imposing the use of a fixed length for the segments is to allow the segments 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 segments, which completely removes the need for a staged search for an acceptable fixed length typical of 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 segments than in choosing how to finally 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. This motivated us to find a way to eliminate the need to imposing a fixed length of blocks.

One initial idea for achieving greater flexibility was to divide the given data into very small segments (e.g., 100 byte per block), compress each segment individually with the best compression model that fits into it, and then glue all the identical subsequent segments (gluing is the process of joining subsequence segments to form bigger blocks, with the intention of processing them as one unit). We eventually discarded this idea for two reasons. Firstly, it relies heavily on the gluing process, which, may 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, more effective) strategy: we ask GP to solve the problem for us. That is, each individual in the population represents how many segments to divide each file into and the size of those segments in addition to the particular algorithm used to compress each segments. 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.

4.2.1 Representation

We used the same primitive set for GP-zip* (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 known compression methods and worked really well in GP-zip, and b) using the same primitives we

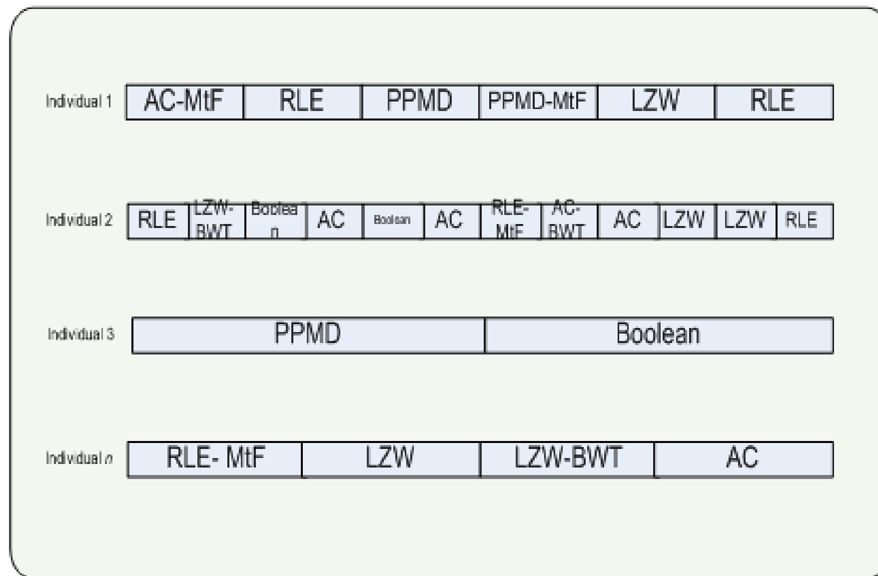

FIGURE 4.4: GP-zip* individuals representation.

can perform a fairer comparison between the two systems. GP-zip* starts by initialising the population randomly. Similarly, to the previous method, all initial individuals contain segments 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 4.4, the resulting population includes individuals with a variety of segment 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 segments, during these processes the size of one or more segments within an individual may change. This makes it possible to evolve any partition of the file into segments. This gives GP-zip* the freedom to explore many more possible solutions in the search space than GP-zip could. The hope is that some of these new possibilities will prove superior.

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

Similarly to the previous method, it is necessary for the decompression process to know which

segment was compressed by which compression/transformation function. A header for the compressed files provides this information, similar to GP-zip. 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 in equation 2.1), evolution always stays clear of solutions that involve too many small segments. In addition, similar to the previous method, GP-zip* presents several advantages as a result of dividing the data into smaller segments. 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.

4.2.2 Crossover

Both GP-zip and GP-zip*'s individuals have a linear representation which may give the impression that the proposed system is a Genetic Algorithm rather than from the Genetic Programming method. However, this is arguably not true for the following reason. GAs are known to have a fixed representation (binary strings) for individuals, while in GP-zip* the system receives 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 forms the GP-zip* compression system. Thus, each element of the representation acts more like an instruction in a GP computer program than as a parameter being optimised by a GA.

Since in GP-zip* individuals are divided into segments of different and heterogeneous lengths, we cannot use the same approach as in GP-zip. 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 segment 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 segment in one parent and swapping it with one or more corresponding segments 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 segment as a crossover point, GP-zip* selects the segment with the lowest compression ratio in the first parent, which arguably is the most promising hotspot.

Naturally, the boundaries of the segment chosen in the first parent will often not correspond to the boundaries of a segment or of a sequence of segments in the second parent. Therefore, before performing crossover, GP-zip* resizes the segment 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 segments from the second parent to replace the (extended) original segment in the first parent. Resizing is the process of extending the selected segment size in the first parent with the intention to fit it within the boundaries of the corresponding segment or the sequence of segments in the second parent. The crossover operator is illustrated in figure 4.5.

This crossover operator (in conjunction with the new representation) is a 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, GP-zip*'s mutation also uses the hotspot idea).

4.2.3 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 thus, search intensity) is maintained.

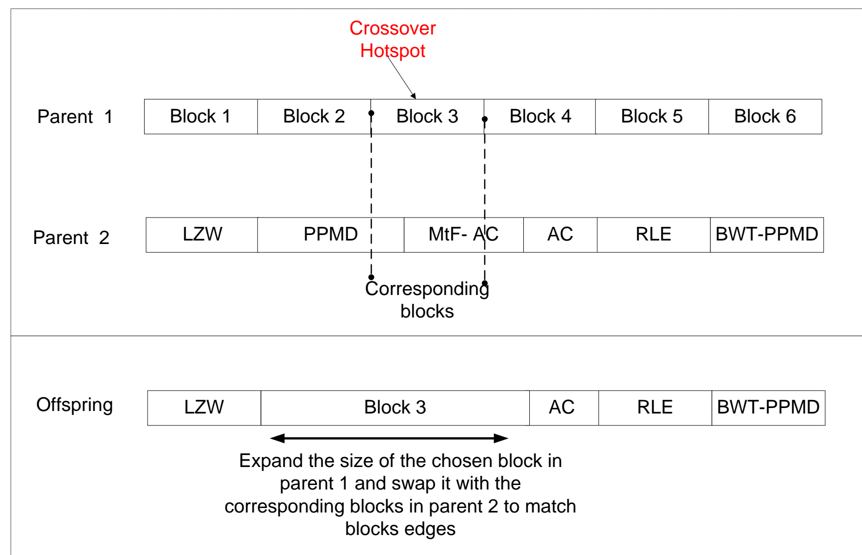

FIGURE 4.5: GP-zip* crossover operator.

GP-zip* mutates individuals differently. Once again we took advantage of the segment-wise nature of the individuals. GP-zip* chooses the segment with the worst compression ratio in the individual. Then it randomly selects a new segment size in addition to a new compression/-transformation function for the segment. 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 segment, 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 segment. In either case, changing the length of one segment will affect all the adjacent segments. The changes may include: extending one or two neighbouring segments, shrinking one or two neighbouring segments, or even entirely removing some segments. Figure 4.6 illustrates two mutation cases.

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

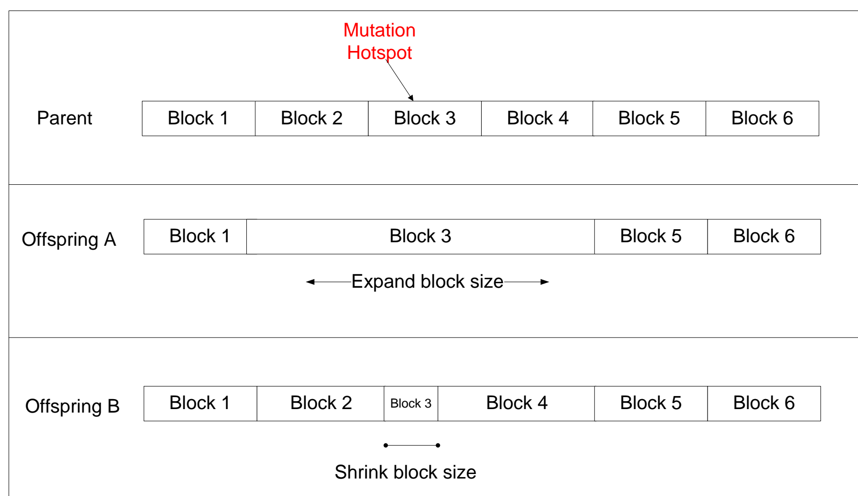

FIGURE 4.6: GP-zip* mutation operator.

4.3 Experimental Results

Experiments have been conducted in order to investigate the performance of the proposed techniques, GP-zip and GP-zip*, with different data types. The experiments covered both homogeneous and heterogeneous archives of data.

The performance is mainly measured by evaluating the compression ratio (see equation 2.1). The time needed to perform the compression process is also reported.

4.3.1 GP-zip Experimental Results

GP-zip's experiments covered three sets of data: *i) an archive of English text files*, *ii) an archive of executable files*, and *iii) a heterogeneous archive file* that include PDF, MP3, Excel sheet, and text files. The total sizes of the data sets are 4.70MB, 4.70MB and 1.43MB, respectively. Both text files and executable files are available in [86]. There is no terminating condition for GP-zip. Therefore, GP-zip runs until it reaches the maximum number of generations. The experiments that are presented here were done using the settings in table 4.1.

TABLE 4.1: GP-zip - Parameters settings.

Parameter	*Value*
Population Size	500
Generations	1000
One-point Crossover Probability	75%
Mutation Probability	20%
Reproduction Probability	5%
Tournament Size	2

After applying GP-zip to the set of English text files and the set of executable files, the system always converged to solutions where each file is treated as a single contiguous block. Experiments show that the compression ratios increase as the number of segments in these files decrease. This is not too surprising since both executable files and plain text, statistically, have a very regular structure. However, when applying GP-zip to our third data set (an archive containing files of different types), the system found that dividing the file into a certain number of segments provided the best performance. Naturally, this had to be expected as it is difficult to process the entire data set effectively using only one algorithm.

To evaluate the relative benefits of GP-zip in comparison to other widely used techniques, we compared the performance of GP-zip against the compression algorithms in the function set. Furthermore, Bzip2 and WinRar, which are amongst the most popular compression algorithms in regular use, were included in the comparison.

The results of the comparison are reported in table 4.2. In the first two data sets, the system decided to process the data as one large single block. Consequently, their compression ratio is based on the performance of one compression function in the function set (in fact, PPMD). Therefore, here GP-zip cannot outperform existing algorithms. It did choose, however, a very good algorithm to do the compression, thereby coming second in both categories.

In the heterogeneous data set, however, GP-zip beats all other algorithms. More precisely GP-zip provides an improvement of compression ratio of 1.28% over all others. This shows that there is potential in an evolutionary approach to data compression.

TABLE 4.2: GP-zip performance comparisons. (Bold numbers are the highest).

Compression \Files	Exe	Text	Archive1
bzip2	57.86%	77.88%	32.93%
WinRar- Best	**64.69%**	**81.43%**	34.04%
PPMD	61.84%	79.95%	33.32%
LZW	35.75%	56.47%	1.14%
RLE	-4.66%	-11.34%	-10.20%
AC	17.47%	37.77%	9.98%
GP- zip	61.84%	79.95%	**35.32%**

Although the proposed technique has achieved a higher compression ratio with heterogeneous files in comparison with the other techniques as mentioned earlier, it suffers from one major disadvantage: running GP-zip is very time consuming (of the order of a day per megabyte). The reason for this is that, each time GP-zip suggests a new segment length, it executes a new GP run which consumes considerably time by itself. Therefore, as discussed previously, most of the time is spent searching for the best possible length for the segments, rather than in choosing how to compress the data.

4.3.2 GP-zip* Experimental Results

A second set of 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 previous method, as well as other widely used compression techniques. Similar to its predecessor, GP-zip* has been compared against the compression algorithms in its function set. In addition, as outlined above, Bzip2 and WinRar were included in the comparison.

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 the previous method. Furthermore, new files have been included such as the Canterbury corpus [28], which is among the most popular benchmarks for data compression algorithms. Table 4.3 presents a list of the files that have been included

TABLE 4.3: Test files for GP-zip*.

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
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

TABLE 4.4: GP-zip* - Parameters settings.

Parameter	*Value*
Population Size	100
Generations	100
One-point Crossover Probability	75%
Mutation Probability	20%
Reproduction Probability	5%
Tournament Size	2

in the experiments. As one can see, the experiments covered both heterogeneous and homogeneous sets of data. The experiments presented here were performed using the parameter settings presented in table 4.4.

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 table 4.5.

As can be seen by looking at the compression ratios obtained with the two homogeneous sets of data (second columns in table 4.5), GP-zip* does very well outperforming six of its seven competitors for the Text Archive, with only WinRar doing marginally better. However, GP-zip* achieve higher compression ratio than all other algorithms in the Exe Archive. 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

TABLE 4.5: GP-zip* performance comparisons. (Bold numbers are the highest, N/T: not tested)

Compression\Files	Exe	Text	Archive1	Archive2	Archive3	Canterbury
Bzip2	57.86%	77.88%	32.90%	3.90%	64.49%	80.48%
WinRar- Best	64.68%	**81.42%**	34.04%	3.19%	65.99%	**85.15%**
PPMD	61.84%	79.95%	33.32%	3.90%	64.36%	81.19%
LZW	35.75%	56.47%	1.14%	-43.98%	43.62%	15.61%
RLE	-4.66%	-11.34%	-10.20%	-11.49%	9.16%	6.55%
AC	17.47%	37.77%	9.98%	0.70%	27.62%	41.41%
GP-zip	61.84%	79.95%	35.32%	N/T	N/T	N/T
GP-zip*	**65.78%**	80.67%	**37.32%**	**4.07%**	**69.62%**	81.65%

files better than GP-zip. 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 presumably to its increased ability to detect heterogeneous data segments (of different sizes) even within one set of homogeneous data. Then by compressing each segment 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*. Here GP-zip* outperforms all other algorithms. With *Archive1* (third column of table 4.5) 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, 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 having run GP-zip* 60 times (10 runs for each file). However, as shown in table 4.6, GP-zip* is very reliable with almost every run producing highly competitive results. The table reports also

TABLE 4.6: Summarisation of 15 GP-zip* runs.

	Exe (4.07MB)	Text (4.07MB)	Canterbury (2.66MB)	Archive1 (1.43MB)	Archive2 (2.39MB)	Archive3 (1.35MB)
Compression Average	59.01%	76.50%	68.50%	31.59%	-0.56%	64.84%
Standard deviation	13.22	3.62	6.59	2.78	5.01	3.83
Best Compression	65.78%	80.67%	81.65%	37.32%	4.07%	69.62%
Worst Compression	21.63%	71.87%	63.12%	28.02%	-7.19%	60.21%
Compression Time	4:30 hours	4 hours	3 hours	2:30 hours	3 hours	2:30 hours
Compression Time Hours/Megabyte	1.11	0.98	1.13	1.75	1.26	1.85
Average of Compression time		3.25 hours				
Average of Compression time/Mega byte		1.35 hours				

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 previous method, it is fair to say that the algorithm is still very slow (we timed the system on an AWS cloud computing using virtual cores with 2 EC2 Compute Units each) [2], although 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.

4.4 Summary

In the lossless data compression domain it is very difficult to outperform standard compression algorithms. These algorithms are difficult to derive and are the result of many person-years of effort (e.g., the LZW have been developed since 1977). So, even a small improvement in the compression ratio is considered important.

In GP-zip we wanted to understand the benefits and limitation of combining existing lossless compression algorithms in such a way to ensure the best possible match between the algorithm being used and the type of data it is applied to.

While other compression algorithms attempt to use different techniques on the file to be compressed, and then settle for the one that provides the best performance, the system we have

[2]One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

developed, GP-zip, goes further. It divides the data file into smaller segments and attempts to identify what combination of compression algorithms provides the best performance.

The GP-zip system was a good starting point for the exploration this new technique of combining compression algorithms. Despite the simplicity of GP-zip realisation of this idea, the results obtained by GP-zip outperforming other compression algorithms on heterogeneous files and never being too far from the best with other types of data.

GP-zip outperform the algorithms in its function set with a considerable margins in most of the cases. Note that none of these algorithms is reliable enough to be used as a universal compression system. However, with this new technique of combining compression algorithms, GP-zip managed to utilise them based on the given coding situation to provide a generic compression model that outperforms these algorithms individually. However, the proposed technique suffers from one major disadvantage: the process of GP-zip is computationally expensive.

A natural alternative to GP-zip's scheme of imposing the use of a fixed length for the segments, is to allow the block length to freely vary (across and within individuals) to better adapt to the data. Further improvement were proposed in GP-zip*, which explored this idea. In GP-zip*, each individual in the population encodes the number of segments into which to divide a file and the size of those segments, in addition to the particular algorithms to be used to compress each segment. To manipulate this representation we designed intelligent crossover and mutation operators that targeted hotspots in the parent individuals. The crossover operator worked by choosing the segment with the lowest compression ratio in the first parent and swapping it with one or more corresponding segments in the other. Mutation worked similarly.

The new representation and operators in GP-zip* resulted in better compression ratios. GP-zip* also outperformed many other compression algorithms (see section 4.3). However, GP-zip* was remaining still significantly slower than other compression algorithms. The time needed for compressing one megabyte ranged from 2:30 to 4 hours. This is because GP-zip* still needs to compress the given file many times with each individual in each generation.

In addition, in order to provide better compression, the division of data files into segments presents the additional advantage for both GP-zip and GP-zip* 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 both GP-zip and GP-zip* can send each decompressed segment into the operating system pipeline sequentially. Further, 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.

Processing times of this magnitude imply that GP-zip* has a relatively small niche in the compression world. Because decompression is fast but compression is very slow, it 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/DVDs or large scale software distributions. For other applications, however, a better solution was needed. In the case of GP-zip, we consider this version of the system as a proof of concept, whose purpose was to ensure the effectiveness of the basic idea (i.e., combining existing compression algorithms and utilise them where they expected to perform best). Thus, GP-zip is not meant to be designed for practical use, rather than to show the ability to achieve high compression ratios.

Chapter 5

The GP-zip2 approach: Compression after Evolution

As defined previously, an ideal compression system (see section 2.2.3) would be one that is able to quickly identify incompatible data fragments (both at the file level and within each file) in an archive, and then allocates the best possible compression model for each in such a way as to minimise the total size of the compressed version of the archive. Both GP-zip and GP-zip* were capable of finding excellent ways of fragmenting archives and of compressing the resulting blocks, thereby obtaining superior results in comparison to other compression methods. However, they suffered from one major disadvantage: the compression process is 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 (see section 3.2.2).

In this chapter we address the main limitations of GP-zip and GP-zip* and present GP-zip2. We aim to eliminate the disadvantage mentioned above from these new systems. The main motivation behind this enhancement is to produce an intelligent and *faster* compression technique that works best on heterogeneous data archives. The objective is still to be able to distinguish different fragments in the data and optimally compress them using the strengths of existing

compression models to achieve high compression ratios. However, we want this to happen very efficiently. The material presented in this chapter on GP-zip2 has been published in [87].

5.1 GP-zip2

5.1.1 The Basic Idea

In the GP-zip2 approach, the problem is to optimally identify file segments and classify these segments into different families based on their compressibility with a particular compression algorithm.

From the point of view of an operating system or standard high-level programming languages, the data to be compressed are normally treated as a sequence of elementary data units, typically bytes. What each unit represents depends on the file type. If the file is plain ASCII text, each unit will represent a character or punctuation. If a file is an executable program, each unit may either represent an instruction, a fragment of an instruction, or some numeric or textual data. Sometimes files contain recordings of signals (e.g., sound), in which case a unit (say a byte) will either represent a sample or part of a sample. In any case, the interpretation of the sequence of units contained in a data file entirely depends on what we know about that file and what our expectations regarding the content of that file are. In most cases, these are determined by the file name and extension (although further information may also be available).

Naturally, one can use such knowledge to decide how to compress a file, and this is what most off-the-shelf compression algorithms do. However, when presented with unknown data (e.g., an archive in a format unknown to the operating system) one cannot exploit this information. Also, as we have seen (in sections 4.1 and 4.2), 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 regularities within files. However, typically no source of information is available on what such regularities are and where they are within each file. In a sense, we are therefore left with a stream of data units but without an interpretation for them.

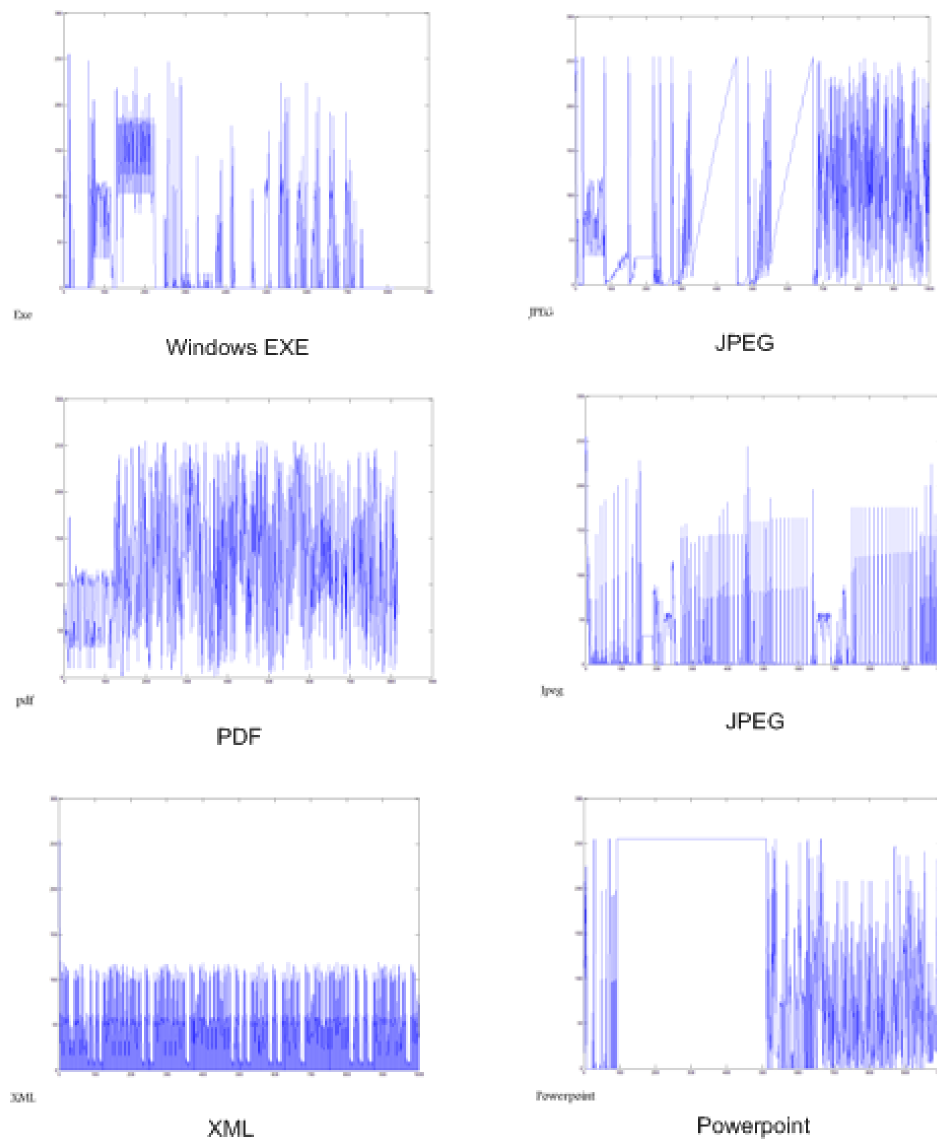

FIGURE 5.1: Plotting the signals associated with different file types.
Different data types often correspond to signals with very different characteristics

What we do in GP-zip2 is that we try to spot regularities within the data. We then try to associate the best compression technique to such regularities. In particular, we treat the stream of data as a signal digitised using an 8-bit quantisation without sign. Thus, each byte in a data file is treated as an integer between 0 and 255. Preliminary tests involving plotting such signals for different file types revealed that different data types often correspond to signals with very different characteristics. This is illustrated in figure 5.1, where the "signals" corresponding to different file types have been plotted. The figure shows that different file types have different characteristics. It also reveals, however, that there is ample variety within each file type. Note

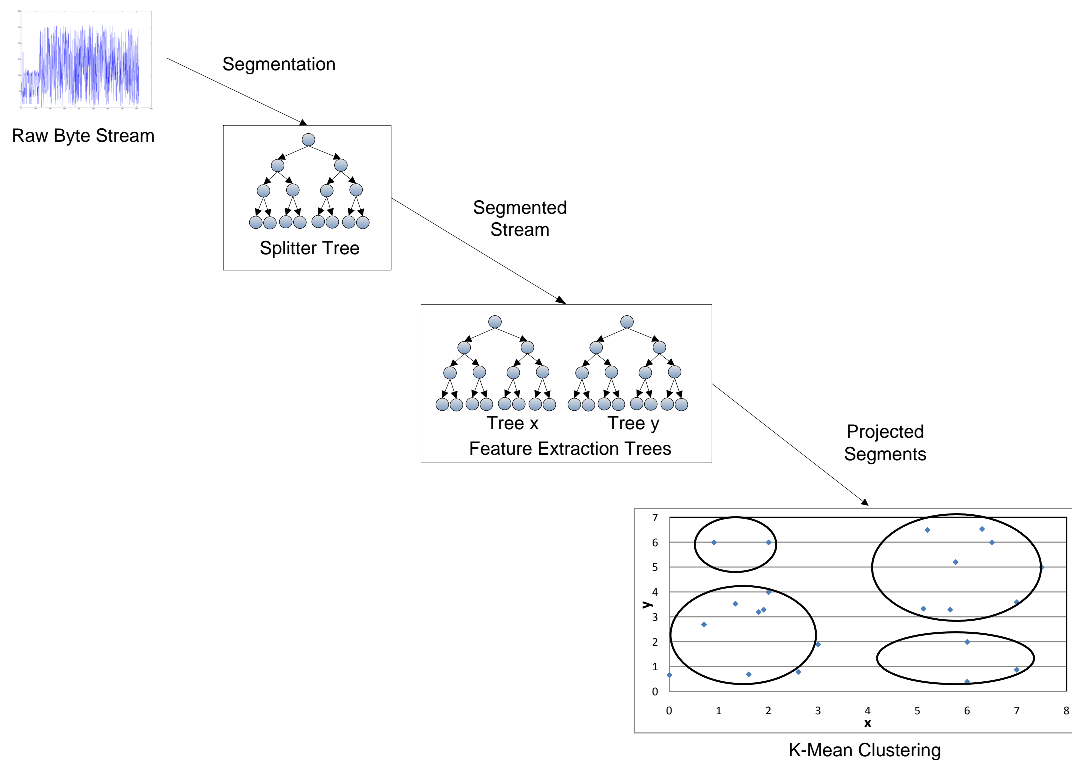

FIGURE 5.2: Outline of GP-zip2 training process.

that, while in principle one could try a variety of interpretations in parallel, in this work we chose to simply focus on one, due to computational cost. As we will show, this has worked well.

5.1.2 System's Operation

The system is used in two main stages: *i) Training*, where the system uses evolution to learn to match different signals characteristics with different compression algorithms, and *ii) Testing*, where the system applies what it has learnt to compress unseen data.

The training phase, broadly outlined in figure 5.2. The system processes the raw byte-series assoicated with the data and performs two major functions: *i) Segmentation* of the signals based on their statistical features and *ii) Classification* of the segments thus identified based on their compressibility with particular compression algorithms.

GP-zip2 uses the following five compression algorithms: AC, LZW, PPMD, RLE, and Bzip2. In addition, GP-zip2 has the option "No-Compression", to give the system the freedom to not

TABLE 5.1: GP-zip2 primitive set

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)

compress some parts of the data. These five compression algorithms were selected because they belong to different compression categories. In addition, these algorithms worked very well with heterogeneous data in GP-zip and GP-zip*. None of them is reliable enough to be used as a universal compression system. However, the appropriate use of these techniques in a combination, based on the given coding situation, may result (as will be shown in the experiments section) in a generic compression model that outperforms these algorithms individually. Further, the transformation algorithms (used in GP-zip and GP-zip*) were removed to reduce the search space size and the evolution time.

GP has been supplied with a language that allows the extraction of statistical features out of the byte series associated to a file. Table 5.1 illustrates the primitive set of GP-zip2. Note that, unlike GP-zip and GP-zip* which used a linear representation, GP-zip2 uses a tree-based representation. Furthermore, GP-zip2 uses a type system to ensure primitives are used correctly. Finally, note that in GP-zip and GP-zip* the decision as to which compression and/or transformation algorithms to use for the data blocks was based on trail-and-error learning. Instead, in GP-zip2, the decision as to which compression algorithm to use for a block is made by a classifier based on K-means.

The system starts by randomly initialising a population of individuals using the ramped half-and-half method [10]. Each individual has a multi-tree representation including one splitter tree, and two feature-extraction trees (a description for the job of these trees is presented in the next subsections). The multi-tree representation used in GP-zip2 is relatively common in GP. A multi-tree representation was probably first proposed by Koza [14] to implement automatically

defined functions. Later Haynes *et al.* [88] used it to adapt GP to evolve cooperative agents for problem solving. Luke and Spector [89] also used it for a similar purpose. Markus and Banzhaf [90] extended the idea to linear GP for the purpose of evolving teams for different classification and regression problems. More recently, multi-tree representations have also been used in data classification and clustering [91] and [92].

We used a similar representation to the one proposed by Haynes in [88]. However, unlike some of the systems mentioned above, in GP-zip2 the splitter tree and the two feature extraction trees use the same primitive set (shown in table 5.1). The contents and size of the *List* terminal changes depending on the tree: 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 tree.

5.1.3 File Segmentation

The main job of splitter trees is to split a given 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. In our experiments we used $L = 100$ bytes and $S = 50$ 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 the program is a single 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 outputs of $f_{splitter}$ in two consecutive windows is more than a predefined threshold θ. This threshold has been set arbitrarily to 10. In preliminary experiments we found that small changes in θ did not affect the performance of splitter trees. This is because evolution is free to change the magnitude of the outputs produced by splitter trees to adapt to the threshold. The operation of the splitter tree is illustrated in figure 5.3.

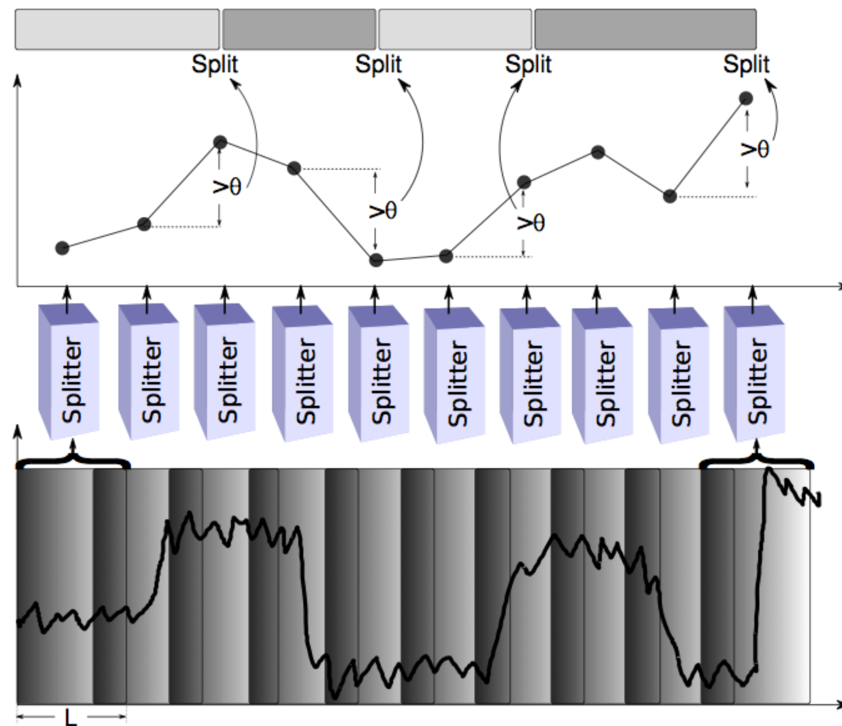

FIGURE 5.3: File segmentation in GP-zip2. The splitter tree is repeatedly applied to the data in a sliding window (bottom). The output in consecutive windows is compared: if it is higher than a threshold θ the data file is split at the window's position (top).

GP-zip2 optimally compresses the divided segments individually (by trying every available algorithm on each segment) and measures the total compression ratio. This is used as a measure of performance for the splitter tree. For example, if the given data was a Microsoft Word file that contained text and graphical charts, a good splitter tree would notice the change in the byte-series values from the text to the pictures and *vice versa*. Moreover, an ideal splitter tree might even detect different fragments within the same data type (e.g., a page full of blank lines within the text or white area in a picture). How the performance of the splitter tree contributes to the fitness of an individual will be discussed in section 5.1.7.

5.1.4 Classification of Segments

The job of each feature-extraction tree in GP-zip2 is to extract statistical information from the segments identified by the splitter tree using the primitives in table 5.1 and to condense it into a single number. This can be considered as a composite, higher-level feature. Using the numbers

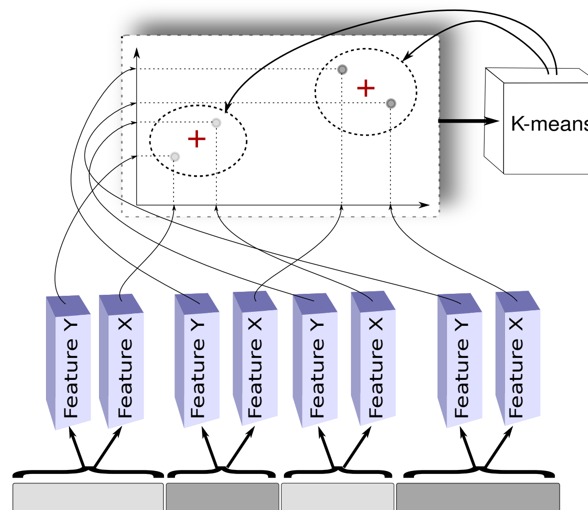

FIGURE 5.4: Illustration of the operation of the feature extraction trees.

produced by the two trees as coordinates, we can then map each segment into a point in a two-dimensional Euclidian space as illustrated in figure 5.4. In principle, segments that share similar statistical features will form dense groups.

We use an unsupervised pattern classification approach on the outputs produced by the two feature-extraction trees to discover regularities in the training files. In particular, we use K-means clustering [93] to organise blocks (as represented by their two composite features) into groups. In this way, objects within a cluster are similar to each other but dissimilar from objects in other clusters. The advantage is that the experimenter does not need to label the training set. Further, the approach does not impose constrains on the shape of the clusters. Once the training set is clustered, if we label each cluster with the compression algorithm (that provides the best compression for the segments in the cluster), we can then use the clusters found by K-means to perform classification of unseen data.

For this to work well, the number of clusters obtained from the projected segments should match the number of compression algorithms used to optimally compress those segments. In other words, 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 5 plus the No-Compression option) which were optimal for at least one of the segments.

Naturally, while we can tell K-means to group items in exactly n 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 for deciding 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.

So, how do we convince K-means to group things differently? Simple: we do not 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.

The advantage of this approach is that it greatly simplifies classification. This is because evolution pushes feature-extraction trees to represent the data in such a way as to optimise the performance of the classification algorithm. Here we used K-means for its simplicity of implementation and its execution speed, but other techniques might work equally well.

5.1.5 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 into account.

There are several options for applying genetic operators to a multi-tree representation. For example, we could apply an operator (that has been selected based on a predefined probability of application) to all trees within an individual. Alternatively, we could iterate over the trees in an individual and, for each, select a potentially different operator. Another possibility would be

to constrain crossover to happen only between trees at the same position in the two parents or we could let evolution freely crossover different trees within the representation.

It is unclear what technique is best. In [92] it was argued that crossing over trees at different positions might result in swapping useless genetic material resulting in weaker offspring. On the contrary, [91] suggested that restricting the crossover positions is misleading for evolution as the clusters are indistinguishable during the evolution. It is likely that what is best appears to depend on the semantics of the representation.

In preliminary experiments we tried these approaches and found that a good way to guide evolution in GP-zip2 is as follows. Let T_c^i be the cth 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 the 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 $feature_extractor_x$ tree or the $feature_extractor_y$ tree of the other. The $feature_extractor_y$ trees are treated symmetrically.

5.1.6 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.

Similar to GP-zip and GP-zip*, 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 header marks the end of the header. Note that the size of the header depends on the number of identified segments. Generally, the header's size is insignificant in comparison with the size of the original (uncompressed) file.

5.1.7 Fitness Evaluation

The calculation of the fitness is divided into two parts. Each part evaluates the quality of one part of the representation and contributes with equal weight to the total fitness. Firstly, the fitness contribution of the splitter tree is measured by calculating the maximum achievable compression ratio for the segments. This is computed by compressing each of the segments identified by the splitter with all available compression algorithms and then labelling each segment with the best compression algorithm for it. The compression ratio obtained by compressing each block with the best algorithm is then used to evaluate the fitness contribution of the splitter tree. The labelling information is retained since this is also used in evaluating the quality of the feature-extraction trees as we will see shortly.

More formally, the quality of the splitter tree can be expressed as follows. Let each compression algorithm be the function C_x where x represents a particular algorithm from the compression pool and let each segment be denoted by S_i. Let us denote with $|C_x(S_i)|$ the length of the ith segment when compressed with C_x. Then, the fitness of the splitter tree can be expressed as:

$$f_{splitter} = \frac{1}{2} \times [100 - (\frac{\sum_{i=1}^{n} \min_x |C_x(S_i)|}{\textit{File Size}} \times 100)] \quad (5.1)$$

Note that this approach is computationally expensive in that it requires the compression of each block in a file numerous times (more precisely 5 times with our set of compression algorithms) in order to determine the optimum compression algorithm. However, this needs to be done only during evolution. During normal operation, GP-zip2 simply applies the splitter tree to the unseen data. When the splitter produces a new block it applies the two feature-extraction trees to it, and then classifies the block based on the classes found by K-means during training. Thereafter, the system applies the compression algorithm corresponding to the chosen class to it. All of this can be done with a small computational overhead with respect to the case where one uses an ordinary compression algorithm.

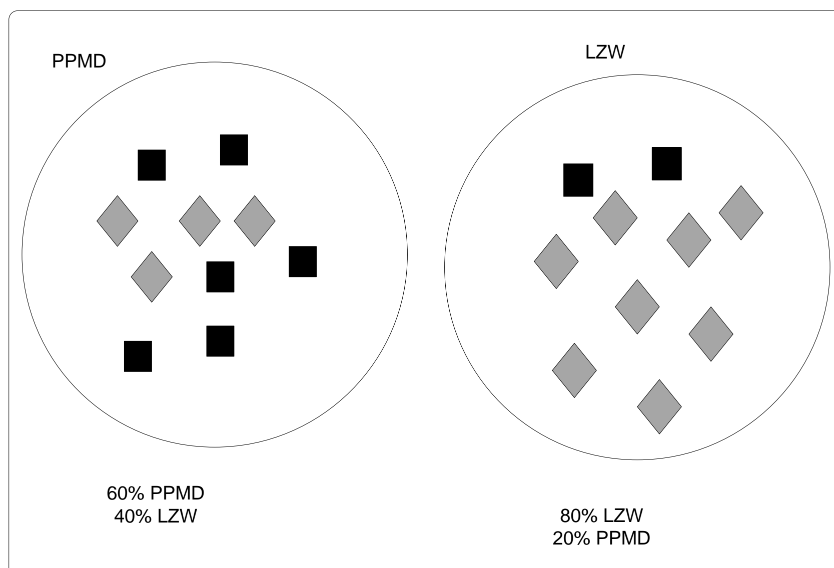

FIGURE 5.5: Homogeneity of clusters. (The cluster on the left has 60% homogeneity while the cluster on the right has 80% homogeneity)

The second part of an individual's fitness is related to the classification accuracy provided by the feature-extraction trees. After performing the clustering using K-means we evaluate the quality of the clusters by measuring *cluster homogeneity* and *cluster separation*.

The homogeneity of the clusters is calculated as follows. As exemplified in figure 5.5, we inspect the members of each cluster, each member of a cluster representing a segment of the data. Since we already know from the previous step the optimal compression algorithm for each segment, we label the clusters according to the dominant algorithm.[1] The homogeneity of clusters 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 compression algorithms 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 penalise extra members in the clusters.

[1]The system prevents the labelling of different clusters with the same compression algorithm even in the cases where proportions in two or more clusters are identical.

More formally, the clusters homogeneity can be expressed as follows. Let H be a function that calculates the homogeneity of a cluster and CL_i be the ith cluster. Furthermore, let K be the total number of clusters and λ the penalty term. Then,

$$f_{\text{Homogeneity}} = \frac{\sum_{i=1}^{K} H(CL_i) - \lambda}{K} \tag{5.2}$$

The penalty term λ is calculated as follows:

$$\lambda = \sum_{i=1}^{K} \frac{|Seg(C_i)|}{|All_Segments|} \times [|mem(CL_i)| - |mem_true_label(CL_i)|] \tag{5.3}$$

$|mem(CL_i)|$ 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). Finally, $|All_Segments|$ is the total number of segments in all clusters.

The homogeneity of the clusters is not the only measure of the quality of the classification performed by K-means. Homogenous clusters with objects far apart within a cluster will extend the cluster's 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. Therefore, we also measure and reward the separation of the clusters.

The Davis Bouldin Index (DBI) [94] is used to measure clusters' quality. 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 (see equation (5.2)), we push evolution to separate clusters (i.e., minimise the DBI). In other

words, we treat the DBI as a penalty value. Here, we used a modified Davis Bouldin Index (DBI) similar to the definition used in [95]. The formulation is as follows:

Let C_i be the centroid of the ith cluster and d_i^n be the nth data member of the ith cluster. In addition, let the Euclidean distance between d_i^n and C_i be expressed by the function dis(d_i^n , C_i). Furthermore, again let k be the total number of clusters. Finally, let the standard deviation be denoted as $std()$. Then,

$$DBI = \frac{\sum_{i=0}^{k} std[dis(d_i^0, C_i), ..., dis(d_i^n, C_i)]}{\sum_{i=0}^{k} \sum_{j=i+1}^{k} dis(C_i, C_j)}. \quad (5.4)$$

The fitness of the feature extraction trees is defined as follows:

$$f_{\text{Feature_Extraction}} = \frac{f_{\text{Homogeneity}} - DBI}{2}. \quad (5.5)$$

Finally, the total fitness of a program can then be calculated as:

$$f = f_{\text{Feature_Extraction}} + f_{\text{Splitter}}. \quad (5.6)$$

A significant advantage of our approach is that it does not impose any constraints on the shape of the clusters. The GP system can then ensure the clusters formed by K-means are suitable for the classification of unseen data (in the sense indicated above). It should be noted that the performance of the feature-extraction trees depends on the output of the splitter tree. In certain conditions, the splitter tree may force the feature extraction trees 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 (compressed with two different algorithms), 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 the DBI index as minimum, resulting in a very high $f_{Feature_Extraction}$. However, each cluster has only one data member and, therefore, 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. As a result, the evolution process will discriminate against them in the following generations.

5.1.8 Training and Testing

GP-zip2 builds its knowledge about the features of the data and their relationships with the performance of different compression algorithms during a training phase (a GP run). Naturally, the ability of GP to find good solutions which generalise well on unseen data depends crucially on the data used in the training set.

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 in order to ensure the generality of the evolved compression algorithms. In deciding which data types 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 durations under control.

Table 5.2 presents the data types we used within our training archive. The training archive contains 11 different data types. The total size of the archive is 332KB. As we will see, this ensures the generalisation 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 that

TABLE 5.2: Training and testing files for GP-zip2

Phase	File types	Total size of training 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

project file blocks in a two-dimensional Euclidian space. Each cluster prototype is defined as the centroid of the cluster's members.

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 involves compressing unseen files using the trees and clusters identified during the training phase. It should be noted that the 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. These are then fed into the feature-extraction trees and projected into a two-dimensional space. A block is assigned to the cluster whose centroid is closest (in terms of Euclidian distance) to the block's two-dimensional feature vector. The block is then compressed with the compression algorithm associated to the chosen cluster.

Files can be compressed in two different ways: *gluing mode*, where consecutive segments that have been assigned to the same compression algorithm are compressed as one (bigger) block, and *non-gluing* mode where 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.[2]

Table 5.2 presents a list of file types that have been used to test the generalisation of evolved compression algorithms. The test set contains 22 different data types within 12 archives, for

[2]In our experiments, the difference in compression ratios between gluing and non-gluing was always $\leq 1\%$. This is because the system rarely uses the same compression method for many consecutive blocks. This is the results of the splitter tree doing a good job at finding the boundaries between different data types.

a total of 133.5MB of 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.

5.2 Experiments

This section will present the experiment results for GP-zip2. The performance is mainly measured by evaluating the compression ratio (see equation 2.1). The time needed to perform the learning and the actual compression process is also reported.

5.2.1 Experimental Results

In the field of compression, researchers primarily focus on measuring compression ratios. However, the time taken to perform compression and decompression is also often considered important. Unfortunately, due to several factors, including, for example, the size of the compressed file, disk speeds and the power of the CPU used, it is difficult to obtain precise time comparisons for different compressions models.

Table 5.3 presents a list of the 12 test archives that have been used in the experiments. Archives contain both homogeneous and heterogeneous sets of data. The files within the archives are distributed in such a way as to ensure that each archive contains a unique combination of 22 file distinct types (see table 5.2). 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 which requires no machine learning techniques. In this method we apply all compression algorithms available to GP-zip2 on each individual file

TABLE 5.3: Test files for 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 (rsum), 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

* The archive contains data types to which the algorithm has not been exposed during training.

in the test archives, we keep the best result and we place the resulting compressed files into an archive.

The experiments presented here were performed using the parameter settings shown in table 5.4. These parameters were set by performing a variety of preliminary experiments and selecting the values that gave us good results while keeping the processing time under control. Since it is practically impossible to determine what the best fitness level is that could be reached with our training set, there is no special terminating condition for GP-zip2. Simply, the system runs until the maximum number of generations is reached.

TABLE 5.4: 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

5.2.1.1 GP-zip2's Behaviour Across Runs

GP-zip2's performance has been measured through 15 independent runs, each of which evolves a compression system. Plots of the overall fitness of the best-of-generation program averaged across runs are shown in figure 5.6 together with their two components: the one associated with the splitter tree and that associated with the feature-extraction trees. The fitness of the splitter tree starts relatively high because it represents the compression ratio (for the *training set*) obtained by trying all compression algorithms on each block and picking the best. So, even with random blocks (such as those produced 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, part of 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 have a knock on effect on the former.

To evaluate the performance of GP-zip2, we used each of the 15 best-of-run programs obtained in our runs to compress the 12 different archives in the *test set*.

Table 5.5 summarises the results. The second column reports the average compression achieved for each archive by the 15 best-of-run programs evolved in our runs. The third column shows

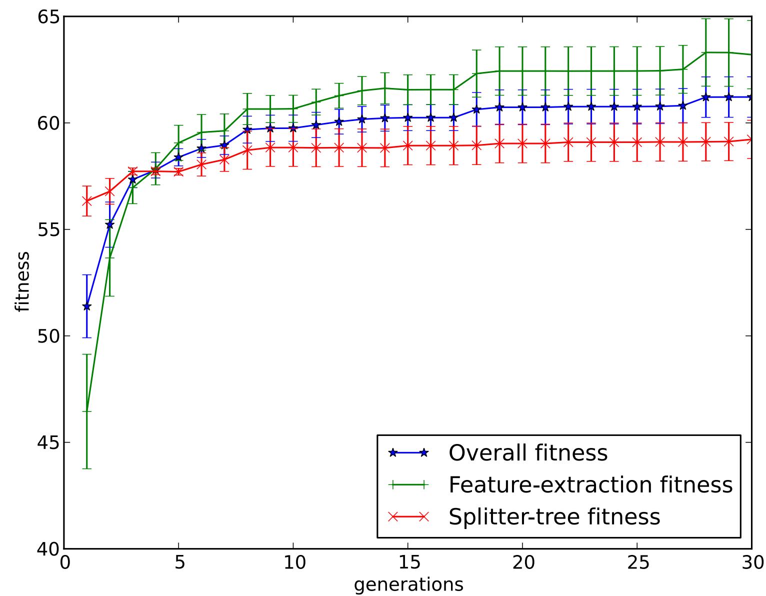

FIGURE 5.6: Evolution of best-of-generation fitness during our experiments. The two components of the fitness (the fitness associate with the splitter tree and the fitness associated with the feature-extraction trees are also plotted). Bars represent the standard errors of the means.

the corresponding standard deviations. The fourth column reports the compression ratios obtained by the best overall best-of-run algorithm on each archive. The fifth column illustrates the performance of the worse overall best-of-run algorithm on each archive. Finally, the last two columns show the best and the 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. Firstly, 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, GP-zip2 is very reliable at producing effective compression algorithms.

The fourth column of table 5.5 reports the performance of the best-of-run program evolved in our runs that showed the best average performance over the 12 archives in the test set. This is probably the compression algorithm the user would choose after our set of 15 run; all other results would be discarded. However, there is also an alternative, albeit more expensive, strategy

TABLE 5.5: GP-zip2 summary of results in 15 independent GP runs

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

which would avoid wasting all the other best-of-run algorithms: when given a file to compress, one could 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 the table, in many cases improved results could be obtained using this approach. 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.

5.2.1.2 GP-zip2 vs. other Compression Algorithms

Table 5.6 shows the compression ratios obtained on each of our 12 test archives by standard compression algorithms, the baseline method, 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, on average, both GP-zip2 compression systems outperform all other algorithms. In fact, the sequential GP-zip2 strategy is best on 8 out of 12 archives, second best on further three archives and fourth best in the remaining archive. The best-overall GP-zip2 strategy is

best in 3 archives and second best in further 4. Only WinRar-Best can get anywhere near our evolved algorithms, providing the best results on 4 archives, second best on two and third best on further four. However, if one compares average compression ratios, it is clear that even WinRar-Best is behind the two GP-zip2 strategies (note that WinRar is not in GP-zip2). The one thing to remember is that WinRar is proprietary. Thus, it is hard to improve on it. If GP can match it (with an open-source result) or improve on it, this should already be considered human competitive and a great result.

The sequential GP-zip2 strategy was not overall best on the *Text, Archive4, Archive5* and *Canterbury* data sets. However, we should note that the *Text* archive is composed of homogeneous sets of files, for which is difficult for the splitter tree to detect statistical differences and split the data correctly. Thus, it is best compressed with a single model. We should also 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. Also, Archive5 included WMV (6:25 minutes) video, which GP-zip2 had not been exposed to in its training set. GP-zip2 was also outperformed on the Canterbury corpus. Because this dataset is often used as a reference for comparison of compression algorithms parameters, highly optimised compression software is often tuned to maximise compression on such a dataset, 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, overall we feel that GP-zip2 did rather well to keep as close as it did to its competitors in the few cases where it did not beat them.

GP-zip2 outperformed the algorithms in its function set with a considerable margins in most of the cases. These algorithms are not reliable enough to be used as a universal compression system. However, GP-zip2 managed to successfully utilise them based on the given coding situation and provided a generic compression model that outperforms these algorithms individually.

Table 5.7 shows that the best-of-run individuals produced by GP-zip2 took only *between 5 and 10 minutes to perform the compression on 133.5MB* on an AWS cloud computing using virtual

TABLE 5.6: 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 Sequintial
Archive1	32.90	34.04	33.32	1.14	-10.20	9.98	33.90	35.36	**35.98**
Archive2	3.90	3.19	3.90	-43.98	-11.49	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	-372.89	58.25	91.00	91.46	91.65
Archive5	7.98	**11.17**	8.64	-36.17	-5401.27	2.73	8.75	9.73	9.90
Archive6	50.61	56.33	49.07	-3.66	-5002.04	11.89	51.83	59.89	**59.90**
Archive7	68.64	64.21	70.76	33.62	-4377.23	33.98	71.19	**72.76**	**72.76**
Archive8	15.68	14.41	18.74	-47.74	-7003.42	5.40	18.73	19.65	**19.95**
Archive9	2.94	3.31	2.28	-41.54	-6333.82	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	-2376.05	20.62	47.73	49.05	49.46
StdDev	32.20	33.24	32.43	37.67	2937.4	18.90	32.5	32.60	32.74

Numbers in **bold** and underlined are the best and the second best compression ratios achieved in each row respectively.

cores with 2 EC2 Compute Units each [3] , i.e., on average GP-zip2 evolved programs can compress 16.97MB per minute. The table reports also the run-by-run training times for GP-zip2. On average runs lasted just under 7 hours.

TABLE 5.7: GP-zip2 training and compression times.

Run	Evolution time **(Hrs)**	Compression Time **(Min)**
1	6	5
2	7	5
3	6	6
4	5	8
5	7	6
6	7	8
7	6	10
8	7	8
9	7	8
10	6	8
11	7	10
12	6	10
13	7	10
14	6	10
15	7	6
Average	**6.4**	**7.8**

[3] One EC2 Compute Unit (ECU) provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor.

TABLE 5.8: Processing times required by standard compression algorithms when compressing our 133.5MB test set.

Algorithm	Time in seconds	Megabyte/Min
WinZip-bzip2	103	77.7
WinRar-Best	116	69.1
PPMD	320	25.0
LZW	54	149.4
RLE	96	83.2
AC	154	52.0

For comparison, in table 5.8, 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, including their closest competitor, WinRar-Best, which can compress 69.1MB per minute. However, we should note that most of these algorithms have highly optimised implementations, while the optimisation of the implementation was a secondary objective in our research agenda for GP-zip2. 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 almost on par with its commercial rivals while delivering superior compression.

5.2.1.3 GP-zip2 Generalisation

Different file types present similarities (e.g., PDF and PS). Thus, even if GP-zip2 uses a relatively small training set, generalisation is possible. Let us have a closer look at this.

To further verify the generality of GP-zip2's evolved solutions, we used the best evolved program to compress 8 more archives, which are completely independent from the training set (the sequential strategy performance is also reported). The files within the archives are distributed in such a way as to ensure each archive contains a unique combination of file types that did not exist in the training set. New file types included: *doc, xlsx, ram* and *dat*. Also, we included two of the most popular benchmarks in the compression field: *Calgary* and *Worms2Demo*. Finally,

TABLE 5.9: GP-zip2 - Archives containing file types not in the training set.

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

we included *ChessMAC* which is a chess game for the Macintosh platform. Table 5.9 describes the 8 archives.

Table 5.10 reports the performance of the best evolved program and the sequential strategy. GP-zip2 is best in 5 out of 8 archives and the second best on further one case. This is a remarkable result given that the algorithm had to deal with new file types. The reason that the algorithm did not outperform its competitors on the remaining 2 cases (Calgary and Worms2Demo) is because they have a very different nature from the training set. The Calgary is an old benchmark that contains file types which are not in common use anymore, while the Worms2Demo is a game. Hence, it contained some system files that are completely different from the training set. Therefore these further tests confirm that GP-zip2's evolved solutions generalise very well.

TABLE 5.10: GP-zip2 performance generalisation

File	Winzip - Bzip2	WinRar - Best	PPMD	LZW	RLE	AC	Baseline Method	GP-zip2 Best	GP-zip2 Sequin-tial
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	**47.9**
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	-13.27	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	-11.42	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	-1.80	10.78	32.78	32.95	33.95
StdDev	28.27	27.98	29.38	36.55	13.32	13.92	30.19	27.77	28.36

Numbers in **bold** and underlined are the best and the second best compression ratios achieved in each row.

5.2.1.4 Compression Algorithms Allocation

It is interesting to look at the behaviour of the final solutions evolved by GP-zip2. One way to do this is to look at how they split archives of files and what algorithms they allocate to the data segments. Figure 5.7 illustrates the typical behaviour of the best-of-run algorithms on a small test archive –Archive10– containing a PDF, a Word document and an Excel spreadsheet see table 5.9.

As shown in figure 5.7(a) there is significant consistency in the behaviour of solutions. For example, most compressors only use two out of the five available compression algorithms on this archive. Also, most GP-zip2 compressors treat the PDF file and to a lesser extent also the Excel file as a single block (see also figure 5.7(b)) allocating either AC or PPMD to them. Furthermore, virtually all compressors attempt to fragment the DOC file into multiple block (three on average), which is unsurprising given the high variability in its byte-values as illustrated in figure 5.7(c).

5.3 Summary

The GP-zip and GP-zip* systems we presented in the previous chapter influenced the development of GP-zip2. For example, we adopted the primitives and the block splitting strategy developed earlier. However, in order to overcome the severe limitations in terms of computational load of prior systems during the compression phase, we had to completely re-design the decision making and learning strategies.

Our guiding idea was to divide the task into one that involves first splitting files into blocks (using a function especially evolved for that task), then 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.

GP-zip2's approach has produced a system that outperforms its predecessors as well as most other compression algorithms. It comes top of most compression algorithms tested on heterogeneous files and is never too far behind the best with other types of data. In addition to providing better compression, GP-zip2 share the advantages of its predecessors. The division of data files into blocks, for example, presents the additional advantage that, in the decompression process, one can decompress a section of the data without processing the entire file. Also, the decompression is much faster than the compression, and can be parallelised, as GP-zip2 can ask different CPU cores to decompress separate blocks.

In the field of lossless compression it is very difficult to significantly improve on the already excellent performance provided by current algorithms. Thus, any improvement in the compression ratio is considered important and makes the difference between one algorithm and another. Note that all of the algorithms against which we have compared GP-zip2 are human-designed and they are among the best compression algorithms ever developed. Furthermore, some of them have been covered or are still covered by patents. This suggests that *GP-zip2 evolved solutions are human-competitive* based on Koza's 8 criteria for human-competitiveness [10, 96].

Although GP-zip2 has achieved good results, its performance depends on the knowledge it acquires during evolution. Thus, users need to select the training set according to their needs. A significant advantage with the GP-zip2 model, 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.

Despite of success achieved by the proposed technique it suffers from one major disadvantage. GP-zip2 runs require several hours (approximately 7 hours) in the training phase and several minutes to perform the actual compression of our test files. This is still significantly slower than standard compression algorithms. In the next chapter will explore this particular aspect in GP-zip2 and propose an improvement to reduce the long training time.

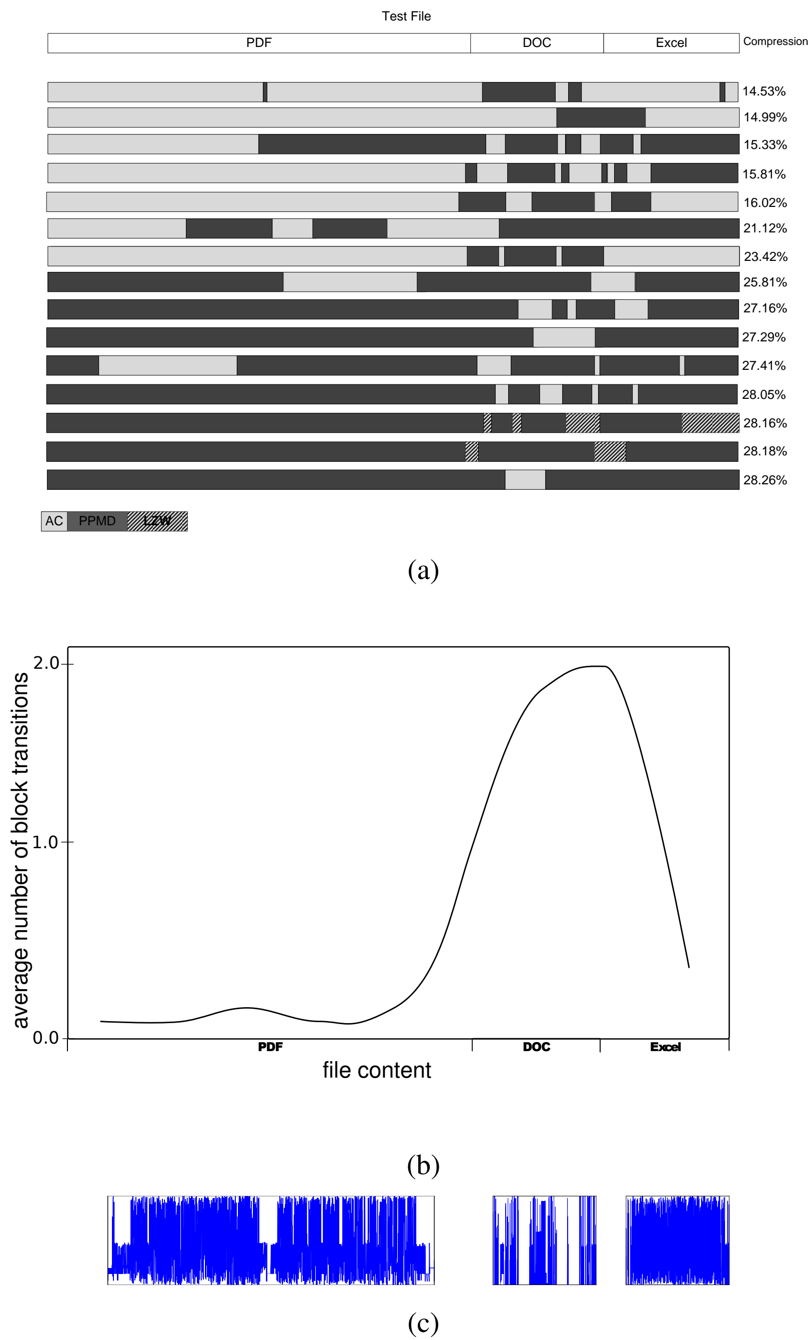

FIGURE 5.7: Allocation of compression algorithms to a test archive file by different best-of-run individuals (a), average number of block transitions along the length of the archive (b) and plot of the byte values in the archive (c). Algorithm allocations in (a) have been sorted by compression ratio to highlight similarities and differences in behaviour. The transitions plot in (b) has been smoothed for a clearer visualisation. Data in (c) were sub-sampled by factor of 10 for the same reason.

Chapter 6

Speeding up Evolution in GP-zip2 by Compression Prediction

In GP-zip2 we developed a training based universal compression system. Experimentation with GP-zip2 revealed superior performance in comparison with its predecessors, as well as most other compression algorithms (see section 5.2). However, as mentioned previously, GP-zip2 runs very slowly. This is largely due to the costly fitness evaluation adopted in GP-zip2, which requires the compression of data fragments using multiple compression algorithms. This long training time is an obstacle to the practicality of the system.

In this chapter, we present a substantial improvement of this system, called *GP-zip3*. GP-zip3 solves this problem by using an alternative fitness evaluation strategy. More specifically, GP-zip3 evolves and then uses decision trees to predict the performance of GP individuals without requiring them to be used to compress the training data. As we will see, predicting the performance of individuals, rather than actually evaluating them, reduces training time significantly and further achieved almost the same results .

In GP-zip2, the system optimally compresses the divided segments individually by trying every available algorithm on each segment and measures the total compression ratio (see section 5.1.1). This is the bottleneck of the training phase. One idea to accelerate this step would

be to use some form of prediction of the compression ratio that would be achieved by different compression models when applied to each data segment. If the prediction was accurate and faster than the actual compression with a model, presumably one could speed up the fitness evaluation of GP-zip2 by using such a prediction, without modifying the course of evolution too much. However, predicting compression ratios is a very challenging task. Therefore, before attempting to employ this strategy within an evolutionary compression system, we decided to test whether the problem could be solved at all to an acceptable degree.

This was done as an independent compression ratio prediction system. We tested a new approach based on GP to predict data compression ratios when applying different compression algorithms. The approach was successful and forms the basis for the new fitness evaluator used in GP-zip3. Details are described in the next section.

GP-zip3 follows exactly the same steps as its predecessor (i.e., GP-zip2), except for the fitness evaluation technique. Thus, GP-zip3 uses splitter trees to divide the data into fragments based on their statistical characteristics and then projects them onto a two-dimensional space using two feature-extraction trees. Later, K-means is applied to find different clusters within the projected segments. Experiments with this new fitness evaluation technique (results are provided in section 6.3) showed that GP-zip3 achieved almost the same results as GP-zip2. However, it required only 2 hours 30 minutes on average to perform the learning process, while GP-zip2 required 6.4 hours. The material presented in this chapter on GP-zip3 has been published in [97] and [98].

6.1 Compression Ratio Prediction

Our prediction system works in two stages: *i) Training* and *ii) Testing*. The prediction system processes each file via two different representations. Firstly, each file is represented as a series of byte values (i.e., integer in the range 0 - 255). Secondly, each file is represented as a byte frequency distribution (BFD).

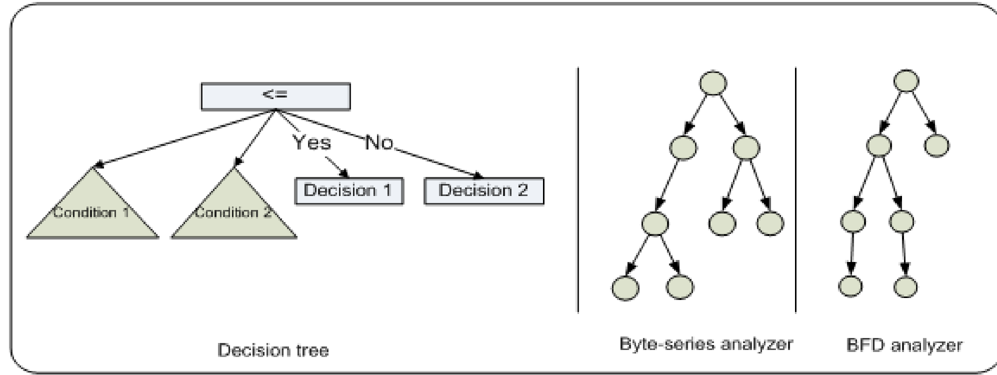

FIGURE 6.1: Individuals representation within the prediction system's population.

TABLE 6.1: Compression prediction system - Primitive set of the Byte-Series tree and Byte-Frequency Distribution tree.

Function	Arity	Input	Output
Median, Mean, Average deviation, Standard deviation, Variance, Skew, Kurtosis Entropy	1	List	Real Number
+, -, /, *, Sin, Cos, Sqrt, log	2 1	Real Number Real Number	Real Number Real Number
List	0	N/A	Vector of Integers (0-255)

TABLE 6.2: Compression prediction system - Primitive set of the decision tree.

Function	Arity	Input	Output
$IF <$,$IF \leq$,$IF \geq$, $IF >$ +, -, /, *,	4 2	Real Number Real Number	Real Number Real Number
Byte-Series tree Byte-Frequency Distribution tree	0 0	N/A N/A	Real Number Real Number

Thus, GP evolves programs with multiple component trees (see figure 6.1). The next subsections will describe the role of each component in solving the problem (i.e., the prediction of files' compression ratios without the need to implement a particular compression algorithm).

GP has been supplied with a language that allows the extraction of statistical features out of the two data representations and then combines them into a single decision tree. Tables 6.1 and 6.2 illustrate the primitive set of the system.

The system starts by randomly initializing a population of individuals using the ramped half and half method. The three standard genetic operators (crossover, mutation, and reproduction) have

been used to guide evolution through the search space.

We let evolution optimise the three components during the training phase. The objective of the system is to build two statistical models and a decision tree that approximates the compression ratio for the files in the training set when applying a particular compression model. After evolution, we test the performance of the evolved components on unseen data.

6.1.1 Analysing the Byte-Series and Byte-Frequency Distribution

Each file is stored within a computer, as a series of unsigned bytes. In our system we use this series as a reference interpretation for the data. In particular, we treat the stream of data as a signal, digitised using an 8-bit quantisation. Hence, each byte in a data file is treated as an integer between 0 and 255.

The task of the Byte-Series analyses tree is to evolve a non-linear function that extracts features out of the given byte-series that spot regularities and redundancy in the data stream to predict the compression ratio when apply a particular compression algorithm. Naturally, in practice, spotting such characteristics is not always straightforward. This depends on the nature of the given data stream. For example, it is easy to spot a regular pattern in an English text (e.g., 'th', 'qu'), but complex in an executable file. Also, large byte-series might conceal some useful features found only in some parts of the file.

Preliminary experimentation showed that analysing files' byte-series alone does not provide enough information to build a generic compression estimation model. Therefore, we also look at the Byte Frequency Distribution (BFD). BFD is defined as a histogram of the number of times that each character appeared divided by the total number of characters.

The basic concept of any compression model is to identify and remove the redundancy during the compression process. Here, we used BFD because it contains features about the amount of available information in the data (entropy) and also the symmetry in the data (from the point of

view of characters frequencies). Thus, the BDF allows GP to reveal these characteristics and spot commonalities among different characters in the data stream.

The task of the BFD analyses component of each GP individual is also to evolve a non-linear function that extracts features out of the given byte-series that spot regularities and redundancy in the data stream to predict the compression ratio when apply a particular compression algorithm. The advantage of this representation is that it is easy to process, since it is a list of 256 values only, and contains valuable information regarding the data stream. A disadvantage is that it ignores the order of the data in the stream.

Thus, the system analyses each representation of the data independently via two compression-prediction trees. Each component tries to estimate the compression ratio of the given file from its own point of view. The output of each compression-prediction trees is a real number. Because the complex nature of files' structure, none of these two component is reliable enough to provide enough information to build a generic compression estimation model on its own. [1] Therefore, our system look at the output of the two components and compare them with the given coding situation to make an educated decision as to estimate the compression ratio.

The system integrates the outputs of the two compression-prediction trees into a single evolved decision tree. The output of the decision tree is the final estimated compression ratio (i.e., a real number). The decision tree produces its result based on a series of comparisons among statistical features extracted from the files and the outputs of the other two evolved components. The evolved decision tree has the choice to either select the outputs of the compression-prediction trees or alternatively, to integrate them into an evolved mathematical formula in an effort to improve the prediction. Section 6.1.2 will describe the decision tree's structure in detail.

[1] Most common files include complex structures that store data of different types simultaneously. For example, a single powerpoint file might contain text, video, pictures and background music.

6.1.2 Decision Tree

A decision tree is a model that maps from the attributes of an item to a conclusion about its value [93]. Leaves in decision trees represent classifications and branches represent conjunctions of features that lead to classifications. Decision trees can be represented as an if-else-if series for human readability [93].

The decision tree receives the two compression-prediction trees (described in section 6.1.1) as terminals. For our compression prediction system we customised the decision tree representation to fit our objective. As displayed in the decision tree in figure 6.1, each comparison node (i.e., a node that contains a comparison condition such as, $<$, $>$, $\leq$ or $\geq$) has four children. The first two represent conditions, while the other two represent decisions. There are two types of condition trees and three types of decisions. The details of each type are as follows:

- Condition types:
  1. *Byte-series Condition tree*: this is a component tree (see middle of figure 6.1) that extracts features from the given Byte-series representation and abstracts them to a single number (the estimated compression ratio from Byte-series tree point of view).
  2. *BFD Condition tree*: this is a component tree (see figure 6.1) that extracts features from the given Byte-Frequency Distribution representation and abstracts it to a single number (the estimated compression ratio from BFD tree point of view).
- Decision types:
  1. *BFD Decision*: the decision tree decides that the output of the BFD analyser tree is closer to the actual compression ratio of the file.
  2. *Byte-series Decision*: the decision tree decides that the output of the Byte-series analyser tree is closer to the actual compression ratio of the file.
  3. *BFD/Byte-series Decision*: the decision tree decides that neither of the two components is close enough to the actual compression ratio and joins the output of both

components into an evolved mathematical formula to approximate the its output to the actual compression ratio of the file.

The output of the decision tree is the estimated compression ratio. Thus, the evolved decision tree has three choices: *i) select the output of the Byte-series analyser tree*, *ii) select the output of the BFD analyser tree* or *iii) integrate both components* (Byte-series analyser and the BFD analyser trees) into a mathematical formula and use that to produce the output.

6.1.3 Search Operators

We used crossover, mutation and reproduction. Naturally, the genetic operators take the multi-tree representation of the individuals into account.

After experimenting with a variety of approaches we settled for the following. Let T_c^i be the cth tree of individual i, where $c \in \{Byte_series_analyser, BFD_analyser, Decision_tree\}$. The system selects an operator with a predefined probability for each T_c^i. Thus, offspring can be generated by using more than one operator.

In crossover, as each component has a particular task, only homologous components are allowed to cross. Also, the system has to take the structural constrains of the decision tree into consideration and ensure its syntax is maintained. The system crosses condition branches with other condition branches from corresponding trees and decision branches with other decision branches from corresponding trees.

6.1.4 Fitness Evaluation

As mentioned previously, each individual has a multi-tree representation, where one tree is used to analyse the byte-series, another tree is used to analyse the BFD and a third tree is used to decide which is the most accurate prediction of the compression ratio. Thus, there are three different objectives for the system. The first is to optimise the performance of the byte-series

analyser tree, the second is to optimise the performance of the BFD analyser tree, and the third is to optimise the decision maker's performance.

We look at this as a multi-objective problem with three fitness functions. Each fitness function measures the quality of one component. The system randomly selects a fitness measure each time it produce a new individual. In this way evolution is forced to jointly optimise all objectives.

The fitness function for each component is simply the average of the absolute difference between the estimated compression ratio and the actual achieved compression for all files in the training set as we explain below.

The fitness of the BFD analyser tree can be expressed as follows: let the output of the BFD-tree be denoted as $BFD(file_i)$,where $file_i$ is the ith file in the training set. Furthermore, let $C(y, file_i)$ be the compression saving of $file_i$ when applying the compression model y.

$$f_{BFD_tree} = \frac{\sum_{i=1}^{n} |BFD(file_i) - C(y, file_i)|}{n}. \quad (6.1)$$

The fitness of the byte-series analyser tree can be expressed as follows:

$$f_{BS_tree} = \frac{\sum_{i=1}^{n} |BS(file_i) - C(y, file_i)|}{n} \quad (6.2)$$

where $BS(file_n)$ is the output of the byte-series tree.

Finally, the fitness of the Decision tree can be expressed as follows. Let the output of the Decision tree be denoted as $DT(bs, bfd, file_i)$, where bs is the output of the byte-series analyser tree, and bfd is the output of the BFD analyser tree. We set,

$$f_{DT_tree} = \frac{\sum_{i=1}^{n} |DT(bs, bfd, file_i) - C(y, file_i)|}{n}. \quad (6.3)$$

TABLE 6.3: Training file types for the compression prediction system.

File types	**Total Size**
pdf, exe, C++ code, gif, jpg, xls, ppt, mp3, mp4, txt, xml, xlsx, doc, ps, ram	5.14 MB
Total number of file types	**26**

6.1.5 Training and Testing

The system extracts knowledge concerning the features of the data and their relationships with the performance of a particular compression algorithm during a training phase (a GP run).

Several factors have been considered while designing the training set. These are the same as with GP-zip2: the training set has to contain enough diversity of data types in order to ensure the generality of the system, and the size of the training set should be small enough to maintain time-efficient training. In addition, it is essential to avoid over-fitting the training set.

Table 6.3 presents the data types within our training set. The training set contains 15 different file types within 26 files for a total of 5.14MB. The training set is completely independent of the test set.

GP's output at the end of the evolution consists of a byte-series analyser tree, a BFD analyser tree and a decision tree estimating compression ratios based on the other components. Because GP is stochastic, the user needs to run the system several times until it achieves adequate performance on the training set. Testing involves processing unseen files using these trees.

Table 6.4 presents a list of file types that have been used to measure the algorithms' performance. The test set contains 19 different data types within 27 files. It contains some file types similar to those used in the training set, while others are different data types to which the algorithm has not been exposed during training. Details regarding the algorithm's performance are given in section 6.3.

Experimentation with this technique has produced accurate estimations for different files when applying different compression modules. Also, the estimation of compression was much faster

TABLE 6.4: Testing files types for the compression prediction system.

Files types	Total Size
tif, jpg, bmp, accdb, xml, c++ code, txt, mht, doc, docx, ppt, pdf, exe, msi, wmv, flv, mp4, mp3, ram	67.9 MB
Total number of files types	**27**

than performing the actual compression itself (results details presented in section 6.3). Although the main motivation for developing this system was to improve the performance of GP-zip2, one cannot ignore the potential and benefits of such system as an independent application to support other existing systems. For example, it is evident that testing alternative compression algorithms to determine the best one to use is extremely time consuming when the given data is large (e.g., > 1GB). Applying a random compression model in this case might result in loss of efficiency with regards to storage space or it might even cause the file size to increase. Consequently, estimating the compression ratio when applying different compression models could be very advantageous in saving both computational resource and the time required to perform the compression with traditional techniques.

6.2 GP-zip3

In order to estimate compression ratios we used the GP system described in the previous section. Naturally, the evolved prediction models work well only for the particular compression algorithm they have been trained to predict. Thus, the user has to evolve a prediction model for each compression algorithm of interest.

In GP-zip3, prediction programs evolved with this approach were used to determine the best way to compress the segments produced by the splitter tree, rather than trying every available algorithm on each segment. This has accelerated the training phase significantly. Naturally, the evolved estimation programs are not 100% accurate. Thus, the system could occasionally allocate a sub-optimal compression module to some of the segments. We expected this to reduce GP-zip's ability to produce high performance compression algorithms. However, as we will

see, this did not happen. On the other hand, GP-zip3 has less generalisation ability than its predecessor. We suspect the reason is that prediction errors acted as noise on the training set.

Naturally, the compression-prediction system proposed in the previous section was designed and used for the prediction of the compression ratio of entire files. Therefore, we had to customise this system in such a way as to make it work well with the segments of files produced by GP-zip's splitter tree. To achieve this, the training set was divided into blocks of predefined lengths, namely $200, 350, 450, 800, 1000, 2000, 3000$ and 5000 bytes. The sizes of the blocks were selected so as to approximately match the typical segments produced by splitter trees. These blocks were treated by the compression prediction system as independent files for which we wanted to estimate, as accurately as possible, the compression ratio achieved by a specific technique. Compression estimation modules were evolved for all the compression modules available to GP-zip3, i.e., AC, PPMD, Bzip2, LZW and RLE. These five pre-evolved estimation modules were then used in GP-zip3 as basis for fitness evaluation (see section 6.2.2)

6.2.1 GP-zip3 Search Operators

GP-zip3 uses the same search crossover and mutation operators as GP-zip2. If the ith individual of the population is denoted as I_i and T_c^i is the cth tree of individual i, where $i \in \{splitter, feature_extractor_x, feature_extractor_y\}$, 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 system is able to freely crossover feature-extraction trees at any position.

6.2.2 GP-zip3 Fitness Evaluation

GP-zip3 fitness function shares the same components as GP-zip2. The main difference is that GP-zip3 uses compression prediction programs to determine the best way to compress the segments produced by the splitter tree, rather than trying every available algorithm on each segment.

As before, the calculation of the fitness is divided into two parts. Each part contributes with equal weight to the total fitness. Firstly, the fitness contribution of the splitter tree is measured by *estimating* the total compression ratio on the training set. This is computed by evaluating the pre-evolved estimation functions for the segments identified by the splitter. The system will label the segments with the algorithm providing the highest estimated compression ratio. The estimated compression ratios obtained in the previous phase are then used to evaluate the fitness contribution of the splitter tree.

More specifically, let $ES_x(S_i)$ be the output of the estimation function for algorithm x, where $x \in \{AC, PPMD, LZW, RLE, Bzip2\}$ and let segment i be denoted as S_i. Furthermore, let n be the total number of segments. Then, the fitness of the splitter tree can be expressed as:

$$F_{Splitter} = \frac{1}{2} \times [100 - (\frac{\sum_{i=1}^{n} \max_x(ES_x(S_i))}{\mathit{File\ Size}} \times 100)]. \tag{6.4}$$

Note that this approach is computationally much less expensive than actually compressing each segment in the training set as was required by the fitness function used in GP-zip2.

The second part of an individual's fitness is the classification accuracy provided by the feature-extraction trees, as in GP-zip2. After performing the clustering using K-means, the quality of the clustering is evaluated by measuring each clusters homogeneity and separation. The formula used in GP-zip3 to do this are provided in equations 5.2, 5.3, 5.4 and 5.5.

6.2.3 GP-zip3 Training and Testing

In order to ease the comparison of the performance of GP-zip3 against its predecessor, the same data sets used to train and test GP-zip2 were included in our dataset (see tables 5.2 and 5.3). This allows us to monitor the learning speed and compare the generalisation ability in the two systems. Furthermore, the same testing files were used to measure the performance of GP-zip* and GP-zip. Thus, a comprehensive comparison is possible.

TABLE 6.5: GP-zip3 parameters settings.

Parameter	*Value*
Population Size	100
Generations	30
One-point Crossover Probability	50%
Mutation Probability	50%
Tournament Size	5
Window size for splitter tree (L)	100
Window step for splitter tree (S)	50
Splitter tree threshold (θ)	10

6.3 Experiments

This section will present the experiment results for GP-zip3. In addition, the experiments with the compression prediction system are presented. As mentioned previously, the main motivation for developing this system was to improve the performance of GP-zip2. However, this technique can be used independently as a compression recommendation system to support other existing systems.

The performance of GP-zip3 is mainly measured by evaluating the compression ratio (see equation 2.1). The time needed to perform the learning and the actual compression process is also reported.

6.3.1 GP-zip3 Experimental Results

The main aim of GP-zip3 experiments was to investigate its performance and speed in comparison with its predecessors. Thus, GP-zip3's performance was measured under the same conditions as GP-zip2. In other words, the same training set and testing set were used in the experimentation (see tables 5.2 and 5.3).

The experiments presented here were performed using the parameter settings shown in table 6.5. Similar to GP-zip2, there is no special terminating condition for the run. Simply, the system runs until the maximum number of generations is reached.

TABLE 6.6: GP-zip3 summary of results in 15 independent GP runs

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	33.52	3.64	35.49	23.36	35.67	23.36
Archive2	3.51	0.54	3.88	2.23	3.88	2.23
Archive3	59.30	16.31	65.32	25.66	66.51	13.48
Archive4	90.57	1.05	91.59	88.67	91.59	87.84
Archive5	8.70	0.58	9.56	8.72	9.56	7.32
Archive6	56.21	5.19	59.91	57.83	59.91	45.22
Archive7	69.48	8.33	72.65	39.99	72.90	39.99
Archive8	18.37	1.71	19.25	17.99	19.85	12.81
Archive9	2.95	0.51	3.47	3.67	3.89	2.12
Canterbury	74.40	14.12	81.17	27.84	81.55	27.84
Exe	52.77	16.89	64.55	9.44	66.94	9.44
Text	74.60	12.02	80.29	71.48	80.29	35.39
Mean	45.38	6.74	48.93	31.41	49.63	25.59
StdDev	30.84	6.50	32.68	27.90	32.99	24.45

6.3.1.1 GP-zip3's Behaviour Across Runs

GP-zip3's performance has been measured through 15 different runs, each of which trains the system and uses the output of the training to compress the 12 different test archives.[2] The aim of the experiments is to compare the performance of GP-zip3 against GP-zip2. Naturally, this includes obtaining a reasonably general compression algorithm that performs well, on average, for all test files.

Table 6.6 summarises the 15 GP runs. The first column reports the average compression ratio in all runs, archive by archive. The second column illustrates the standard deviation of the achieved compression ratios in all runs for each archive, to show the system's robustness when dealing with different archives. The third column provides the results of the best evolved compression program. By way of comparison, the fourth column reports the results of the worst evolved compression program, in order to show the performance of GP-zip3 in its worst cases. The last two columns report the best and worst achieved compression ratio respectively for each file in any run.

[2]The same number of runs as in GP-zip2 was performed to allow fair comparisons between the two systems.

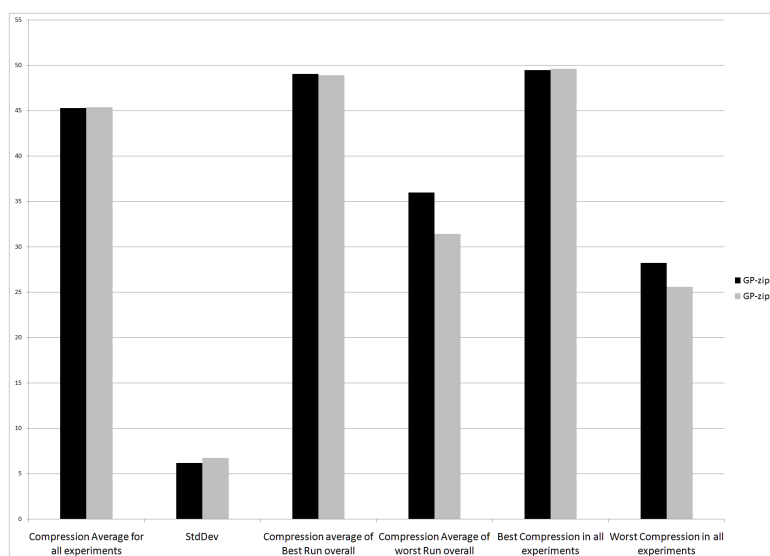

FIGURE 6.2: GP-zip2 vs. GP-zip3.

There are several things worthy of note in this table. Firstly, we find that, as illustrated by the second column, runs of GP-zip3 were able to evolve compression algorithms which generally do a very good job at compressing multi-file archives, almost similar to GP-zip2. However, GP-zip3 outperforms its predecessor when comparing the Canterbury corps (with 3.32% difference). Overall, we can see that GP-zip3 has almost similar results as its predecessor. GP-zip3 has inferior compression ratio than GP-zip2 when comparing the worst run overall and worst Compression in all experiments (i.e., columns 5 and 7). These results are confirmed in figure 6.2 where an overall comparison is presented (comparing the mean performances in tables 5.5 and 6.6).

The fourth column of table 6.6 reports the performance of the best-of-run program evolved in our runs which showed the best average performance over the 12 archives in the test set. This is probably the compression algorithm the user would choose after our set of 15 run; all other results would be discarded. However, the user is also free to try all possible evolved solutions and execute each of the 15 best-of-run programs and thereafter pick the best result (see section 5.2.1.1).

6.3.1.2 GP-zip3 vs. GP-zip2

Table 6.7 shows the compression ratios obtained on each of our 12 test files by the best best-of-run program evolved by GP-zip3, and GP-zip2 and the sequential strategy highlighted above of testing all 15 best-of-run programs sequentially. As one can see, on the left side of the table, GP-zip2 is slightly superior to GP-zip3. This is no surprise; the reason is that GP-zip2 is getting accurate information during the training phase while GP-zip3 has to work with a noisy fitness evaluation. Actually, it is remarkable that GP-zip3 managed to stay this close to GP-zip2. In fact, in most cases GP-zip3 still produced competitive compression results, outperforming traditional compression algorithms in its function set with considerable margin.[3] This indicates that the prediction technique used to evaluate the individual did a good job in guiding the course of evolution. On the other hand, the right side of table 6.7 presents the best evolved program in all 15 runs. This is probably the compression algorithm the user would choose after our set of 15 runs; we can see that GP-zip2 outperforms its successor in 7 out of 12 testing cases. However, on average, the difference between the two systems is still small, being 0.12%. It should be noted that the best evolved program by GP-zip3 still outperforms other traditional compression algorithms, with a considerable margin in most cases. This indicates that GP-zip3 was able to evolve a good program that can effectively compress different heterogeneous archives.

Based on these observations we can conclude that both systems are able to achieve similar compression performances when dealing with test cases that are similar to its training set.

GP-zip3 required only two and a half hours on average to perform the learning process, while GP-zip2 required 6.4 hours. Both GP-zip2 and GP-zip3 take between 5 and 10 minutes to perform the actual compression on an AWS cloud computing using virtual cores with 2 EC2 Compute Units each (One EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor).

[3] Reporting the best performance achieved by programs evolved in different runs may seem unfair. However, this level of performance is achievable, albeit with some computational cost: all one needs to do is to compress a file with each of the 15 evolved programs and pick the shortest compressed version.

TABLE 6.7: Performance comparison (compression ratio %) between GP-zip3 and GP-zip2

File	GP-zip2 sequential	GP-zip3 sequential	GP-zip2 Best	GP-zip3 Best
Archive1	**35.98**	35.67	35.36	**35.49**
Archive2	**3.93**	3.88	3.34	**3.88**
Archive3	**66.69**	66.51	**66.66**	65.32
Archive4	**91.65**	91.59	91.46	**91.59**
Archive5	**9.90**	9.56	**9.73**	9.56
Archive6	59.90	**59.91**	59.89	**59.91**
Archive7	72.76	**72.90**	**72.76**	72.65
Archive8	**19.95**	19.85	**19.65**	19.25
Archive9	3.71	**3.89**	**3.71**	3.47
Canterbury	80.76	**81.55**	77.94	**81.17**
Text	**80.44**	80.29	**80.29**	**80.29**
Exe	**67.86**	66.94	**67.86**	64.55
Mean	**49.46**	49.38	**49.05**	48.93

Bold numbers are the highest

Experimentation with GP-zip3 demonstrated that, despite the much shorter learning time in comparison to GP-zip2, GP-zip3 achieves almost the same level of performance in terms of compression ratios. The disadvantage of GP-zip3 is that it depends on evolved estimation functions. Thus, the user must spend extra time evolving accurate estimation functions. Also, the user has to evolve an estimation function for each of the compression algorithms available to GP-zip3. This, however, needs to be done only once. When estimation functions are available, GP-zip3 can efficiently be applied to evolve general compression algorithms as well as compression algorithms tailored to specific domains of application.

6.3.1.3 GP-zip3 Generalisation

The estimation approach used to accelerate the evaluation of the splitter trees acted as noise (i.e., produced errors in the estimated values) on the training set. While in small amounts, noise can improve a learning process by increasing the algorithm's generalisation, beyond a certain level, noise may result in deleterious consequences for the learning process and limit the algorithm's abilities to generalise its knowledge beyond the training set. In order to compare the generalisation performance of GP-zip3 against its direct predecessor, here, the best evolved program in both GP-zip2 and GPzip3 were used to compress 8 more archives (the same 8 archives used

TABLE 6.8: GP-zip3 generalisation performance

Archive	*Gp-zip2*	*Gp-zip3*
Archive10	**29.29**	28.22
Archive11	**27.57**	26.57
Archive12	**47.25**	46.76
Archive13	**5.70**	4.69
Archive14	**9.20**	7.46
ChessMac	**72.35**	71.31
Worms2Demo10OctA	2.41	**2.46**
Calgary	69.84	**75.39**

Numbers in **bold** are the highest

to test GP-zip2 generalisation), which are completely independent from the training set (see table 5.9).

Table 6.8 illustrates the results achieved when compressing the aforementioned archives. Both GP-zip2 and GP-zip3 achieved almost similar compression ratios. However, we can see that GP-zip3 has slightly inferior compression ratios. This may indicate that GP-zip3 has less generalisation abilities than GP-zip2. Hence, it is fair to say that GP-zip3 is ideal in situations where the user needs to quickly evolve a general purpose compression system and the types of data to be compressed are roughly expected.

The user is expected to train GP-zip3 a number of times until it achieves adequate performance on the training set. Thereafter, training outputs are used (1 splitter tree and 2 feature extraction trees) as an independent program without the need for any further evolution.

6.3.2 Compression Prediction System Experimental Results

As mentioned previously, the main motivation for developing this system (i.e., compression ratio prediction) was to improve the performance of GP-zip2. However, this technique can be used as an independent application. This is useful in situations where the users have limited computational power and are interested in both time and computational savings.

TABLE 6.9: Compression prediction system parameter settings.

Parameter	*Value*
Population Size	200
Generations	40
One-point Crossover Probability	90%
Mutation Probability	5%
Maximum tree depth	10
Reproduction Probability	5%
Tournament Size	5

The approach has been tested to predict the compression ratio for the files in the test set for the following compression algorithms: Prediction by Partial Matching (PPMD) and Arithmetic coding (AC).[4]

The experiments presented here were performed using the parameters shown in table 6.9. There is no special terminating condition for the run. Simply, the system runs until the maximum number of generations is reached.

Figures 6.3 and 6.4 summarise the results of the experiments when evolving predictors for AC and PPMD (we performed 10 independent GP runs for each compression model). The graphs plot the best, worst and average prediction error for each file. Prediction error is measured as the absolute difference between the actual compression ratio and the estimated compression ratio (expressed as a percentage). Also, the standard deviation of the achieved predictions for each file in all runs is recorded. Each figure also shows the average of the best and worst predictions in all runs.

As one can see the average of the best achieved prediction errors is small (with values ranging from 0.8% to 1.83%). The small standard deviation with the reasonable predictions average indicates that our system is likely to produce accurate models within a few runs.

As we mentioned before, the test set contained some files that consist of data types, to which the algorithm has not been exposed during training (see section 5.1.8). Table 6.10 illustrates the average achieved prediction errors for the seen file types and the unseen file types. Generally,

[4]Predictors for LZW, Bzip2 and RLE are also evolved and integrated into GP-zip3. However, these two algorithms were used to verify whether GP is able to solve this problem at all.

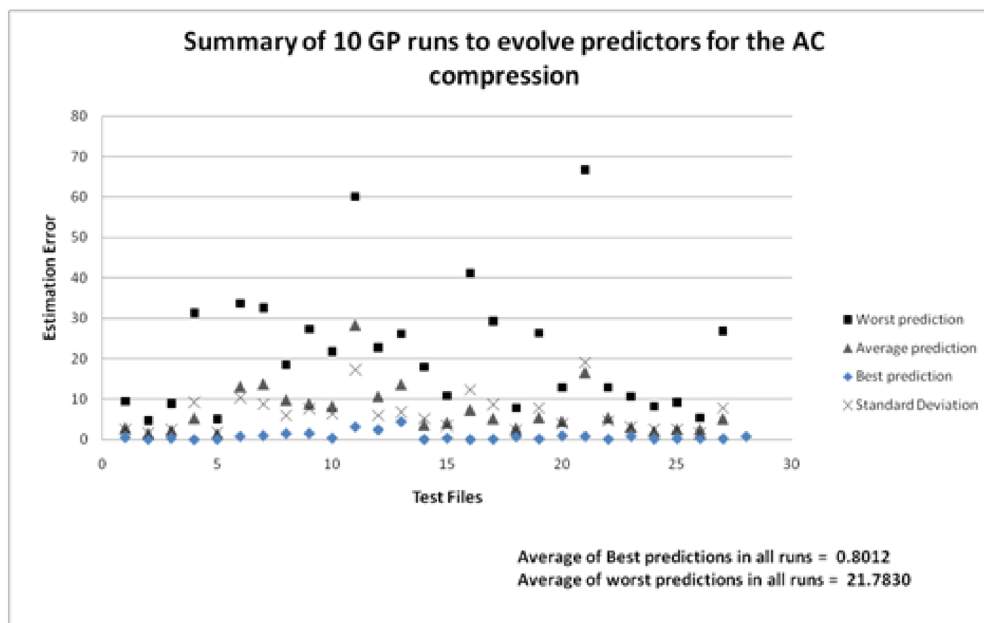

FIGURE 6.3: Summary of 10 GP runs to evolve predictors for the AC compression.

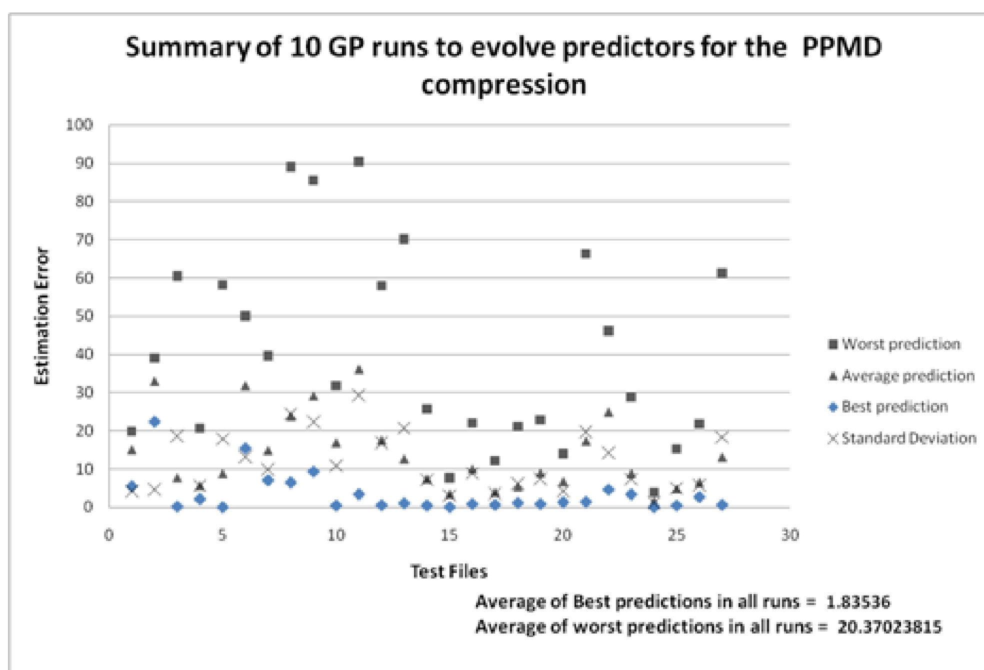

FIGURE 6.4: Summary of 10 GP runs to evolve predictors for the PPMD compression.

the algorithm gives slightly less accurate results for those types which it had no prior experience. Nonetheless, the achieved predictions when the algorithm deals with new files types are satisfactory. Consequently, the algorithm must have learnt some general knowledge which can be used in different situations.

TABLE 6.10: Compression prediction system's performance Comparison (Seen file types vs. Unseen file types)

Compression Model	*Avg .prediction error for trained files types in all runs*	*Avg .prediction error for untrained files types in all runs*
AC	5.95	7.31
PPMD	11.96	16.98

TABLE 6.11: Compression prediction system - Decision tree performance

Decision	*Percentage*
Improved decision	7.05%
Right decision	61.78%
Wrong decision	31.17%

Each component in an individual has a particular task, as explained previously. The final output is produced by the decision tree using the estimates constructed by the two other components. Hence, it is interesting to study how the decision tree integrates such information.

As previously mentioned, the evolved decision tree has three choices: select the output of the Byte-series analyser tree, select the output of the BFD analyser tree or integrate both Byte-series analyser and BFD analyser trees into a mathematical formula (see section 6.1). Thus, if the decision tree selected the closest estimate to the actual compression ratio then it is considered to be a right decision. If it chose to return the estimate of the less precise component, we count it as a wrong decision. If the decision tree decided to integrate the Byte-series and BFD analysers into a mathematical formula producing a more accurate estimate we count that as an improved decision.

Table 6.11 shows the proportion of *correct/wrong/improved* decisions for the decision tree in all 20 runs. Decision trees were able to select the correct compression estimation in most of the cases. Table 6.12 shows how often the BFD and Byte series trees were used to produce the right estimation for the compression ratio. Both components have been used to estimate the compression ratio.

Naturally, the aim of estimating the compression ratio via different compression modules to save both computational resource and the time required to perform a compression. Thus, table 6.13

TABLE 6.12: Compression prediction system - Decision tree- Right decisions statistics

Decision	*Percentage*
BFD tree	73.87%
Byte series tree	26.13%

TABLE 6.13: Compression prediction system - Compression time vs. Prediction time.

Compression algorithm	*Compression time*	*Prediction time*
PPMD	148 seconds	62 seconds
AC	52 seconds	45 seconds

reports a comparison between the times needed to compress all files in the test set (67.9 MB) against the time needed to predict their compressions. It is clear that our approach is faster than performing the actual compression. This is no surprise, as compression algorithms involve a lot of I/O operations while our approach only scans the file and extract statistical features that correlate with the compression ratio.

6.4 Summary

Despite the success achieved by GP-zip2 (see previous chapter) it suffers from one major disadvantage. GP-zip2 runs require approximately 7 hours in the training phase. This is still significantly slower than standard compression algorithms.

To solve this limitation, an improvement to GP-zip2 was proposed; the improvement is referred to as GP-zip3. To achieve this we eliminated the need for repetitive compression of the training set with multiple compression algorithms in the fitness evaluation. We proposed to use evolved estimation functions to predict the compression ratio achieved on a segment of data with different compression algorithms without applying the actual compression algorithms themselves. We found that doing so reduced the training time significantly and achieved almost the same results.

The proposed method to accelerate GP-zip2's evolution uses GP to evolve programs that predict the compression saving for the data via two different interpretations: i) looking at a file's byte-series and ii) considering a file's bytes frequency distribution. Each interpretation is used by a separate tree within our representation in conjunction with a collection of statistical measures. We require the two trees to predict as best as possible the compression ratio achievable when applying a particular compression model. A third component of the representation, a decision tree, attempts to distinguish the most accurate of the two predictions and if necessary integrates them into an even better prediction. Evolution is guided by a single fitness measure (prediction error), which is applied randomly to one of the tree components in the representation. This forces GP to evolve accurate predictors in the first two components but in such a way that the third component can easily understand which predictor is more accurate, thereby effectively performing a kind of multi-objective optimisation. The final outcome of the prediction system (evolved prediction program) is integrated into GP-zip3 as an estimation function. Although the main motivation for developing this technique was to improve GP-zip2, it can be used as an independent application. This is useful for checking whether it is worth to apply a compression process to some data before transmitting them over a network or storing them on disk.

Experimentation with GP-zip3 demonstrated that, despite the much shorter learning time in comparison to GP-zip2, GP-zip3 achieves almost the same level of performance in terms of compression ratios. The disadvantage of GP-zip3 is that it depends on evolved estimation functions that a user needs to evolve before the system can be used. This, however, needs to be done only once and, in fact, users could easily share their best evolved predictors for each compression algorithm, effectively allowing new users of GP-zip3 to immediately start using the system. When estimation functions are available, GP-zip3 can efficiently be applied to evolve general compression algorithms as well as compression algorithms tailored to specific domains of application. In addition, experimentation revealed that GP-zip3 may have less generalisation abilities than its predecessor. Thus, GP-zip3 is ideal in situations where the user need to quickly evolve genetic compression system and the types of data to be compressed are reasonably not

too different from the training set (e.g., media databases that include a variety of images, videos and music).

Chapter 7

Extensions of GP-zip2

The main objective of GP-zip2 was to perform intelligent universal compression that can work effectively with different data types. To this end, GP-zip2 used a learning technique to analyse and understand the given data (treated as digital signals). These signals are processed by using an evolved splitter tree to analyse statistical characteristics within the data and split the data based on them. Additionally, two feature-extraction trees are used to project the data into a reduced-dimensionality space from the original space. These three trees are co-evolved together and act as one program in order to solve the given problem in collaboration. Experimentation revealed that this technique works very well with heterogeneous archives where it can outperform state of art compression algorithms (see sections 5.2.1 and 6.3.1).

Although the central objectives of the work presented in this thesis are related to intelligent data compression, in this chapter, we aim to study the benefits and potentials of the learning strategy employed by GP-zip2 beyond the domain of data compression.

Two main factors were considered whilst selecting the problems to be used to test the generality of GP-zip2's learning technique. Firstly, the selected problems should share similar characteristics with data compression. In particular, the selected problems should consist of a classification

task that deals with a continuous stream of data where the user needs to identify different fragments and associate them with different classes. Secondly, the selected problems should be non-trivial and lead to practical applications of GP.

Out of the many problems that satisfy these criteria, two problems were selected to validate the generality of GP-zip2's learning model. Firstly, we considered the application of the model in analysing human muscles EMG signals to predict fatigue onset. Results of this approach were quite promising as we will see below. Secondly, we considered a novel application of GP where we used GP-zip2's model for identification of file types via the analysis of raw binary streams (i.e., without the use of meta data). Experimentation with this application showed that the identification accuracy obtained by the GP-zip2 model was far superior to those obtained by a number of standard algorithms. The next sections will describe each test problem in details.

The material presented in this chapter on the two problems mentioned above has been published in [99] and [100].

7.1 Detecting Localised Muscle Fatigue during Isometric Contraction

7.1.1 Introduction

Electro-myo-graphy (EMG) is a technique used to record the electrical activity of muscles [101]. Muscles produce an electrical potential that is (non-linearly) related to the amount of force produced in a muscle. Analysing these signals and associating them with muscle state has been an area of active research in the biomedical community for many decades. However, the problem itself is extremely hard. Consequently, researchers have begun to combine statistical methods with Artificial Intelligence (AI) techniques in order to gain more understanding of EMG and utilise them successfully. This has produced significant improvements that are starting to lead to practical applications. For example, Sony [102] has presented a hardware system for musical

applications that is controlled by EMG signals. The system is able to recognise different gestures and associate them with particular commands to the machine.

Some problems has also been made in the detection of muscle fatigue. For instance, Atieh *et al.* [103] tried to design more comfortable car seats by trying to identify and classify EMG signals using data mining techniques and statistical analysis to determine localised muscle fatigue. Artificial neural networks have been used to detect muscle activity by Moshou *et al.* [104]. Wavelet coefficients were proposed as features for identifying muscle fatigue. Song and collaborators [105] proposed an EMG pattern classifier of muscular fatigue. The adaptation process of hyperboxes of fuzzy Min-Max neural networks was shown to significantly improve recognition performance. Detecting muscle fatigue, however, is still an extremely challenging task.

In this application we investigate the idea of using GP-zip2's model to predict localised muscle fatigue by identifying a transition state which resides between the non-fatigue and the fatigue stages within the EMG signal. A collection of statistical measures have been used to generate new composite, higher-level features and correlate muscle activity with particular patterns (details will be given in the following sections).

There are two main motivations behind this application. The first motivation is to study the limitation and the capabilities of using GP-zip2's model in the mentioned domain. The second is the desire to achieve real time processing of the EMG signal and provide an early warning before the onset of fatigue.

As we will illustrate in the following subsections, the proposed approach was able to detect the onset of fatigue in most of the experimental cases we looked at. Therefore, this approach shows some potential for application domains such as ergonomics, sports physiology, and physiotherapy.

7.1.2 Applying GP-zip2 Model

To test the application of GP-zip2's learning techniques to analyse EMG and predict muscle fatigue, we used the same implementation of GP-zip2 as in chapter 5. The difference, however, is that we feed the system with EMG signals instead of binary streams from computer files. Thus, instead of associating data fragments with compression algorithms, here the system associates them to muscles status classes.

An ideal fatigue prediction system would be one that non-invasively reads data from the athletes' muscles and alerts them when the onset of fatigue approaches. The use of GP to identify localised muscles fatigue has not been explored thus far. The system works as follows:

The system tries to spot regularities within the EMG data and to associate them to one of three classes: *Non-Fatigue, Transition-to-Fatigue,* and *Fatigue*. Each class indicates the state of the muscle at a particulate time. The system works in two main stages: *i) Training*, where the system learns to match different signals' characteristics with different classes, and *ii) Testing*, where the system applies what it has learnt to classify unseen data.

In the training phase, the system processes filtered EMG signals and performs two major functions: *i) Segmentation* of the signals based on their statistical features, and *ii) Classification* of the identified segments based on their types (i.e., Non-Fatigue, Transition-to-Fatigue, or Fatigue).

For these tasks, GP has been supplied with a language that allows it to extract statistical features from EMG. This is similar to the primitive set used in GP-zip2 for data compression. However, zero-crossings is an additional primitive. Table 7.1 reports the primitive set of the system.

The system starts by randomly initialising a population of individuals using the ramped half-and-half method. Similar to GP-zip2's model, each individual has a multi-tree representation. In particular, each individual is composed of one splitter tree, and two feature-extraction trees (see figure 5.2). In the next subsections we will describe the job of these trees in detail and how they have been utilised to serve the problem of analysing EMG signals.

TABLE 7.1: Fatigue prediction system - Primitive set

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

7.1.3 Splitter Tree

The main job of splitter trees is to split the EMG signals in the training set into meaningful segments, where by "meaningful" we mean that each segment indicates the state of a muscle at a particular time.

As in GP-zip2, the system moves a sliding window of size L over the given EMG signal with steps of S samples. At each step the splitter tree is evaluated. This corresponds to applying a function, $f_{splitter}$, to the data under the window. The output of the program is a single number, λ. The system splits the signal at a particular position if the difference between the λ 's in two consecutive windows is more than a predefined threshold θ. Similar to GP-zip2, the threshold θ = 10, The size for the sliding window is $L = 300$ and $S = 50$. Preliminary tests revealed that sliding window of 300 bytes is best to capture statistical difference within the EMG signals.

Once the data have been divided into blocks, the system labels each block with one of the three identified classes, based on the number of *zero crossings* in the raw EMG signal, i.e., the number of times the signal crosses the zero-amplitude line (details are in section 7.1.5). A good splitter tree should be able to detect three types of blocks: *Non-Fatigue, Transition-to-Fatigue,* and *Fatigue*.

Preliminary tests showed that an average EMG signal in our set has 50% non-fatigue, 10% transition-to-fatigue and the remaining 40% is fatigue. Naturally, these numbers are variable from one individual to another. However, what is common among signals is that the smallest portion of the signal represents transition-to-fatigue while the largest portion is non-fatigue.

Thus, the splitter tree can be considered to be good if it divides the signal into the three types of blocks with both meaningful proportions (i.e., fatigue $>$ non-fatigue $>$ transition-to-fatigue) and a meaningful sequence (non-fatigue should appear before transition-to-fatigue and fatigue). Splitter trees that violate these conditions are discouraged by penalising their fitness value (see section 7.1.7).

An effective splitter tree would be able to detect the statistical differences within the signal and isolate the boundaries between the non-fatigue parts and the fatigue parts with a transition-to-fatigue block in the middle.

7.1.4 Feature-Extraction Trees

The main job of the two feature-extraction trees is to extract features using the primitives in table 7.1 from the blocks identified by the splitter tree and to project them onto a two dimensional Euclidian space, where their classification can later take place. As in GP-zip2, feature-extraction trees represent a transformation formula which maps the subset of the features used as terminals in the tree into a single output, which can be considered as a composite, higher-level feature.

Again we used K-means clustering to organise blocks (as represented by their two composite features) into groups. The advantage with this approach is that the experimenter does not need to label the training set which may be very difficult to do. Once the training set is clustered, we agian use the clusters found by K-means to perform classification of unseen data.

One might wonder why we need to perform a feature extraction step via the two feature-extraction trees if we are then applying K-means to the data. In principle, one could imagine applying the K-means algorithm directly to the raw EMG features. However, there is a problem with this way of proceeding. K-means has no way of knowing what each cluster is meant to represent. Consequently, with the raw features K-means might easily produce results that are not ideal or even not useful for classifying unseen blocks of signal. By evolving feature-extraction trees, instead of forcing the clustering algorithm (the K-means in our case) to group segments

based on their raw statistical features directly, we let evolution optimise two features-extractions trees and use them to project the training segments on a two-dimensional Euclidian space. If the trees are successfully evolved, K-means will now be able to group together blocks that, based on their raw statistical features, would have otherwise been grouped separately and vice versa. The advantage of this approach is that GP might discover new features that a human may not think about.

In the testing phase, unseen data go through the three components of the evolved solution. Blocks are produced by the splitter tree and then projected onto a two-dimensional Euclidean space by the two feature-extraction trees. Thereafter, they are classified based on the majority class labels of their k-nearest neighbours. Unlike GP-zip2, here use a weighted majority voting, where each nearest neighbour is weighted based on its distance from the newly projected data point. More specifically the weight is $w = 1/distance(x_i, z_i)$, where x_i is the nearest neighbour and z_i is the newly projected data point. This is different than GP-zip2 in data compression, where unseen data are classified based on the closest centriod. Preliminary tests showed that weighted majority voting produced better classification results in this particular problem. Due to the difficulty of the problem and the overlap between clusters, the closest centriod did not always produce accurate classifications.

Once the system detects a transition-to-fatigue, it alerts the user about a possible approaching fatigue. Naturally, given the amount of noise and variability in EMG signals, the prediction is not very precise and might results in a false alarm in some cases. In principle, we can evaluate the confidence level of the prediction by measuring the average Euclidean distance between the coordinates of a segment and its n nearest neighbours.[1] Low confidence predictions could then be ignored or produce a low level warning. However, we have not explored this particular aspect.

[1] n is a predefined number which represent the number of closest neighbours.

7.1.5 Labelling the Training Set

Identifying muscle state has been an active research area in sports science, biology, and physiology communities for some time. There are several ways to recognise muscle state from the EMG signal. In [106–108] the authors argued in favour of the idea of counting the number of times the amplitude of the signal crosses the zero line based on the fact that a more active muscle will generate more action potentials, which overall causes more zero crossings in the signal. However, at the onset of fatigue the zero crossings will drop drastically due to the reduced conduction of electrical current in the muscle.

Hence, our system decides the labels of the blocks found by the splitter tree in the training set based on the number of zero crossings of the whole EMG signal. Before the system starts the evolution process, it scans the training set signals (offline) and divides them into blocks of a predefined length. Each block is 300 bytes, which is similar to the sliding window size (see section 7.1.3). The number of zero crossings from each block is stored in a sorted vector that does not allow duplicated elements. Then, the system divides this vector into three parts. The lower 40% is taken to represent fatigue, the middle 10% represents transition-to-fatigue, and the higher 50% is non-fatigue. These three propositions (10%, 40% and 50%) were selected based on preliminary tests with the EMG signals (as mentioned in section 7.1.3). The number of zero-crossings of the blocks in these three groups are then used to identify three intervals. Later, the output of the splitter trees is classified into one of these three groups based on the interval in which the number of zero crossings in it falls.

It should be highlighted that the system uses this approach during the training phase only. In the testing phase, unseen EMG signals are classified according to the statistical features that GP found during the training phase.

Despite the simplicity of this labelling technique, the labels that it assigned to the output of the splitter trees were consistent and were judged by an expert to indicate the muscle state in most of the cases. However, noise in some parts of the EMG signal can result in wrong labels.

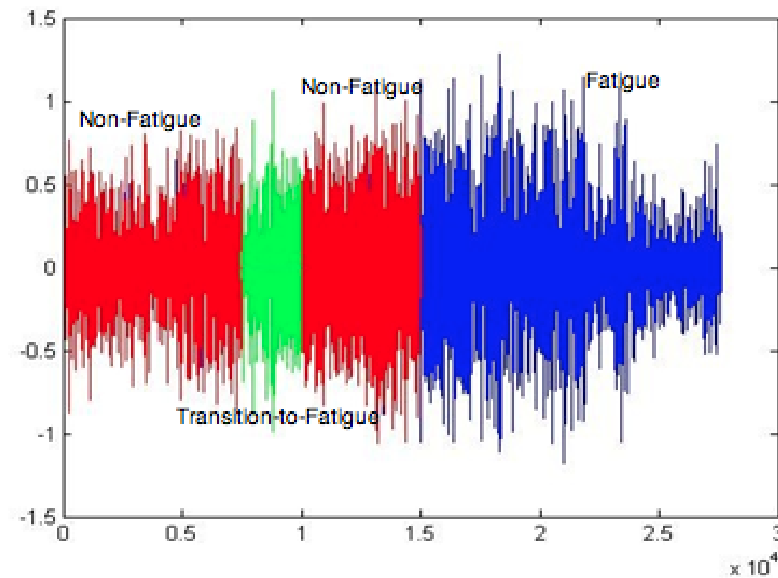

FIGURE 7.1: Labelled EMG signal using the zero crossing approach. Fatigue = blue, Non-Fatigue = red, Transition-to-Fatigue = green.

Sometimes when the EMG signal is noisy, the labels tend not to be in a meaningful sequence and do not reflect the actual state of the muscle. To solve this problem, we carefully selected a training set with the least noisy signals. Also, each fragment of signal in the training set was manually screened to ensure that it had the correct label. Figure 7.1 illustrates the labels for one signal in the training set.

7.1.6 EMG Filtering

Although EMG signals may contain valuable information about the state of individual muscles, there are several problems associated with their analysis. One of the major obstacles is their stochastic and noisy nature. This is due to subject movement during the recording of the EMG signal, neighbouring muscle interference, and other factors such as environmental factors [105].

Analysing a typical EMG signal shows an average amplitude range of $\pm 100mV$ depending on the muscle being analysed. Usable energy in the signal ranges from 10 to 500Hz although the dominant energy is somewhat smaller and falls in the 50-150Hz range [109].

Preliminary experiments with our approach failed to understand the raw EMG signals. This was no surprise. De Luca [109] highlighted a disadvantage of zero crossings: that they are prone to being effected by noise and are, therefore, not very reliable. Therefore, it was necessary to pre-process the EMG signals beforehand.

The signals of all participants were scaled and then filtered with a dual pass fifth-order Butterworth filter, with the band positioned between 1 and 500Hz to ensure accurate training of the GP system. The GP training was based on the zero crossings. However, to ensure the results were not too affected by noise, we developed a method to increase the signal to noise ratio in the data.

Our noise reduction method used a wavelet transform with the *'Sym4'* mother wavelet. In particular, we filtered the data using a wavelet function that closely resembles a motor unit action potential. This filtered out unwanted data, which in turn increased the reliability and accuracy of the zero crossing method. The method was tested rigorously for all trials against a force gauge reading to confirm the method's accuracy.

7.1.7 Fitness Evaluation

As mentioned previously, the same individual representation as in GP-zip2 was applied to solve this problem: One tree is used to split the EMG to identify the boundary among different muscle states, and two other trees provide composite features that are then used to classify the data blocks. Similar to GP-zip2, the calculation of the fitness is divided into two parts. Each part contributes with equal weight to the total fitness. The fitness contribution of the splitter tree is measured as follows.

The splitter tree is considered good if it divides the signal into homogeneous block of the three types discussed above with meaningful proportions and meaningful order. Those splitter trees that contravene these conditions are penalised. Hence, the evolution process will discriminate against them in the following generations.

The splitter tree fitness is measured by assessing how much it helped the feature extraction trees in projecting the segments into tightly grouped and well separated clusters, plus a penalty if required. More formally, the quality of the splitter tree can be expressed as follows. Let $f_{Feature_extraction}$ be the fitness of the feature-extraction trees, and μ a penalty values:

$$f_{Splitter} = \frac{f_{Feature_extraction} + \mu}{2} \tag{7.1}$$

where μ is added if, for example, the splitter does not respect the non-fatigue/transition-to-fatigue/fatigue sequence. The value of μ is fixed and applied whenever the splitter tree fails to divide the given EMG signal into blocks.

The second part of the individual's fitness is the classification accuracy (with K-means) provided by the feature-extraction trees. After performing the clustering using K-means we evaluate the accuracy of the clustering by measuring cluster *homogeneity* and *separation*. This is calculated as in GP-zip2. The system counts the members of each cluster (see figure 7.2). [2] Since we already know (from the previous step) the label for each block, we label the clusters according to the dominant members. The fitness function rates the homogeneity of clusters in terms of the proportion of data points – blocks – that are correctly labelled with the muscle's state that labels the cluster. As with GP-zip2, the system prevents the labelling of different clusters with the same label even in cases where the proportions in two or more clusters are equal. When two or more clusters have equal proportions of label types, the system will choose randomly between the alternatives. More formally, the clusters' homogeneity can be expressed as follows.

Let H be a function that calculates the homogeneity of a cluster and let CL_i be the ith cluster. Furthermore, let K be the total number of clusters (three clusters in our case: fatigue, transition-fatigue and non-fatigue). Then,

$$f_{Homogeneity} = \frac{\sum_{i=1}^{K} H(CL_i)}{K} \tag{7.2}$$

[2] It should be noted that each data point in the cluster represents a block of the EMG signal.

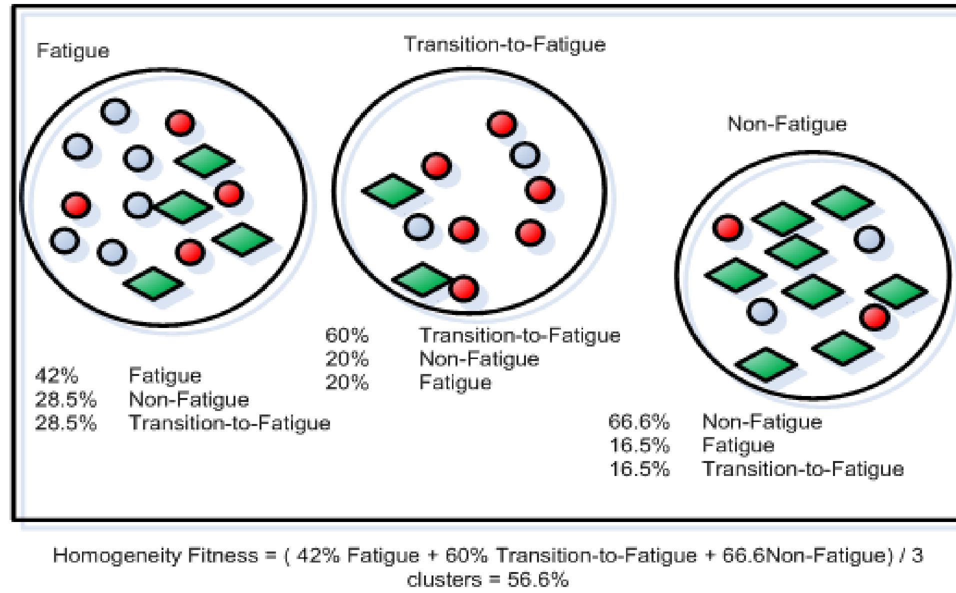

FIGURE 7.2: EMG analysis - Homogeneity of the clusters

Clusters separation is measured using a modified Davis Bouldin Index (DBI) (see equation 5.4). A small DBI index indicates well separated and grouped clusters. Therefore, we add the negation of the DBI index to the total feature extraction fitness in order to encourage evolution to separate clusters (i.e., minimise the DBI). Thus, the fitness of feature extraction trees is as follows:

$$f_{Feature_extraction} = \frac{f_{Homogeneity} - DBI}{2} \tag{7.3}$$

The total fitness of the individual is:

$$Fitness = f_{Feature_extraction} + f_{Splitter} \tag{7.4}$$

Thus, a GP individual's quality is defined by its ability to identify muscle state from the EMG signal and classify them correctly. GP explores the search space of possible programs using the standard search operators; crossover, mutation and reproduction. The fitness function in equation 7.4 rates each individual based on the criteria mentioned above.

7.1.8 Search Operators

Search operators in any GP system are important as they guide the search though the search space to discover new solutions. Given that the system uses the same implementation of GP-zip2, it can be seen that the same search strategy was implemented. See section 5.1.5.

7.1.9 Experiments

Experiments have been conducted in order to investigate the performance of GP-zip2's technique in this domain. The aim of these experiments is to measure the prediction accuracy with different EMG signals.

7.1.9.1 EMG Recording

The data was collected from three healthy subjects (aged 23-25, non-smoker, athletic background). The local ethical committee approved the experiment's design. The three participants were willing to reach the state of physical fatigue, but not psychological fatigue. Selection criteria were used to minimise the differences between the subjects, which would facilitate the analysis and comparison of the readings. The participants had comparable physical muscle strength. It was also preferred that the subjects had an athletic background with a similar muscle mass, which would facilitate the correlation of the results. It was also important that the volunteers were non-smokers, as it is known that smoking affects physical abilities, which again could lead to inaccurate readings of muscle fatigue in a participant who smokes.

The participants were seated on a chair. Each participant was asked to hold a weight training bar – dumbbell – until their muscle fatigued. The steps in our test bed set up are the following:

- A bipolar pair of electrodes was placed on the right arm's biceps muscle for EMG recording.

- The force gauge was perpendicular to the dumbbell to ensure that the force gauge was taking the correct reading.
- A strap was handed to participants to enable the force gauge reading.
- A protractor was used to ensure a 90 degree angle of the elbow for the initial setup.
- A dumbbell was handed to the participant.
- A laser was embedded in the dumbbell to give visual guidance of the elbow angle.
- The elbow position was padded, so the participant was comfortable.

The myoelectric signal was recorded using two channels; Double Differential (DD) recording equipment at 1000Hz sampling rate with active electrodes on the biceps branchii during two isometric dumbbell exercises, with 30% Maximum Voluntary Contraction (MVC) and 80% MVC respectively. The readings of the exerted muscle force were also measured with the force gauge and recorded to aid in analysis, as it gives a reliable indication of the development of fatigue. The force gauge reading was then correlated with the EMG signal. For each of the three participants, 6 trials were carried out, providing 18 trials in total.

7.1.9.2 GP Setup

Of the 18 EMG signals (trials) acquired, we used 3 trials for the training set, one trail from each subject. In this way we hoped we would allow GP to find common features for the three different participants and build a general prediction model. The experiments that are presented here were done using parameters settings in table 7.2.

Since GP search is a stochastic method, the performance of our approach has been measured through 18 independent runs, each of which trains the system and uses the output of the training to predict the muscle state of 15 EMG signals (5 signals for each participant). The aim is to obtain a good prediction for each participant and a reasonably general prediction algorithm that performs well on average for all participants.

TABLE 7.2: Muscle fatigue prediction system - Parameter settings.

Parameter	*Value*
Population Size	100
Generations	30
One-point Crossover Probability	90%
Mutation Probability	5%
Maximum tree depth	10
Reproduction Probability	5%
Tournament Size	5

7.1.9.3 Results and Analysis

Each GP run results in one splitter tree and two feature extraction trees. As mentioned previously, the splitter tree splits the EMG into blocks based on their statistical differences. Since during off-line tests it is possible to know the actual label for each block by counting its number of zero crossings, we compared the GP predictions against the actual labels and counted the proportion of times this was correct (hit rate).

Table 7.3 reports the best achieved hit rate for each test signal in all runs, as well the average hit rate for each signal in all runs (18 GP runs). Also, the worst hit rates are presented to show the algorithm performance in its worst case. Moreover, the corresponding standard deviations are presented to illustrate the system's reliability. The last row of table 7.3 reports the approximate average processing time for both training and testing.

Table 7.4 summarises the runs' information. We measured the quality of each run by calculating the average hit rate of the best-of-run on all test signals for each run. This reflects the accuracy of the evolved programs when dealing with signals from different participants. More specifically, the first row in table 7.4 shows the average of the runs' qualities (i.e., the average of the averages). In other words, we measure the average-hit of each best-of-run evolved program on all test signals, then we measured the average of average-hits of all 18 evolved programs. The second row in the table shows the average of the best hit-rate for each signal in all runs (i.e., the average of column 3 in table 7.3). On the contrary, the third row shows the average of the worst-hit for each signal in all runs (i.e., the average of column 4 in table 7.3). Finally, the last

TABLE 7.3: Muscle fatigue prediction system - Summary of performance of 18 different GP runs.

Signal/ results	*Average Hit*	*Best Hit*	*Worst Hit*	*Standard Dev.*
A1	60.36%	77.27%	37.84%	8.99
A2	57.54%	77.36%	38.89%	9.03
A3	60.69%	85.71%	39.39%	11.24
A4	62.40%	100%	25.00%	19.31
A5	57.96%	81.82%	9.09%	17.92
B1	56.62%	90%	39.29%	13.1
B2	62.73%	82.35%	41.18%	12.98
B3	41.67%	100%	0.00%	28.58
B4	65.08%	100%	0.00%	23.78
B5	58.24%	100%	25.00%	20
C1	67.08%	100%	42.31%	14.73
C2	62.28%	78.79%	43.59%	8.27
C3	66.05%	85.71%	40.00%	11.76
C4	56.75%	100%	0.00%	16.56
C5	65.34%	100%	16.56%	42.84
Average Testing Time			*1 min/test signal*	
Average Training Time			*18 Hours*	

row shows the standard deviation of the best-of-run in all experiments. The low standard deviation in conjunction with the reasonable average prediction accuracy indicates that our system is likely to produce accurate models within a few runs.

TABLE 7.4: Muscle fatigue prediction system - Summary of 18 runs.

Average Hits	60.05%
Best-Hit Average	90.60%
Worst-Hit Average	27.94%
Standard Deviation	7.87

As we mentioned previously, our aim is to obtain a generic predictor that performs well on average on all signals in the test set. Table 7.5 reports the performance of the best overall evolved predictor. This is the programs that achieved the highest average hit-rate on all test signals. This is the program that the user will use to predict the fatigue onset after running the system multiple times. Also, the performance of the worst overall is reported. It should be noted that the achieved results are promising, especially considering that the system is predicting muscle fatigue for three different individuals.

TABLE 7.5: Muscle fatigue prediction system - Best GP test run vs. worst GP test run.

Signal/ results	*Best run Overall*	*Worst run Overall*
A1	66.67%	37.84%
A2	66.66%	53.12%
A3	75.47%	48.91%
A4	25%	50%
A5	66.67%	40%
B1	90%	40.74%
B2	81.25%	65.52%
B3	50%	33.33%
B4	100%	50%
B5	100%	42.86%
C1	100%	42.31%
C2	78.79%	43.84%
C3	75%	56.25%
C4	80%	69.23%
C5	100%	42.86%
Average	77.03%	47.79%

To illustrate the results, figure 7.3 shows a visualisation of the system's performance in one of the test signals (B4). The figure illustrates the difference between the actual muscle's status and the prediction of our system. The actual muscle's status was measured by counting the number of zero crossing offline (see section 7.1.5) and has been compared against the GP's performance. The figure highlights the intervals where the system failed to correctly classify the signal.

7.1.10 Muscle Fatigue Prediction Summary and Future Work

An ideal system would be one that non-invasively reads data from an athlete's muscles and then alerts before the onset of fatigue. Here, we have proposed a system based on GP-zip2's learning technique, which is perhaps one step closer to this ideal fatigue prediction system.

There were two main motivations behind this application: firstly, to study the limitations and the capabilities of using GP-zip2's learning model for the mentioned domain; and, secondly, to allow real time processing of the EMG signal and provide an early warning before the onset of fatigue. We feel that our GP-zip2 based approach has also made steps in the direction of satisfying these objectives, as shown by the illustrated experimentation.

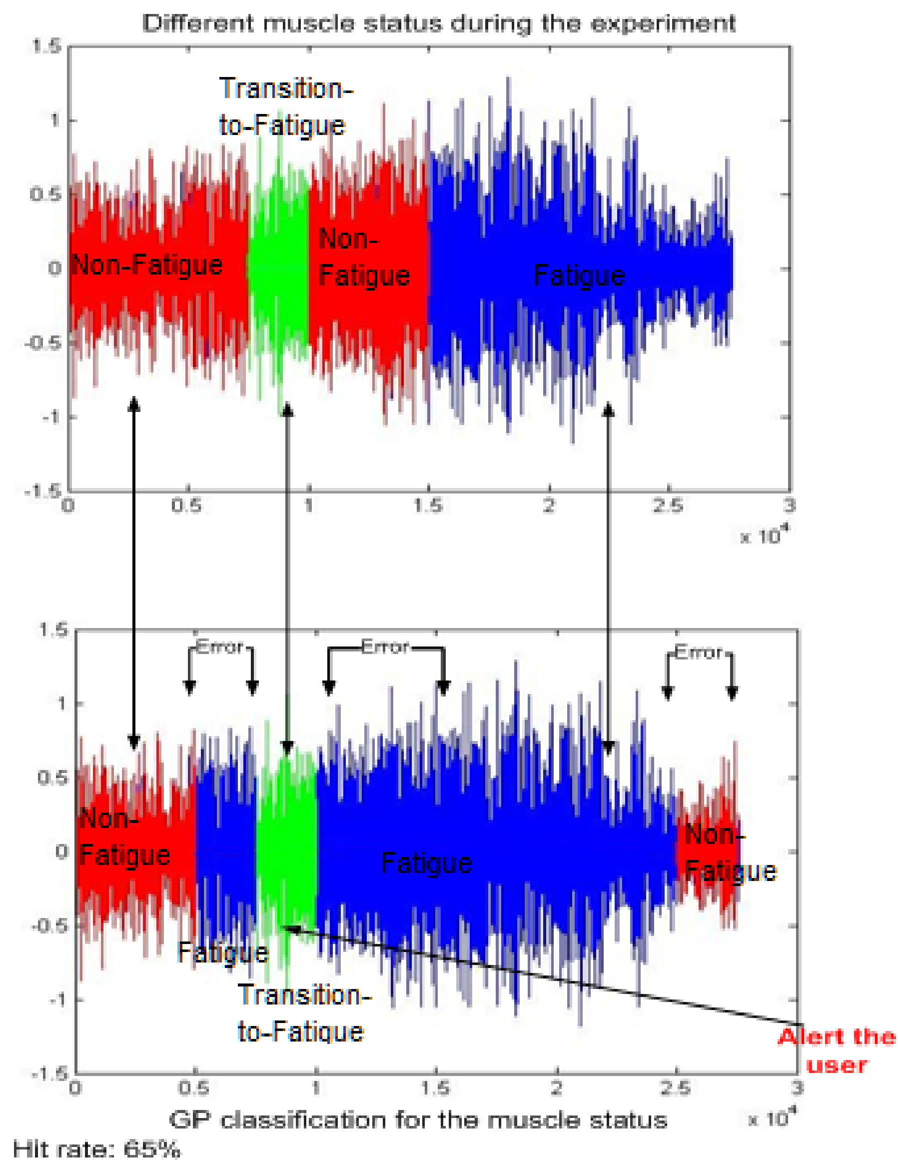

FIGURE 7.3: Visualised illustration of GP performance in one of the runs for test signal B4. GP prediction (bottom) is compared against actual muscle's status (top). Fatigue = blue, Non-Fatigue = red, Transition-to-Fatigue = green.

Our results are encouraging, in the sense that a good prediction has been achieved and further significant improvements could be obtained. Therefore, this approach shows potential for applications in several domains, such as, ergonomics, myopathy and physiotherapy.

Although, this approach has achieved good prediction rates (100% in some cases), it suffers from a major disadvantage. The proposed labelling mechanism — being based on zero crossings — is unreliable (as described in section 7.1.5). This is due to the fact that the number of zero

crossings is affected by noise. Thus, the labels attributed to the signal's blocks might not be accurate. This has the potential to hamper or prevent learning in the system. We avoided this problem altogether by carefully selecting the least noisy signals for the training set and suitably preprocessing the signals. Nevertheless, there remains the possibility that some errors in the labelling could still occur.

There are many routes which could be pursued in order to further improve the performance of this application. For example, a simple extension in the set of statistical functions available in the primitive set might improve the classification accuracy. Also, the use of a more sophisticated technique to label the EMG blocks (e.g., fuzzy classification) might improve the system's reliability. Finally, a comparative study with other fatigue prediction methods is required to evaluate the significance of the conducted results.

The work presented in this section has been further extended by the co-authors in [110] where some of these directions have been investigated.

7.2 GP-Fileprints: File Types Detection

7.2.1 Introduction

This section presents the second test problem for GP-zip2 learning technique. In this section we use it to detect file types by analysing their binary streams.

From the point of view of an operating system or standard high-level programming languages, a file is normally treated as a sequence of elementary data units, typically bytes. File format is a particular way to encode information for storage in a computer so that the file can be correctly interpreted by the operating system. Unfortunately, there are no universal standards for file types and there are hundreds of file types. This makes file type identification a difficult, but increasingly significant, problem. Operating systems have traditionally used different approaches to solve this problem (i.e., file extensions and magic numbers). This, however, is very unreliable

given that any user or application can easily change the extension of a file or change the file's meta data. A method is required to identify files' contents. This is useful for applications such as email spam filtering, virus detection, forensic analysis and network security.

In [111], McDaniel and Heydari proposed an approach for automatically generating "fingerprints" for files. These fingerprints were then used to recognise the true type of unknown files based on their content instead of using the metadata associated with them. The authors used three algorithms to build these fingerprints: Byte Frequency Analysis (BFA), Byte Frequency Cross-Correlation (BFC) and the File Header/Trailer (FHT) algorithm. The BFA algorithm works by drawing a frequency distribution of the number of occurrences with which each byte value occurs in a file. This frequency distribution is useful in determining the type of a file because many file types have consistent patterns in their frequency distribution. The BFA algorithm presents some limitations related to the fact that it compares overall byte-frequency distributions. This issue is addressed by the BFC algorithm by taking the relationship between byte value frequencies into account. In BFC, two values are calculated: the average difference in frequency between all byte pairs and correlation strength. Finally, the FHT algorithm works by using file headers and file trailers. These are patterns of bytes that appear in a fixed location in the file: at the beginning and the end of the file. The authors tested the described algorithms and constructed thirty file-type fingerprints using four test files for each file type (i.e., a total library of 120 files). They reported that BFA and BFC showed poor performance (i.e., an accuracy in the range of 27.5% and 45.83%) compared to FHT algorithm (which had an accuracy of 95.83%).

Later, this work has been criticised by Li *et al.* [112], who claimed that a single fingerprint is insufficient to represent whole classes of files. Li *et al*. proposed to analyse the data using n-grams to identify multiple centroids – fingerprints – for each file type. They applied three different techniques: *i) Truncation*, where part of the file header is analysed and compared with single representative fingerprints; *ii) Multi-centroids*, where a group of fingerprints is used to form clusters with K-means, each cluster representing a particular file type, then unseen data are

classified according to minimal distance; *iii) Exemplar files*, where unseen data are compared to all fingerprints from all trained data types and classified based on the closest fingerprint. The authors reported some problems when classifying similar data types such as GIF and JPG. Also, some difficulties appeared when classifying PDF and MS office file types, as embedded images and figures mislead the algorithms.

Karresand and Shahmehri [113] proposed a method called Oscar that allows classification of data fragments based on their structures without the need for any other meta data (e.g., header information). For this purpose, they used the Byte Frequency Distribution (BFD) of data fragments and calculated the mean and the standard deviation for each byte value. Together these measures form a model which is used to identify unknown data fragments. In [114], the same authors extended this approach by calculating the Rate of Change (RoC) (i.e. the absolute value of the difference between two consecutive byte values in a data fragment). RoC allows the incorporation of the ordering of the bytes into the identification process. The authors reported that their approach, tested using only JPEG files, gave a 99.2% detection rate.

Hall and Wilbon [115] used a sliding window of fixed size and measured the entropy and the data compressibility with LZW compression to identify file types. For each file type, these measurements were averaged from training examples and the standard deviation calculated. Later, unseen data was compared with these models to predict their contents. The authors reported that entropy was not successful in associating the correct file type with unseen data. Also, this work reported that some file types such as BMP can vary greatly and it is very difficult to correctly classify them based on the proposed method.

Erbacher and Mullholland [116] focused their attention on the location and identification of data types embedded within a file, to offer analysts a technique to more efficiently locate relevant data on a hard drive. For this purpose, the authors used a range of statistical analyses with a variety of file types and were able to discover the types of data embedded within a file. The authors were able to identify five statistics that gave sufficient information to differentiate the different

types of data: average, standard deviation, kurtosis, distribution of averages and distribution of standard deviations.

No prior work has used evolutionary algorithms, including GP, to solve the problem of identifying file types from their raw binary streams.

7.2.2 Applying GP-zip2 Model

This section proposes an application based on the use of GP-zip2's learning technique to identify the file contents by analysing the raw binary streams. The question that we investigated is whether it is possible for GP to extract regularities from the raw byte-series of files and reliably correlate them with particular data types without the need for any meta data. For this task, we used the standard implementation of GP-zip2 with the only difference that we added another tree to the individuals' representations to adapt the system to the given problem (details will be presented shortly).

The analysis process, broadly outlined in figure 7.4, works as follows. We try to spot regularities within the raw byte series and to associate them to different file types (e.g., TXT, PDF, JPG). Each class indicates the contents of the data. The system works in two main stages: *i) Training* and *ii) Testing*. The system processes raw byte-series signals and performs three major functions: *i) Segmentation* of the byte-series based on their statistical features, *ii) Fileprints creation* which are a sort of fingerprint of the contents of a file, and *iii) Classification* of the identified fileprints into their types (e.g., TXT, PDF, JPG). For these tasks, GP has been supplied with a language that allows it to extract statistical features from byte-series. Table 7.6 reports the primitive set of the system.

The system starts by randomly initialising a population of individuals using ramped half-and-half. As illustrated in figure 7.5, each individual has a multi-tree representation comprising one

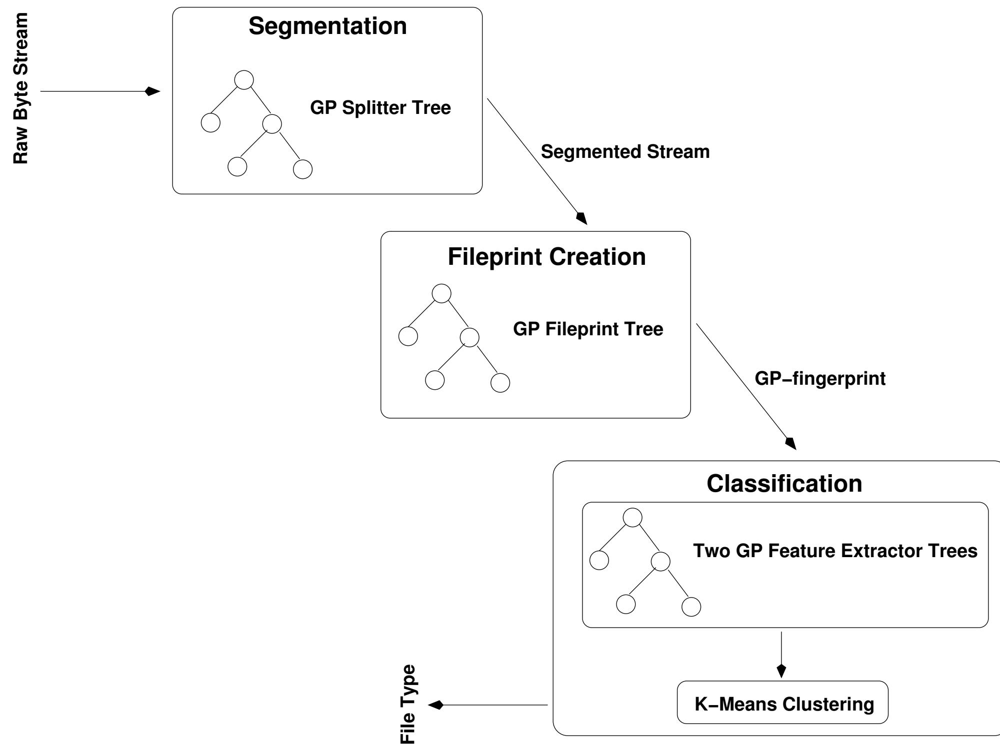

FIGURE 7.4: Outline of the file-type detection process.

TABLE 7.6: File-type detection system - Primitive set.

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

splitter tree, one fileprint tree and two feature-extraction trees. [3] In the next sections we describe the role of each tree in detail.

7.2.3 Splitter Tree

It is very difficult to extract statistical features from the raw byte-series and directly correlate them with a particular data type. Furthermore, over time there has been an increase in the use of files with complex structures that store data of different types simultaneously. For example,

[3]The fileprint tree is an additional component that was not in GP-zip2 design. This extra component is required due to the nature of this particular problem (i.e., file-types identification).

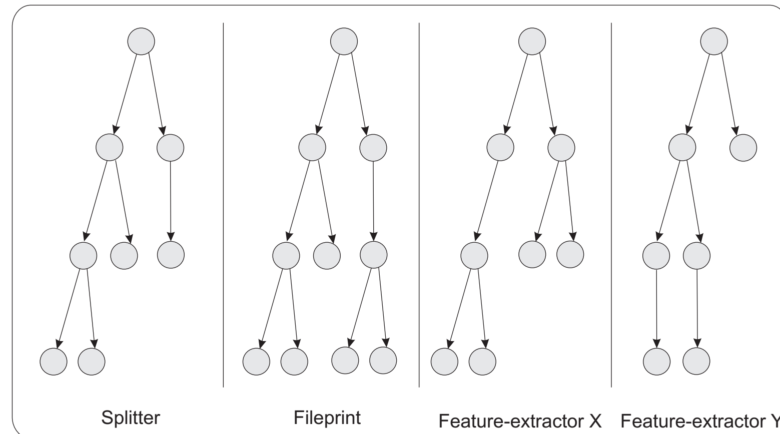

FIGURE 7.5: File-type detection system - Individuals' representation.

a single game file might contain executable code, text, pictures and background music. Also, many file types, e.g., OpenOffice's ODT, Microsoft's DOCX or a ZIP file, are in fact archives containing inhomogeneous data. This makes the task of recognising file types today even more difficult and traditional methods unreliable. It is, therefore, necessary to properly address the fact that files may contain multiple data types. The main job of the splitter trees is to split the given raw byte-series into smaller segments based on their statistical features in such a way that each segment is composed of statistically uniform data.

The implementation of the splitter tree is exactly the same as for GP-zip2. Thus, the size of the sliding window $L = 100$ and the step by which it is moved is of size $S = 50$. In preliminary tests we tried different sizes for L and we found that $L = 100$ has achieved the best results.

7.2.4 Fileprint Tree

Unlike other techniques where files are processed as single units, our approach attempts to divide the file into smaller segments via the splitter tree and understand the type of each segment separately via the *fileprint* tree before making a final determination about the file type. The main job of the fileprint tree is to identify a unique signature for each file. These signatures are meant to be similar for files of the same type and different for files of different types. Hence, the outputs of the fileprint tree are easier to classify into different classes.

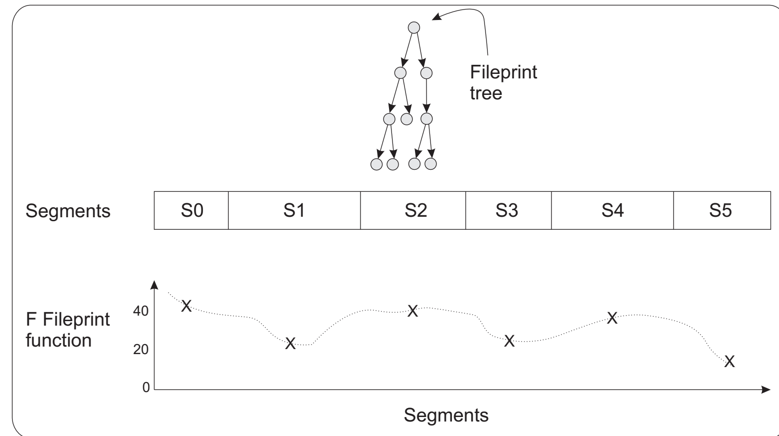

FIGURE 7.6: The fileprint tree processes the segments identified by splitter tree (top). Its output produces a GP-fingerprint for the file (bottom).

As illustrated in figure 7.6, the fileprint tree receives the segments identified by the splitter tree for each file and processes each segment individually. This corresponds to applying a function, $f_{fileprint}(S_i)$, to the data within each segment, S_i. The output of the program is a single number, α, which is an abstract representation of the features of the data within the segments. Thus, the fileprint tree will produce a vector $v = (a_1, a_2, \cdots, a_i)$. Each v contains a series of abstracted numbers that describe the contents of a particular file. Each vector v is referred to as a GP-fingerprint. A collection of vectors, $V = (v_1, v_2, \cdots, v_n)$, is obtained after execution of the fileprint tree with all the files in the training set.

7.2.5 Feature-Extraction Trees

The main job of the feature-extraction trees in our GP representation is to extract features (using the primitives in table 7.6) from the GP-fingerprints identified by the fileprint tree and to project them onto a two-dimensional Euclidian space.

Similar to GP-zip2, K-means is used to find clusters to organise blocks (as represented by their two composite features) into groups. Once the training set is clustered, we can then use the clusters found by K-means to perform classification of unseen data. At the end of the evolution, K-means is able to distinguish file types based on their contents.

7.2.6 Fitness Evaluation

The performance of each individual is evaluated by measuring the classification accuracy of the training examples.

Fitness is evaluated after performing the clustering of the outputs of the feature extraction trees using K-means. Our fitness evaluation is based on the quality of the clustering in terms of cluster homogeneity and cluster separation.

The homogeneity of the clusters is calculated as follows. As exemplified in figure 7.7, we count the members of each cluster, each data point in the cluster representing a GP-fingerprint of one file in the training set. Since with the training data we already know the content type for each GP-fingerprint, we label the clusters according to the dominant data type. The fitness function rates the homogeneity of clusters in terms of the proportion of GP-fingerprints that are labelled as the file type that labels the cluster.[4]

Using the information about the GP-fingerprints and their labels we can easily find the total number of data points that belong to the same data type. Any deviations from this optimal value due to clusters containing extra members is discouraged via a penalty term in the fitness function.

Formally, the clusters homogeneity and penalty term is expressed in equations 5.2 and 5.3, respectively.

The homogeneity of the clusters is not the only measurement of the quality of the classification performed by K-means. As before, we also measure and reward the separation of the clusters. For this task, the modified DBI index defined in equation 5.4, which treated as a penalty value (the lower the DBI the lower penalty applied to the fitness). Thus, the fitness function is as follows:

[4] As usual, the system prevents the labelling of different clusters with the same file type even in the cases where the proportions in two or more clusters are equal.

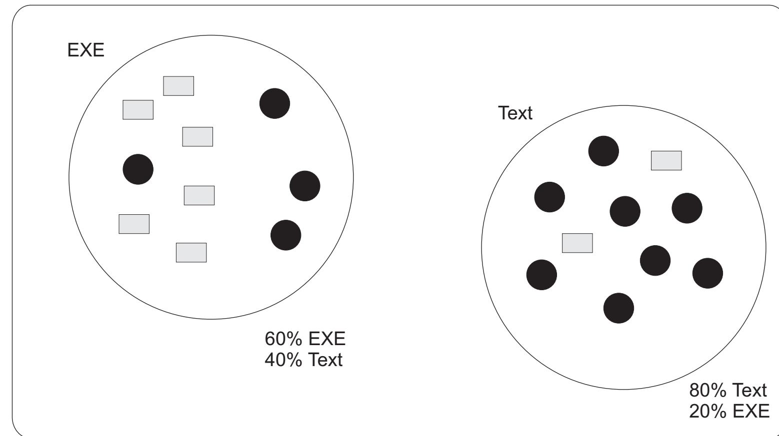

FIGURE 7.7: GP-fileprint - Homogeneity measure of the clusters.

$$Fitness = f_{Homogeneity} - DBI \tag{7.5}$$

A significant advantage with our method is that the approach does not impose any constraint on the shape of the clusters. Once the training set is clustered, we can then use the clusters found by K-means to perform classification of unseen data.

In the testing phase, unseen data go through the three components of the evolved solution: blocks are produced by the splitter tree, the GP-fingerprint is obtained by the fileprint tree, and, finally, GP-fingerprints are projected onto a two-dimensional Euclidean space by the two feature-extraction trees. Then, these are classified based on the majority class labels of their K-nearest neighbours. We use weighted majority voting, where each nearest neighbour is weighted based on its distance from the newly projected data point. More specifically the weight is $w = 1/distance(x_i, z_i)$, where x_i is the nearest neighbour and z_i is the newly projected data point.[5]

7.2.7 Search Operators

The system uses the same search operators are in previous application extended to deal with the extra tree present in this application.

[5]This is different than GP-zip2 in data compression where unseen data are classified based on the closest centriod. In this particular application, tests showed that weighted majority voting produced a slightly better classification results than using the closest centroids.

TABLE 7.7: GP-fileprint - Parameter settings.

Parameter	*Value*
Population Size	100
Generations	30
One-point Crossover Probability	50%
Mutation Probability	50%
Elitism	20%
Tournament Size	5

Let the ith individual of the population be denoted as I_i and let T_i^c be the cth tree of individual i, where $c \in \{splitter, fileprint, feature_extraction_x, feature_extraction_y\}$. The system selects an operator with a predefined probability for each T_i^c. In the crossover, a restriction is applied so that splitter and fileprint trees can only be crossed over with their equivalent tree type. However, as before, the system is able to freely crossover feature-extractions trees at any position.

7.2.8 Experiments

The results presented in this section were obtained by using the parameter settings illustrated in table 7.7. Evolution halts when 30 generations have elapsed.

Experiments were performed on various file types. Experiments have been divided into four sets. Each set involved 10 independent runs (40 runs in total). In the first set we trained the system to distinguish between two different file types. We increased the number of file types to three in the second set of experiments, to four in the third set, while the last set included five file types. The files types that have been included in the experiments were selected because they are among the most commonly used files types. During the experiments, the algorithm has been trained to distinguish between similar files types (JPG and GIF or TXT and PDF). This allowed us to stress the algorithm and study its ability in distinguishing similar file types.

Several considerations were taken into account when designing the training set. The training set is processed many times by each individual in each generation. Thus, it has to be small enough to avoid over-fitting and yet big enough to contain enough examples to aid the learning process.

TABLE 7.8: GP-fileprint - Training and test sets for the experiments.

		Training set		Test set	
Function	*File types*	*Total size*	*Number of files*	*Total size*	*Number of files*
2 file types	JPG, GIF	618 KB	20	5.44MB	60
3 file types	JPG, GIF, TXT	987 KB	30	7.9MB	90
4 file types	JPG, GIF, TXT, PDF	1.55 MB	40	16 MB	120
5 file types	JPG, GIF, TXT, PDF, EXE	2 MB	50	17.7MB	150

Table 7.8 presents the contents of the training sets for each set of experiments. The training sets included 10 different files of each type.

To assess the system's learning and generalisation we evaluated the accuracy of the evolved programs with a test set. The test set is composed of 30 different files for each type. Table 7.8 presents the contents of the testing cases for each set of experiments. The test sets are completely independent of the training sets. It should be noticed that the size of test set is bigger than the training set. Also, the test set included complex files such as EXE games, and large PDFs that contain figures and charts. This is to allow us to probe the generalisation capabilities of the evolved solutions.

In order to evaluate our approach against other state of the art classification techniques we compared our results with standard Neural Networks [117], Bayes Network [117], and J48 decision trees [117] (a variant of C4.5). For these classification algorithms we used the implementation provided by WEKA [118]. We provided to these algorithms the same training sets and the same primitive sets as our GP system in order to obtain fair comparisons. For each of the Neural Networks and Bayes Network systems, we performed 10 different runs for each data set, [6] as we did for our GP system. J48, being a deterministic algorithm, was only executed once for each data set.

Table 7.9 summarises the results of our experiments comparing GP-zip2's learning model with other techniques. For each non-deterministic algorithm we report the best results obtained within independent 10 runs. GP-zip2's learning appears to outperform the other classification methods considered by a significant margin. We believe that the good classification accuracy of our

[6]Neural Networks and Bayes Network use deterministic learning models but initial networks are random.

TABLE 7.9: GP-fileprint - Test-set performance results.
Numbers in **boldface** represent the best performance achieved.

Method	*2 file types*	*3 file types*	*4 file types*	*5 file types*
Neural Networks	58.33%	66.67%	74.17%	39.33%
Bayes Nettworks	50.00%	66.67%	83.33%	48.67%
J48	51.67%	51.67%	74.17%	48.67%
GP-fingerprint	**85.00%**	**88.90%**	**85.00%**	**70.77%**

approach is largely attributable to the segmentation of the data into smaller parts and obtaining fingerprints.

For all algorithms in table 7.9 we can see that performance increases as the number of file types increases from 2 to 4. One might wonder why this happens: recognising two file types would appear to be an easier task than recognising three or four. To understand this, we need to consider that these results represent test-set performance. The reason why performance is lower in the two-file case than in the three- or four-file cases is over-fitting. As one can see in table 7.8, the fewer the file types in a data set, the smaller the data set. Thus, it is easier for a learning algorithm to over fit training sets with 2 and 3 file types than the set with 4. As a result, the fewer the file types the worse the generalisation performance of the corresponding learning systems. GP, however, approach not to be affected by this problem.

It should be noticed that the lowest performance for all classification algorithms shown in table 7.8 occurred when classifying 5 file types. This is due to the fact that the complexity of the problem increases with the number of classes. However, GP performance degraded less than the other approaches, indicating that our system may be more robust.

The disadvantage of our approach, however, is training time: our GP-zip2 technique entails a learning process of several hours, while other classification techniques only consume a few seconds for the whole learning process. On the other hand, it has to be pointed out that the solutions evolved by our system take only a few seconds to predict the files contents. Consequently, they are not only accurate, but also entirely practical.

Although in table 7.8 we reported the best performance for all the systems (including GP-zip2's learning model) in fact our system is remarkably reliable across runs. Table 7.10 summarises the results obtained in all 40 runs. We measured the quality of each experiment set (four sets, each set included 10 runs) by calculating the following statistics: the average classification accuracy across the test files, the corresponding standard deviation, and the best and worst classification accuracies in all runs. Note that the average worst performance of our GP system is better than the average best performance of the other systems in table 7.8. The low standard deviation and the reasonably high average performance of our system suggest users are that likely to obtain accurate file-type prediction models within few GP runs.

TABLE 7.10: GP-fileprint - Summary of results of 40 runs.

Average of Averages	76.18%
Average of Best run	82.38%
Average of Worst run	68.72%
Standard Deviation	4.85

7.2.9 File Types Detection Summary and Future Work

In this application we have proposed a system based on GP-zip2 to evolve programs that can identify file contents without making use of any meta data. This is a novel application of GP. The classification accuracies obtained by GP were far superior to those obtained by a number of classical algorithms from WEKA [118], namely artificial neural networks, Bayesian networks and J48 decision trees. While evolution is relatively slow with the large training sets required by this application, the resulting programs are entirely practical, being able to process tens of megabytes of data in seconds. In the future we intend to extend the work to larger and more varied data sets and, hopefully, to turn the best solutions evolved by GP in public-domain stand-alone programs. These could perhaps be integrated in spam filters and anti-virus software. Also, one idea to improve the system is to allow the system to say "I don't know" when dealing with new data types beyond the scope of its training set. Among other things this would allow the user to train several instances of the system with different data types and use the evolved programs sequentially. If a prediction model fails to identify a file, it passes it to the next model.

7.3 Summary

In this chapter we studied the benefits and potential of the learning strategy employed by GP-zip2 with problem domains other than data compression. The main aim of the work presented in this chapter is to test the generality of GP-zip2 as a pattern recognition model. GP-zip2 receives a stream of data as input, and returns different patterns within the data categorised into different classes as output (the number of classes and their labels are predefined by the user). GP-zip2 treats the input data as digital signals. An evolved splitter tree is used to analyse the data and split them into different fragments based on their statistical features. Two feature-extraction trees are used to project the data in a reduced dimensionality space from the original space. These three trees are co-evolved together and act as one program to solve the given problem in collaboration.

Two problems were selected to validate the generality of GP-zip2's learning model. Firstly, we presented a successful application of the model in analysing human muscles EMG signals to predict fatigue onset. The results of the experiments with this problem were encouraging in the sense that a good prediction has been achieved and further significant improvements could be obtained. The experiments, also, revealed a few limitations of the current approach (see section 7.1.10). However, overall, the approach is practical and capable of processing EMG signals in real time and produces an early warning before the onset of fatigue. Therefore, this approach shows potential application domains such as ergonomics, sports physiology, and physiotherapy.

Secondly, we used the model for identification of file types via the analysis of raw binary streams (i.e., without the use of meta data). In this particular application, an extra component has been added to the individuals' representation (the file-print tree). Experimentation showed that the identification accuracy obtained by the GP-zip2 model was far superior to those obtained by a number of standard algorithms, while being able to process tens of megabytes of data in seconds.

The problems mentioned above were selected for several reasons. Firstly, the selected problems share similar characteristics as in data compression. Both problems contain continuous streams of data and the user needs to classify them into different classes. Secondly, both problems are non-trivial and of practical importance.

GP-zip2 technique showed its ability to successfully deal with other domains, in addition to data compression. Thanks to the used multi-tree representation which divides a complex problem into a collection of simpler tasks to be solved individually in order to solve the whole problem in collaboration. The results of the two test problems mentioned above indicate that GP-zip2 technique can be applied to a wider range of problems. Also, the technique used can be easily customised to fit a particular problem.

Chapter 8

Conclusions and Future Work

This chapter presents some conclusions on the work presented in this thesis as well as some ideas for future research directions where this work can be extended.

8.1 Conclusions

We will start by reviewing the main motivations and objectives of this thesis. Then, we will summarise of the basic idea behind the work reported in thesis and provide a summary of its principal contributions.

8.1.1 Motivations and Objectives

Data compression is one of many technologies that has enabled the information revolution. Evidently there is an explosive growth in the information that need to be transmitted or stored. There is a stable improvement of storage and transmission technologies in response to this (e.g., Blue Ray CDs and optical fibre networks). However, the development of these technologies would need to be twice fast as to match the information growth [1].[1]

[1] Parkinson's law states that data expands to fill the space available for storage.

The original motivation for this thesis was the realisation that the development of data compression algorithms capable to deal with heterogeneous data has significantly slowed down in the last few years. Furthermore, there is relatively little research on using Computational Intelligence paradigms to develop reliable universal compression systems.

The main objective of this thesis was to develop an intelligent system that is able to automatically generate lossless compression algorithms based on the given encoding situation. We used GP to learn the structure of the given data and adapt the system's behaviour in order to provide the best possible compression level.

To achieve this, we expected adjustments to the standard form of GP would be required for the purpose of improving evolution speed and reliability in our application domain. As a by-product, we hoped that these adjustments would result in a system that could be applied to other applications in comparable domains.

GP was selected for various reasons. Firstly, its application in the data compression domain is relatively unexplored. Also, the flexibility and expressiveness of the GP representation allows the creation of perhaps more complex programs than many other machine learning techniques. This was considered an important requirement in relation to our desire to develop intelligent models that could provide universal data compression. Finally, the reported successes achieved by GP in solving other complex problems in several domains was an additional reason to explore its capabilities in the data compression domain.

8.1.2 Contributions of the Thesis

8.1.2.1 The GP-zip Family

Available evidence shows that there is no effective algorithm that can compress all data types [3]. 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.

In this thesis we started to make some progress on turning this idea into practice. In particular, we proposed a series of intelligent universal compression systems (the GP-zip family) the main idea of which is to evolve programs that attempt to identify what is the best way of applying different compression algorithms to different parts of a data file so as to best match the nature of such data. We presented four members of the GP-zip family, namely, *GP-zip, GP-zip*, GP-zip2* and *GP-zip3*. Each new version addresses the limitations of previous systems and improves upon them.

With GP-zip we wanted to understand the benefits and limitation of combining existing lossless compression algorithms in a way that ensures the best possible match between the algorithm being used and the type of data it is applied to. To explore this idea we implemented a simple decision making mechanism. GP-zip uses a linear GP representation for its individuals. The system divides the given data file into segments of a certain length and asks GP to identify the best possible compression technique for each segment. GP-zip was executed multiple times until it could find the best size for the segments. The function set of GP-zip is composed of primitives that are themselves basic compression and transformation algorithms. These algorithms are treated as black boxes that receive input and return output, although some algorithms may use another learning mechanisms and/or multiple layer of compressions themselves (see section 4.1).

In GP-zip*, the second member of the family, we proposed 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. GP-zip*, also, uses a linear GP representation for its individuals. The system evolves the length of the blocks within each run, rather than use the staged evolutionary search, possibly involving many GP runs, of GP-zip. Each individual is divided onto blocks of different sizes to each of which a compression and/or transformation algorithm

is allocated. The system changes the blocks' sizes and their allocated algorithms via new intelligent crossover and mutation operators which targeted hotspots in the selected individuals. The intelligence of these operators comes from the fact that, instead of acting on random blocks in the parents, they select blocks with the lowest compression ratio, which arguably are the most promising places where variations can be beneficial. This approach was found to achieve higher compression ratios and to require less computational effort than its predecessor.

Both GP-zip and GP-zip* outperformed state-of-the-art compression systems in terms of achieved compression ratios. However, the compression process is very slow and impractical for large files. This is a common problem for compression systems based on CI paradigm (see section 3.2). With GP-zip2, we aimed at addressing this particular weakness. We re-designed the whole decision making process. GP-zip2 uses a tree-like representation for its individuals. The system treats the data to be compressed as digital signals represented using 8-bits per sample. GP evolves programs with multiple components. One component analyses statistical features extracted from the raw byte series and divides the data to be compressed 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 into clusters. Each cluster is labelled with the optimal compression algorithm for its member blocks. After evolution, evolved programs can be used to compress unseen data. Experimental results showed that GP-zip2 compares very well with established compression algorithms in terms of the compression ratios achieved. Also, once compression programs are evolved, they can be used over and over again to perform compression. This compression process is much faster than other evolutionary compression techniques. These features make GP-zip2 appealing for practical use (see section 5.1).

The major disadvantage of GP-zip2 that it requires on average 6.4 hours to evolve a practical solution. This is largely due to the costly fitness evaluation adopted in GP-zip2 which requires compressing data fragments using multiple compression algorithms. In GP-zip3, we tried to solve this particular problem in the system by using a novel fitness evaluation strategy. More

specifically, GP-zip3 evolves and then uses decision trees to predict the performance of GP individuals without requiring them to be used to compress the training data. As shown in a variety of experiments, this speeds up evolution in GP-zip3 considerably over GP-zip2 while achieving similar compression results, thereby significantly broadening the scope of application of the approach. The major disadvantage of GP-zip3 is that its fitness evaluation depends on evolved estimation functions. Thus, the user must spend extra time evolving accurate estimation functions for each of the compression algorithms available to GP-zip3. This, however, needs to be done only once and then it can be reused many times. (see section 6.2).

In summary, in this research we have produced systems that outperform most other compression algorithms. They come top of all compression algorithms tested on heterogeneous files and are never too far behind the best with other types of data. Note that all of the algorithms against which we have compared GP-zip2 are human-designed and they are among the best compression algorithms ever developed. These algorithms are difficult to derive and are the result of many person-years of effort (e.g., LZW has been developed since 1977). Furthermore, some of them have been covered or are still covered by patents. This suggests that *GP-zip family evolved solutions are human-competitive* based on Koza's 8 criteria for human-competitiveness [10, 96]. In addition to providing better compression, dividing the data into blocks and compressing them individually has other advantages. For example, 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 is much faster than the compression, and could be parallelised by using different CPU cores to decompress separate data blocks.

8.1.2.2 Generalisation of GP-zip2 Beyond Data Compression

GP-zip2 uses a special learning technique to analyse and understand the given data (treated as digital signals) where the main problem is divided into smaller sub-problems to be solved in cooperation. After the successful experimentation with this technique in the compression of

heterogeneous archives we decided to test the benefits and potentials of the learning strategy employed by GP-zip2 beyond the domain of data compression.

We considered two problems. In the first problem, we applied the technique to analyse human EMG signals to predict muscle fatigue onset. In this application, we used the standard implementation of GP-zip2 without any modification. Results of this approach were quite promising (see section 7.1). In the second problem, we considered a novel application of GP where, we used GP-zip2's model for identification of file types via the analysis of raw binary streams (i.e., without the use of meta data). In this particular application we added another tree to the individuals' representation to adapt the system to the given problem. Also, we guided the search using a single objective-function (i.e., the final classification accuracy of individuals). This is different than the original implementation of GP-zip2 where the search is guided using two objective-functions (i.e., the splitter tree fitness and feature-extraction fitness). Experimentation with this application showed that the identification accuracy obtained by the GP-zip2 model was far superior to those obtained by a number of standard algorithms (see section 7.2).

We feel that these contributions meet the original objectives stated in this thesis. Hopefully, this thesis will inspire other researchers working in the same area or along similar lines.

8.2 Future Work

Many interesting questions were raised in the thesis, and have been answered by the discussions and the experiments that were reported. Some of these discussions have raised further challenging question which would be interesting to answer in future research.

This research can be extended in many different ways, including the following:

- In various systems (e.g., GP-zip2) we analysed the data in a file using statistical descriptors, such as the mean and the standard deviation, which were computed using one particular interpretation of the data: as a sequence of bytes. It would be important to explore

the benefits and drawbacks of using a different interpretation of the data (e.g., as short integers with or without sign) and/or using multiple interpretations at the same time.

- In the thesis we used K-means to cluster data blocks. This was an effective technique, but many more sophisticated classification techniques exist which in principle could provide further improvements to the system's performance. Their use should be explored.

- In the experiments with the systems in the GP-zip family we always used a very small number of compression algorithms as building blocks for the systems. However, it is reasonable to expect that by providing a larger number of compression models to GP-zip, further improvements in performance might be obtained.

- The experiments have revealed that the size of the sliding window has significant influence on system performance. It is likely that the optimal size will depend on the particular domain of application (as expressed by the training set). It might make sense to ask GP to optimise also this parameter as to best match the nature of the given data.

- The learning strategy in GP-zip2 has worked well with other domains, in addition to data compression. Thus, applying and extending the model to make it possible to tackle other problems would also be an interesting future research direction.

- Also, currently, each compression model the system can use is treated as a black box. A promising extension for this research would be to decompose each model into more elementary entities and allow the system to use compositions of multiple such elements. Such a lower-level language could express both classical and novel compression algorithms.

- In the future the use of a form of hyper-GP-zip should also be investigated. This would choose the best program from a pool of solutions for each unseen archive, based on statistical features from the data.

- Treating each component of the fitness function as a separate objective, rather than adding them up, might also provide significant benefits.

- 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.

We hope to explore some of these avenues in future research.

Bibliography

[1] Khalid Sayood, *Introduction to Data Compression*, Morgan Kaufmann Publishers, San Francisco, CA, USA, second edition, 2000.

[2] David Salomon, *Data Compression: The Complete Reference*, Second edition, 2004.

[3] Ida Mengyi Pu, *Fundamental Data Compression*, Butterworth-Heinemann, Newton, MA, USA, 2005.

[4] Leandro Nunes de Castro, *Fundamentals of natural computing: basic concepts, algorithms, and applications*, CRC Press, 2006.

[5] Zbigniew Michalewicz and David B. Fogel, *How to solve it: modern heuristics*, Springer-Verlag New York Inc, 2004.

[6] Sushil John Louis, *Genetic algorithms as a computational tool for design*, PhD thesis, Bloomington, IN, USA, 1993.

[7] James Kennedy, Russell C Eberhart, and Yuhui Shi, *Swarm intelligence*, Morgan Kaufmann, 1 edition, 2001.

[8] Nils J. Nilsson, *Artificial intelligence: a new synthesis*, Morgan Kaufmann, 1998.

[9] Andries Petrus Engelbrecht, "Computational intelligence: An introduction.", *J. Artificial Societies and Social Simulation*, vol. 7, no. 1, 2004.

[10] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee, *A Field Guide to Genetic Programming*, Published via `http://lulu.com`, 2008, (With contributions by J. R. Koza).

[11] Marcos I. Quintana, Riccardo Poli, and Ela Claridge, "On two approaches to image processing algorithm design for binary images using GP", in *Applications of Evolutionary Computing, EvoWorkshops2003: EvoBIO, EvoCOP, EvoIASP, EvoMUSART, EvoROB, EvoSTIM*, Günther R. Raidl, Stefano Cagnoni, Juan Jesús Romero Cardalda, David W. Corne, Jens Gottlieb, Agnès Guillot, Emma Hart, Colin G. Johnson, Elena Marchiori, Jean-Arcady Meyer, and Martin Middendorf, Eds., University of Essex, England, UK, 14-16 April 2003, vol. 2611 of *LNCS*, pp. 422–431, Springer-Verlag.

[12] Astro Tellet, *Algorithm evolution with internal reinforcement for signal understanding*, PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

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

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

[15] Pat Langley, *Elements of Machine Learning*, Morgan Kaufmann, San Francisco, 1996.

[16] Wolfgang Banzhaf, Peter Nordin, Robert E. Keller, and Frank D. Francone, *Genetic Programming – An Introduction; On the Automatic Evolution of Computer Programs and its Applications*, Morgan Kaufmann, San Francisco, CA, USA, January 1998.

[17] Jeroen Eggermont, J. N. Kok, and W. A. Kosters, "Genetic programming for data classification: Refining the search space", in *Proceedings of the Fivteenth Belgium/Netherlands Conference on Artificial Intelligence (BNAIC'03)*, T. Heskes, P. Lucas, L. Vuurpijl, and W. Wiegerinck, Eds., Nijmegen, The Netherlands, 23-24 October 2003, pp. 123–130.

[18] Hammad Majeed and Conor Ryan, "A less destructive, context-aware crossover operator for GP", in *Proceedings of the 9th European Conference on Genetic Programming*, Pierre Collet, Marco Tomassini, Marc Ebner, Steven Gustafson, and Anikó Ekárt, Eds., Budapest, Hungary, 10 - 12 April 2006, vol. 3905 of *Lecture Notes in Computer Science*, pp. 36–48, Springer.

[19] Hammad Majeed and Conor Ryan, "Using context-aware crossover to improve the performance of GP", in *GECCO*, Mike Cattolico, Ed. 2006, pp. 847–854, ACM.

[20] Peter Nordin, Frank Francone, and Wolfgang Banzhaf, "Explicitly defined introns and destructive crossover in genetic programming", in *Proceedings of the Workshop on Genetic Programming: From Theory to Real-World Applications*, Justinian P. Rosca, Ed., Tahoe City, California, USA, 9 July 1995, pp. 6–22.

[21] Wolfgang Banzhaf Markus Brameier, *Linear genetic programming*, Springer-Verlag New York Inc, 2007.

[22] Edgar Galvan-Lopez, *An Analysis of the Effects of Neutrality on Problem Hardness for Evolutionary Algorithms*, PhD thesis, University of Essex, 2009.

[23] Julian F. Miller and Stephen L. Smith, "Redundancy and computational efficiency in cartesian genetic programming.", *IEEE Trans. Evolutionary Computation*, vol. 10, no. 2, pp. 167–174, 2006.

[24] Simon. Ronald, John Asenstorfer, and Millist Vincent, "Representational redundancy in evolutionary algorithms", in *Proceedings of the 1995 IEEE International Conference on Evolutionary Computating*, 1995, vol. 2, pp. 631–637.

[25] Ahmed Kattan, "Universal lossless data compression with built in encryption", Master's thesis, School of Computer Science and Electronic Engineering, University of Essex, 2006.

[26] Mark Nelson, *The Data Compression Book*, M & T Books, 1991.

[27] David H. Wolpert and William G. Macready, "No free lunch theorems for optimization", *IEEE Transactions on Evolutionary Computation*, vol. 1, no. 1, pp. 67–82, April 1997.

[28] Ross Arnold and Tim Bell, "A corpus for the evaluation of lossless compression algorithms", in *DCC '97: Proceedings of the Conference on Data Compression*, Washington, DC, USA, 1997, p. 201, IEEE Computer Society.

[29] William H. Hsu and Amy E. Zwarico, "Automatic synthesis of compression techniques for heterogeneous files", *Softw. Pract. Exper.*, vol. 25, no. 10, pp. 1097–1116, 1995.

[30] Gordon V. Cormack and R. Nigel Horspool, "Data compression using dynamic markov modelling.", *Comput. J.*, vol. 30, no. 6, pp. 541–550, 1987.

[31] Jeffrey Scott Vitter, "Design and analysis of dynamic huffman codes.", *J. ACM*, vol. 34, no. 4, pp. 825–845, 1987.

[32] John G. Cleary, W. J. Teahan, and Ian H. Witten, "Unbounded length contexts for PPM", in *Data Compression Conference*, 1995, pp. 52–61.

[33] Jacob Ziv and Abraham Lempel, "A universal algorithm for sequential data compression", *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[34] Julian Seward, "Bzip2", Website, 2009, Nov, http://www.bzip.org/.

[35] WinZip, "The compression utility for windows", Website, 2009, Nov, http://www.winzip.com/.

[36] Peter Deutsch, "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951 (Informational), May 1996.

[37] Robert D. Dony and Simon Haykin, "Neural network approaches to image compression", *Proceedings of the IEEE*, vol. 83, no. 2, pp. 288–303, 1995.

[38] S. Anna Durai. and E. Anna Saro, "Image Compression with Back-Propagation Neural Network using Cumulative Distribution Function", *International Journal of Applied Science, Engineering and Technology*, vol. 3, no. 4, pp. 185–189, 2006.

[39] B. Verma, M. Blumenstein, and S. Kulkarni, "A Neural Network based Technique for Data Compression", in *Proceedings of the IASTED International Conference on Modelling and Simulation, MSO*. IASTED, vol. 97, pp. 12–16.

[40] Mohamad Hassoun and Agus Sudjianto, "Compression net-free autoencoders", in *Workshop on Advances in Autoencoder/Autoassociator-Based Computations at the NIPS*, 1997, vol. 97.

[41] Aran Namphol, Steven H. Chin, and Mohammed Arozullah, "Image compression with a hierarchical neural network", *IEEE Transactions on Aerospace and Electronic Systems*, vol. 32, no. 1, pp. 326–338, 1996.

[42] J. Jiang, "Image compression with neural networks- a survey", *Signal Processing: Image Communication*, vol. 14, no. 9, pp. 737–760, 1999.

[43] Stanley C. Ahalt, Ashok K. Krishnamurthy, Prakoon Chen, and Douglas E. Melton, "Competitive learning algorithms for vector quantization.", *Neural Networks*, vol. 3, no. 3, pp. 277–290, 1990.

[44] Syed A. Rizvi and Nasser M. Nasrabadi, "Finite-state residual vector quantization using a tree-structured competitive neural network", *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 7, no. 2, pp. 377–390, 1997.

[45] Fu-Lai Chung and Tong Lee, "Fuzzy competitive learning.", *Neural Networks*, vol. 7, no. 3, pp. 539–551, 1994.

[46] Nader Mohsenian, Syed A. Rizvi, , and Nasser M. Nasrabadi, “Predictive vector quantization using a neural network approach (Journal Paper)”, *Optical Engineering*, vol. 32, no. 07, pp. 1503–1513.

[47] Nikolaos A. Laskaris and Spiros Fotopoulos, “A novel training scheme for neural-network-based vector quantizers and its application in image compression.”, *Neurocomputing*, vol. 61, pp. 421–427, 2004.

[48] Jrgen Schmidhuber and Stefan Heil, “Predictive coding with neural nets: Application to text compression.”, in *NIPS*, Gerald Tesauro, David S. Touretzky, and Todd K. Leen, Eds. 1994, pp. 1047–1054, MIT Press.

[49] Jrgen Schmidhuber and Stefan Heil, “Sequential neural text compression”, *IEEE Transactions on Neural Networks*, vol. 7, no. 1, pp. 142–146, 1996.

[50] Matthew V. Mahoney, “Fast text compression with neural networks.”, in *FLAIRS Conference*, James N. Etheredge and Bill Z. Manaris, Eds. 2000, pp. 230–234, AAAI Press.

[51] Thomas Krantz, Oscar Lindberg, Gunnar Thorburn, and Peter Nordin, “Programmatic compression of natural video”, in *Late Breaking Papers at the Genetic and Evolutionary Computation Conference (GECCO-2002)*, Erick Cantú-Paz, Ed., New York, NY, July 2002, pp. 301–307, AAAI.

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

[53] Suman K. Mitra, C. A. Murthy, and Malay Kumar Kundu, “Technique for fractal image compression using genetic algorithm.”, *IEEE Transactions on Image Processing*, vol. 7, no. 4, pp. 586–593, 1998.

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

[55] 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.

[56] Anargyros Sarafopoulos, “Automatic generation of affine IFS and strongly typed genetic programming”, in *Genetic Programming, Proceedings of EuroGP'99*, Riccardo Poli, Peter Nordin, William B. Langdon, and Terence C. Fogarty, Eds., Goteborg, Sweden, 26-27 May 1999, vol. 1598 of *LNCS*, pp. 149–160, Springer-Verlag.

[57] Lucia Vences and Isaac Rudomin, “Fractal compression of single images and image sequences using genetic algorithms”, *The Eurographics Association*, 1994.

[58] L. Vences and I. Rudomin, “Genetic algorithms for fractal image and image sequence compression”, *Proceedings Computacion Visual*, pp. 35–44, 1997.

[59] Faraoun Kamel Mohamed and BOUKELIF Aoued, “Optimization of fractal image compression based on genetic algorithms”, in *2nd International Symposium on Communications, Control and Signal Processing*, Marrakesh, Marocco, 2006.

[60] Mehrdad Salami, Masahiro Murakawa, and Tetsuya Higuchi, "Data compression based on evolvable hardware.", in *ICES*, Tetsuya Higuchi, Masaya Iwata, and Weixin Liu, Eds. 1996, vol. 1259 of *Lecture Notes in Computer Science*, pp. 169–179, Springer.

[61] Mehrdad Salami, Masaya Iwata, and Tetusya Higuchi, "Lossless ImageCompression by Evolvable Hardware", in *Proc. Fourth European Conference on Artificial Life, MIT Press, Cambridge, MA*, 1997, pp. 28–31.

[62] Tetsuya Higuchi, Masahiro Murakawa, Isamu Iwata, Masayaand Kajitani, Weixin Liu, and Mehrdad Salami, "Evolvable hardware at function level", in *IEEE International Conference on Evolutionary Computation*, 1997, pp. 187–192.

[63] Lukás Sekanina, "Evolvable hardware as non-linear predictor for image compression", 1999.

[64] Alex Fukunaga and Andre Stechert, "Evolving nonlinear predictive models for lossless image compression with genetic programming", in *Genetic Programming 1998: Proceedings of the Third Annual Conference*, 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, Eds., University of Wisconsin, Madison, Wisconsin, USA, 22-25 July 1998, pp. 95–102, Morgan Kaufmann.

[65] Gktrk oluk and I. Hakki Toroslu, "A genetic algorithm approach for verification of the syllable-based text compression technique", *Journal of Information Science*, vol. 23, no. 5, pp. 365, 1997.

[66] Tomas Kuthan and Jan Lansky, "Genetic algorithms in syllable-based text compression", in *DATESO*, Jaroslav Pokorný, Václav Snásel, and Karel Richta, Eds. 2007, vol. 235 of *CEUR Workshop Proceedings*, CEUR-WS.org.

[67] Wee Keong, Sunghyun Choi, and Chinya Ravishankar, "Lossless and Lossy Data Compression", *Evolutionary algorithms in engineering applications*, pp. 173 – 188, 1997.

[68] Wu Youfeng and Msuricio Breternitz Jr, "Genetic algorithm for microcode compression", Nov 2008, US Patent 7,451,121.

[69] Mohammed Javeed Zaki and M Sayed, "The use of genetic programming for adaptive text compression", *Int. J. Inf. Coding Theory*, vol. 1, no. 1, pp. 88–108, 2009.

[70] Farhad Oroumchian, Ehsan Darrudi, Fattane Taghiyareh, and Neeyaz 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*, New York, NY, USA, 2004, pp. 478–479, ACM.

[71] Martin Burtscher and Paruj Ratanaworabhan, "gFPC: A Self-Tuning Compression Algorithm", *Data Compression Conference,*, vol. 3, no. 4, 2010.

[72] Martin Burtscher and Paruj Ratanaworabhan, "High throughput compression of double-precision floating-point data.", in *DCC*. 2007, pp. 293–302, IEEE Computer Society.

[73] Mauro L. Beretta, Gianni Degli Antoni, and Andrea G. B. Tettamanzi, "Evolutionary synthesis of a fuzzy image compression algorithm", 1996.

[74] Johan Parent and Ann Nowe, "Evolving compression preprocessors with genetic programming", in *GECCO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, 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, Eds., New York, 9-13 July 2002, pp. 861–867, Morgan Kaufmann Publishers.

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

[76] Uli Grasemann and Risto Miikkulainen, "Effective image compression using evolved wavelets.", in *GECCO*, Hans-Georg Beyer and Una-May O'Reilly, Eds. 2005, pp. 1961–1968, ACM.

[77] Wim Sweldens, "The lifting scheme: A custom-design construction of biorthogonal wavelets", *Applied and Computational Harmonic Analysis*, vol. 3, no. 2, pp. 186–200, 1996.

[78] Harald Feiel and Sub Ramakrishnan, "A genetic approach to color image compression.", in *SAC*, 1997, pp. 252–256.

[79] Ahmed Kattan and Riccardo Poli, "Evolutionary lossless compression with GP-ZIP", in *2008 IEEE World Congress on Computational Intelligence*, Jun Wang, Ed., Hong Kong, 1-6 June 2008, IEEE Computational Intelligence Society, IEEE Press.

[80] Ahmed Kattan and Riccardo Poli, "Evolutionary lossless compression with GP-ZIP*", in *GECCO '08: Proceedings of the 10th annual conference on Genetic and evolutionary computation*, Maarten Keijzer, Giuliano Antoniol, Clare Bates Congdon, Kalyanmoy Deb, Benjamin Doerr, Nikolaus Hansen, John H. Holmes, Gregory S. Hornby, Daniel Howard, James Kennedy, Sanjeev Kumar, Fernando G. Lobo, Julian Francis Miller, Jason Moore, Frank Neumann, Martin Pelikan, Jordan Pollack, Kumara Sastry, Kenneth Stanley, Adrian Stoica, El-Ghazali Talbi, and Ingo Wegener, Eds., Atlanta, GA, USA, 12-16 July 2008, pp. 1211–1218, ACM.

[81] I.H. Witten, R.M. Neal, and J.G. Cleary, "Arithmetic coding for data compression", 1987.

[82] A. Lempel, "Compression of individual sequences via variable-rate coding", *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978.

[83] S. W. Golomb, "Run-length encodings.", *IEEE Trans. Inform. Theory*, vol. IT-12, pp. 399–401, 1966.

[84] M. Burrows and DJ Wheeler, "A block-sorting lossless data compression algorithm", *Digital SRC Research Report*, 1994.

[85] Z. Arnavut, "Move-to-front and inversion coding", in *Data Compression Conference, 2000. Proceedings. DCC 2000*, 2000, pp. 193–202.

[86] ACT Archive Compression Test, "Archive compression test", Website, 2007, Dec, `http://compression.ca/act/act-win.html`.

[87] Ahmed Kattan and Riccardo Poli, "Evolutionary synthesis of lossless compression algorithms with GP-zip2", in *Submitted to GPEM*. Springer, 2010.

[88] Thomas Haynes, Sandip Sen, Dale Schoenefeld, and Roger Wainwright, "Evolving a team", in *Working Notes for the AAAI Symposium on Genetic Programming*, E. V. Siegel and J. R. Koza, Eds., MIT, Cambridge, MA, USA, 10–12 November 1995, pp. 23–30, AAAI.

[89] Sean Luke and Lee Spector, "Evolving teamwork and coordination with genetic programming", in *Genetic Programming 1996: Proceedings of the First Annual Conference*, John R. Koza, David E. Goldberg, David B. Fogel, and Rick L. Riolo, Eds., Stanford University, CA, USA, 28–31 July 1996, pp. 150–156, MIT Press.

[90] Markus Brameier and Wolfgang Banzhaf, "Evolving teams of predictors with linear genetic programming", *Genetic Programming and Evolvable Machines*, vol. 2, no. 4, pp. 381–407, December 2001.

[91] Neven Boric and Pablo A. Estevez, "Genetic programming-based clustering using an information theoretic fitness measure", in *2007 IEEE Congress on Evolutionary Computation*, Dipti Srinivasan and Lipo Wang, Eds., Singapore, 25-28 September 2007, IEEE Computational Intelligence Society, pp. 31–38, IEEE Press.

[92] Durga Prasad Muni, Nikhil R Pal, and Jyotirmay Das, "A novel approach to design classifier using genetic programming", *IEEE Transactions on Evolutionary Computation*, vol. 8, no. 2, pp. 183–196, April 2004.

[93] Tom M. Mitchell, *Machine Learning*, McGraw-Hill, New York, 1997.

[94] James C. Bezdek and Nikhil R. Pal, "Some new indexes of cluster validity", *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 28, no. 3, pp. 301–315, 1998.

[95] F. Sepulveda, M. Meckes, and BA Conway, "Cluster separation index suggests usefulness of non-motor EEG channels in detecting wrist movement direction intention", in *Proceedings of the 2004 IEEE Conference on Cybernetics and Intelligent Systems*, Singapore, pp. 943–947, IEEE Press.

[96] John R. Koza, Martin A. Keane, Matthew J. Streeter, William Mydlowec, Jessen Yu, and Guido Lanza, *Genetic Programming IV: Routine Human-Competitive Machine Intelligence*, Kluwer Academic Publishers, 2003.

[97] Ahmed Kattan and Riccardo Poli, "Genetic programming as a predictor of data compression saving", in *Evolution Artificielle, 9th International Conference*, Pierre Collet, Ed., 26-28 October 2009, Lecture Notes in Computer Science, pp. 13–24.

[98] Ahmed Kattan and Riccardo Poli, "Evolutionary synthesis of lossless compression algorithms with GP-zip3", in *2010 IEEE World Congress on Computational Intelligence*, Barcelona, 18-23 July 2010, IEEE Computational Intelligence Society, IEEE Press.

[99] Ahmed Kattan, Mohammed Al-Mulla, Francisco Sepulveda, and Riccardo Poli, "Detecting localised muscle fatigue during isometric contraction using genetic programming", in *International Conference on Evolutionary Computation (ICEC 2009)*, Agostinho Rosa, Ed., Madeira, Portugal, 5-7 October 2009, pp. 292–297.

[100] Ahmed Kattan, Edgar Galvan-Lopez, Riccardo Poli, and Michael O'Neill, "GP-fileprints: File types detection using genetic programming", in *Proceedings of the 13th European Conference on Genetic Programming, EuroGP 2010*, Anna Isabel Esparcia-Alcazar, Aniko Ekart, Sara Silva, Stephen Dignum, and A. Sima Uyar, Eds., Istanbul, 7-9 April 2010, vol. 6021 of *LNCS*, pp. 134–145, Springer.

[101] Richard L. Lieber, *Skeletal muscle structure, function & plasticity: the physiological basis of rehabilitation*, Lippincott Williams & Wilkins, 2002.

[102] Gilles Dubost and Atau Tanaka, “A wireless, network-based biosensor interface for music”, in *Proceedings of International Computer Music Conference (ICMC)*, 2002.

[103] Mirna Atieh, Rafic Youns, Mohamad Khalil, and Herman Akdag, “Classification of the car seats by detecting the muscular fatigue in the EMG signal”, *INTERNATIONAL JOURNAL OF COMPUTATIONAL COGNITION (HTTP://WWW. YANGSKY. COM/YANGIJCC. HTM)*, vol. 3, no. 4, 2005.

[104] Dimitrios Moshou, Ivo Hostens, George Papaioannou, and Herman Ramon, “Dynamic muscle fatigue detection using self-organizing maps.”, *Appl. Soft Comput.*, vol. 5, no. 4, pp. 391–398, 2005.

[105] Jae-Hoon Song, Jin-Woo Jung, and Zeungnam Bien, “Robust emg pattern recognition to muscular fatigue effect for human-machine interaction.”, in *MICAI*, Alexander F. Gelbukh and Carlos A. Reyes Garci’a, Eds. 2006, vol. 4293 of *Lecture Notes in Computer Science*, pp. 1190–1199, Springer.

[106] Gerold Ebenbichlera, Josef Kollmitzera, Michael Quittana, Frank Uhlb, Chris Kirtleya, and Veronika Fialkaa, “EMG fatigue patterns accompanying isometric fatiguing knee-extensions are different in mono-and bi-articular muscles”, *Electroencephalography and Clinical Neurophysiology/Electromyography and Motor Control*, vol. 109, no. 3, pp. 256–262, 1998.

[107] Josef Finsterer, “EMG-interference pattern analysis”, *Journal of Electromyography and Kinesiology*, vol. 11, no. 4, pp. 231–246, 2001.

[108] Anne F Mannion, Beth Connolly, Kherrin Wood, and Patricia Dolan, “The use of surface ENIG power spectral analysis in the evaluation of back muscle function”, *Development*, vol. 34, no. 4, pp. 427–439, 1997.

[109] Gianluca De Luca, “Fundamental concepts in emg signal acquisition”, *Delsys Inc*, 2001.

[110] Mohmaed R. Al-Mulla, Francisco Sepulveda, Martin Colley, and Ahmed Kattan, “Classification of localized muscle fatigue with genetic programming on sEMG during isometric contraction”, in *Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC 2009*, Minneapolis, Minnesota, USA, 2-6 September 2009, pp. 2633–2638.

[111] Mason McDaniel and M. Hossain Heydari, “Content based file type detection algorithms”, in *HICSS ’03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS’03) - Track 9*, Washington, DC, USA, 2003, p. 332.1, IEEE Computer Society.

[112] Wei-Jen Li, Salvatore J. Stolfo, and Benjamin Herzog, “Fileprints: Identifying file types by n-gram analysis”, in *Proceedings of the 2005 IEEE Workshop on Information Assurance*, 2005, pp. 64–71.

[113] Martin Karresand and Nahid Shahmehri, “Oscar – file type identification of binary data in disk clusters and ram pages”, in *Security and Privacy in Dynamic Environments*, pp. 413–424. Springer Boston, 2006.

[114] Martin Karresand and Nahid Shahmehri, “File type identification of data fragments by their binary structure”, in *Proceedings of the 2006 IEEE Workshop on Information Assurance*, NY, 2006, pp. 140–147, IEEE Computer Society.

[115] Gregory A. Hall and Wilbon P. Davis, “Sliding window measurement for file type identification”, Tech. Rep., Computer Forensics and Intrusion Analysis Group, ManTech Security and Mission Assurance, Rexas, 2006.

[116] Robert F Erbacher. and John Mulholland, “Identification and localization of data types within large-scale file systems”, in *SADFE ’07: Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering*, Washington, DC, USA, 2007, pp. 55–70, IEEE Computer Society.

[117] Ian H. Witten and Eibe Frank, *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2005.

[118] University of Waikato, “Weka”, http://www.cs.waikato.ac.nz/ml/weka/, July 2009.