Evolving Chord Progressions as Neural Networks

Hansen Schwartz

12 April, 2006

Abstract

Systems which evolve music are becoming increasingly popular under the domain of interactive evolution. Steady state neuroevolution can be used for both learning an example and learning through interaction. This provides an avenue to creative composition. However, too much freedom in creation can lead to undesirable results and thus knowledge of music must be incorporated. Furthermore, creation of artworks with patterns requires some sort of *memory* to store the patterns. We attempt to solve both of these issues through providing a representation for chords based on simple music theory, and using a structure which inherently builds a type of *memory* . In this paper, we present and perform preliminary evaluations of two distinct, yet related ideas. The first idea is dealing with new implementations for methods of steady state NEAT. The second idea demonstrates the ability of the modified NEAT structure to learn and produce basic chord progressions.

1 Introduction

There are many reasons one can find interest in a computer, interactively, composing music. Applications exist across a wide range of applications. Real-time systems can interact with the environment or a set of users as a whole: For example, a department store which adjusts the style and mood of music based on observed client behavior. From a more direct point of interaction, composer's can ask a computer to generate a set of possible next measures, based on previous measures in order explore paths they may not have thought of. Similarly, another set of applications are catered toward entertainment value or for those without a big musical background. Many people know what they like, but they do not have the knowledge to be able to create it. Finally, it can simply be seen as a form of entertainment to interact with a computer exploring music. The research presented here is applicable to all forms of interactive composition listed above, however the focus of implementation is on the later ideas of direct user interaction.

Neural networks are conducive to this task because due to their ability to encode structure that may not be so obvious to a human. Peter Todd explains that while many artists may have deep within their minds an algorithm or set of rules, it is difficult to state just exactly what they are [8]. It becomes even more difficult when trying to produce a variety of pieces of art. Thus, although theory on music may restrict our search, a network is able to give structure to a series of chords. This structure can be used to base further progressions.

In order to provide users with a pleasant experience it is desirable to limit the amount of bad output created. Figure 1 provides a visualization of the hope from bootstrapping the system to learn a desired chord progression before interactive evolution. Of course for this to have a better chance of working, one should be certain neural networks and evolution systems are capable of producing decent chord progressions. Consequently, there is a focus on this step of creating a NEAT implementation which is able to learn chord progressions. These learned chord progressions can then be fed into the system to allow interactive evolution, which more desirable output. We present an approach steady state neuroevolution similar to Real-Time NEAT [6], and demonstrate it's effectiveness at learning to produce chord progressions.

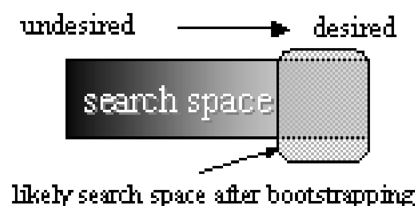

Figure 1: Reduction of search space by bootstrapping the system to learn a desired progression first

The paper is organized as follows. Background is presented both on our domain of chord progressions, and on similar work done in the past. Next, we explain our approach, which includes some differences from traditional NEAT. Experiments and Results show our approach works for XOR as well as basic chord progressions. Finally, we discuss the implications and future of our work.

2 Background

There have been numerous strategies to evolve music used in the past. Additionally, NEAT is no stranger to real-time or even interactive evolution. Before examining the past work in evolving music, one should be aware of concepts in our domain.

2.1 Chord Progression

Music typically consists of melody, harmony, and rhythm section. Melody and harmony have a rhythm as well, but a rhythm section is distinguished in that it is not given distinct pitches to play in rhythm. Chord progressions represent the harmony sans rhythm of a musical piece. In the most basic form these progressions simply represent a successive set of chords. Chords on the other hand, are multiple notes played in parallel. For example, playing the notes, (C, E, G) together is a chord, while a progression consists of multiple chords such as: ((C, E, G), (F, A, C), (G, B, D)) where order matters. In this paper, a less strict defin Many times a single unison note is desirable in a chord progression or harmony.

In composing music, several different approaches are used. One of the most popular approaches is to first create the harmony and later add melody and rhythm on top [1]. The harmony is responsible for the mood created by a piece which typically elicits the most emotions in a listener. Although it is not likely that a composer will strictly adhere to creating a harmony first [1], the idea of a computer doing so simplifies this project somewhat. One could attempt to create a system which evolves all aspects of music. However focusing on just one aspect(Harmony) and providing additional restrictions makes for a much more achievable (yet challenging) project.

The chord progressions in this work are represented as a series of chords, defined with the following characteristics:

- *Key* : Restricts the possible notes to be based on one (tonic) note which a scale can be based.
- *Quality* : Determines the type of scale to use with the key, (Major or Harmonic Minor in our case).
- *Root* : Gives which note in the key is the base which determines all other notes relatively.
- *Degrees* : List of degrees up the scale which should be played (listed in order of most desirable: fifth, third, seventh, ninth, thirteenth, second, forth, sixth).
- *Degree Adjustments* : List adjustments to be made to each degree. Typically changes the pitch -1, 0, or +1 halfstep.
- *Inversion* : Determines which will be the lowest in the chord.
- *Octaves* : Determines how many octaves (repetition of notes with higher frequencies) the chord progression will span.
- *Bottom Octave* : Which octave should the lowest note belong.
- *Spacing* : Open - The top octave has only two notes, Closed - the top octave contains more than two notes.

These next two more appropriately fit the idea of harmony as whole. However, they may be important variables to consider when listening to chord progressions.

- *Volume* : The loudness which the notes in the chord are played
- *Duration* : The length which each chord lasts.

It is very important to note these characteristics are based on ideas in music theory. A learning system typically should not ignore knowledge and representation on a subject when it is available. Yet, there are still many representations which could be used based on theory of music. Obviously, this representation is different than that used in score or part to a piece of music read by musicians. Although a score provides the same information, it is less restricted. This representation provides a limit on the search space for chords, and can be limited furthermore by setting some of the characteristics as constants.

2.2 Past Work

There are a few distinctions from this research, and past work on the same subject. Systems have not used an approach to music based on NEAT. The focus on chord progressions alone is somewhat novel. Representations of chords which we part of other systems have taken a different approach. Finally, the idea of bootstrapping the system with learned chords before interacting has not been explored in quite the same way.

It seems the most popular methods of composing music take on the task of learning a style of music. This is obviously similar to the approach of learning chords before interacting. David Cope has a system which "Generate quasi-random music using genetic algorithms, Markov chains, fractal and chaotic algorithms, and various mathematic, connectionist, or pattern-matching algorithms" [2]. The system learns from a database of music a style of a composer, and then tries to recreate pieces in the database but with slightly random changes. This seems to be the most developed of such systems available. However, the

complexity of the system makes it difficult to establish guidelines on guidelines on how to replicate or perform quantitative comparisons.

More specifically, neural networks have been evolved to create music. Recurrent Neural Networks were used by Franklin to experiment with several pitch representations. A novel circle of thirds representation is used to store information about notes in a chord which requires little output information from a network [4]. Eck and Schmidhuber present an approach using Long Short-Term Memory recurrent networks to create melodies which is able to consider the global structure more easily [3]. In [7], two types of networks are evolved in a coevolution process. One network produces music, and the other is a critic of the music. All of these systems use representations of chords as triads, where output always corresponds to predefined set of 3 or 4 notes. These sets of notes were based on music theory. However, the representation we present provides for more exploration in that the number of notes in a chord are not restricted, and other characteristics of the chord are considered.

Lastly, we should mention the work of Reil and Husbands [5]. The ideas presented in their paper, correspond to the idea of a leaky integrator neuron. This allowed for a rhythmic type of execution necessary to control bipedal walking. A node similar to this is being implemented in our system in order control the intervals at which chords or notes appear and reappear.

3 Approach

3.1 Variations on NEAT

There are several ideas presented that vary slightly from steady state NEAT as described in [6]. One significant change is in dealing with new genomes and species. After calculating a new genome's fitness, it inherits fitness from both the species and it's parents. The child's fitness linearly progresses (along a gradient) from this shared fitness to it's own fitness in a certain number of evaluations proportionate to the population size. The gradient rate at which it moves per evaluation can be defined by this function: $rate = 1 + \frac{populationsize}{n}$, where n is a constant to determine the portion of the population that is likely to be a child. Adding the parents to this mix allows more fit parents to correspondingly protect their children slightly better. It also helps when a child gets placed in a new species, as the new species will have a chance to grow even if the representative genome does not. On the other hand, shared species fitness is based on the species champ rather than the average fitness. This simply speeds up operations, since it is necessary for the system to quickly evolve an initial network before interactive evolution (which does not include species).

Dealing with fitness and selection there are several other differences to discuss. First, In addition to the methods described above designed to help new species, there is the ability to explicitly raise the fitness of all genomes in a new species. Next, typically the worst node is always selected to be lost in steady state. In this research we allow for any genome of a certain portion of the population to be lost. The portion of the population is already ordered by fitness due to being stored in a sorted list. With each rank closer to the end it becomes more likely that a node will be selected. This was implemented from the point of view that in nature the worst organism is not always lost, although there is little basis from an artificial point of view. Additionally, we implement a method to evaluate fitness favoring outputs produced early. Because the later characteristics of chords depend on previous values, the outputs for early chords seem more significant.

Several other small differences should be noted. We use a bisigmoid as the standard function, and have the ability to mutate functions such as add, multiple, threshold, greatest, least, and guassian rbf. Historical markings are stored forever. The addlink routine will just mutate a link if it discovers that first, which increases the probability of mutating a link as a network becomes more fully connected.

Lastly, several variances on initializing the population of networks were explored. Starting with recursive links at the outputs was possible. To take this further, *memory* was also possible to initialize a population with. As mentioned previously, *memory* is something desired from our structure. Therefore, the initial genome, which is copied to the population has the ability to come prepackaged with a set of nodes extending from the outputs. The networks can then evolve to determine when to connect a portion of the memory back to the output in order to achieve repetition.

3.2 Chord Representation

The representation of the chords described previously is gathered from output nodes of the network. Typically, only four or five of the characteristics were set by the network and the rest were left constant. *Root, Inversion, Spacing, and Degrees(not adjusted): Fifth, Third, and Seventh* were chosen from most often to encompass the evolved characteristics. These few characteristic can actually provide a large variety of progressions in a conservative manner by not changing constants such as *Key, Quality, and Degree adjustments* .

Output nodes interpret their activation to produce appropriate values for each characteristic. *Spacing and Degree on/off*were represented as binary output,

thresholding the activation value at a middle point. *Inversion* values were represented by a three level continuum to produce no inversion, first inversion, or second inversion. Finally, a variety of outputs were used for representing the *Root* . One approach was to sum changes -1, 0, 1 of values for the root over all timesteps the network was activated. A second method was to use a larger continuum to get values of -3, -2, -1, 0, 1, 2, 3 in which to move the root. It was important to be able to move the root from it's current location as this could help provide more pleasing results. There are only a total of seven pitches in the scale which a root can be. When values went below -3 or above 3 they were cycled to the opposite 3 or -3.

The output could not simply be converted directly into a chord. In addition to correctly choosing notes depending on the quality, key and root, some interpretation was necessary to produce pleasing sounds. For example, by playing a different inversion of the chord, the lowest note could be far from the root note. The interpretation algorithm adjusts the inversion so it is in the closest possible octave to the root. Additionally, when playing over multiple octaves, many times it does not sound good to have many notes in the bottom octave. Therefore, when octaves are greater than 2, notes are spread out in the lower octave. The end result of interpretation is simply a list of pitches to be used as a chord.

A neural network was a desirable structure for this task because of it's innate ability to remember. Recurrent links allow information produced many times steps sooner to remain within the structure. When turning a neural network into a chord progression, it is activated $((numberofchords) * (timestepsperchord))$ timessteps. Thus, the neural network is in essence a complete representation for the chord progression, given the mappings from outputs described early. The network is not reset during activation, and as a standard a single sensor(bias) with value of one was used. The network includes a warm up value, that states for the network to activate a certain number of times before starting to pull chords. This approach will also allow for a leaky integrator node [5] or a series of nodes to be deployed as input instead in the future.

4 Experiments

We present two types of experiments. The first subsection on xor is used to validate the effectiveness of our variations on NEAT. Second, we experiment with learning (through evolution) chord progressions, which serves as a bootstrapping technique before interactively evolving.

4.1 XOR

This experiment attempts to solve the xor problem on a variety of parameters. The test is run 60 times for each set of parameters, and the min, max, and average number of evaluations are recorded. A test is aborted if the same best fitness is repeated over 8000 evaluations. The population and networks are reset after each test. The following parameters were used:

pfd Parent and species fitness sharing / species fitness sharing only

lp Linear probability species choice / standard species choice based on fitness probabilities function

rl Recurrent links: whether to allow recurrent links in general

hn Help new species: provide extra fitness to new species

rf RemoveFactor: genomes of worst 1 / rf percentage of population are possible candidates for removal

The variable to the left of each item is used to identify whether the item was implemented (+) or not (-). For rf, many possibilities are experimented with.

For all of the XOR tests a standard set of probabilities was used that was found to be ideal for this domain. It included a high probability of adding a node. Keep in mind that adding a link could mutate a link if it already exists and is enabled. also used the following probabilities (in percentages):

a) probMateInSpecies = 98

a) probMateInPop = 2

a) probMate = 45

b) probMateAvg = 40

b) probClone = 15

c) probAddLink = 65 (subsumes mutate weight)

c) probAddNode = 20

c) probMuteFunc = 0

c) probMuteWeight = 10

c) probNoMute = 5

When creating an offspring, the algorithm first uses class "a" probabilities to choose from where. Next, class "b" probabilities choose how it will be created. Finally, class "c" probabilities decide how to mutate the new offspring. Only one choice from each class is permitted, and the sum of probabilities for each class must be 100

The fitness function used for xor evaluation is given below:

$$fitness = t - \sum_{i=0}^{t-1} error_i^2 \ ,$$

where $t = 4$, the number of xor situations to test, and $error_i (<= 1)$ is the difference from the output value and the ideal output value needed, scaled between 0 and 1. Keep in mind the activation function is a bisigmoid by default, so this scaling is necessary.

A few other parameters should be noted. The child grow up time is $1 + \frac{populationsize}{3} = 84$. This value was based on standard NEAT's newly created genomes per population (equivalently 1/3 in this case, rather than the standard 1/2). The min and max weights are -12 and 12, where the most a link can mutate each evaluation is between -4 and 4. All compatibility coefficients were simply 1. The network starts fully connected with the two standard inputs, 1 being true and -1 being false, plus a single bias set with value 1.

4.2 Chord Progressions

A variety of chord progressions are attempted to be learned through steady state evolution. The best parameters used during XOR were used unless specified otherwise below. The higher probability of node additions used with XOR is not as applicable to chord progressions. Tests were not performed on other varieties such as using initial *memory*

Unfortunately there was not time to permit a complete test of additional parameters when evolving chords. However, during development of the system, the values used for the experiments in this paper were discovered to produce more pleasing results than others. The additional parameters, and those differing from XOR included:

- increment the sensor(bias) with each chord: false
- timesteps per chord: 3
- recurse initial outputs: false
- add initial memory: false
- allow recurrent links: true
- warm up network: 0 timesteps

c) probAddLink = 65

c) probAddNode = 5

c) probMuteFunc = 0

c) probMuteWeight = 25

c) probNoMute = 5

The following chord progressions were used to be tested with:

1. (C E G), (F A C), (G B D), (C E G)
2. (C E G), (C F A), (D G B), (E G C)
3. (C E G), (A C E), (G B D), (C E G)
4. (B D F), (B E G), (A D F), (A C E)
5. (C E G), (C F A), (E G C), (D G B), (C E G)
6. (F C), (G C), (F A C), (E G C), (D F B), (C E G)
7. (C E G), (D F A), (C E G)

The chords are listed in order according to inversions desired as well. For example, the first and last chord of 2 are both C major, but the last chord is a first inversion. Keep in mind the lowest note is placed nearest the key, in this case of C. The first and second progression play the same chords, but the first include no inversions. (C E G) is common, especially as an ending, because it is the tonic, of the key, and it is common (a rule in many cases) to always end on the tonic. The first three progressions and the last two are mostly major, the forth is all minor, although it is still in the major key (the ending chord, A minor, is the relative minor for C major). 5 and 6 are obviously longer than the others. Notice not all chords contain three notes. The last sequence is given simply as an easy test. The *root, fifth, third,* and *inversion* outputs were needed to evolve these progressions, and they were they only outputs used.

Notice that while the length of the chord progressions varied, the *timesteps per chord* was set at 3. Each test is run for up to 40,000 evaluations. The length of time to reach 40,000 evaluations on a 2000mgz 896MB memory system is roughly 5 minutes. It is not practical for an interactive system to require more than a 5 minute bootstrapping wait time. The overall experiment is thus to determine how often each chord progression can be learned within 30,000 evaluations.

The fitness function was designed to weight earlier chords higher. Some output values rely on earlier values and thus a change to early value has a bigger effect. The fitness function is given below:

$$fit = \sum_{n=1}^{crds} ot * (rng^2 * n) - \sum_{i=0}^{crds-1} \sum_{j=0}^{ot-1} (err_i^2) * (crds - i),$$

where ot is the number of output nodes, rng is the range of output ($1 - -1 = 2$), crds is the number of chords to play, and err is the difference between the actual output and the output that would be ideal to get the desired value a specific output node, i. Notice the term $(error^2) * (crds - i)$ weights earlier chords higher. The desired chords are stored as the same type as what the system expects for NN output. The fitness was found never be less than 100 from the best possible fitness, so the following fitness function was actually used in the experiments:

$$fit = 100 - \sum_{i=0}^{crds-1} \sum_{j=0}^{ot-1} (err_i^2) * (crds - i)$$

To be safe, negative and zero values results from fitness were set to 1. Each chord progression was given only 5 chances to be learned, due to limits on time available to test.

5 Results

5.1 XOR

These results come from running each test 60 times. The tests encompass a variety of circumstances. under a variety of parameters described previously. The maximum number of evaluations, minimum, and average number of evaluations are reported in table 1. It should be noted that no tests were aborted, meaning the best fitness did not stay stagnant for more than 8000 evaluations.

pfs	lp	rl	hn	rf	max	min	avg
-	-	-	-	256	3604	175	1758.9
-	-	-	-	128	6175	590	2049.033
-	-	-	-	64	7775	126	2152.217
-	-	-	-	32	4029	209	1794.0
-	-	-	-	16	3888	86	1878.833
-	-	-	-	8	4274	226	1822.75
-	-	-	-	4	3482	259	1272.033
-	-	-	-	2	3064	320	1552.283
-	-	-	-	1	4941	280	1693.383
+	-	-	-	4	4216	403	1595.783
-	+	-	-	4	4213	147	1455.08
-	-	+	-	4	2343	25	728.61
-	-	-	+	4	5288	358	1294.5

Table 1: Evaluation statistics for creating a network to solve XOR under a variety of parameters: min, max, and average(mean) evaluations, see previous section for parameters

These results show that many of the different approaches in this paper might hinder the performance of evolution. The standard deviation in evaluations was 620 +-150 across all of the rf experiments. Notice that the best results were actually received when rf was somewhat low. This is when a greater portion of the population could be chosen for loss. Figure 2 displays this more clearly, by showing the portion of the population considered for removal with each average value.

The best results occurred when recursive links were allowed. By not allowing recursive links the network's abilities were restricted.

5.2 Chord Progressions

Results for the experiments described earlier are reported in table 2. One can see that the network more often was not able to learn a chord. Certain chord progressions were learned more often.

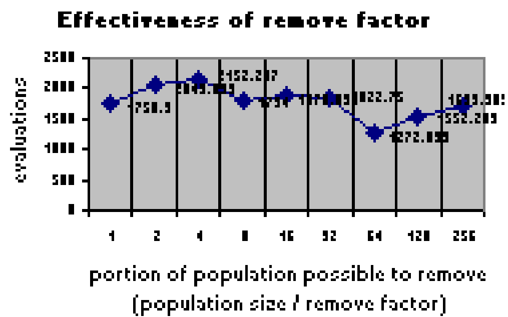

Figure 2: Average evaluations to solve xor as the portion of the population valid for replacement increases.

progression	learned	avg evals
1	3	7485.66
2	1	33500
3	0	0
4	1	17352
5	0	0
6	1	34147
7	5	11295

Table 2: Number of tests which a chord progression was learned in less than 40,000 evaluations and average number of evaluations when the progression was learned.

A neural networks are being provided to show the complexity used to learn these progressions. Figure 3 is a network which was able to play the first chord progression with 825 evaluations. If looking closely, one can see less complexity in this network than that in figure 4 which was used to produce the same chord progression with 15230 evaluations. The second progressions network was only slightly more complex than figure 4, and the second progressions pattern. Green lines represent links directing downward, or curved links directing left, while blue links refer to upward straight links or right curved links. Dotted links indicate the weight is negative. Results for the longer chord progressions were extremely rare, while the last shorter, and less complicated progression was always learned.

6 Discussion and Future Work

Much of this work is preliminary. More tests should need to be run to evaluate all the differences in our system versus real time NEAT, such as using a gradient fitness sharing rather than a period of all out fitness sharing. Additionally, the research should include some

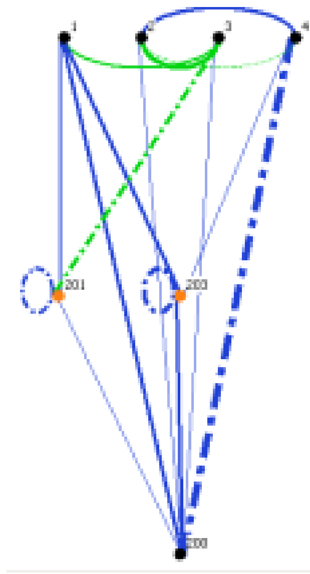

Figure 3: Neural network which plays chord 1

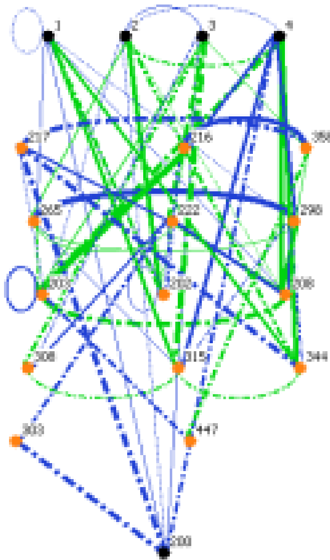

Figure 4: Another, more complex, neural network which plays chord 1

empirical results are how satisfied users were with what they could create. The differences in this approach to NEAT and the chord progressions would probably be best broken up into two papers if these preliminary ideas were expanded upon.

Let us take a moment to discuss the rational behind the gradient child fitness sharing, since this may be one of the biggest ideas proposed. A newly created genome receives the fitness from either a combination of parents and species, or strictly from it's species. This fitness slowly moves toward the child's real fitness at a rate that depends on the difference between both fitness values. The child's fitness is always reached at the same time from when it was created. This models biology in that as new organisms grow up, they slowly become more responsible and start leaving their mark on the world different than their parents. However, the purpose for fitness sharing in steady state neat is to, number one, improve the likelihood that an organism will mate or, number two, at least not be killed off immediately. The first of these purposes does not match biology, where as the second more closely does. Next, let's consider standard NEAT fitness sharing, where genomes are able to take advantage of the shared fitness in one generation. It would not make as much since to put this gradual change procedure in there, because the gradient would be steep between generations (possible just the difference if the fitness is only shared for one generation). Additionally, it would be difficult to implement within a generation. This works because a series of genomes will be new each generation (born at the exact same time conceptually). In steady state, genomes are always born in an order, and it makes more sense to help the newest genomes more than those slightly older. A gradient allows us to favor those genomes newer in a more ideal way than a strict drop in fitness at a certain point.

There are some things worth noting based our test results. From table 1, we see that a remove factor of four is best, although with only 60 test runs it is difficult to state that this will always be true. Of course it is somewhat clear that removing the worst genome may not always be best. The algorithm still favors those genomes closer to the bottom, but we theorize that not always removing the worst allows some really bad genome, such that species sharing does not help enough, to survive long enough for it's difference to become beneficial.

The neural network is able to learn the progressions even with only one input sensor. The sensor, or more correctly, the single bias serves as starter and constant source of activation, which then propagated around the network, recurrently many times. This seems to roughly simulate the memory a person may have. The fact that a network with only 2 hidden nodes was able

to learn the first progression, and another had 12 hidden nodes, suggests that more search space is needed mutating weights or possible adding links of earlier networks before moving to more complexity. Some networks seem to come really close to learning a progression, within .00001 of the needed fitness, but they are unsuccessful, perhaps being mislead through local optima.

The bootstrapping time of 5 minutes in the experiments is still likely too big, but it was necessary to give the evolution a fairer shot at find something. In application, it may be possible to just store the bootstrapped networks, and if someone wishes to create a new one, they will have to wait the minutes for evolution. It is also likely this time will speed up or it will learn better with future improvements. There are many variables currently in the system, and unnecessary code due to the variety of implementations there are for many of the evolutionary methods. Removing these may also help once we have a refined system. Furthermore, rather than weighting earlier chords more in the fitness, it may be desirable to first evolve to produce one chord, then to produce two ..etc.. all the way until the entire progression is learned.

It would have been nice to experiment with people, and produce a quantitative study based on human opinions. When doing short simulations of interaction, the neural network still produced desirable chord progressions after mutating. However, it seems that the first priority should be setting up the evolution system to be able to learn chord well. This implies the system will respond better to feedback from the user more acceptably. So, to summarize "near future" work, we will be implementing leaky nodes, experimenting more with initial *memory* , and be adding a friendly graphical user interface in order to evaluate the system on a variety of people.

7 Conclusion

A new implementation of steady state NEAT was introduced, evaluated on XOR, and used to learn chord progressions. The preliminary evaluations left the impressions of some new ideas were not worthwhile, while others such as using a *remove factor* for choosing removed genomes showed somewhat promising results. Additionally, gradient child fitness sharing was shown to at least work since XOR was still solved (and in an arguably low number of evaluations through all tests).

Additionally, a representation for chords was developed which seems to produce decent output in many situations. The output was easy for a neural network to produce, and was able to be mapped to full chords. Additionally, the representation is somewhat more flexible than those used in the past, yet it can be scaled back based to make an easier task as was done for our tests.

Finally, the system was evaluated in learning to produce a few simple chord progressions. Results indicated that this was achievable by the system, though not consistent. More promising, when examining chord progressions made by mutations on a learned progression, pleasing results were found. These results demonstrated there is good potential for this system to an be effective interactive evolution system, although there is still needed improvement in the time and ability to learn chord.

References

[1] J. Bogenschutz. A conversation with arranger and composer john bogenschutz., 2006.

[2] D. Cope. *The Algorithmic Composer*. A-R Editions, Inc., Madison, WI, 2000.

[3] D. Eck and J. Schmidhuber. A first look at music composition using lstm recurrent neural networks.

[4] J. A. Franklin. Recurrent neural networks and pitch representations for music tasks. In *FLAIRS Conference*, 2004.

[5] T. Reil and P. Husbands. Evolution of central pattern generators for bipedal walking in a real-time physics environment. *IEEE Transactions on Evolutionary Computation*, 6(2), 2002.

[6] K. O. Stanley, B. D. Bryant, and R. Miikkulainen. Real-time neuroevolution in the nero video game. *IEEE Transactions on Evolutionary Computation*, 9(6), 2005.

[7] P. Todd and G. Werner. Frankensteinian methods for evolutionary music composition in griffith, 1999.

[8] P. M. Todd. Neural networks for applications in the arts. In M. Scott, editor, *Proceedings of the Eleventh Annual Symposium on Small Computers in the Arts*, pages 3–8, Philadelphia, PA, 1991. Small Computers in the Arts Network, Inc.