Evolving Compression preprocessors with genetic programming

Johan Parent
Vrije Universiteit Brussel
Pleinlaan 2
Brussels 1050, Belgium
parentjo@vub.ac.be

Ann Nowe
Vrije Universiteit Brussel
Pleinlaan 2
Brussels 1050, Belgium
asnowe@info.vub.ac.be

Abstract

We present a new approach for applying genetic programming to lossless data compression. Unlike programmatic compression the evolved programs are preprocessors. These preprocessors aim at enhancing the compression rate of the given data by transforming it. The entropy based fitness function is both fast and independent of the type of information being processed. The obtained results are encouraging in sense that significant improvements can be achieved. Furthermore the required computation time is much smaller than in the case of programmatic compression, making the presented approach more viable. We used a strongly typed GP kernel. The kernel offers the extra advantage of being able to exploit parallel execution through the island model.

1 Introduction

Compression has been a research topic for many years. In 1940 Claude Shannon already studied what later became information theory. His research has determined the theoretical limits of data compression. Current research focuses mainly on the development of application specific compression algorithms. Generic lossless compression algorithms can however be considered at a stand still. A recent algorithm is the Burrows-Wheeler transform (1993) [1] which is used in the bzip2 compression program.

Record compression rates can be achieved using a good model of the data, e.g. true color images. Here we investigate how a program can be evolved to transform the data so that it matches the model used by a given compression algorithm. The exact transformation is not explicitly known but one can formulate certain conditions the latter should possess (see section 3).

Our attention goes toward lossless compression algorithms. Examples of popular lossless compression programs are gzip [8] and Winzip [2].

This document is structured as follows. In section 2 related research is presented. The problem and the chosen approach are described in section 3. Section 4 details the experimental setup. Sections 5 and 6 present the results and the conclusion respectively.

2 Related work

Evolutionary algorithms have been used in the past for data compression purposes. Two approaches can be distinguished.

Genetic algorithms were used to find parameters for a compression algorithm in order to maximize compression Driesen [4]. Feiel and Ramakrishnan [6] have used genetic algorithms to optimize the compression of color images using vector quantization.

Genetic programming was used for what is called *programmatic compression*. This approach is closely related to algorithmic complexity were one looks for the shortest program that produces the given data [13]. De Falco et al. [5] have used genetic programming for string compression. Fukunaga and Stechert [7] have used genetic programming for lossless compression of gray-scale images. Nordin and Banzhaf [10] achieved lossy programmatic compression of images and sound. Noteworthy is the fact that [10] [7] both used a genocompiler for their experiments. This software eliminates the function call overhead incurred by other systems during the evaluation of the individuals. Luke [9] reports on a relative improvement in speed of 2000 times compared to LISP code and of 100 times compared to interpreted C (like used for this experiment).

3 Preprocessing

Data compression is already a highly specialized domain. Therefore it seems too far fetched to use genetic programming to generate an algorithm that would compress data and do so in a competitive way.

We formulate our objective as follows: instead of aiming for a program that recodes the data we seek a transformation.

This transformation is applied to the data before compressing it. It is used as a preprocessing step in the entire compression process. Of course, after decompressing the data the transformation needs to be reversed in order to obtain the original data (since we focus on lossless compression).

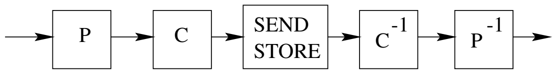

Figure 1: The preprocessing takes place before the actual compression and is reverserd after decompression.

Such a preprocessing program can be formalized as a function P that works on a string S over an alphabet. The length of a string S is denoted as $|S|$. C represents a data compression algorithm. The result we are looking for is a transformation P, so that the condition denoted in equation 1 holds.

$$|C(P(S))| < |C(S)| \tag{1}$$

Stating the problem in these terms makes it easy to fold it into the genetic programming framework since both the program we are looking for, being P, and the fitness function are easily identified. Formulating the problem in this way has some serious disadvantages though. First, computing the result of equation1 is rather expensive. The compression algorithm has to be applied to the transformed data for every individual in the population. Second, the transformation will depend on the compression algorithm used. To avoid this problem we reformulated it using a metric from information theory, the entropy.

[1]Notwithstanding the increase in computation speed [7] [10] report runs lasting several tens of hours on powerful workstations. The results presented here required far less time while working on a bigger amount of data.

3.1 Entropy

Consider a message as a series of symbols. The entropy can be thought of as a measure for the *information content* of a message. The entropy gives the average information content of a symbol [2], this is typically expressed in bits per symbols. The formula for the entropy is given below, note that P_i represents the probability of symbol i in the message.[3]

$$H = -\sum_{n} P_i . \log P_i \tag{2}$$

Using the entropy (equation 2) we have a means to determine how much information is present in a given message. The entropy is the theoretical lower bound on the size of the data (in bits). Any representation of the data with a size lower than the one *predicted* by the entropy loses information. Important is the fact that this measure is independent of the type of data being represented.

3.2 Reducing the entropy

We will evolve a transformation for the given data using the entropy as an objective criterion. The purpose of this transformation is: lowering the entropy of a message (data). The information content can be reduced without loss by exploiting redundancies that might be hidden in the data (as will be shown in section 5.2). By lowering the entropy we reduce the information content, this means that the data can be recoded to occupy less *space*. This property is independent of any compression algorithm. Compression algorithms are designed so as to recode data in order to match the *real* size of the data.

The instruction set of the genetic programming software is designed to reduce the entropy, albeit under the good circumstances. It is up to the evolutionary pressure to bring forth the best transformation. Using the entropy we now can define a new condition for the transformation we wish to evolve.

$$\frac{H_{out} \times L_{out}}{H_{in} \times L_{in}} < 1 \tag{3}$$

Equation 3 is a computationally cheaper fitness function. It is furthermore independent of any data com-

[2]Entropy has a much more rigorous mathematical foundation but the description given here suffices for the purpose of this text.

[3]The model used here is a first order model (marginal probability). Higher order models are based on conditional probabilities.

pression algorithm. Since on the average a symbol represents H bits of information, a message with length L gives $H \times L$ bits of information in total. The formula expresses that the total information of the transformed data has to be lower than the information of the initial data. Note that we impose no limit on the length of the transformed data. Since we do not immediately compress the data, L_{out} can either be greater than or equal to L_{in}.

4 Experimental setup

The setup used for the experiments will now be presented. The transformation one seeks is represented by an S-expression. The function and terminal set used here relies on a simple virtual machine. The use of a virtual machine gives a limited function and terminal set with clear semantics without sacrificing performance or introducing limits on the data size.

The instruction set of the virtual machine has been *wrapped* to form the function and terminal set used by the genetic programming software. To structure the S-expressions strong typing has been used.

4.1 Input data

For the experiments the Canterbury Corpus [3] as well as various bitmaps and word processor files were used. This means that the size of the input tape usually exceeded the order of several kilobytes and grew even up to more than 1 megabyte.

4.2 Parallel and strongly typed

The approach presented here uses a strongly typed genetic programming kernel written in C that produces LISP-like programs. This package is a modified version of the Lil-gp package [12]. This modified version can run on parallel using multi processor machines and clusters of workstations [11].

4.3 Virtual machine

The virtual machine bears some resemblance with an automaton as it uses an input and an output tape. The instruction set can be divided into two categories. Instructions that control the input tape and instructions that process the data read from it. Note that there are no operations that directly influence the output tape in the instruction set. The output tape is manipulated implicitly whenever new data is read from the input tape. This is done through two buffers inside the machine as depicted in the figure 2. The processed data, which is stored in the output buffer, is copied to the output tape when new data is loaded in the input buffer.

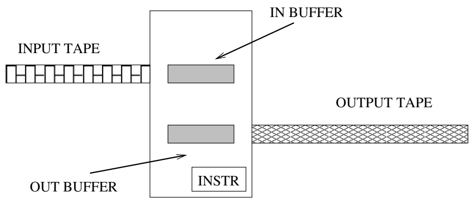

Figure 2: A simple virtual machine has been defined which bears some resemblance with an automaton. The internal buffers are resized to accommodate the data read from the tape.

A status variable is used in order to control the correct operation of the virtual machine. In certain circumstances the execution of an instruction can be illegal. In this case the status of the virtual machine is set accordingly. Whenever an attempt has been made to perform an illegal instruction the execution of subsequent instructions is aborted. The execution of the program is thereby limited to the correct portion of the program.

4.3.1 Tape operations

The machine needs to read data from the input tape. To that end two instructions were defined. Note that the size of the internal buffers is adjusted so that it can contain all the symbols read from tape.

- load : this instruction takes 1 parameter, an integer. This instruction reads the specified number of symbols from the input tape and copies them in the input buffer. This operation is always valid.
- fwd : is identical to load except that it does not take any parameter. Instead it uses the value of the parameter passed to the last *load* instruction. If there was no load instruction preceding the fwd this operation is invalid.

A third rewind instruction makes it possible to apply several operations on data read from the input tape.

- rew : no new data is read or written from or to the tapes. Instead the previously read **and** processed data is processed again. The content of the output buffer is simply copied in the input buffer.

4.3.2 Data operations

Next to operations for reading data, operations were defined for processing the data. The data operations are performed on the data present in the input buffer. Data produced by data operation is stored in the output buffer.

Since we are working with lossless compression each of the data operations needs to be reversible. Each of these operations needs to be reversible in order to be useful within the lossless compression context. Hence, for each data operation there is a reverse operation defined in the virtual machine. These operations are however not made available in the instruction set of the genetic programming software.

The choice of the data operations was guided by their capability to reduce the entropy of the data. This is typical done by exploiting patterns that exist in the data. A stride parameter was introduced for some operations. The stride specifies the step size when going over the data. For example all the symbols 3 positions apart (in this case the stride would be 3). The stride allows data operations to work on patterns that may be spread within the data.

- dpcm : stands for differential pulse code modulation. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_0, x_1 - x_0, x_2 - x_1, x_3 - x_2, \ldots, x_n - x_{n-1}\}$ This operation has a stride parameter.

- min : This operation uses the average of all the symbols (binary representation) in the input buffer. A symbol is replaced by the difference between the average and the symbol. This operation does expand the data by 1 symbol, the symbol which represents the average[4].

  $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_{avg}, x_0 - x_{avg}, x_1 - x_{avg}, x_2 - x_{avg}, x_3 - x_{avg}, \ldots, x_n - x_{avg}\}$ This operation has a stride parameter.

- raw : no transformation is applied at all to the data. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{x_0, x_1, x_2, x_3, \ldots, x_n\}$ This may seem like a quite useless instruction. But it allows for *jumps* in the data processing since we maybe do not need to transform the entire tape. An alternative would be the definition of a *jump* tape instruction.

- pec : pseudo exponential code. This operation produces an output with double size of the input. The output data represents an input symbol as a couple of numbers. The output is based on the number of the input symbol. A couple is a *pseudo* exponent and a *pseudo* remainder. The pseudo exponent is the largest exponent used as a power of 2 which doesn't exceed the input number, e.g. 3 for 10. The remainder is the difference between the power of two and the input number, in the previous example this would be 2 (10 - 2 ^{3}). The reason for calling this *pseudo*[5] is that it has been modified for numbers bigger than 128.

- mtf : move to front. This operation is a standard encoding scheme which uses a map of all the possible symbols. The symbols are replaced by their position in the map. Each time a symbol is replaced by its current position in the map the latter is updated. The symbol is put in front of the map thereby assigning it a small number. This allows to encode the symbols that appeared recently with small numbers.

- inv : inversion. $\{x_0, x_1, x_2, x_3, \ldots, x_n\} \Rightarrow \{MAX(x) - x_0, MAX(x) - x_1, MAX(x) - x_2, MAX(x) - x_3, \ldots, MAX(x) - x_n\}$ Here $MAX(x)$ represents the maximum value that can be represent by the data type used to represent the symbols. This operation has a stride parameter.

- sub : substitution. This operation substitutes the symbol that appears most frequently in the input buffer with the most frequent symbol in the output tape so far. This operation has a stride parameter.

4.4 Types

Typing is used in the first place to structure the programs that can be evolved. Evaluating the functions and terminals corresponds to the execution of an instruction by the virtual machine. For the experiments four types were used for the functions and terminals of the genetic programming software. The first two types are integer types, **int** and **num**. The **num** type represents small integers in the range [0,20]. In order to integrate the instruction of the machine two types were defined for the operations: **tape** and **data**.

The reason for the definition of two separate types for the instruction is the need to combine these instructions to make correct programs. Indeed, the data operations are invalid when no data has been read from the tape. And for the same reason tape operations make no sense if the data is not processed afterwards.

[4]min (stride=1) changes the series 2_5_3_2_5_6_7 to -2_1_-1_1_2_3 + 4 (the average)

[5]pec The symbols 1_123_250 are replaced by (3,4)_(6,59)_(13,58). The last couple allows to represent 250 with two smaller numbers (2^7 + 2^6 +58) where as log $_2$ would give the couple (7,122).

4.5 Function and terminal set

Here we present the function and terminal set provided to the genetic programming system. Apart from the operations provided by the virtual machine a few other functionalities have been introduced.

The programs one can evolve consist of series of tape operations. In order to obtain those series a special function called SEQ has been used. The SEQ function will evaluate its first then its second arguments. Arguments which in turn can be other SEQ functions or real operations on the data.

The ERC_INT and ERC_NUM are used to create ephemeral random constants. These values are either used for the stride parameter expected by some data operations or for the number of symbols to read from tape using **load**. Related to this is the terminal **END**. This terminal returns the number of symbols remaining on the input tape. The introduction of this terminal made it possible to evolve programs that processed the entire input tape.

Table 1: Overview of the return types of the terminal and function set.

Name	**ret type**	**type arg1**	**type arg2**
LOAD	tape	int	data
FWD	tape	data	/
REW	tape	data	/
SEQ	tape	tape	tape
END	int	/	/
DPCM	data	num	/
INV	data	num	/
MIN	data	num	/
SUB	data	num	/
MTF	data	/	/
RAW	data	/	/
PEC	data	/	/
DIV	int	int	num
ERC_INT	int	/	/
ERC_NUM	num	/	/

4.6 Parameter settings

The experiments presented here were done using 2 populations of 500 individuals. The selection probabilities of the different genetic operations were:

- standard uniform subtree crossover 75%
- standard uniform mutation 20%
- reproduction 5%
- no depth or node count limit
- The selection strategy for the 3 genetic operators was tournament selection with a tournament size of 4

The island model requires additional parameters:

- exchange rate of 3 individuals every 10 generations
- the exchanged individuals (the ones being exported) were selected using tournament selection
- The worst individuals in destination subpopulation were replaced by the imported ones

5 Results

5.1 Tool-box and decoder

To validate the presented results a set of software tools have been implemented. The functions and terminals needed by the gp software are provided by an implementation of the virtual machine.

In order to reverse the transformation an encoding format has also been designed. When executing an instruction, the data produced by the virtual machine actually comprises a header and the processed input data. This header describes the instruction used to produce the data, the length of the data as well as possible parameters required by the instruction. In the present implementation this header is 9 bytes in size. It is important to point out that the program that produced the transformed data is encoded in the output data. The entropy used as an objective criterion in the fitness function, is thus the entropy of the data **and** the program that created it.

Of course a decoder is required to reconstruct the original data. Using the headers present in the data the different transformations can be reversed one by one. One can decode the preprocessed files and compare them with the original data with a program such as *diff*. The size of the statically linked decoder is 23534 bytes on a Linux x86 platform. Both the data and the decoder should be transmitted to obtain the original data. The gain in compression should exceed the size of the decoder.

5.2 Entropy reduction

Files from the Canterbury Corpus were used as input data and a separate preprocessor evolved for each of them. The reduction of the entropy obtained for some of the files is given in table 2. The standard image compression image *lena*, for example, has been transformed so that is retains only 68.3% of its original entropy. And this without any loss of information.

Table 2: Comparison between the initial entropy and the entropy after preprocessing

File	original H	new H	% reduction
kennedy.xls	3.57	0.7	78.4
laptop.bmp	7.76	3.3	56.5
lena_std.ppm	7.75	5.2	31.7
mosaic.pnm	7.78	4.1	46.6
peppers.ppm	7.66	5.3	30.1

This reduction in entropy without loss of information is not impossible. The evolved preprocessor exploits the redundancy that is present in the data. This can not be modeled by the formula for the entropy given by equation 2. One would have to use conditional probabilities when computing the entropy to account for this kind of redundancy. This *higher order redundancy* is however very much there and is (partially) exposed by transforming the data. Although reducing the entropy of given data is not an easy task, it is of no immediate use as such. The idea behind reducing the entropy is to improve the compression rate. To illustrate this table 3 gives a comparison between the compression ratio of the data with and without preprocessing. The compression algorithm used here is bzip2. [6]

Table 3: Difference in compression ratio after preprocessing. Initially the lena could be compressed to 74% of its initial size. After preprocessing the compression ration is 68% of the size.

File	ratio	new ratio	% reduction
kennedy.coded	0.12	0.02	80.8
laptop.coded	0.53	0.43	19.1
lena_std.coded	0.74	0.68	7.9
mosaic.coded	0.72	0.52	28.3
peppers.coded	0.80	0.69	14.3

One can notice some difference between the gain in entropy and the gain in compression size. The gain in compressed size can be smaller than the reduction in entropy because Burrows Wheeler transform [1] used by bzip2 can to some extent model the redundancy beyond the single symbol probabilities. In other words, bzip2 already breaks the theoretical limit computed using the entropy formula in equation 2. That is why the gain is somewhat lower in this case.

[6]Bzip2 is free implementation of the Burrows-Wheeler transform. Bzip2 was invoked with the maximum compression parameter -9.

5.3 Filters

It was initially expected that, given the operations which can process chunks of data, the evolved programs would process the entire input data in a piecewise manner. That is, the programs would consist of a sequence of operations on contiguous parts of the data. The function set provided to the genetic programming framework certainly would allow for such programs to be evolved.

This was however not the case, in the first experiments only small fractions of the data were being processed. In their experiments Nordin and Banzhaf [10] have chosen to evolve separate programs for segments of fixed size, *chunking*, to avoid a similar problem.

The reason behind this phenomenon is that the entropy is a global measure over the entire output tape. Initial programs may indeed reduce the entropy. But the growth of these programs can adversely affect the initial changes to the entropy. Thereby making the results of the genetic operations like crossover less fit. This has been observed even when the initial population was seeded with full grown trees.

```
(SEQ (SEQ (SEQ (LOAD (DIV 67 2)(DPCM 1))
               (SEQ (LOAD END (DPCM 1))
                    (REW (DPCM 3))))
          (SEQ (REW (SUB 1))
               (REW (DPCM 3))))
      (SEQ (REW (SUB 1))
          (REW (DPCM 3))))
```

However with the introduction of the special **END** terminal in the set an interesting result showed up. This *dynamic* terminal returns the number of symbols left on the input tape. The evolved programs became filters. Most of the data goes through several transformations. In the example above, the program can be divided in 2 parts. First, 33 symbols are load into the machine and DPCM coded using the (LOAD (DIV 67 2) (DPCM 1)) instruction. The second part of the program processes far more symbols. The instruction (LOAD END (DPCM 1)) instruction loads *all the symbols* remaining on the input tape and DPCM codes these. Using the (REW (DPCM 3)) instruction the DPCM coded data is used again as input. This time the data is DPCM coded again but only every 3-th symbol is being processed. The data will undergo 4 more transformation after this step. The remain transformation are in order: SUB, DPCM, SUB, DPCM.

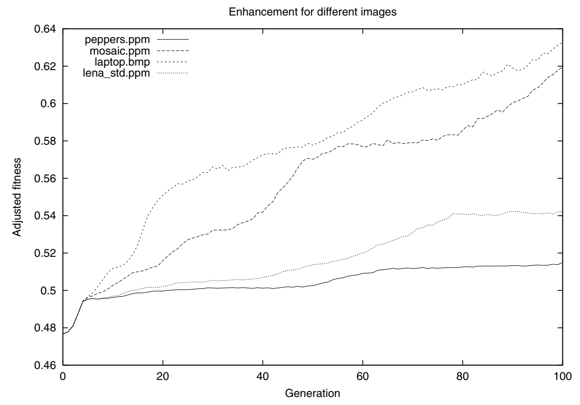

Figure 3: Evolution of the adjusted fitness of the best individual for the different data files, average of 8 runs.

6 Conclusion

This experiment presents a new approach for combining genetic programming and lossless data compression. The chosen approach develops preprocessing programs which are tailored to the data one wishes to compress. The obtained results are encouraging both in terms of gain in compression as for the computation time required to evolve the programs. One should note that, although the experiments were done using a parallel software package, the speed improvement results from the fitness function as well as the fact that we focus on preprocessors. Surprisingly the evolved programs were mostly filters although the provided functions and terminals allowed to evolve to more complex preprocessors.

References

[1] M. Burrows and D.J. Wheeler. A block-sorting lossless data compression algorithm. *Digital System Research Center research report 124*, 1994.

[2] Winzip Corporation. www.winzip.com.

[3] Canterbury Corpus. http://corpus.canterbury.ac.nz/.

[4] Karel Driesen. Compressing sparse tables using a genetic algorithm. In *Proceedings of the GRONICS'94 Student Conference, Groningen*, February 1994.

[5] I. De Falco et al. A kolmogorov complexity-base genetic programming tool for string compression. In *Proceedings Seventh International Conference on Genetic Algorithms (ICGA97)*, 1997.

[6] H. Feiel and S. Ramakrishnan. A genetic approach to color image compression. In *ACM 0-89791-850-9*, 1997.

[7] Alex Fukunaga and Andre Stechert. Evolving nonlinear models for lossless image compression with genetic programming. In *Genetic Programming 1998: Proceedings of the Third Annual Conference*, pages 92–102, 1998.

[8] The Gzip homepage. www.gzip.org.

[9] Sean Luke. Code growth is not caused by introns. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Conference*, 2000.

[10] Peter Nordin and Wolfgang Banzhaf. Programmatic compression of images and sound. In David B. Fogel John R. Koza, Davis E. Goldberg and Stanford University Rick L. Riolo ed., editors, *Proceedings of the First Annual Conference on Genetic Programming*, pages 354–350. CA, USA, Mit Press.

[11] J. Parent. Parallel lil-gp technical report. Vrije Universiteit Brussel, http://parallel.vub.ac.be./~johan/Projects/.

[12] B. Punch and D. Zonker. lil-gp genetic programming version 1.1 beta. michigan state university, http://garage.cps.msu.edu/software/lil-gp/index.html.

[13] P. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications.* Addison-Wesley, Reading, Mass, 1991.