Pitching Computer Algorithms

Judy A. Franklin, Elizabeth A. Laverty, Shana R. Negin
Computer Science Department
Smith College
Northampton, MA 01063
jfranklin@cs.smith.edu, elaverty@smith.edu, snegin@cs.smith.edu
http://www.cs.smith.edu/~jfrankli/

Abstract

Computer scientists and practitioners have a wealth of algorithms at their disposal for enterprises such as sorting and searching lists of numbers, compiling computer programs, and enabling machines to learn to perform tasks, just to name several. The purposes of these algorithms are broad in nature yet generally are task-driven and analytic/scientific. We are using these algorithms in unconventional, artistic ways, to generate new computer music. We outline several ideas for using algorithms in new ways and have several short compositions available that exemplify this notion.

Generating Sound Grains using Sorting and Searching

We have used the bubble sort, the quick sort and the binary search as methods of algorithmic granular synthesis. Algorithmic granular synthesis is a method in which computer algorithms manipulate small "grains" of sound. These grains can vary in duration from the shortest duration that can be distinguished by the human ear to, say, 1 second. Many, many grains combined make up a new sound wave or a song. Grains may be separated by silence or may overlap to a small or large degree. Granular synthesis is a broad term within the field of computer music in that a variety of techniques are used to achieve it. Some techniques manipulate sound in the time domain and others manipulate it in the frequency domain. At present we use time domain techniques.

This work is inspired by that of Roads [1985] and Helmuth [1991] in granular synthesis in which clouds of grains are manipulated by e.g. time-varying spectral parameters. These parameters can be set by a composer, or can be changed according to a probability distribution. In our work, the mechanics of the algorithms choose the grains. Roads and Helmuth have also "time-granulated" sound files as we do, and discuss below.

To generate grains using a sorting algorithm, a list of numbers must be available to be sorted, say into increasing order. Any list can be used; we generate a list randomly. The list is typically from 1000 to 10000 numbers in length. The bubble sort compares two consecutive numbers in a list and swaps them if the first is larger than the second. This causes the largest (heaviest) value to sink to the bottom of the list and the other values to "bubble up to the top." Once a value reaches the bottom, that part of the list is no longer sorted, but another pass is made over the remaining part of the list in the same manner. When we use the bubble sort, each time there is a swap, the index of the larger of the two numbers is recorded in a second list. It is only the *index values* that are retained and that reflect the mechanics of the sort. The actual values being sorted are discarded. They are irrelevant as they are not used to generate sound directly. However, different lists of numbers will generate different lists of indices when sorted. The choice of list to be sorted can be used to shape the composites of grains; an unbiased random list will generate a different set than a biased list.

To clarify, suppose we have a list of four numbers, shown below with their indices on the left:

1	12
2	10
3	3
4	2

In stages, that are passes through the list, the bubble sort will sort them out: pass 1 makes a complete pass through, pass 2 passes through only the first 3 numbers (since the 12 has sunk to the very bottom), and pass 3 only compares (and here swaps) the top two numbers:

pass1:							pass2:					pass3:		
12		10		10		10	10		3		3	3		2
10	=>	12	=>	3	=>	3	3	=>	10	=>	2	2	=>	3
3	=>	3	=>	12	=>	2	2	=>	2	=>	10	10	=>	10
2		2		2		12	12		12		12	12		12

Whenever there is a number swap, the index of the larger number is saved. In pass 1, the saved indices are 1, 2, 3. In pass 2, the saved indices are 1, 2, as the value 10 sinks down, and in pass 3 the saved index is 1. The final list for this example is 1, 2, 3, 1, 2, 1.

After the sort, a grain that is a 0.1 second long sinusoid with frequency scaled by one of these index values, is generated; one grain is generated for each index value that was stored. The grains are placed side by side in time to create one long sound wave that both demonstrates the algorithm and generates a song. RTcmix (2001) is an open source software package used for this purpose. It can be used to save the sound wave to a file that can be played and translated to various audio formats. Figure 1 shows a sound clip with 17 grains, each with a different frequency. The dark areas show grains with a very high frequency. It shows the heaviest index value dropping to the bottom of the list in the first pass, as it is stored in the list at a higher and higher index, provoking a higher and higher frequency.

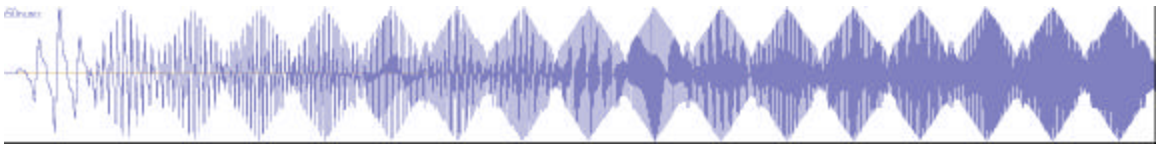

Figure 1. Bubblesort grains at beginning of sort (of 50 numbers).

The quick sort is another algorithm that we used to generate grains of sound in order to form a composite sound wave. There are several versions of the quick sort algorithm. In the version we use, the recursive[1] algorithm first finds the midpoint value of the list. Then it starts at the endpoints and moves toward the center. Whenever a value on the left is greater than the value at the center, it is swapped with a value on the right that is less than the center in value. The result is a list that is composed of two unsorted sub-lists, but each value in the first sub-list is less than each value in the second sub-list. Quick sort is then called, recursively, on each sub-list. The sub-lists become smaller and smaller and eventually there is nothing left to sort and the result is the whole list sorted. As the quick sort is sorting our randomly generated list, whenever two values are swapped, the indices of both values are stored in a second list, in a manner similar to that described above for the bubble sort. The sound however is quite different.

Once again, the actual values being sorted are discarded after sorting. It is only the index values that are important and that reflect the mechanics of the quick sort. As before, after the sort, a 0.1 second long sinusoid with frequency scaled by an index value, is generated; one grain is generated for each index value in the list. The grains are again placed side by side to create one long sound wave that both demonstrates the

[1] A recursive algorithm is one that refers to itself. In software, it would make a call to itself, passing it some smaller part of the problem it is solving.

algorithm and generates a song. It is possible in RTcmix to manipulate the grains further, such as by adding harmonics to the sinusoid, or by passing the grain through an amplitude envelope (amplitude corresponds to volume). Figure 2 shows a short clip from the sound wave generated by the quick sort grains. Here two levels of harmonics have been added and the grain is passed through an envelope that simply ramps up linearly with a slope of 1 and then ramps down with the same slope. (Note the grains in Figure 1 are also passed through an amplitude envelope).

Figure 2. Quicksort grains at beginning of sort. Swapping between right and left sides of list can be readily seen in the jump from high frequency grain to low frequency grain with the difference in frequency diminishing as the middle of the list is neared.

Once a list of values is sorted, it can be searched efficiently by, for example, the binary search. The binary search operates much as one does when looking for a name in an alphabetized phone book. The value being searched for is compared to the value at the midpoint. If it is lower, the right side is ignored. The value is then compared to the midpoint of the left side of the list and we can determine if it falls within the right or left side of that list. With just two comparisons, we can rule out three-fourths of the list. Comparisons of the value with sub-list midpoints continues until the value is found or until there are no more sub-lists. We performed the binary search and when the midpoint of each sub-list being searched was calculated, we saved this midpoint index value in a second array. These index values were then used as frequencies for sine waves as before and grains generated. The result was a little disappointing, ironically because of the efficiency of the algorithm. Only a handful of grains is generated (about 9 for finding a value in a sorted list of 5000 numbers). The frequencies change quite a bit in the beginning, but at the end they are very similar, as the search narrows.

Generating Grains using Musical Recordings

For fun and exploration we also used the three algorithms above to choose grains of sound from digitized recordings of people playing instruments or as Roads puts it “time-granulating” the sound files. We were able to use the resulting sound waves in our own musical compositions. We made a recording of a guitar improvisation and used the same list of index values obtained from the binary search. This time however, these indices are used as locations in the digitized recording and a sample that is 0.1 seconds in duration is taken from that location. In other words a grain of guitar improvisation is extracted. These grains are placed in sequence with time in between and Figure 3 shows the result.

Figure 3. Grains of guitar improvisation chosen using indicies from binary search.

Notice that the grains are quite distinguishable in the beginning but are difficult if not impossible to distinguish by the end. The midpoints of the binary search are so close as the search is narrowed that they are used to choose samples that are highly overlapping and start at nearly the same location in the sound wave. A recording was also made of a flute improvisation and the list of indices resulting from a bubble sort were used in the same way to take out granular samples of the improvisation, using each index as a location in the original sound wave. While it is difficult to see the result in a visual representation of a clip of the composed sound wave, it can be seen that there is a varied but repetitive structure that is a result of the bubble sort.

Computer Language Compilers as Song Generators

A compiler is a large computer program that translates another program, written in a language such as C/C++ or Java, for example, into machine usable form. A compiler is a complex algorithm that has several parts. Compilers have to understand the syntax of the computer language that they are translating, and check to see that programs follow the syntax by doing a syntactic analysis. For example, language features such as loops or an if-then-else statement can be used as long as the programmer follows a very particular form. Computer languages also have semantic rules that require certain language forms to be embedded in a certain context. For example, some operators require both numbers they use to be the same type; perhaps both numbers need to be integers. Some languages require that every identifier, such as the name of a variable, must be declared in the program and the type of value that may be stored in that variable must be stated. Therefore, the compiler must also perform a semantic analysis of the program. After all of this checking is done, and some translation has taken place, the compiler must then complete the translation, into the form required by the target machine (platform) on which the program is to be run, or executed [Watt and Brown 2000]. Generally the form that is usable by the machine (the executable machine code) is at a monumentally lower level than the original language. The third part, or pass, is called code generation.

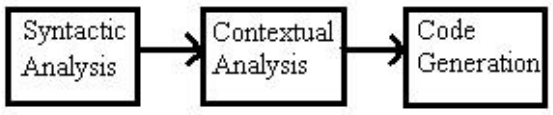

Figure 5. Three passes in a programming language compiler as described in [Watt and Brown 2000].

There are variations on these steps, and different compilers are written to handle these steps in one or more passes. We are using an educational compiler that makes one pass through the program it is compiling for each of the three steps described above (also written by [Watt and Brown 2000]) and shown in Figure 5. The compiler is written in the Java programming language, an object-oriented language that, when used with good software-engineering practices, results in a large group of small subprogram modules called methods. The mechanics of the compiler algorithm, reflected in the order in which these many methods are used, can again be used to choose grains.

We simply associate a number with each of the methods in the compiler program. When the compiler, a program itself, is run in order to compile some program (in this case in an educational language called Triangle), many of these methods are called and run, and their associated numbers are recorded in a file. A different set of numbers will be recorded for each different user program that is compiled. Various schemes could be used to choose the numbers. We simply use the method name. For example, “ParseIdentifier” has associated with it the number 9 (I is the 9th letter in the alphabet). ParseSingleCommand is 193: 19 for the S, and 3 for the C. There are many method names that start with the word “Parse” because the syntactic analyzer is also called the parser. So we ignored the “P” and used the rest of the method name.

A separate program, written in RTcmix, converts the recorded numbers into frequencies to be used with a software oscillator to generate a pitch. The logarithm of the number is taken first, and then the pitch is generated, and sent to a plucked string software instrument called strum. The result is a catchy very computer-sounding song that has three clearly demarcated segments, corresponding to the three passes.

Composing using these Tools

While these techniques alone do not necessarily produce compositions that one might think of as musical, we have used them as tools for composing electro-acoustic music. We have combined the bubble sort as pure sine wave pitches with bubble sort time-granulated manipulation of recorded flute improvisation, along with recordings of thrashers (related to mocking birds) to create a composition called "Thrashing Out" that has been submitted to an electro-acoustic competition [eContact2002]. RTcmix is useful in combining the sound waves from the several sources. Figure 6 shows a small snapshot of sound from "Thrashing Out" that includes both grains from the flute improvisation and grains using sinusoid generation, both either extracted by or chosen in frequency by indices from the bubble sort.

Figure 6. In composition Thrashing Out. Flute grains juxtaposed with bubble sort grains.

We are currently using Csound [Csound 2002] and its built-in filters to generate songs that use the compiler-generated granular, very computer sounding music, but yield ethereal compositions that one can listen to and enjoy as art. As a method of self-similarity or self-reference, we are also exploring using the listing of numbers from the compiler program as parameters to the Csound filters that filter the Triangle song, originally made from those same numbers! We have recordings of these pieces.

RTcmix and Csound

RTcmix (Real-Time cmix) [Rtcmix 2002] is software written in C/C++ that is freely available on the web. It runs under Linux, Irix, and on Next and Sun machines and PCs. The complete source code is available and one can programmatically implement instruments (as they are called) for granular synthesis or other kinds of synthesis. Furthermore, functions needed to create and read and write sound files are included.

Csound is free software that runs under PCs (Windows), Macs, and Linux operating systems. Csound is also a software synthesis program. Sound is created in Csound through the use of *orchestra* files and *score* files. An orchestra file is a text document that defines the instruments to be played by the score file through the use of Csound functions called opcodes. The score file is a text document that determines how and when each instrument will be played. The score file contains such information as note start time, amplitude, and duration. In one orchestra file we use the *soundin* opcode to import a wav file created from the compiler algorithm and then pass it to the *comb* opcode that applies a comb filter to it. Furthermore, we have created a program in the programming language C, that is independent of Csound, that can read in from

the file the numbers generated from the compiler, and generate a score file that uses these numbers as various filter parameters in the score file. This is one way to explore the self-similarity mentioned above. In another set of experiments, Csound's *grain* opcode is providing us a means to time-granulate the compiler's already once granulated sound file.

Summary

We have explored using computer algorithms in unconventional ways to generate sound and songs. During this work we have also used the results to demonstrate the algorithms to students of computer science [Franklin 2001]. There are many, many untapped algorithmic resources in the computer science field, as in other scientific fields in which computers are used. We are continuing to delve into these algorithms for future sound.

Acknowledgements
The granular synthesis figures in this paper are snapshots of windows Cakewalk's Sonar [Cakewalk 2002], software in which sound can be visually edited. The authors would like to thank the Smith College Computer Science Department and the John Payne Music Center for their support.

References

Cakewalk. 2002. http:// www.cakewalk.com/, Visual sound editing program.

Csound. 2002. http://www.csounds.com/, Software synthesis program.

Franklin, Judy. 2000. Listen to Computer Science Algorithms. http://www.cs.smith.edu/~jfrankli/music/algorithms.html

Franklin, Judy. 2001. Computer Generated Music as a Teaching Aid for First Year Computing. The Journal of Computing in Small Colleges. Vol 16, No 4. Proceedings of the Sixth Annual CCSC Northeastern Conference. Middlebury College VT.

Griffith, Niall and Todd, Peter 1999. Musical Networks: Parallel Distributed Perception and Performance. Bradford Books. MIT Press. Cambridge MA.

Helmuth, Mara. 1991. Patchmix and StochGran. Proceedings of the International Computer Music Conference. Montreal: McGill University.

ICMC 1996. International Computer Music Conference CD. International Computer Music Association (http://www.computermusic.org).

Keykit. 2001. http://nosuch.com/keykit/, Music programming software with nice MIDI interface.

Roads, Curtis 1985. Granular Synthesis of Sound. In *Foundations of Computer Music*. eds. Roads, C. and Strawn, J. MIT Press. Cambridge, MA.

Roads, Curtis 1996. The Computer Music Tutorial. MIT Press. Cambridge MA.

RTcmix. 2002. http://www.music.columbia.edu/cmix/, Software synthesis program.

Watt, D. A. and Brown, D. F. 2000. *Programming Language Processors in JAVA*, Prentice-Hall (Pearson Education), Harlow England.