Evolving Musical Scores Using the Genetic Algorithm

Adar Dembo
3350 Thomas Drive
Palo Alto, California 94303
adar@stanford.edu
(650) 494-3757

Abstract: This paper describes a method for applying the genetic algorithm onto musical scores, causing them to evolve to better match a target musical score. The encoding used is outlined, followed by a discussion of the tradeoffs of using this specific encoding. The method of evaluating fitness is also explained, followed by several examples from well known musical scores and runs of the GA based on them. The paper concludes with a look towards the future, and possible modifications that would improve this method.

Introduction

We have studied at length how the genetic algorithm can be applied to engineering and scientific problems, such as the 10-member truss problem (Goldberg and Samtani 1986), the hamburger problem, the antenna design problem (Altshuler and Linden 1998), and others. However, not once have we tried to apply the genetic algorithm to concepts closer to our lives. That is, to a problem that may be applicable for all of us, not just to a certain branch of science. It is this motivation that brings us to the problem I have chosen to analyze. Suppose we have a musical score. Is it possible to use the genetic algorithm on a certain initial population to evolve "songs" that better and better match this target musical score? Is it possible to create a perfect individual, using the GA, that will perfectly match the target score? For that matter, how does one represent a musical score in such a way that a genetic algorithm can understand?

I chose to tackle these questions, again, because music is a field that everyone can relate to and understand, and also because I've always had an affinity for bridging my love for music with my love for computing and programming. In the past, I had done this by working for Napster, and for learning as much as possible regarding the file structure of an MP3. Now, I can apply the genetic algorithm to a musical score and once again connect music and computer science.

Structure – First Attempt

With the problem decided upon, I had to implement an encoding for a piece of music. I began by searching online if such a problem had been attempted before. I came across a language called "abc" that is used to represent music in simple ASCII (Walshaw 2002). In abc, a music is written in a line of characters, preceded by a header that specifies certain default values for this line. For example, the header may specify the "default length" for any given note, the musical key that the line is in, the meter, or the tempo. Then, the line contains different letters representing the pitches that the notes are found in (A, B, C, D, E, F, or G). If the pitch is an octave higher or lower, the note is represented in lower case, or appended with a comma, respectively. If a note's duration is modified (in relation to the default specified in the header), a division sign followed by the fraction of the default note is used, and if the duration is amplified, a number indicates by how much. Coupled with pipe signs to indicate the end of a measure, it is possible to represent nearly any kind of musical score (Fig. 1), making abc a very versatile yet simple language.

Unfortunately, converting abc to something the GA can work with is nigh to impossible. Using abc, variations in pitch cause the length (in characters) of a note to grow (by adding a comma, for example). This causes a representation with fixed-length chromosomes to fall apart. Furthermore, a representation in binary would be difficult, as a toggle of a bit from 0 to 1 (or vice versa) is not guaranteed to be safe. The encoding would degenerate into dozens of special case checks, which would be very difficult to account for. It is for these reasons that although I found abc to be a useful format, I had to create my own.

Structure – Second Attempt

In creating my own encoding, I had to decide how the structure, generally speaking, will be viewed. I chose to represent each "event" in a song as an object with a fixed length. Events can include nearly anything in music, such as notes, rests, time signatures, measure bars, repeat signs, fermatas, etc. Using this method, it is possible to classify every piece of musical notation into an object. Then, one can use a binary chromosome to differentiate between each type of notation. For example, one bit can toggle whether the event is a note or a rest. Another can toggle whether the note is slurred or not. With this approach, creating an encoding for the structure of a note or rest is easy, and an entire score can be created by stringing together multiple notes and/or rests. However, the tradeoff lies in evaluating the fitness for the musical score. With such a random assortment of "events" globbed together, evaluating fitness can degenerate into multiple special case analyses, quickly slowing down any genetic algorithm in terms of processor power. So, while the encoding created was efficient and clean, I was forced to examine only a small subset of musical ideas in my algorithm.

More specifically, each chromosome was composed of seven bits. The first two bits marked the rhythm, which, having four values, was either a whole note, half note, quarter note, or eighth note. The next bit marked if the note was slurred or not (1 is slur, 0 is not). The final 4 bits represented two octaves of pitches, beginning on middle C and continuing to the C two octaves above. These two octaves take only 15 out of the possible 16 arrangements these four bits can make, so the 16^{th} arrangement (1111) represents if the note is actually a rest. If it is a rest, in evaluating the note, the slurred value is ignored.

To better understand the structure, let me give some examples:

- An eighth note, unslurred with notes around it, on a middle C, is 1100000.
- A quarter note, slurred, on the B above middle C, is 1010110.
- A whole rest is 0001111.
- A half note E an octave above middle C slurred to a quarter note G an octave above middle C is 0111001-1011011.
- A scale in quarter notes from middle C to an octave higher would be represented by Figure 2.
- "Mary had a little lamb" would be represented by Figure 3.

Fitness Measure

With the encoding taken care of, it was now time to decide how to measure the fitness for a given song. The most important thing to remember is that to the fitness of an evolved score can only be measured in comparison to another musical score. We are measuring how "alike" the two scores are. This is keeping in line with the original problem, which was to create a method in which songs can be evolved to match a target song. It is also important to note that each musical score consists of 7-bit chromosomes (notes), with between 1 and infinity (theoretically) of these chromosomes stuck together. Thus, the problem of deciding how alike two scores are exists on two levels: How alike a note is to another note, and how alike the entire song is to the other song. Because songs have different amounts of notes, some kind of recursive edit distance solution is necessary. To illustrate an example of why this is the case, suppose there existed two pieces of music, nearly identical by nature, except in between the first and second notes of one of them existed an extra note. Thus, the two pieces are identical, except there exists a frame shift in one of them, causing all notes to be shifted down one spot, making them no longer line up with their counterparts in the other score. Thus, we cannot simply compare pairs of notes side by side to establish a reasonable fitness; we must calculate the edit distance needed to transform one song into another.

To achieve the goal outlined above, I used an algorithm established by known as the Levenshtein Distance, or edit distance. This algorithm calculates the minimum path necessary to transform one string into another by using a combination of insertions, deletions, and substitutions. It is recursive by nature, and highly expensive (in terms of CPU power). It was fairly difficult to adapt the algorithm to compare songs rather than strings, but it was do-able. The LD provides the raw fitness measure for a song compared to another song. To calculate the normalized fitness, this raw fitness measure is applied as the numerator in a fraction

where the denominator is the “theoretical maximum distance” that the song that is furthest away from the target has. This “theoretical maximum” relies on the fact that for any given note, there exists a note we can deem “furthest”, and assign the highest raw fitness for that note. Coupled with the length of one song compared to the length of the other, we can calculate the most distant song to any other song, and thus achieve the maximum raw fitness. When the actual raw fitness is divided by the maximum raw fitness, a normalized fitness is achieved, with 1 being the worst fit, and 0 being the best fit.

Other parameters

Because the Levenshtein Distance algorithm is so expensive, and because the actual building, comparing, and modifying songs exists in high-level but expensive strctures, I had to limit both the scope of the project (as explained earlier), and the evaluation parameters. The population size was chosen to be 50. A high population size would be optimal, given the variety of the potential songs (2^7 possible chromosomes, with an undetermined amount of notes). The number of generations was likewise chosen to be 50. Both of these values were as high as I could allow without causing the computer I was working on to slow to a halt. The termination criteria was chosen to be the arrival at generation 51, or the rise of a perfect individual. Finally, the results were deemed the “best of run” individuals. The tableau for this GA is summarized in Figure 4.

Size of Search Space

This problem has both an undefined and a defined search space. On one hand, each chromosome is 7 bits long, giving only 2^7 = 128 note possibilities. On the other hand, there can be an infinite number of notes strung together in a single score. Due to CPU constraints, “Mary had a little lamb” was essentially the longest score I could attempt, with 26 notes. Because of that, I limited the number of notes in a score to 30. This gives $\Sigma(2^{7*N})$ (N = 1, 2, 3…30) possibilities, which is still an extraordinarily high amount of possible songs.

Genetic Operators

The three normal operators were applied to each individual of the population. Reproduction was executed in a deterministic tournament style. Crossover was chosen to be 0.8, and mutation was chosen to be 0.01. Thus, nearly every individual was crossed over in some area with another individual, and, for long individuals, mutation for one bit was almost assured. Because of the way the encoding was structured, there were no cases where certain individuals became illegal. Every bit toggle was legal and produced a perfectly viable note or rest. We are very lucky for that, because special cases to repair or discard damaged individuals would result in the algorithm being even more expensive than it already is.

The operators also give access to the entire search space. Given that the maximum score length is 30, and that the number of individuals in the population is 50, it is very likely that there will be individuals from long and short lengths. Thus, through crossover, we can ensure that even the longest patterns remain alive. Mutation allows even the more obscure bit patterns to be accessed, guaranteeing that rests do occur from now and then. Finally, fitness based reproduction will make sure that as a whole, our population is moving towards the more fit individual pool.

Methodology

By this point I had assembled all the necessary ingredients for a working genetic algorithm. The variable-string length fixed-chromosome length encoding was in place, ready to be used. The evaluation function pseudocode was outlined. For variable-length GA’s, the GENESIS software we used in Problem Set 3 would not work, as it works only for fixed-length GA’s. The only apparent alternative was to use ECJ8, a Java-based GA program that could handle variable-length GA’s. This presented several problems; the fundamental one being that I do not know Java. Nevertheless, I acquired ECJ8 and attempted to install it on one of Stanford University’s servers. The servers were both SPARC based and x86 based, running either Solaris or SULinux. In both cases, I failed to install the software. The java compiler would not compile the

entire software suite, citing unresolved symbols and other errors. This occurred even after I followed the directions provided on the ECJ8 webpage to the letter. Not discouraged, I tried to install ECJ8 on my own server, a Debian Linux based machine. The advantage to using my own server is that I can install whatever software needed without needing the permission of others. I installed the JDK and set the appropriate environmental variables, yet ECJ8 would still not compile. I then moved on to my Windows 2000 computer, with the same results. Each time the Java compiler would report me different error messages, yet fatal ones nonetheless. I tried with a different Java compiler (jikes instead of javac) with no luck. At this point, I gave up on ECJ8 and scoured the net looking for an additional resource. I did not want to convert my variable-length encoding into a fixed-length one; that would severely handicap my structure, not to mention be difficult to accomplish well. Finally, I came across EO, an Evolving Objects library built in C++. Not only did it install perfectly on my Debian Linux server, but it was written in C++, a language that I had total familiarity with. Furthermore, all of its classes were in one way or another derived from STL, which made it even easier to use. The version of EO that I used was 0.9.2. The tutorials explained how the system was put together, and not long after I had created two classes (note and song) to represent my encoding scheme. The evaluation function was written, and the parameters were inputted.

Before I launch into the results, I will give further information about the server that I used. This was a Debian Linux (testing distribution) based server, running a 2.4.18 kernel with a low latency patch applied. The motherboard was an ASUS CUR-DLS (Serverworks LE chipset), with two Pentium III-933's and 512 megabytes of ECC SDRAM. The server was running many additional CPU consuming processes at the time, but they were all on only one of the CPU's, so I had an entire P3-933 at my disposal.

Results and discussion

Unfortunately, EO only shows the user the initial population and the final population of the GA, so I was unable to create a graph showing fitness increase over time. Furthermore, because EO was written in C++ and STL, the LD was even more expensive than I originally had predicted. Running EO with a max number of chromosomes at 30 (in order to compare the results to "Mary had a little lamb") proved to be impossible, as it took nearly 5 minutes for the LD to compute one fitness measure! This forced me to again limit the scope of the algorithm. I instead ran it on the C scale indicated in Figure 2. This scale consisted of 8 notes, so I set the maximum number of notes to 10. With this done, the algorithm was able to compute approximately five LD's every second, bringing the entire GA time to about 45 seconds (since more fit individuals took less time to LD). I have included a snippet of the initial and final populations in Figures 5 and 6 respectively. It is very apparent that the initial population is comprised of completely random songs, some one note long, others far more. It is also clear that by generation 50, a group of perfect individuals has risen. They match the target song exactly. During the actual GA run, as the individuals got more and more fit, the LD algorithm began taking less and less time to compute. This is because one of the main base cases of the LD recursive algorithm is that if two notes are equal, run the LD on the remainder of the song without those two notes. In fit individuals, this can shorten songs to far less notes than the original songs contained.

I had doubts that the LD algorithm could really be converted effectively from strings (for which it was designed) to songs. However, their structures are really more or less the same. Each song is composed of a variable number of notes, just as a string is composed of a variable number of characters. The only difference then, between LD for strings and LD for songs is that in an LD for strings, any two characters are either equal (0 distance) or not equal (1 distance). In an LD for songs, each note has a variable distance from another note, a distance that cannot be expressed in 0/1 terms alone. This is the shortcoming I had to account for in transcribing the algorithm for songs. It was also interesting to note that, like most other GA's, the amount of memory used by the program was very small, in fact, under a kilobyte of memory. The bottleneck lay in the CPU usage, since the GA is comprised of thousands upon thousands of iterations.

Conclusion

This paper has demonstrated that, although musical scores are extremely complex, they can be broken down into smaller components that still represent the whole. By encoding a musical score as a list of

"events", one can quantitatively assess how alike two pieces of music are. With this measure, it is possible to construct a genetic algorithm that will evolve a random initial population of songs to match a target song. Using a modified Levenshtein Distance algorithm in conjunction with fitness measurements, one can account for both frame shifts and note differences in a piece of music, and thus assess its structure completely and entirely. The end result is a genetic algorithm that is not only elegant, but also applicable to everyone's lives, and is not just an application of engineering or science.

Further Work

There is much further work to be done on this project. Firstly, there are a multitude of additional events to be represented in the encoding. Repeat signs, fermatas, additional octaves, sharps, flats, staccatos, accents, additional rhythms, time signatures, tempos, playing styles, grouped notes, and dotted notes are just a few. The project's scope can scale to whatever dimension one can see fit: It can analyze simple children's melodies, piano etudes, or an entire opera, if one is willing to put that much work into it. Perhaps it is a viable project for a future PhD thesis ☺. Furthermore, the actual structure can be adapted to better much a more versatile structure, such as abc. As it stands, music that is formed is very disjointed, and can be even invalid depending on the specified format. For example, if we chose to include time signatures, each measure would require a certain number of beats, a requirement that we cannot hope to achieve with our current "random rhythm" technique.

Beyond the structural realm, much can be done to analyze the closeness of two pieces of music. We can examine the intervals rather than the actual pitches, we can differentiate between whole steps and half steps. We can look into dissonances and consonances as a fitness measure. We can compare the hertz values of pitches rather than their notation. Beyond the realm of musical scores altogether, we can extend the comparison to a specific file format, such as WAV, MP3, or MIDI. Each of these contain a specification that could probably be translated into an encoding.

In the realm of code, it is painfully apparent that the GA and associated code used in this project need to be as tight and efficient as possible, given how expensive the algorithms that evaluate fitness are. An ideal GA would be coded in low-level C, with very little overhead for classes or structures. Memory allocation should be kept to a minimum, and all complex functions should minimize their O(n) running time. Of course, a very powerful processor would be optimal as well, or a multi-threaded GA that could distribute its load over several processors. Either way, to run a music evolving GA on any large piece, such as Mozart's Clarinet Concerto (thousands of events), would require some severe overhauls in the equipment used, whether code or hardware.

Acknowledgements

Scott Cruzen for the concept and original discussion.
Sean Montgomery for assistance and further brainstorming.
Darren Lewis for convincing me that it wasn't impossible.

References

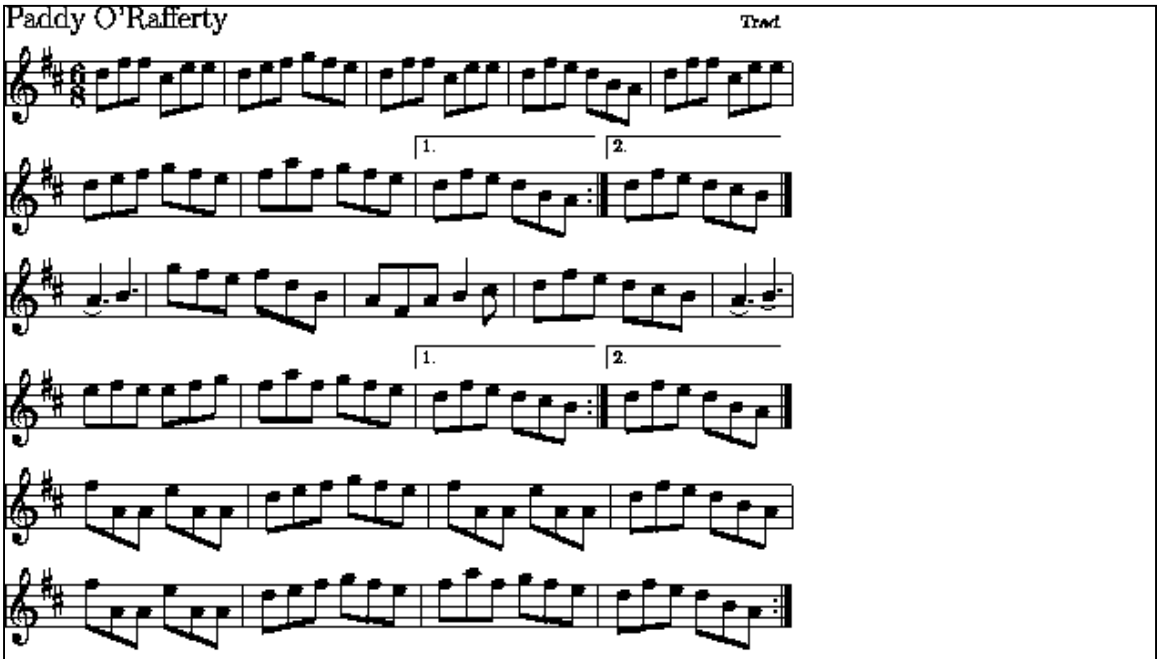
Goldberg, David E. 1989. *Genetic Algorithms in Search, Optimization & Machine Learning*. Addison Wesley Longman, Inc.

Walshaw, Chris. http://www.gre.ac.uk/~c.walshaw/abc/ *the abc musical notation homepage*. Last updated March 12, 2002.

Merelo, JJ. http://eodev.sourceforge.net *EO Evolutionary Computation Framework*. Last updated March 6, 2002.

Luke, Sean. http://www.cs.umd.edu/projects/plus/ec/ecj/ *ECJ*. Last updated 2001.

Zelenski, Julie. Recursive Levenshtein Distance Problem. *CS106X Fall Practice Handout*. December 5, 2001.




```
M:6/8
O:I
R:J

X:1
T:Paddy O'Rafferty
C:Trad.
K:D
dff cee|def gfe|dff cee|dfe dBA|dff cee|def gfe|faf gfe|1 dfe dBA:|2 dfe dcB|]
~A3 B3|gfe fdB|AFA B2c|dfe dcB|~A3 ~B3|efe efg|faf gfe|1 dfe dcB:|2 dfe dBA|]
fAA eAA|def gfe|fAA eAA|dfe dBA|fAA eAA|def gfe|faf gfe|dfe dBA:|
```

Figure 1: Sample song and its translation into abc.

1000000
1000001
1000010
1000011
1000100
1000101
1000110
1000111

Figure 2: A quarter-note scale in C in my encoding scheme

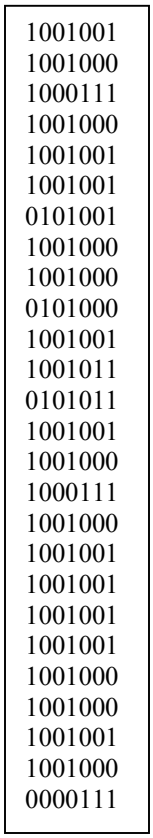


Figure 3: "Mary had a little lamb" in my encoding scheme

Objective:	Find the globally optimum musical score to match a target musical score.
Representation Scheme:	• Structure = Variable length strings consisting of fixed length chromosomes. • Alphabet Size K = 2 (binary) • String Length L = 7 (per chromosome) • Mapping = Each chromosome represented either a note or a rest, with the first 2 bits designating rhythm, the 3rd bit designating slur, and the 4th-7th bits representing pitch or presence of a rest.
Fitness cases:	Only one (comparison of target song to source song).
Fitness	• Raw fitness = How "distant" a song is from the target. • Normalized fitness = Raw fitness divided by "maximum theoretical distance" for the target song.
Parameters:	• Population size M = 50 • Max number of generations G = 50
Termination criteria:	The GA has run for G generations, or a perfect individual is found
Result designation:	The best-so-far individual in the population.

Figure 4: Tableau for the genetic algorithm

```
0.618056  56 00000010101101010111011111011010101111001001011001010011
0.701389  21 110011101101111100001
0.625  56 01100110011000110100111000010011001000011100100001010001
0.611111  49 0001001000010011011110110110011001101100011110011
0.576389  21 011111101011000000001
0.604167  56 00011000010111010101111100111001101110001101110000001011
```

Figure 5: Excerpt from initial population of GA run.

```
0     56    10000001000001100001010000111000100100010110001101000111
0     56    10000001000001100001010000111000100100010110001101000111
0     56    10000001000001100001010000111000100100010110001101000111
0     56    10000001000001100001010000111000100100010110001101000111
0     56    10000001000001100001010000111000100100010110001101000111
0     56    10000001000001100001010000111000100100010110001101000111
0     56    10000001000001100001010000111000100100010110001101000111
0.02333    56    10000001000001110001010001011100110010001010001101000111
```

Figure 6: Excerpt from final population of GA run.