Gene expression synthesis

Alo Allik
http://tehis.net
alo@tehis.net

ABSTRACT

Gene expression programming presents an alternative approach in the evolutionary computation paradigm evolving populations of candidate solutions as valid computer programs that can be used for a potentially wide range of problem solving tasks, including sound synthesis. This paper proposes Gene Expression Synthesis (GES) as a method to evolve sound synthesis functions as nested graphs of unit generators. The functions are encoded into linear chromosomes according to the principles of gene expression programming and then evolved by subjecting the functions to genetic operations and evaluating fitness. The design of the fitness functions involves statistical methods and machine listening algorithms in an attempt to automate the supervision of the synthesis process. The specification of synthesis parameters explores the idea of artificial co-evolution. A parallel population of functions share their fitness values with their respective synthesis functions for which they compute parameter values, while being subjected to genetic operations including recombination separately.

1. INTRODUCTION

Since the first artificial life experiments by Nils A. Barricelli in the 1950s, evolutionary computing has inspired numerous problem solving and model building techniques including ways to evolve sound synthesis algorithms inspired by processes of evolution by adaptation and natural selection. In our attempts to understand these natural algorithmic processes, which are purposeless and devoid of any intention, but nonetheless directly responsible for all the complexity and intelligent behavior in the natural world, we keep developing increasingly more powerful technology that enables us to model and simulate, albeit on a vastly simplified scale, the power of cumulative selection. Genetic algorithms and genetic programming have been firmly established as efficient and productive stochastic search and optimization methods within the artificial intelligence field and have been widely used in various disciplines for years. Gene expression programming was introduced as an improvement to the existing paradigms, proposed by Candida Ferreira in 2001, by combining the best features of genetic algorithms and genetic programming [1]. The fundamental differences between gene expression programming and its predecessors stem from the separation of genotype-phenotype representations and the modular multigenic structure of the chromosomes. These improvements account for significant increases to the efficiency of the algorithm for a number of benchmark problems. The following account describes an experimental approach to evolving sound generating programs with the proposed principles and explores creative applications of evolutionary computation which do not necessarily presume a definite solution to a problem, but rather an open-ended solution space to be explored for aesthetic experimentation.

Copyright: ©2014 Alo Allik et al. This is an open-access article distributed under the terms of the Creative Commons Attribution 3.0 Unported License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

2. SOUND SYNTHESIS WITH EVOLUTIONARY ALGORITHMS

The evolutionary paradigm has been harnessed in a broad spectrum of applications in the realm of computer music, applying the processes of gene expression, selection, reproduction and variation on many different levels of compositional hierarchy. Examples can be drawn throughout all musical time levels, including producing waveforms directly by expressing binary genotypes as sample level time functions, evolving synthesis graphs and optimizing parameters, generating longer time structures and patterns of motives and phrases, all the way to composing comprehensive artificial environments inhabited by listening and sound-generating agents. Magnus [2] developed a modified genetic algorithm that works directly on time-domain waveforms to produce genetically evolved electroacoustic music. Garcia [3][4] proposed using evolutionary methods for selecting topological arrangements of sound synthesis algorithms and for optimizing internal parameters of the functional elements. On the phrase and motive level, there are two classic studies that paved the way for countless later explorations: John Biles [5] hierarchical GenJam system that generates on-the-fly jazz chord progressions and the "sonomorphs" proposed by Gary Lee Nelson [6]. Jon McCormack [7] developed an interactive installation of evolving agents influenced by the presence and movement of audience as an example of a comprehensive digital sonic ecosystem. These are but a few examples of the wide range of applications for evolutionary algorithms and by no means meant as a review, rather a random sampling of applications on different levels of the compositional process.

The abundance of different possibilities explored demonstrates the potential inherent in evolutionary processes which can exhibit unparalleled efficiency and problem-solving resourcefulness even in a vastly simplified form as compared

to the forces operating in the natural world. The idea of automating the design process of sound synthesis algorithms using evolutionary methods has to be considered in the context of computer music specification. Generating waveforms by the direct principle of sample-by-sample calculation, for example, does not necessarily require any higher level infrastructure or a specialized programming environment, however, such an approach may complicate the design of an efficient fitness function, especially considering unsupervised learning methods. Since the Music N programming languages (most prominently Csound), the encapsulation of sound generating and processing functions into unit generators has cultivated a modular graph based concept of synthesis with interconnectable functions as building blocks. Most contemporary synthesis software, regardless of whether the interface is graphical or text-based, operates based on this model. The method presented here has been implemented in the SuperCollider environment, but is applicable in any audio programming environment that has adopted the graph based paradigm, where sound synthesis programs are defined as interconnected unit generator graphs. These graphs can be evolved by evolutionary programming principles just like any other computer programs that serve as the solution space for a particular problem. The question then becomes how to define or, in other words, encode these graphs in terms of evolutionary programming.

SuperCollider synthesis topologies have previously been studied in the context of evolutionary programming. Dan Stowell [8] presented a genetic algorithm for live audio evolution at the first SuperCollider symposium in Birmingham 2006. The system demonstrates how genetic methods can be used in a live setting, with modifications to the synthesis process occurring in real time. Fredrik Olofsson [9] released a similar algorithm for sound synthesis through his personal website. The goal of his project was to create genomes that would translate into realtime synthesis processes and allow the user to evaluate the results in a framework of a realtime sequencer. The algorithm is, similarly to the one described above, based on arrays of floating point values serving as genomes, which were translated into SuperCollider synthesis definitions.

The SuperCollider implementation of the gene expression programming proposed here expands on the foundations of the methods described above. The problem addressed is how to encode SuperCollider unit generator graphs as populations of chromosomes and evolve these graphs using genetic operators. In a similar way, there is a constrained selection of unit generators that are included in the graphs and the translation process produces valid sound generating functions that are evaluated for fitness. However, the following description introduces a number of modifications and distinct features in accordance with the techniques of the gene expression algorithm to introduce an alternative strategy for evolutionary sound synthesis.

3. COMPONENTS OF GES

Gene expression programming (GEP) is a method of evolutionary computation providing an alternative to the established paradigms of classic genetic algorithms (GA) and genetic programming (GP) [1][10]. The basic premises that these methods share in common have been inspired by biological evolution and attempt to model the natural selection process algorithmically in computers. All these methods use populations of individuals as potential solutions to a defined problem, select the individuals from generation to generation according to fitness, and propagate genetic variation within the population by random initiation and applying genetic operators. The differences between these algorithms are defined by the nature of individuals. In GAs the individuals are fixed length strings of numbers (traditionally binary); in GP the individuals are non-linear tree structures of different sizes and levels of complexity. GEP combines these approaches by encoding complex expression trees as simple strings of fixed length to overcome the inherent limitations of the previous methods. In GEP the genotype and phenotype are expressed as separate entities, the structure of the chromosome allowing to represent any expression tree which always produces a valid computer program. Another feature to set GEP apart from its predecessors is the structural design of GEP individuals that allows encoding multiple genes in a single chromosome. This facilitates encoding programs of higher complexity and expands the range of problems that can be solved with evolutionary computing.

GEP consists of two principal components: the genes (genotype) and the expression trees (the phenotype). The information decoding from chromosomes to expression trees is called translation. The genome or chromosome consists of a linear, symbolic string of fixed length composed of one or more genes. Each gene is structurally divided into two sections: a head and a tail. There are two types of smallest units called codons that make up a gene: functions and terminals. Terminals operate as placeholders for static variables or arguments to the functions. The head of a gene contains symbols representing both functions and terminals with the start codon always holding a function while the tail is entirely made up of terminals. This structure and the particular rules of translation in GEP ensure that each gene encodes a valid computer program. Despite the fixed length of the genome, each gene has the potential to encode for expression trees of different levels of complexity and nesting. The translation from genotype to phenotype follows a simple, breadth-first recursive principle: as the codons of a gene are traversed, for each function encountered, the algorithm reserves a number of following unreserved codons as arguments to that function regardless whether they are functions or terminals. The number of codons reserved depends on the number of arguments the function encountered requires. In order to illustrate this process, encoding of a simple phase modulation graph is shown in Figure 1. Such a gene would have to consists of a head section with at least 3 codons and tail with at least 6. The first 3 positions in the head of this gene contain the two sine oscillator functions and a terminal in between (the head part of the gene is indicated by a shaded grey background). The tail is entirely made up of terminals.

In the *Karva* notation[11] this chromosome is represented

0	1	2	3	4	5	6	7	8	9	0	1	2
O	a	O	b	c	d	e	f	g	a	c	d	b

Figure 1. Encoding a phase modulation instrument as a single-gene sequence

as a string of upper and lower case letters with position reference numbers above:

```
0123456789012
OaObdefghcdbc
```

The expression tree that emerges form this gene after the translation process is shown in Figure 2.

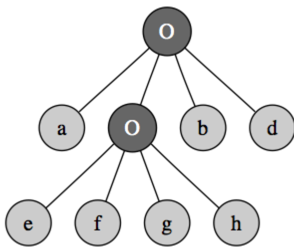


Figure 2. Decoded expression tree of phase modulation as a graph of sine oscillators. Sine oscillators are designated by captial O and terminals by lower case letters

The first codon designating a sinewave oscillator audio rate method (represented here by symbol O) - which in SuperCollider language specification expects four arguments: frequency, phase, mul and add - is translated as the root node in the expression tree with four branches deriving their values from codons in positions 1 to 4 in the chromosome string as they get reserved according to number of arguments into the function. When the algorithm encounters a terminal, there is no need to reserve anything and the terminal is assigned its position in the tree with no further branching, however, when it comes across another function at position 2 in the head of the gene, it looks ahead to reserve the next sequence of codons, in this case four arguments are expected again, therefore terminals at positions 5 to 8 fill these nodes. Once the algorithm has filled all the function arguments, the process stops and the rest of the terminals in the tail section of the gene are ignored. This mechanism allows to define the potential complexity and nesting in the resulting computer programs as a function of overall gene length. The expression tree above translates into a corresponding SuperCollider unit generator graph function:

```
{arg a, b, c, d, e, f, g;
 SinOsc.ar(a, SinOsc.ar(d, e, f, g), b, c)
}
```

The size of the gene tail t is calculated based on the size of the head h and the number of terminals n required by the function with the largest number of arguments.

$$t = h(n - 1) + 1$$

Another feature that sets GEP apart from other evolutionary algorithms is the use of multigenic chromosomes. Multigenic chromosomes can be combined together by a function that serves as a linker. In order to provide an example of a multigenic chromosome, let us consider a slightly more complex example than the phase modulation graph above. This time there are four unit generators involved: sine oscillator SinOsc (O), sawtooth wave oscillator LFSaw (S), random values oscillator with quadratic interpolation LFNoise2 (N) and band-limited pulse wave generator Pulse (P) [1] . Since the generator with largest number of arguments is the sine tone oscillator and the head size remains the same for the time being, the gene size is also the same as above, but this time the chromosome consists of two genes which are linked together by mathematical multiplication function in the translation process.

The gene expression tree of this chromosome consists of two independent sub expression trees corresponding to the multigenic structure: the first one has a noise generator as the root codon and the second one a sawtooth oscillator. There is an additional linker function, in this case multiplication, which combines the genes together into a single composite function, as shown in Figure 3

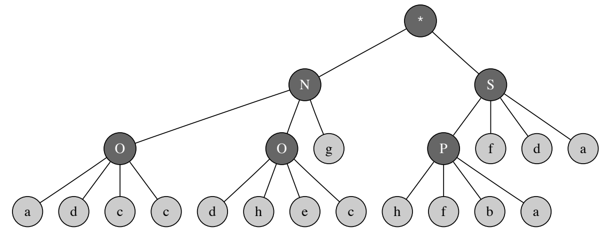


Figure 3. Expression tree of a multigenic chromosome with multiplication function serving as a linker.

This expression tree translates to a unit generator graph function in SuperCollider:

```
{arg a,b,c,d,e,f,g,h; LFNoise2.ar(SinOsc.ar(a,d,
c,c),SinOsc.ar(d,h,e,c),g) * LFSaw.ar(
Pulse.ar(h, f, b, a ), f, d, a )
}
```

GEP chromosomes contain several genes each coding for structurally and functionally unique expression trees. Depending on the problem to be solved, these sub-trees may be selected individually according to their respective fitness or they may form a more complex multi-subunit expression tree and be selected according to the fitness of the whole chromosome. The linker between the individual expression trees can also be any function and depends on the context of the task at hand. For example, in the above structure, the multiplication could be substituted by

[1] All the examples presented in this paper use these four unit generators with their corresponding abbreviations

addition to produce additive synthesis instead of modulation synthesis or any other function that requires two arguments.

4. THE SELECTION PROCESS

The gene expression process does not differ much from that of the classic genetic algorithms. It begins with the random generation of chromosomes of a certain number of individuals for the initial population. In the next step, these chromosomes are translated into computer functions to be executed and the fitness of each individual is assessed against a set of desired examples which act as the environment to which the individuals are to be adapted. The individuals are then selected according to their fitness (their performance in that particular environment) to reproduce with modification, leaving progeny with new traits. These new individuals are, in their turn, subjected to the same developmental process: expression of the genomes, confrontation of the selection environment, selection, and reproduction with modification. The process is repeated for a certain number of generations or until a good solution has been found.

The initial population in gene expression programming is created in the same way as in other evolutionary computation algorithms either by randomly populating the gene codons with functions and terminals determined to be part of the solution space or using pre-existing individuals from a pool of previous successful runs. In case of random generation of the population, which is by far the most common method used, the genes are constructed, first, by randomly selecting a root node from the included function definitions, then the head codons are filled by randomly selecting a function or a terminal for each position and, finally, the tail only includes random selections of terminal values. Although, it is not absolutely necessary to define a root node as a function according to GEP principles, especially in multigenic chromosomes, it proves more crucial of a factor in the special case of sound synthesis. Sound synthesis is a special case for more than one reason and the many constraints that it imposes on the GEP paradigm will be discussed in detail in the following sections.

As in any other evolutionary programming model, the most important and challenging component in GEP is the design of the fitness cases as this is what drives the fitness of the population and ultimately decides the success of the problem solving algorithm. In most cases which are trying to find the single best solution to a particular problem, the goal must be defined clearly and precisely in order for the system to evolve in the intended direction. Although it may not always be the case, particularly while evolving candidate solutions for complex, open-ended situations including sound synthesis or musical phrase composition, a poorly designed fitness function tends to produce random meaningless results and either converges on an inappropriate solution or will not converge at all producing consistently large error values in individuals with the highest fitness.

The selection process commences once each individual in the population has been assigned a fitness value. The purpose of this phase of the algorithm is to propagate the fittest solutions to the following generation. Again, there are a number of different methods by which to select the individuals, stochastic and deterministic, however in the long run it makes little difference which one is used as long as the best traits of the current population are preserved in the new population. The preferred method in GEP is stochastic, which entails assigning each chromosome in the population a probability weight value proportional to its relative fitness. This may mean that the fittest individual may not always survive the selection process while mediocre individuals might be selected.

5. GENETIC OPERATORS

The selection process has a tendency to converge towards a single high scoring solution and, without genetic operators, would rapidly get stuck in a local optimum. Therefore it is essential to maintain genetic diversity, which is mainly achieved by several modifications introduced during the replication process of the genomes. There are a variety of genetic operators in GEP divided into three main categories: **mutation**, **transposition**, **recombination**.

Mutation entails modifying a single value in a randomly chosen position and can occur anywhere in the chromosome. However, the structural organization of the chromosome must be preserved to ensure that when expressed the individual still produces a valid program. This means that the root can only be replaced by another function, any codon in the head section of the chromosome can be substituted by a function or a terminal and only terminals are allowed as replacements in the tail section. Mutations of a single codon can have a dramatic effect on the phenotype a chromosome is encoding, especially if it occurs in the head section. The following Karva notation strings display a mutated chromosome before and after the mutation, in which a terminal that occurs in position 1 in the original gene has mutated into a sine oscillator in the next generation:

```
0123456789012
NcOgadccdhecc

0123456789012
NOOgadccdhecc
```

Figure 4 shows the effect on corresponding expression trees of this single-point mutation.

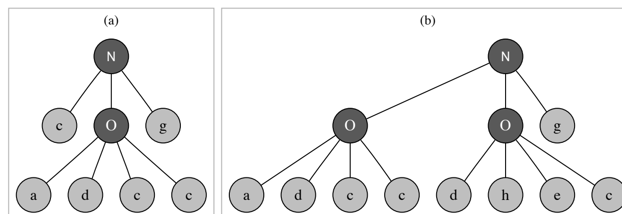


Figure 4. Single-point mutation. A terminal in (a) changes into a sine oscillator function in (b)

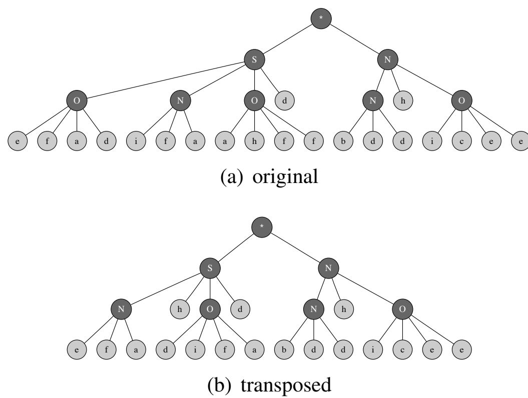
Mutation rate is defined as a global constant in the GEP algorithm and can be specified as a probability percentage

which each chromosome is subjected to. If the mutation rate is defined as 0.1, it means each chromosome has a 10% chance of being subject to a random one-point mutation.

The transposition operations in GES copy short fragments of the genome from their original locations to another location in the chromosome. For example the already familiar gene from two previous examples is subjected to transposition of a short codon sequence shown in Karva notation and Figure 5. The terminals at locations 5 and 6 are copied into the head section of the gene, which results in the first two parameters - frequency and phase in this case - of the root codon sawtooth oscillator of the first gene to be replaced by a noise generator and a terminal instead of a sine oscillator and a noise generator.

0123456789012345678901234567890123
SONOdefadifaahffbN**NhO**bddiceedaebcd

0123456789012345678901234567890123
S**NhO**defadifahffbaNNhObddiceedaebcd



(a) original

(b) transposed

Figure 5. Transposition of a codon sequence

The SuperCollider synthesis function that is derived from the transposed tree is shown in the code listing below:

```
SynthDef('r00_g02_s001', {arg a,b,c,d,e,f,g,h,i;
 Out.ar (0,
  (LFSaw.ar(LFNoise2.ar(e,f,a), h,
       SinOsc.ar(d,i,f,a),d))*
  (LFNoise2.ar( LFNoise2.ar(b,d,d),h,
       SinOsc.ar(i,c,e,e)))
  )
})
```

Recombination involves choosing chromosomes from the pool of individuals that have successfully passed the selection process and exchanging their genetic material. This process results in creation of two new individuals. A defined number of points are randomly chosen along the two parents and their codons are copied to the child chromosomes as mixed set containing codons from each of the parents. In order to illustrate the basic principles and effects of recombination let us consider two chromosomes derived from the same four unit generators presented previously. The listings below display two parent chromosomes in Karva notation (head sections in bold):

0123456789012345678901234567890123
SONOdefadifahffba**NNhO**bddiceedaebcd

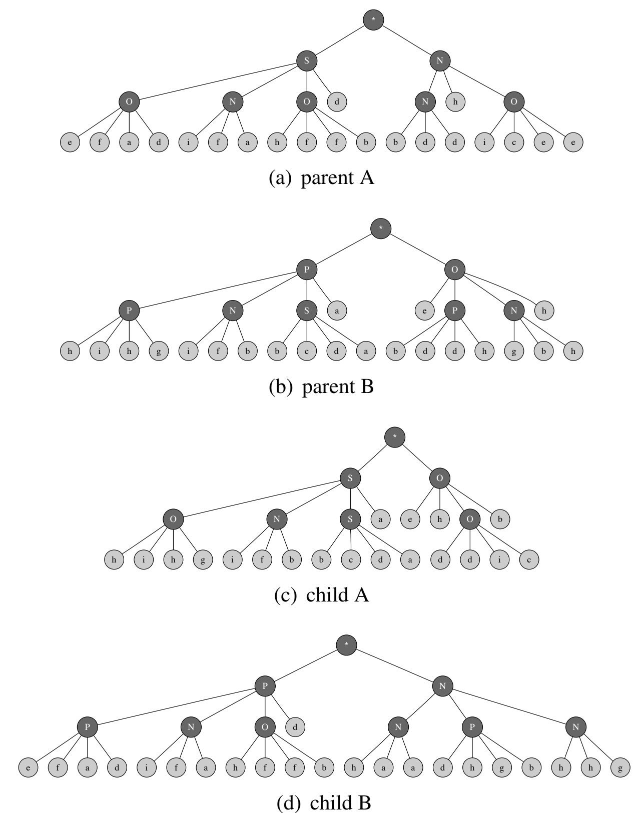
0123456789012345678901234567890123
PPNSahihgifbbcdaf**OePN**hbddhgbhhgdee

After subjecting these chromosomes to recombination, the result is two new individuals that have characteristics of each of the parents. In the symbol strings below, the components that made up the original chromosome 1 are indicated in bold to illustrate the effect of recombination. The first of the two randomly selected recombination points is located at position 3 of the chromosome and the second occurred at position 27 located in the head section of the second gene.

0123456789012345678901234567890123
SONSahihgifbbcdafOe**hObddiceedaebcd**

0123456789012345678901234567890123
PPN**OdefadifahffbaNN**PNhaadhgbhhgdee

The corresponding expression trees of the two parents and their progeny is shown in Figure 6



(a) parent A

(b) parent B

(c) child A

(d) child B

Figure 6. Recombination

These are relatively simple examples in order to demonstrate the principles of genetic operations in gene expression synthesis. The synthesis functions that have been evolved so far using this technique typically originate from chromosomes consisting of at least 4 up to 8 genes and head sizes ranging between 8 to 16, resulting in much more complex graphs with more levels of nesting. While the genetic operations ensure variability within the population, evolution towards a goal is largely determined by a fitness function.

6. EVOLVING UNIT GENERATOR GRAPHS

The algorithm works in a cyclical pattern as illustrated in Figure 7, first an initial population of n individuals is generated, then each individual is expressed as a recursive expression tree beginning with the root node which can then be translated into a sound synthesis function string. The function string is evaluated and a synthesis process is started on the server. An analyzer agent then assigns a fitness value to each individual. The selection process is stochastic and associates a probability weight to each individual based on their relative fitness. Replicated individuals are then subjected to a series of genetic operations depending on the settings of the algorithm. Once every new individual has been exposed to the genetic operator phase, the cycle is completed by replacing the original population with the new individuals which then are ready for the subsequent repeat of these steps.

Figure 7. The cyclic gene expression programming algorithm.

In order to begin the process of evolving unit generator graph functions for sound synthesis, there are two crucial components to be defined: (1) the specification of terminals as synthesis function arguments and (2) the design of the fitness function so that the evolution proceeds towards desired goals with minimal human supervision. The specification of terminals was solved by introducing a parallel population of calculation functions in which each individual becomes expressed as a list of floating point values. This parallel population is evaluated simultaneously with the sound generating functions and each individual receives the same score as its counterpart in the sound generating population. However, the selection process and the genetic operators are applied separately so the population retains a certain degree of independence. The number of genes in this parallel population corresponds to the number of terminals necessary to fill all the parameters. The functions used in this population are not sound generating functions, but binary arithmetic operators of addition, subtraction, multiplication, and division and the terminals are static floating point values. This solution imitates the phenomenon of co-evolution in the natural world where two interdependent species indirectly cause mutual evolutionary changes across the confines of their genotypes.

7. DEFINING THE FITNESS FUNCTION

The fitness function uses machine listening algorithms to analyze the candidate solutions once they have passed an initial basic compilation test on the SuperCollider server. Before the machine analysis can commence, any individual that fails the basic fitness check and the expressed function fails to compile, is automatically assigned a weight value of 0 and consequently excluded from the selection process. Compilation may fail for any number of reasons, the most common being invalid input type and since initialization is completely random, unsuitable function arguments become quite frequent in case unit generators that have arguments of specific type. A good example of an invalid unit generator argument would be in case of a filter algorithm which expects the first argument to be a signal of the same rate (typically audio rate in this case) as it is running itself, therefore a floating point number is not accepted and compilation fails. There is an option to start the process by filling the initial population exclusively with candidate solutions that pass this check.

The machine listening process analyses a set of 20 mel frequency cepstral coefficients (MFCC), spectral flatness, spectral centroid, and amplitude features into running mean and standard deviation values over a desired duration, 3 to 8 seconds in the runs reported in this account. Invalid output from any of the analysis processes (mostly NaN or unrepresentable value as a result of a calculation, dividing 0 by 0 for example) is assigned an error value greater than one which gets treated the same way as uncompilable functions and is thereby excluded from the selection process. The fitness function that was used in all the variants of the gene expression experiments under investigation in this case used example analysis sets extracted from sound examples towards which the algorithm was expected to converge. A number of different reference sounds were used including sounds synthesized with GES, other types of synthesized sounds as well as sounds of traditional musical instruments. The score of each individual was determined as the difference between maximum possible score and the total actual error in each of the analysis categories. The mean and standard deviation statistics of each of the MFC coefficients were given double weighting relative to other statistical values and the maximum error in each of the statistical categories was set to 1.0. Spectral centroid values, which are expressed in frequency values, were mapped to range between 0.0 and 1.0. This meant a maximum individual score of 10.0 as the sum of scores from MFC coefficients adding up to 2.0 for both mean and standard deviation statistics, and to 1.0 for spectral flatness, spectral centroid, and amplitude.

Table 1 represents the assignment of initial fitness scores which were calculated as difference measures from the corresponding features of the reference sound. Each of the 20 mean MFCC coefficients were each assigned a weight value of 0.1, which means that the maximum score possi-

FEATURE	NUMBER	WEIGHT	MAX
MFCC (mean)	20	0.10	2.00
MFCC (std dev)	20	0.10	2.00
Flatness (mean)	1	1.00	1.00
Flatness (std dev)	1	1.00	1.00
Centroid (mean)	1	1.00	1.00
Centroid (std dev)	1	1.00	1.00
Amplitude (mean)	1	1.00	1.00
Amplitude (std dev)	1	1.00	1.00
		Total max	10.00

Table 1. The weights of features used in the fitness function. The maximum score of each feature is calculated by multiplying the number of features by the weight

ble from the sum of these features is 2.0. The same weight is assigned to the standard deviation values of MFCC. The remaining 6 features - mean and standard deviation values for Spectral Flatness, Spectral Centroid, and Amplitude - were each assigned a weight of 1.0. Therefore the maximum similarity score possible is 10.0 in case of identical features.

In order to imitate the condition of limited resources of natural selection, each candidate solution is assigned a CPU usage value measured during the execution of the synthesizer. At the end of each evaluation cycle, the CPU usage percentage is normalized relative to the minimum and maximum values of the population and the scores recalculated adding in the CPU percentage as 10 percent of the total score. This pressure introduces a tendency in the population of favoring simpler synthesizer graphs over more complex ones. To counteract this tendency a conflicting fitness pressure is introduced to encourage structural complexity. Maximum depth of unit generator nesting is a straightforward indicator of complexity in graphs, so the scores are adjusted according to the maximum depth of a chromosome relative to the maximum of the population. This way, the complexity can be maintained in populations, while still encouraging resource usage effectiveness. These parameters can be adjusted depending on the purpose of the experiment.

8. DISCUSSION

The most striking feature of the implemented synthesis system that emerged during the experiments is perhaps the phenomenon of high fitness scores being present starting from the initial randomly generated population. The maximum score remained fluctuating within a limited range at the top of the fitness landscape and did not seem to improve. Figure 8 shows a graph of mean and maximum fitness scores plotted against each generation in an experiment, in which a synthetic bass sound was used as reference.

This reveals the crucial characteristics of the algorithm and informs of inherent properties and constraints of sound generating functions going forward. One of the factors for this outcome is the additional server compilation check applied prior to exposing the functions to the statistical fitness

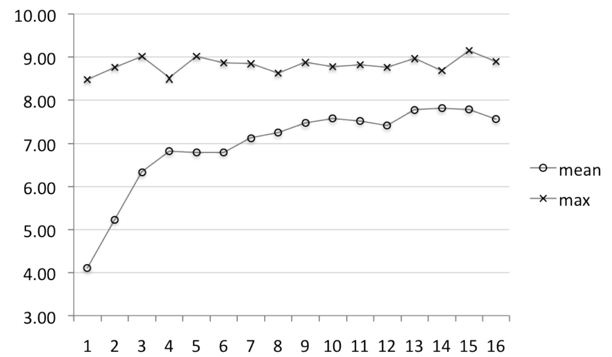


Figure 8. Maximum and mean scores of a gene expression synthesis experiment with the fitness scores on the x-axis plotted against the generation number

evaluation procedures. This is an additional layer, which in standard framework of GEP problem solving would seem redundant, but has been devised to accommodate for the specific language-server architecture of the SuperCollider environment, which is essentially comprised of 2 different computer languages. Another characteristic that emerged from these experiments was the explosion of variety combined with rapidly improving mean fitness of the entire population after 2-5 generations, which produced the most interesting (as subjectively judged by the author) synthesis functions in large variety compared to later generations. The variety tends to diminish after a number of generations similar to classic GA-s as strands of highest fitness individuals take over the population. Furthermore, evolving sound-generating functions imposes rather strict constraints on the algorithm, which in these experiments were largely ignored as much as possible. For example, the co-evolution of parameters to the functions from a set of random calculations exposed the audio system to unexpected output values which were weeded out by the statistical analysis of amplitude tracking and bad value checks. Further normalizers and limiters were employed to try and keep the synthesis output within a perceivable (and tolerable) range. The persistence of relatively high but static maximum scores also underlines the limitations of the statistical fitness functions used in the experiments. The mean and standard deviation statistics of the example sounds do not provide sufficient time-domain information, therefore, there was always a significant variety of functions with differing spectral and temporal characteristics attaining high scores. The main limitation to the sonic output is the selection of unit generators, which were naturally not expected to conjure up complex spectra of the human voice or traditional musical instruments. The fitness statistics were intended rather as rough guides to acceptable ranges of spectral characteristics of the candidate solutions and near exact matches were understood to be virtually impossible from the outset even in the case of GEP evolved synthesis experiment. In total, over 3000 new synthesis definitions have been selected as additions to the GES database as of this publication. These synthesis definitions are stored together with their statistical analysis data and linearly en-

coded chromosomes to be utilized in the interactive autonomous mikro improvisation system, live coding improvisation performances, and as genetic source material for further experiments in GES.

Garcia [4] presents a somewhat similar approach to evolving sound synthesis graphs. The sound synthesis algorithms are similarly represented as topologies of interconnected nodes in a graph while the fitness function also uses an audio feature based distance metric to compare the candidate solutions to a target sound. The evolution of graphs is driven by genetic programming, in which the population consists of individuals each of which represents a nonlinear tree of varying complexity. This variation makes it difficult to apply genetic operators and thereby new solutions are not introduced into the populations. This means that in order for the algorithm to converge, the components for the optimal solution would already have to be present in the initial populations.[1] Since GES uses the gene expression programming algorithm to guide the search, this problem is solved by encoding the graph represenation (phenotype) into a linear representation (genotype) which enables more complex and efficient genetic operators to be applied. In Garcia's experiments, the algorithm is designed to converge to the target and in the example presented shows the error diminish significantly lower over 200 generations than in experiments reported here. This is mainly because (1) the implementation of Garcia's algorithm is on a lower representational level of synthesis using C++ and matlab while the experiments presented here utilize the large variety of unit generators available in SuperCollider, and (2) Garcia's experiments are designed to converge to an optimal solution whereas GES experiments are designed for open-ended exploration.

9. CONCLUSIONS

The claims that the gene expression programming algorithm is superior to the traditional evolutionary algorithms[1] appear to be corroborated in the GES experiments based on the speed of population mean fitness increase and high maximum scores starting with initial populations. There are no direct comparisons with previous sound synthesis algorithms due to the fundamentally different nature of the studies considered. Existing genetic algorithms that have made use of the SuperCollider programming environment used human listeners as the fitness function, therefore it is not possible to compare the evolution of the fitness landscapes. These initial experiments have provided a rich insight into the myriad of sound synthesis possibilities latent in the GES algorithm. Based on these experiments and taking aboard the methodology explored in previous evolutionary programming attempts with SuperCollider, the algorithm can be expanded to include range limitations for unit generator parameters to safeguard against unreasonable output values. Special classes of unit generators - such as filters, buffer players, reverberation and spatialization functions can only be incorporated by providing structural constraints in the design of the chromosome, but would significantly increase the complexity of the potential sonic output. Further, the parameter definition could be optimized more efficiently.

The gene expression synthesis method was designed as a tool to be used in the creative process, particularly to build up a database of synthesizers that could be employed in different live performance situations. This objective has guided the decisions made while designing fitness functions and experiments with the recognition that there are many alternative ways to implement the basic concept presented here.

Source code, a tutorial and further examples of GES are available at `http://geen.tehis.net`

Acknowledgments

This research was partly funded by the University of Hull 80th Anniversary PhD Scholarship.

10. REFERENCES

[1] C. Ferreira, "Gene expression programming: a new adaptive algorithm for solving problems," *Complex Systems*, vol. 13, 2001.

[2] C. Magnus, "Evolving electroacoustic music: the application of genetic algorithms to time-domain waveforms," in *Proceedings of the 2004 International Computer Music Conference*, 2004.

[3] R. A. Garcia, "Automating the design of sound synthesis techniques using evolutionary methods," 2001.

[4] ——, "Growing sound synthesizers using evolutionary methods," in *In Proceedings ALMMA 2001: Artificial Life Models for Musical Applications Workshop*, 2001.

[5] J. Biles, "Genjam: A genetic algorithm for generating jazz solos," in *Proceedings of the 1994 International Computer Music Conference*, 1994.

[6] G. L. Nelson, "Sonomorphs: An application of genetic algorithms to the growth and development of musical organisms," in *Proceedings of the Fourth Biennial Art & Technology Symposium*, vol. 155, 1993.

[7] J. McCormack, "Eden: An evolutionary sonic ecosystem," in *Advances in Artificial Life, 6th European Conference, ECAL 2001, Prague, Czech Republic, September 10-14, 2001, Proceedings*, 2001, pp. 133–142.

[8] D. Stowell, "Supercollider code written by Dan Stowell," http://www.mcld.co.uk/supercollider/, accessed: 2012-08-27.

[9] F. Olofsson, "Work with Mark: Genetics. (a blog post.)," http://www.fredrikolofsson.com/f0blog/?q=node/144, accessed: 2012-08-27.

[10] C. Ferreira, *Gene Expression Programming: Mathematical Modeling by an Artificial Intelligence*, ser. Studies in Computational Intelligence. Springer, 2006, vol. 21.

[11] ——, "Karva notation and k-expressions. from gep tutorials: A gepsoft web resource." http://www.gene-expression-programming.com/tutorial002.htm.