Programmatic Compression of Images and Sound

Peter Nordin and Wolfgang Banzhaf
Universität Dortmund LS11, Fachbereich Informatik, Dortmund, Germany
nordin,banzhaf@ls11.informatik.uni-dortmund.de

Abstract

The importance of digital data compression in the future media arena cannot be overestimated. A novel approach to data compression is built on Genetic Programming. This technique has been referred to as "programmatic compression". In this paper we apply a variant of programmatic compression to the compression of bitmap images and sampled digital sound. The work presented here constitutes the first successful result of genetic programming applied to compression of real full size images and sound. A compiling genetic programming system is used for efficiency reasons. Different related issues are discussed, such as the handling of very large fitness case sets.

1 Introduction

Programmatic compression is a very general form of compression. The basic idea behind this technique is that any system, which evolves programs or algorithms for generating data, can be viewed as a data compression system. The data that should be compressed are presented to the Genetic Programming system as fitness cases for symbolic regression. After choosing a function set that facilitates an accurate reproduction of the uncompressed data, the system then tries to evolve an individual program that, to a certain degree of precision, outputs the uncompressed data. If the evolved program solution can be expressed by fewer bits than the target data, then we have achieved a compression. In principle, this technique could address any compression problem with an appropriate choice of the function set and the method has a potential for both lossy and lossless compression. The initial studies performed by Koza [1]in the domain of very small bitmap images displays the difficulty of the problem and the huge computational effort needed for real world examples. We have applied a Compiling Genetic Programming System (CGPS) to the programmatic compression problem. A CGPS is up to 2000 times faster than a LISP Genetic Programming System (GPS) [5],[6], which enables the application of programmatic compression to real-world problems.

1.1 Compiling Genetic Programming

A CGPS is a GP system where all genetic manipulation of individuals is performed in binary machine code. Almost all computers today are of Von Neumann type, which means that programs and data reside in the same memory. Program information can thus be regarded as just another sort of data structure and it can be manipulated as any other data, in this case by the genetic operators. This arrangement enables the deletion of all interpreting steps, which results in a speed up of up to 2000 times compared to a LISP system, and 100 times compared to an interpreting C implementation. Some experiments described below consumed 10 CPU-days of execution. This kind of experiments would not be feasible with other types of GPSs. A LISP version of such an experiment could take more than 50 years while an interpreting C version still would need two years of execution time. These compression examples thus are real-life experiments and research, only feasible with a CGPS. For a more detailed description of the CGPS and comparison with other implementation methods, see [5],[6],[8].

2 Programmatic Compression (PC)

Programmatic image compression is briefly mentioned in [1]. Koza uses the 900 pixel values of a small 30x30 bitmap of a regular mathematical pattern as fitness cases for symbolic regression by a GPS. The system finds a good solution in less than 100 generations with a generation size of 2000 individuals and a function set consisting of arithmetic operators. This method has an important relation to the theoretically maximal compression achievable. If a Turing-complete function set is used then the shortest program that accurately produces the target data could be seen as one estimation of the Kolmogorov Complexity [3] which opens possibilities of a deeper theoretical analysis of the application. The GPS has at least the potential of finding this shortest possible compression of the data.

In our work we have used two real-world applications which both call for the use of very large fitness case sets. In the experiments described below we apply programmatic compression to bitmap images and sampled sound. The large potential of these two fields in the growing di-

gital media arena cannot be overestimated.

We examine the compression in several different ways for instance with different function sets, fitness evaluation methods and parameter settings.

The majority of experiments were performed with a simple function set consisting of the most usual low level machine code operations such as arithmetic and logic operations. It is surprising that the most efficient search runs were performed using these machine instructions as the function set, regardless of application. These results strongly support the universal applicability of GP. Sometimes critique has been heard that GP only would manage to solve different problems because there was a specially adapted function set corresponding to the problem domain. In most of our runs we have used a function set that is not decided by us but by the manufacturer of a certain computer. The same set of operators was used for the completely different domains of sound and images, suggesting that it is *not* the choice of function set that is primarily responsible for the success of GP.

The advantage of programmatic compression is the large flexibility possible with a programming language as output of a compression process. This flexibility implies a possible larger compression factor sometimes approaching the theoretical limit. The use of machine code instructions makes decompression very fast. The decompression algorithm is simply a small and efficient binary machine code program. Decompression rates of 150 million bits/second per processor are sometimes possible on a SUN 20. This figure definitely exceeds the demand of, for instance, real-time full size video.

Another advantageous feature of the system is that each element of the target data (sample or pixel) can be accessed independently of the values of other elements. For instance, a sample with a certain index number can be obtained without the need to calculate the values of samples before and after that index number.

2.1 Chunking

The simplest and most obvious method to be applied when using PC is to present the target data as a continuous sequence of fitness cases. The evolved program then tries to reproduce the entire list of data when executed. While uncomplicated and elegant, this method has some disadvantages when used with very large fitness case sets. If the target data are very complex, there is a risk that an evolutionary search will not converge to a solution with acceptable quality. It is also harder to predict how many generations are needed before a solution is found. Therefore, we have used two different systems applied to each of the two domains sound and images. The first system treats all the fitness cases at the same time. The other system applies programmatic compression to equally sized sub-sets of the fitness cases and evolves a solution to each of them. Below we refer to this method as *chunking*. A time limit is imposed on each of the chunks which keeps the overall conversion time under control. The disadvantage of chunking is that the compression ratio could be lower since similarities between chunks will not be expressed in the individuals.

Chunking for sound is done by letting a CGPS evolve a solution for a fixed size sample chunk typically below the resolution time of the ear (70ms). In the Image domain the pictures were divided into small quadratic blocks, 8x8 or 16x16 pixels each.

The system was in this way presented with fitness case sets of sizes from 32 to 100000 integers. The largest fitness case set corresponded to 10 seconds of sound or a 256x256x8bit pixel image.

2.1.1 Two methods for long term GP-memory

In order to speed up the search in the chunked data approach two different "memory methods" were introduced. The first method *kept* the entire generation between chunks which speeded up the search when similar chunks were adjacent. The second method made a few individuals "read-only" in every generation. They could take part in reproduction and crossover but only as parents. They were never allowed to be over-written by any offspring. For every generation, less than 10 individual were frozen, to help maintain diversity and to save valuable genetic code for future chunks. The second method has the advantage that genetic information can be reused between non-adjacent chunks. In this way, the GPS is given an association capability that enables it to use knowledge obtained from a similar problem earlier in time. The results of this experiments show that these mechanisms can save considerable amounts of search time while the best solutions found tend to converge to a slightly worse fitness. The tradeoff between search time and quality makes the method worth further investigation.

2.1.2 Multi-level Compression

When this method is used in practice it is advisable to compress the resulting machine code segments further. In most experiments only half of 32 bits in the instructions were used in the programs, the rest of the instructions did not have any relevance in a CGPS. This suggests that the instructions in the result can be further compressed by a factor of two. In addition the resulting "16-bit instructions" can be ordered according to their frequency and then Huffman-coded. This method will have only a marginal effect on the overall decompression speed, because decompression of programs only has to be performed once per chunk.
Note that introns will be removed before packing the programes. The introns will be identified with the method described in [2].

3 Compression of Sound

Our first experiments with PC concerns sound generating programs. The difficulty in generating and predicting large and complex time series has previous been reported by Oakley [4]. In this case, the time series consists of sampled sound. The data are produced by the built in sampler in a SUN workstation, which produces sound samples with 8 bit resolution and 8kHz sampling frequency. A recording of a one second sound, is thus 8000 bytes long and corresponds to 8000 fitness cases for the CGPS.

3.1 Sound Generating Program architectures

There are several program architectures possible for individuals for time series problems. The first obvious individual structure is a function without inputs that has the ability to store values as side-effects. The individual is placed in a loop, the variables initialized to zero and then output of the program only depends on what is stored by the program in these variables during the cycles of the loop. The variables can be accessed as a stack, as fixed variables or as index memory. In a similar way the individual can be directly fed by its previous output. A different approach is to feed the individual programs with the index number of the fitness case and expect it to compute the output only from this index number. A combination of side effects and index number is also possible. Other architectures of the individuals are also possible. The individual could also be fed with recent fitness case output, but most of these methods are not suitable for efficient compression, because you have to supply output values. This method, however can be used for time series prediction into the near future. Table 1 summarizes the parameters used during training.
Of these approaches the index version without side-effects gave the best results during evolution of sound. An integer is thus given as input to the individual program and the output is taken to be a single sound sample. The integer index is subsequently incremented and fed to the same individual, which then produces the next sample. In its simplest case, the fitness is just the sum of the absolute values of the differences between the actual sound samples and generated samples. There are many other methods to establish fitness values and as we have experienced, they can have a considerable impact on the quality of the evolved results.

3.2 Fitness measurements

Oakley has previously reported the importance of choosing an appropriate fitness measurement when evolving a chaotic and oscillating target function [4]. He observed that when fitness was measured as the difference between the target output and the actual output, the system would only converge to a constant output. The constant output represented the mean value of the target function. When we used the fitness measure of absolute value of differences, we noticed that for many initial generations, sometimes more than 100, the system displayed this behavior. However, a constant value is not of much use when the target is to evolve a sound. In the long run though the system always converged to a more accurate, oscillating output. Figure 3.2 shows the output of a program after the oscillation in the output has been adapted to a close fit of the target samples[1]. The original sound in this case is a sampled two-tone door bell. We can here see, how the output represented by the dotted line shows a very good adaption to the target function's frequency, amplitude and phase. A view of the entire sample also shows an adaption to the envelope, that is the long term changes in amplitude. However, the waveform in the output is simplified to a square wave. This behavior seemed hard to change by changing the parameters and varying the function set. When we changed the fitness function to the squared difference instead of the absolute value of the difference the result changed significantly. Not only did the convergence to an os-

Terminal set :	Integers 0-8192
Function set :	ADD, SUB, MUL, SHL, SHR, XOR, OR, AND desired value
Maximum population size :	100 - 100000
Crossover Prob :	90%
Mutation Prob :	5%
Selection :	Tournament Selection
Maximum number of nodes:	16-1024

Table 1: Summary of parameters used during training.

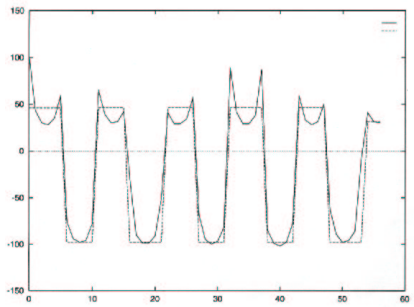

Figure 1: Adaption of generated sound to target sound (GP output is dotted)

[1]This figure, like most figures below, shows an excerpt of a longer sequence of samples. The details of the wave form of the full sequence is rather irregular, which might explain some irregularities in the output. It is impossible to show the full sequence in one diagram. If the training sequence was restricted to only the short sample spaces shown in the figures an even closer convergence would have been expected.

cillating output happen earlier, but the resulting curve also displayed much more details. We also tried fitness differences raised to three and raised to four without significantly better results. The crucial point could be that non-linearity is needed.

The results with the squared error both looked and sounded much better, but the details of the waveform in the generated output lacked the extreme peaks of the target sample. To increase the selection pressure towards accurately reproducing these pointed features we tried with a difference between two adjacent points as the fitness value. This fitness measure did not work well when adopted alone but when it was in a weighted sum with the normal absolute value fitness it displayed a more pointed behavior.

All of the fitness methods above have the disadvantage that they do not take into account the way the ear perceives sound. The output can look quite similar to the samples and there could still be a lot of audible distortion. The evolved sounds often had metallic overtones and sometimes noise added to it. To some extent this could probably be filtered out with a filter similar to an anti-aliasing filter, but the fact remains that there is a difference between looking similar and sounding similar.

The human inner ear could be considered to perform a transformation similar to a Fourier transform of the sound we hear. This is the reason why we perceive tones and not quickly changing sequences of pressure variations. This mechanic Fourier transform is necessary because the 'clock frequency' of the brain is not fast enough to process sound directly.

Another fitness measurement would thus be a comparison with the Fourier transformed spectra of the generated samples and the target functions. Oakley [4] used Fourier transforms as a means to force the system away from local minima of constant output. Our system showed oscillation without the Fourier transform but we instead applied it to improve sound quality by aligning fitness function judgment of the output with that of the human ear.

We thus implemented a fast Fourier transform (FFT) taking real-valued data and transforming them into the frequency domain represented by complex numbers. Fitness is measured as the squared difference between the transform of output and the transform of the target samples. Both the real and imaginary part of the transform were separately taken into account by the fitness function in order to assure that the phase of the output was treated appropriately.

The sound produced by the FFT fitness system was audibly better. The CPU time needed for this system was about twice that of the system without FFT, which was reasonable considering the quality improvement.

When FFT is applied in the fitness function then the exact amplitude of the curve is less important, instead

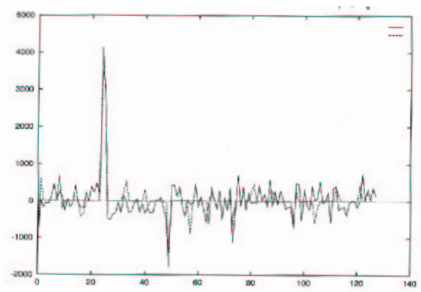

Figure 2: FFT spectrum of the sample(GP output is dotted)

features that influence the overtones of the system are dominating. Figure 3.2 shows the spectrum of the same sample. An external function in a CGPS is a function that is not part of the processor's instruction set and has to be used by a call instruction. Any C-function can be linked to the system and used as an external function.
The function set in the experiments presented above was eight low-level machine code instructions: addition, subtraction, multiplication, shift left, shift right, logical and, logical or and logical exclusive or.
Different experiments where also performed with the use of side effects in functions. The programs were given the possibility to store values in memory between the processing of the samples. In the extreme case the individual was not provided with an index number for the sample that should be generated instead the individual had to do its own book keeping with the use of memory variables. These programs took longer time to evolve and performed worse in fitness but had a softer sound with less overtones. Side effects were introduced as Save/Restore functions,the Swap function and indexed memory of different sizes.
The first reflection over sound generating programs is that it would be advantageous to use functions related to oscillation in the function set. We therefore tried to use the sinus function directly in the function set as well as functions that are components of a discrete differential equation for oscillation and thus together could produce a variety of different curves. To our surprise did none of these functions perform better than the individuals built with the simple machine code building blocks.
Automatically defined subfunctions (ADFs) [2] were also used without significantly improving results.

3.3 Evolving Large Samples Sequences

As mentioned above two methods were tried for evolution of large samples sequences. The simple linear method that tried to evolve the sample with one individual proved insufficient for large and complex sound. This method also had the disadvantage that its convergence behavior and evolution time requirements were

Figure 3: Target bitmap with fast evolution

Figure 4: An example of output from the programmatic compression system

hard to predict. It was possible to evolve the frequency, amplitude, phase, envelop and, to some extent, the wave form of a 16.000 samples sound lasting two seconds, if the sound was regular. However, to evolve changes in frequency in a sample, turned out to be harder. We succeeded to generate two different tones, that change at a specific point in the sample frequency, but we did not have any good results with more complex sounds. The more random output from earlier generations sometimes expressed complex melodies, but it did not evolve such melodies corresponding to a specific fitness case set. Even more complex sounds, like spoken words were impossible to mimic with this linear method and with sample length of audible size.
The solution we use instead for evolution of longer and more complex sounds is chunking. The sample sequence is divided into equally sized sequences and the CGPS in then applied to each chunk in turn. The chunk size is chosen so that the length of a chunk is below the resolution of the human ear. Pulses that come faster than 70ms are perceived as tones and not individual pulses. The fact that the evolved sound always is in phase assures that the sound pieces fit together well in the joint of two chunks. With this method it is possible to evolve complex sound lasting for a longer time. We have for instance evolved human speech with an understandable quality. Melodies of 100000 samples have also been evolved. In future work we would like to investigate the possibility of evolving the division points between chunks allowing for adaptive and varying chunk lengths.

3.4 Results

Compression ratios vary much depending on the regularity of the target sound. For regular sound the compression ratio can easily be 10 times or more, while the ratio for speech is around two times. The quality is still below telephone quality but we will in the future investigate several methods to improve it.

The CPU time for generating a ten second chunked sample is in the range of tens of hours on a SUN 20.

4 Compression of Pictures

Our system for programmatic compression of images bears many similarities with the sound compressing system. Instead of trying to evolve a one dimensional array of samples we here try to evolve a two dimensional array of pixels. In our experiments every pixel holds 8 bits of information. In the experiments we have used gray scale images because of the technically more complex handling of color in the display palette. The technique is however equally applicable to color pictures and to pictures with more than 8 bit quantification, i.e. 24 bit color. The output of the evolved individuals consists of eight bit numbers. If very fast decompression is required, it is possible to use the full 32 integer bits of the architecture. In this way more bits per second can be decompressed.

4.1 Program architecture

Basically the same method is used for images and sound. As a first method an individual is fed by the X and Y coordinates of the pixel and the output is interpreted as the value of the pixel. However a simpler method proved to give better results. Instead of supplying both the X and Y coordinates to the programs, we supplied a *one dimensional* index value of the pixel.

Chunking is a must when working with realistically sized images. In our experiments we have used pictures consisting of 256x256 pixel which were divided into squares of either 16x16 or 8x8 pixel. The definitely best results were achieved with 8x8 pixel blocks. This size corresponds to the size used by several other image compression techniques such as JPEG [9]. Different methods for side effect were also used, and unlike the sound example the pictures had a slightly better quality when the

Figure 5: The target bitmap

programs also had access to a small set of memory cells.

Figure 4.1 shows an example of an evolved image that used figure 4.1 as target data. Figure 3.2 shows a quickly (one hour) evolved picture with lower quality .

4.2 Fitness functions

Different fitness measurements were also applied, and similar to the sound domain, the squared error differences gave better results than the absolute value.

In future work we would like to use two dimensional FFT in the fitness function, as well as functions measuring differences between adjacent pixels in several directions.

The blocks of chunking are visible to a varying degree in the decompressed pictures. By incorporating the differences in color around the edges of blocks into the fitness function we hope to reduce this effect. We will also try different conventional smoothing techniques used i.e. with JPEG for the same purpose.

The CPU times for compressing a picture in our experiments on a SUN20 are in the range of 30 minutes to 10 days depending on the requested quality and compression ratio.

5 Summary and Conclusion

We have described experiments that have shown that programmatic compression can be used with other than toy problems in both the image and sound domain.

With these compression experiments we have demonstrated that GP can be used with very large fitness case sets if a compiling approach is applied. Some of our experimental runs have evaluated hundreds of billions of fitness cases, something that would take years on an interpreting GP system. The machine code approach provides for very fast decompression suitable even for the demand of real time full size video.

Acknowledgment

One of us (P.N.) acknowledges support by a grant from the Ministerium für Wissenschaft und Forschung des Landes Nordrhein-Westfalen under contract I-A-4-6037-I

Bibliography

[1] Koza, J. (1992) *Genetic Programming*, Cambridge, MA: MIT Press.

[2] Koza, J. (1993) *Genetic Programming II, Automatic discovery of reusable programs*, Cambridge, MA: MIT Press.

[3] Li M., Vitani P. (1990) Inductive Reasoning and Kolmorogov Complexity. In *Journal of Computer and System Sciences*, pp343-384

[4] Oakley H. (1994) Two Scientific Applications of Genetic Programming: Stack Filters and Non-Linear Equation Fitting to Chaotic Data In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA , USA, MIT Press.

[5] Nordin, J.P. (1994) A Compiling Genetic Programming System that Directly Manipulates the Machine-Code. In *Advances in Genetic Programming*, K. Kinnear, Jr. (ed.), Cambridge, MA , USA, MIT Press.

[6] Nordin, J.P. , Banzhaf, W. (1995) Evolving Turing Complete Programs for a Register Machine with Self-Modifying Code. In: *Proceedings of the Sixth International Conference of Genetic Algorithms*, Pittsburgh, Penn., USA, Morgan Kaufmann Publishers

[7] Nordin, J.P. , Banzhaf W. (1995) Complexity Compression and Evolution. In: *Proceedings of the Sixth International Conference of Genetic Algorithms*, Pittsburgh, Penn. ,USA, Morgan Kaufmann Publishers

[8] Nordin, J.P. (1995) Compiling Genetic Programming. To appear in: *Handbook of Evolutionary Computation*, T. Bäck, D. Fogel (ed.), New York, NY ,USA , Oxford University Press.

[9] Pennebaker, W.B., Mitchell, J.L. (1993) JPEG still image data compression standard. Van Nostrand Reinhold, New York , USA