

# Automated Musical Part Writing

Kevin Deisz

05/27/2009

## **Abstract**

This program writes music. Simply put, that is actually a relatively accurate summary of what this program does. Through a series of pseudo-randomly selected chords, user-generated constraints and constraint satisfaction code, this program writes music. The output is handled by a very handy programming language called lilypond, that engraves the generated musical progression into an outputted png and pdf. The output is also ported to a midi file so the user can hear what the program just wrote.

## **1 Introduction and Background**

### **1.1 Variations and Themes**

In 1680, a man named Johann Pachelbel came up with a theme written for three violins and a string bass. That theme was then arranged for a wide variety of other ensembles, but the most important part was the theme itself. Known as the Canon in D major, the sequence of notes became the basis for many pieces of the era. Handel used it for the main theme in the second movement of his Organ Concerto No. 11 in G minor, Mozart used it in Die Zauberflöte, and even Haydn used it (albeit many years before) in his Opus 50 No. 2. This progression can be found in many modern pieces; unwitting artists perform this progression the unknowing crowds almost every day. Such artists as Matchbox 20, Avril Lavigne, Green Day, Blues Traveler and Aerosmith, to name a few. As demonstrated by this piece, it is not only the progression that separates pieces, but the overlying, more ornamented parts.

What this program attempts to do is to show that the same progressions can produce entirely different, random music. That even though the pieces may be quoting each other, the generated output is actually almost entirely unique. Therefore, this program writes its own music from user-specific variables, so as to allow the user to create unique music that may or may not be based off of others work.

### **1.2 Background**

Many people have attempted (and successfully too), to create almost this exact same project. Two examples are listed below.

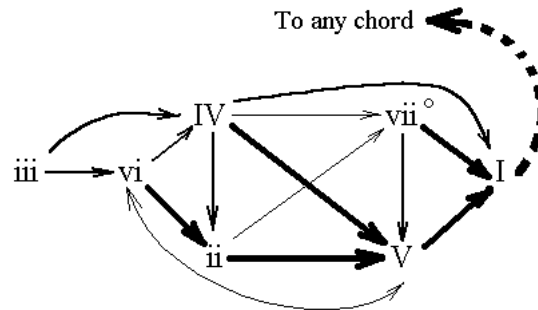
1. Microsoft at one point created software that would take input from a microphone, and then create harmonies and a chordal progression that would play beneath the singer, as a sort of back-up music [7]. Their product was called "Songsmith", and it involved simply singing into the mic, and they advertised that it would do the rest for you. The product recieved mixed review, with some articles saying that the quality of the music was very poor, but that that was probably a good thing, because music being generated by a computer being as good as that of a human would be dangerous.
2. A man by the name of Bruce Jacob produced a product called "variations", which he used to write music for him. It is viewable on his webpage, along with all of its source code and the scant documentation he came up with [3]. The program goes by a set of constraints that he set up to model his own style of composition, which he then filters through a program he called the "ear". This ear program is actually a genetic algorithm designed to filter out music that he himself doesn't like. Obviously this program is very specifically designed to his tastes, but this is pretty much what I would be doing anyway. In reality, this is the embodiment of my project, simply put to a different genre. Jacob also came out with an essay on the subject of using algorithmic composition as a model for creativity, which ended up being very influential in how I looked at this project [2].

In addition to the products, there are many people and organizations that have written numerous detailed reports on the subject of algorithmic composition. Florida International University came out with a great essay on the subject [1]. Stanford came out with a very helpful "Brief History of Algorithmic Composition" [6]. They also compiled an extensive bibliography in which can be found many useful resources on the subject [5]. In short, this is a subject that has been studied extensively, so I am only just brushing the tip of the iceberg with this project.

## 2 Process

### 2.1 Bass Progression

The first step in the development of this project was to create a working bass progression that would follow a general map of chords. This map of chords was taken from the Tonal Harmony textbook, and works only for major progressions. I then translated this map into a CSV file, where the first item of each line was the number of the chord (called scale degrees), and every other item in that line was a scale degree that that chord could progress to. All of these chords are then read in, processed, and added to a progressions array that is returned from the `readin()` method.



### Major Key Chord Progressions

I ran into a problem when I began this process, because writing the chord progressions in the forward direction would cause the program to eventually run out of options if one of them was determined to not work (later in the program). So, I created another array that held all of the progressions backwards. The program would then start from the end of the music and write backwards, and this actually solved the problem. There were many more options this way, and it was much more evenly distributed between the scale degrees.

The next step, following the creation of the progression array, is to pass through the `step()` method  $x$  number of times, where  $x$  is the user-specified integer that was given at the very beginning. The incredibly simple `step()` method simply takes the array of possible neighbors from the progressions array and returns a random neighbor. This ensures that each and every composition is unique, so as not to make it a formula. However, this does pose some problems as to how to make the piece fit together as a whole, because at some point there needs to be a repeated motif, and currently returning a random chord does not lend itself well to this eventual goal. Obviously this is a place for further work to be done.

## 2.2 Domains

Once the bass progression has been established, the domains for each chord for each voice can be established. For the most basic music, without deviating from the specific chordal tones, in each chord each voice can only be one of three notes. These notes are scale degrees one, three and five, but these can be in any octave. So, for each chord, each voice has an array of possible values, and those values are filled with the first, third and fifth scale degree, repeated for about three octaves. This gives each voice a relatively large domain, and the constraint satisfaction code a slightly larger area to work in, allowing the code a little more freedom, meaning that it caused fewer errors.

The domains themselves are created differently for different genres. In jazz, for example, the scales are completely different, and the progressions are determined entirely differently. Obviously Bach and Benny Goodman composed with different approaches. In order to bridge this gap and make the code more

general, I've created a command-line argument for the `musicwriter.py` file that takes the filename of a python file and pulls the methods from it (also in Appendix A). The methods are named the same in each file so that it will not have to change names, and this way multiple files can be created in the future. Hopefully in the future this could be a web-based program, and these files could be generated by the user. For now however, they are all created manually, and the only one created so far is `voice.py`, which writes for four voices.

## 2.3 Constraints

The constraints, which are the basic partwriting rules broken down into their own binary components, are relatively simple to implement. There is only one constraint function, and the output is a boolean on whether or not the attempted chord was valid. The four inputs are voice A, note a, voice B, note b. The voices are simply integers from 0 to 3, signifying soprano through bass. The notes are numerical values which are indexes in the `notes` array.

The first, and probably most important constraint so far, is that a higher voice cannot have lower notes than a lower voice. Other constraints include parallel fifths, which mean that if two voices are four notes apart, the next chord cannot see the same two voices at the same interval. Parallel octaves are another, which is the exact same as parallel fifths, except that they are seven notes apart. A more interesting constraint, that resulted in some peculiar learned behavior, was that the voices could not be more than one octave apart. In the beginning (before the domains were properly implemented), because of this constraint, the entire piece would go as low as it possibly could. The last constraint, which is still not done being written and implemented, is that each chord must have a first degree (called the root), a third, and a fifth. This constraint is going to be the most difficult to implement, because not long is everything binary. In order for this to work, the note has to be linked to not only its neighbor, but also every other note in the chord. This involves ternary constraints, which will take quite a bit more research before they are implemented.

In order to link all of the voices together, and to make sure that they follow all of the rules created by the constraints, a map is created, and each voice is added to the map. Each voice is then given an array, that contains other mini-arrays that are simply pairs. Each pair contains a neighboring voice (which in this case, neighboring means any and all), and a pointer to the constraint function. This way, when it is time to choose the notes, the code need simply parse through this array, pull out the neighbor, and pass the neighbor and its attempted note to the constraint function dictated by the pointer.

## 2.4 Constraint Satisfaction

Once the domain and constraint maps are created, it is time to pass everything to the constraint satisfaction code. This code is actually entirely generic, and has many other uses. A new map is created, called `assignments`, which will be the output. This blank map is then passed to the main method in the `csp` file

(Appendix A, under `csp_solver_4.py`), because the main method is a recursive method that will stop when all of the assignments have been made. The code itself implements its own relatively complex algorithms that are separate from the actual music writing. These include a least-constraining value function, which chooses the note that will leave the most options up to its neighbors, a function that sorts by the amount of unassigned neighbors, and a few other tricks to make it execute more quickly.

## 2.5 Ornamenting the Output

For now, this process is relatively simple, but in the future this is going to be the portion of the process that makes the music the most unique. This is the portion of the code that will add the non-chordal tones (NCTs), which are the notes not defined by the bass progression (i.e. not the root, third or fifth). Currently, the only NCTs that have been implemented are passing tones, which are notes that are added between other chordal tones to produce a scale-like effect. They are generated if a voice makes a jump over one note, C to E for example. The `ornament()` method would then make the C an eighth note, add in a D eighth note proceeding the C, and then end with an E quarter note. This makes the music far more interesting, as up until this point there was no variance in the rhythm. In the future, other NCTs will be added to make it more interesting, like *appoggiatura*, escape tones, and suspensions.

As of now, the only rhythmic deviation from straight quarter notes is described above, except for the very end. The `add_ending()` method takes the array and very simply makes the last chord a whole note. This adds to the music a sort of definitive ending that allows the listener to know that is the point at which to clap. In the future, other rhythmic devices could be implemented as well, such as syncopation and hemiola.

## 2.6 Output

Once all of the bass progression is decided, the constraints and domains established, everything passed through the constraint satisfaction and the output ornamented, it is finally time to print out the result. Thankfully, there is a programming language much like LaTeX for music, called *lilypond*. *Lilypond* has sections defined by back slashes, that allows the user to very simply output their music with minimal effort. As is, the `printout` method works wonders, which takes all four voices, and prints them out individually. The top two voices are printed out in the upper staff, which is housed in the treble clef, while the lower two voices are written in the bass clef, in the lower staff. This paper will not go into the actual syntax of *lilypond* files, but you can see the output in Appendix B.

*Lilypond* thankfully has a helpful utility that allows the user to also output a midi file that contains the generated music. This means that every time this program is run, it generates the sheet music, and then a midi file that plays the sheet music. Since it is the same naming convention every time, it was very

easy to make a webpage in which to view this output. And so, a simple html page was written to house this project, the output and the midi.

### 3 Results

The outputted music was surprisingly complex. The program generated relatively good music, though it does sound a bit formulaic. While one could discover general themes throughout the music, its very clear that these are completely circumstantial and not intended upon. The next major step in this research is going to be how to have the program generate a general theme for the piece and elaborate on that, because currently the randomness weighs on the listener. Additionally, because the triads (chords with a root, third and fifth) have not been fully implemented, at times the music seems empty in the middle parts.

## 4 Appendix A - Code

### 4.1 musicwriter.py

```

1 #!/usr/bin/python
2 # Kevin Deisz
3
4 import random, os, csp_solver_4
5 num_chords = 128
6
7 chords = [ 1 ]
8 current_chord = []
9
10 note_names = [ None, "a", "ais", "b", "c", "cis", "d",
    , "dis", "e", "f", "fis", "g", "gis", "a", "ais",
    "b", "c", "cis", "d", "dis", "e", "f", "fis",
    "g", "gis", "a", "ais", "b", "c", "cis", "d", "dis", "e",
    "f", "fis", "g", "gis", "a", "ais", "b", "c", "cis", "d", "dis",
    "dis", "e", "f", "fis", "g", "gis", "a", "ais", "b", "c",
    "cis", "d", "dis", "dis", "e", "f", "fis", "g",
    "gis", "a", "ais", "ais", "b", "c", "cis", "d", "dis",
    "dis", "e", "f", "fis", "fis", "g", "gis", "a", "ais",
    "ais", "b", "c", "cis", "cis", "d", "dis", "dis", "e",
    "f", "fis", "fis", "g", "gis", "gis", "a", "ais", "ais",
    "ais", "b", "c", "c", "c" ]
11
12 def constraint_func( A, a, B, b ):
13     global current_chord
14     if (a<b and A>B) or (a>b and A<B) or a==b:
15         return False
16     if abs(A-B)==1 and ( abs(a-b)>12 ):
17         return False
18     if A!=0 and B!=0 and ( abs(current_chord[A]-current_chord[B])
19         ==24 and abs(a-b)==24 ):
20         return False

```

```

20     if A!=0 and B!=0 and ( abs(current_chord[A]-current_chord[B])
21         ==12 and abs(a-b)==12 ):
22         return False
23     if ( abs(current_chord[A]-current_chord[B])==7 and abs(a-b)==7
24         ):
25         return False
26     return True
27
28 def findPossibles( degree ):
29     if degree == 1: return [ 4, 8, 11 ]
30     elif degree == 2: return [ 1, 6, 9 ]
31     elif degree == 3: return [ 3, 8, 11 ]
32     elif degree == 4: return [ 1, 4, 9 ]
33     elif degree == 5: return [ 3, 6, 11 ]
34     elif degree == 6: return [ 1, 4, 8 ]
35     elif degree == 7: return [ 3, 6, 9 ]
36
37 def limit_domains( domains ):
38     for x in domains[0]:
39         if x<20:
40             domains[0].remove( x )
41     for x in domains[1]:
42         if x<28:
43             domains[1].remove( x )
44     for x in domains[2]:
45         if x<36:
46             domains[2].remove( x )
47     for x in domains[3]:
48         if x<44:
49             domains[3].remove( x )
50     return domains
51
52 def readin( filename ):
53     file = open( filename, "r" )
54     temp = file.read().split('\n')[:-1]
55
56     progressions = {}
57     for i in temp:
58         item = i.split( ',' )
59
60         item2 = []
61         for j in item:
62             item2.append( int( j ) )
63         progressions[ item2[0] ] = item2[1:]
64     return progressions
65
66 prog_back = readin( "prog_backward.csv" )
67 prog_forw = readin( "prog_forward.csv" )
68
69 def step( progressions, curchord ):
70     neighbors = progressions[ curchord ]
71     return neighbors[ random.randint( 0, len( neighbors )-1 ) ]
72
73 def write( current, next, index ):
74     global current_chord, chords
75     current_chord = current
76
77     possible = findPossibles( next )

```

```

75     options = []
76
77     min, max = current[0], current[3]
78     for opt in possible:
79         for x in range( 1, 6 ):
80             val = opt+x*12
81             if val > 59:
82                 continue;
83             if val>min and val<max:
84                 if val not in options:
85                     options.append( val )
86             elif abs( val-min ) < 8:
87                 if val not in options:
88                     options.append( val )
89             elif abs( val-max ) < 8:
90                 if val not in options:
91                     options.append( val )
92 #     print "%i:" % (next),
93 #     print options
94 #     print current_chord
95
96     domains = {}
97     for x in xrange( len( current ) ):
98         domain = []
99         for y in options:
100             if abs( current[x]-y ) < 4:
101                 domain.append( y )
102         domains[ x ] = domain
103     domains = limit_domains( domains )
104
105     grid, constraints = {}, {}
106     for x in xrange( 4 ):
107         grid[ x ] = []
108         for y in xrange( 4 ):
109             if x!=y:
110                 grid[ x ].append( y )
111     for node in grid:
112         constraints[ node ] = []
113         for neighbor in grid[ node ]:
114             constraints[ node ].append( [ neighbor, constraint_func
115 ] )
116
115     assignments = {}
116     if csp_solver_4.backtracking( domains, assignments, constraints
117 ):
118         retval = []
119         for x in xrange( len( assignments ) ):
120             retval.append( assignments[x] )
121         return retval
122     else:
123 #         next = step( prog_forw, chords[index] )
124         return write( [28,35,40,44], next, index )
125
126 for x in xrange( num_chords-1 ):
127     chords.append( step( prog_back, chords[x] ) )
128 chords.append( 1 )
129 chords.reverse()

```



```

130 #print chords
131
132 parts = [ [28,35,40,44] ]
133 for x in xrange( len( chords )-1 ):
134     parts.append( write( parts[x], chords[x+1], x ) )
135 #print parts
136
137 fh = open( "output_numbers.txt", "w" )
138 for x in xrange( len( parts ) ):
139     for y in xrange( len( parts[x] )-1 ):
140         fh.write( "%s," % ( parts[x][y] ) )
141     fh.write( "%s\n" % ( parts[x][ len( parts[x] )-1 ] ) )
142 fh.close()
143
144 os.system( "python parse.py" )

```

## 4.2 parse.py

```

1 import random, os
2
3 note_names = [ None, "a,,", "ais,,", "b,,", "c,,", "cis,,", "d
,," , "dis,,", "e,,", "f,,", "fis,,", "g,,", "gis,,", "a,,", "
ais,,", "b,,", "c,,", "cis,,", "d,,", "dis,,", "e,,", "f,,", "fis,,",
"g,,", "gis,,", "a,,", "ais,,", "b,,", "c,,", "cis,,", "d,,", "dis,,", "e,,",
"f,,", "fis,,", "g,,", "gis,,", "a,,", "ais,,", "b,,", "c,,", "cis,,", "d,,", "
dis,,", "e,,", "f,,", "fis,,", "g,,", "gis,,", "a,,", "ais,,", "b,,", "c
'", "cis'", "d'", "dis'", "e'", "f'", "fis'", "g'", "gis'", "
gis'", "a'", "ais'", "b'", "c'", "cis'", "d'", "dis
'", "e'", "f'", "fis'", "g'", "gis'", "a'", "ais
'", "b'", "c'", "cis'", "d'", "dis'", "e'", "
f'", "fis'", "g'", "gis'", "a'", "ais'", "b
'", "c'" ]

4
5 def getnote( num ):
6     return note_names[ int( num ) ]
7
8 file = open( "output_numbers.txt", "r" )
9 lines = file.read().split( "\n" )[:-1]
10 file.close()
11
12 parts = []
13 for x in xrange( len( lines ) ):
14     parts.append( lines[x].split( "," ) )
15
16 parts2 = []
17 for x in xrange( len( parts ) ):
18     line = []
19     for y in xrange( len( parts[x] ) ):
20         line.append( getnote( parts[x][y] ) )
21     parts2.append( line )
22
23 def separate( parts ):
24     retval = []
25
26     for part in xrange( 4 ):
27         line = []
28         for chord in parts:

```

```

29         line.append( chord[part] )
30     retval.append( line )
31     return retval
32
33 def addPassingTones( part ):
34     retval = []
35     need4 = False
36
37     for note in xrange( len( part )-1 ):
38         first , second = getnum( part[note] ), getnum( part[note+1]
39             )
40         if abs( first-second )==3:
41             if random.random() < 0.5:
42                 retval.append( "%s8" % ( part[note] ) )
43                 if first > second:
44                     retval.append( "%s" % (getnote( getnum( part[
45                         note] )-1 ) ) )
46                 else:
47                     retval.append( "%s" % (getnote( getnum( part[
48                         note] )+1 ) ) )
49                 need4 = True
50             else:
51                 if need4:
52                     retval.append( "%s4" % (part[note] ) )
53                     need4 = False
54                 else:
55                     retval.append( part[note] )
56 if need4:
57     retval.append( "%s4" % (part[len(part)-1] ) )
58     need4 = False
59 else:
60     retval.append( "%s" % (part[len(part)-1] ) )
61 return retval
62
63 def addEnding( parts ):
64     for x in xrange( 4 ):
65         if parts[x][len(parts[x])-1][-1] == "4" or parts[x][len(
66             parts[x])-1][-1] == "8":
67             parts[x][len(parts[x])-1] = parts[x][len(parts[x])
68                 -1][0:len(parts[x][len(parts[x])-1])-1]
69             parts[x][len(parts[x])-1] = parts[x][len(parts[x])-1] + "1"
70     return parts
71
72 parts = separate( parts2 )
73 parts2 = []
74 parts2.append( parts[0] )
75 parts2.append( parts[1] )
76 parts2.append( parts[2] )
77 #addPassingTones( parts[2] ) )
78 parts2.append( parts[3] )
79 #addPassingTones( parts[3] ) )
80
81 parts = addEnding( parts2 )
82 for part in parts:
83     if part[0][len(part[0])-1] != "8" and part[0][len(part[0])-1]:

```

```

81         part[0] = part[0] + "4"
82
83 def printout( parts ):
84     fh = open( "output.ly", "w" )
85     fh.write( "\\version \"2.12.1\"\n\n" )
86
87     def printvoice( voice ):
88         for x in voice:
89             fh.write( "%s " % (x) )
90             fh.write( "}" )
91
92     fh.write( "\\header{\n" )
93     fh.write( "\\tttitle=\"Generated Music\"\n\n" )
94     fh.write( "\\tcomposer=\"Kevin Deisz\"\n\n" )
95     fh.write( "}\n\n" )
96
97     fh.write( "\\score {<<\n\t\\new Staff <<\n" )
98     fh.write( "\t\t\\tempo 4=160\n" )
99     fh.write( "\t\t\\clef treble\n\t\t{ " )
100    printvoice( parts[3] )
101
102    fh.write( "\t\t\t\t\n\t\t{ " )
103    printvoice( parts[2] )
104    fh.write( "\n\t\t>>\n" )
105
106    fh.write( "\t\t\\new Staff <<\n" )
107    fh.write( "\t\t\t\t\\tempo 4=160\n" )
108    fh.write( "\t\t\t\t\\clef bass\n\t\t\t\t{ " )
109    printvoice( parts[1] )
110
111    fh.write( "\t\t\t\t\n\t\t\t\t{ " )
112    printvoice( parts[0] )
113    fh.write( "\n\t\t\t\t>>\n>>\n" )
114
115    fh.write( "\t\t\t\t\\midi{ }\n" )
116    fh.write( "\t\t\t\t\\layout{ }\n}\n" )
117
118    fh.close()
119    os.system( "lilypond --png output.ly" )
120
121 printout( parts )

```

## 5 Appendix B - Output

### 5.1 Lilypond

```

1 \version "2.12.1"
2
3 \header{
4     title="Generated Music"
5     composer="Kevin Deisz"
6 }
7
8 \score {<<
9     \new Staff <<
10     \tempo 4=160

```

```

11      \clef treble
12      { e'4 f' g' e' f' f' f' e' e' f' g' e' f' g' e' g' a' f' g'
          a' a' b' e' g' a' g' e' g' a' g' e' g' a' f' f' e' f'
          g' e' f' g' e' e' f' f' e' f' e' g' e' g' a' f' f' e' f'
          ' e' g' a' g' e' g' e' g' a' g' e' f' f' e' g' e' g' a'
          g' e' f' e' f' e' f' e' f' e' g' a' a' b' e' g' a' g'
          e' e' f' f' e' g' a' f' f' e' f' g' e' f' f' e' e' f' g'
          ' e' f' e' g' e' e' f' f' e' g' a' a' b' e' f' e' g' e
          '1 } \\
13      { c'4 b c' b a a b c' a a b c' a b c' b a b c' c' c' b c' b
          a b c' b a b c' b a a b c' a b c' a b c' a a b c' b c'
          b c' b a a b c' b c' b a b c' b c' b a b c' a b c' b c
          ' b a b c' b c' b c' b c' b c' b a c' b c' b a b c' a a
          b c' b a a b c' a b c' a b c' a a b c' b c' b c' a a b
          c' b a c' b c' b c' b c'1 }

14      >>
15      \new Staff <<
16      \tempo 4=160
17      \clef bass
18      { g4 f e e f d d e e f d e f d e e f f e e f d e e f d e e
          f d e e e f d e f d e d d e e d d e d e e e f d e d
          e e f d e d e e f d e f d e d e e f d e d e d e d e d
          e e e f d e e f d e e d d e e f f d e f d e d d e e d d
          e d e d e e f d e e e f d e d e d e1 } \\
19      { c4 b, c b, a, a, b, c c a, b, c a, b, c b, a, b, c a, a,
          b, c b, a, b, c b, a, b, c b, a, a, b, c a, b, c a, b,
          c c a, b, c b, c b, c b, a, a, b, c b, c b, a, b, c b,
          c b, a, b, c a, b, c b, c b, a, b, c b, c b, c b, c b,
          c b, a, a, b, c b, a, b, c c a, b, c b, a, a, b, c a, b
          , c a, b, c c a, b, c b, c b, c c a, b, c b, a, a, b, c
          b, c b, c1 }

20      >>
21      >>
22      \midi{ }
23      \layout{ }
24      }

```

## 5.2 Music

### Generated Music

Kevin Deisz

The musical score is written for piano in common time (C). It consists of four systems of music, each with a treble and bass staff. The tempo is marked as  $\text{♩} = 160$ . The melody in the treble staff is a simple, repetitive sequence of eighth notes: G4, A4, B4, C5, B4, A4, G4. The bass staff provides a steady accompaniment of eighth notes: G3, A3, B3, C4, B3, A3, G3. The piece concludes with a final whole note chord in both staves, consisting of G3, B3, and D4.

## References

- [1] Burns, Kristine H. "Algorithmic Composition". September 1996. Florida International University. 23 Jan. 2009 <<http://eamusic.dartmouth.edu/~wowem/hardware/algorithmdefinition.html>>.
- [2] Jacob, Bruce. "Algorithmic Compositions As A Model Of Creativity". University of Maryland. 23 Jan. 2009 <<http://www.ece.umd.edu/~blj/algorithmic.composition/algorithmicmodel.html>>.
- [3] Jacob, Bruce. "VARIATIONS: Algorithmic Compositions for Acoustic Instruments." University of Maryland. 20 Jan. 2009 <<http://www.ece.umd.edu/~blj/algorithmic.composition/>>.
- [4] Kostka, Stefan and Payne, Dorothy (1996). *Tonal Harmony: With An Introduction To Twentieth Century Music*. 3rd ed. McGraw-Hill, Inc.: San Francisco.
- [5] Kunze, Tobias. "Algorithmic Composition Bibliography". 1998. Stanford University. 16 Jan. 2009 <<http://ccrma-www.stanford.edu/~tkunze/res/algobib.html>>.
- [6] Maurer, John A. "A Brief History of Algorithmic Composition". March 1999. Stanford University. 19 Jan. 2009 <<http://ccrma.stanford.edu/~blackrse/algorithm.html>>.
- [7] "Microsoft Research Songsmith". 2009. Microsoft Corporation. 22 Jan. 2009 <<http://research.microsoft.com/en-us/um/redmond/projects/songsmith/>>.