APPLICATIONS OF LINKED DATA STRUCTURES TO AUTOMATED COMPOSITION

Charles Ames
32-3 Yale Avenue
Egbertville, New York
U.S.A. 14226

ABSTRACT

This paper describes how lists, trees, and networks are used in the set of composing programs developed by the author for the 1985 Ymauka Exposition. Ames first introduces basic terminology associated with linked data structures. He shows how linear lists can be used to collate simultaneous rhythmic layers and to enhance sensitivity to current or recent musical events. He next discusses how linked lists are compounded into a linked tree of finite depth and infinite breadth, which Ames uses to organize a knowledge base of pianing progressions between "source" and "destination" chords. The remainder of the paper discusses three instances in which Ames designed linked networks to represent complex musical structures: 1) a hierarchical form of "origins," "nodes," and "goals," in which each low-level element links to two higher-level elements; 2) a three-tiered monodic network of chords, basic melody notes, and embellished melody notes, and 3) a polyphonic network which organizes notes both "horizontally" into contrapuntal parts and "vertically" into chords.

1.0 INTRODUCTION

This paper describes the linked data structures employed by the set of composing programs which I developed for the 1985 International Exposition in Ymauka, Japan. These programs implement four algorithmic processes according to musical directives supplied by three composers: Toy Harumine (John Nishi), Circum Flage (Learen Miller), and Transitions (Charles McL.). The Toy Harumine program was implemented with assistance from Leonard Winters. For details of the musical directives, of the general strategies used in realizing them, and of the circumstances surrounding the project, I refer readers to a companion paper by Learen Miller, Robert Prunkli, and myself (1).

Though linked structures such as lists, trees, and networks have long been core subjects in undergraduate computer science curricula, their integration into composing programs has been slow. I hope to demonstrate here that advanced data structures for music are much more than an intellectual curiosity. During my recent experience with the Ymauka project, I found myself confronted by many problems of implementation which lent themselves quite elegantly to linked structures; indeed, many of these problems could only have been solved by longer, less general, and much slower programs—had not linked structures been available.

I stress that all of the applications described in this paper were motivated by practical necessity, not by fashionable programming philosophies or by special features of the language (2). As has been my practice over the past several years, each of the Ymauka programs was developed from scratch with the specific goals of the composer(s) in mind. Not only does this practice enable me to optimize algorithms and data structures for the tasks at hand, it continuously encourages me to seek out new ways of solving problems.

2.0 FUNDAMENTALS

A data structure is a construct of elements called either nodes or records, each containing one or more fields of discrete information. For example, a melody might be represented using a data structure with one node of data describing each note. Attributes such as the starting time, duration, and pitch of a note would occupy fields within such a node.

In a linked data structure, each node incorporates additional fields called links or pointers. These additional fields direct a program to relevant items in the structure, for example: to the "predecessor" or "successor" of each note in a melody (3).

251

ICMC '85 Proceedings
3.0 LINEAR LISTS

The simplest of linked data structures are linear lists, and an excellent example of a linear list is the melodic structure described above. Linear lists are used to organise sequences of notes; that is, each node in a linear list has at most one direct "predecessor" and at most one direct "successor" (2). Linear lists are most appropriate when sequences are subject to frequent insertions and deletions, because such modifications can be effected simply by revising a few links (3).

3.1 Rhythmic Layering

John Rhysill's compositional directives for Toy Harmony described a place divided up into segments, each with many simultaneous rhythmic layers. Some of these segments diverge gradually from blocked chords into random cludores others converge gradually from cludores into blocked chords. In order to bring the rhythm back into synchronisation at the end of a segment, Kacar and I adopted the expedient of generating an independent rhythmic sequence for each layer, then "squeezing" this sequence so that it would fit exactly into the segment. The program collates these different layers into a single stream of notes which it passes into a subsequent pitch-selection stage.

Figure 1 shows how the rhythmic layers of Toy Harmony are processed. The mechanism involves three linear lists, as shown in Figure 1 as FREE, LAYER, and BUFFER. FREE holds all nodes not currently being used to describe notes; LAYER serves as a temporary holding queue for notes in the current layer; prior to squeezing; BUFFER holds all previously generated notes in the current segment, sorted in increasing order of starting times. Of the linked-list operations employed in Figure 1, "insertion" and "flushing" require explanation. The program "inserts note into BUFFER relative to START(NOTE)" by 1) setting a pointer PER to the tail of BUFFER, 2) working its way backward through BUFFER until START(FER) no longer exceeds START(NOTE), and 3) "inserting" note into BUFFER just after PER. The program "flushes" BUFFER by repeating the following steps until BUFFER is empty: 1) detecting a node from the head of BUFFER, 2) writing out the note attribute stored in this node, and 3) appending the node to FREE.

3.2 Sensitivity to Current Events

For his Circle Piece, Lejaren Hiller supplied me with a sheet of music paper containing twelve handwritten "modules" which the program was to employ in various transpositions, registers, and instrumentations. He directed me that the proportions in which these modules and attributes of modules were to be chosen should follow certain prescribed statistical distributions, as should the number of modules occurring simultaneously at any given moment in the piece.

In order to realize Hiller's directives, I developed a program which maintains a list of all the modules currently sounding. The starting time for each group of newly entering modules is determined by the earliest ending time in this list. At each moment, the program first computes how many new modules will enter in accordance with Hiller's prescribed distribution. It then selects attributes for these new modules and appends them to the list. Once the attributes of any new
module have been determined, the program collates the contents of this module into a common stream of notes.

Figure 2 shows the process used to compose 

Diaspora Fugues. Explicit in Figure 2 are a list of currently active modules called LIST and a note queue called BUFFER serving much the same purpose as BUFFER in Figure 1. Lists of free notes for modules and notes are also implicit in the process: should the list of free notes for notes become exhausted, the program recycles nodes by writing out the earliest notes in BUFFER. Also consulted by the program are lists of musical material indexed under the generic symbol MATERIAL. MATERIAL[1] contains material for the Ist of Hillier's twelve modules.

Notice that the program inserts modules into LIST with respect to ending times rather than starting times. This procedure insures that the modules which finish soonest will always reside at the head of LIST.

3.3 Sensitivity to Recent Events

The material of my own Transitions is a contextual texture organized horizontally as a progression of chords and vertically as a composite of six "regisrtal channels" (6). As it generates this texture, the program also orchestrates it by selecting one of six "spectral" channels and one of six "spatial" channels for each note. Regisrtal, spectral, and spatial channels are spaced equidistantly between limits established by the program's lowest and highest registers; lowest and highest positions of a band-pass filter relative to the fundamental frequency; left and right speaker. The companion paper (1) details how registral and spectral limits evolve over the course of the work.

The texture-generating mechanism for Transitions is controlled by five parameters selected during an earlier stage of production: 1) speed - the average number of chordal attacks per second of music; 2) syncopation - the random variance around the speed; speed and syncopation supply two inputs to a random function which determines "periods" between chordal attacks; 3) thickness - the average number of notes sounding in each chord (whether new or held over from the preceding chord); 4) dovetailing - a percentage governing how many notes hold over between chords; 5) articulation - a percentage relating chordal durations to periods.

Figure 3 shows how the texture-generating mechanism worked. Explicit in Figure 3 are three lists: CURRENT, PREVIOUS, and
CHORD Start(Chord) = Sustain-Time;  
for each chord in segment,  
compute number of sustaining notes Nsustain(Chord),  
compute number of sustaining notes without Nsustain(Chord),  
compute period between chords without Nsustain(Chord);  
compute period between chords without Nsustain(Chord),  
compute period between chords without Nsustain(Chord),  
compute period between chords without Nsustain(Chord);  
if Nsustain(Chord) = 0,  
select a non-sustaining note from CURRENT,  
DURATION(HEIGHT) = DURATION(HEIGHT) + CHORD DURATION(Chord-1);  
select note from CURRENT,  
select note from CURRENT;  
select note from CURRENT,  
select note from CURRENT;  
end of segment.

BUFF. CURRENT holds notes which are active in the current chord;  
PREVIOUS holds notes which released in the  
preceding chord; while BUFF.  
holds a stream of notes sorted in  
increasing order of starting times.  
Each time it composes a new chord, the program  
first determines which notes do not  
sustain from the preceding chord and  
transfers those notes from CURRENT to  
PREVIOUS. The program next supplements  
with new notes in CURRENT with newly  
entering notes, selecting channels in each  
instance. The basic rule for channel  
selection is that simultaneous notes  
cannot occupy the same channel, and the  
program checks for this condition by  
examining notes in CURRENT. Phrases  
within each predominant channel are  
maintained in spectral and spectral  
channels as well by the conditions  
relating new channels to channels already  
present in PREVIOUS.

BUFF., CURRENT holds notes which are active in the current chord; PREVIOUS holds notes which released in the immediately preceding chord; while BUFF. holds a stream of notes sorted in increasing order of starting times. Each time it composes a new chord, the program first determines which notes do not sustain from the preceding chord and transfers those notes from CURRENT to PREVIOUS. The program next supplements with new notes in CURRENT with newly entering notes, selecting channels in each instance. The basic rule for channel selection is that simultaneous notes may not occupy the same channel and the program checks for this condition by examining notes in CURRENT. Phrases within each predominant channel are maintained in spectral and spatial channels as well by the conditions relating new channels to channels already present in PREVIOUS.

We recall that notes in a linear list are related sequentially as "predecessors" and "successors". Notes of a tree, by comparison, are characterised by hierarchic relationships between "superior" and "inferior". Though a node in a tree may have at most one direct superior, such a node may have an arbitrary number of direct inferiors.

The process used in Mix or Match to compose choral progressions shows how linked trees can be useful in representing contextually sensible "productions" or "rewrite rules" for generative grammars (7). This particular grammar begins with an archetypical progression, such as:

CHN / / PM / 07 /

The "productions" of this grammar elaborate upon this archetype by inserting

ICMC '85 Proceedings 254
passing progressions between consecutive chords. For example:

CM / Gm 07 / FM 07 / G7 /

The form of these productions suggested to me a tree of finite depth in which the number of direct "inferiors" to each node was arbitrarily extendible through subsidiary linked-list structures. Figure 4 illustrates how this tree is organized. At the highest level resides a list of "source" chords, inferior to these "source-iss" are subsidiary lists of "destination" chords, each "destination" directs the program in turn to a list of appropriate passing progressions.

5.3 NETWORKS

Networks (8) are to trees as trees are to lists. Any node within a network may have one, two, or more "superiors"; in addition to this expanded hierarchic organization, nodes at equivalent levels (i.e., hierarchic "peers") may also be linked sequentially.

5.4 A Hierarchic Network

An example of a network in which each node has two superiors appears in the form-composing program for Transitions.

This program first generates an abstract description of the form as a hierarchy of sections, subsections, sub-sub-sections and so on (9). Dividing points of this form are marked by "terminals", with each terminal assuming one of three idiosyncratic functions: "source", "goal", and "node" (in a different sense from the normal usage of "node" in this paper). Figure 5 illustrates relationships between the first fourteen terminals of Transitions; each terminal of the musical structure is represented within the program by one node of a linked network.

Once this abstract description has been obtained, the program proceeds to select attributes for each terminal in such a manner that the three functions are expressed through appropriate similarities and contrasts. This selection process consists of recursive searches (10) which begin with terminals of broadest hierarchic significance and work their way down through more local relationships, backtracking wherever they compose themselves into a corner.

5.5 A Monodic Network

A second example of a network resembles a linked tree to the extent that each "inferior" node links vertically to
exactly one "superior". However, additional sets of links organize nodes sequentially at each hierarchic level. Such a network coordinates the three compositional layers used by the Me or Mach program to represent a melody: 1) chords, 2) basic melody, and 3) embellished melody. Figure 6a illustrates these three layers using music actually composed by the program.

Figure 6b illustrates how the program represents the three layers depicted in Figure 6a as a linked network. Vertical links direct each chord to a subsidiary list of basic-melody notes; similarly, each basic-melody note has its own list of embellished-melody notes. The horizontal lists detail sequential relationships within each layer; these independent lists enable the program to traverse the basic melody without consulting the chordal progression or to traverse the embellished melody without consulting the basic melody.

5.3 A Polyphonic Network

The most elaborate linked data structure in all of the Yokoza composing programs is the network used for the pitch-selecting program in the selection stage of transcription. This pitch-selecting program gathers the polyphonic stream of notes described under heading 3.3 of this paper into "chords" at
Figure 6b: Network representation of a Mix or Reisch tone. Shaded times are indicated at lower numbers, and corresponding durations are shown in the network. The diagonal lines represent functions as: primary, secondary, and incidental.

Figure 7 illustrates how the polyphonics network for Transitions is organized. The basic structure is a "master" list detailing all of the notes currently in memory, as the program proceeds through the piece, each note added to head of the list is continuously being written out in order to recycle memory for new notes being added onto the tail. Superimposed upon this master list is an independent "horizontal" structure of linear lists, one for each regiral channel. "Vertical" structure is provided.

![Diagram](image)

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

![Diagram](image)

Figure 7: Polyphonics network for Transitions. The blocks in the top row depict schedules, the remaining blocks depict notes. Of the two numbers in each box block, the first specifies the note's position in the master note list. The second number indicates the block number and starting time. The segment number in each segment block specifies a position in the list of schedules, while the remaining numbers indicate pointers to notes.
by the schedules, which are themselves organized into an ancillary list.

6.0 CONCLUSION

Linked data structures greatly facilitate the process of acquiring relevant information for compositional decisions, increasing the speed and musical sensitivity of composing programs. Properly employed, such structures can also enable a program to adapt itself to fluid data requirements, both from moment to moment during program execution and from day to day during program development as compositional input is expanded and "tuned".

In combination with heuristic decision-making algorithms, analytic feedback, and constrained searches with dependency-directed backtracking, linked data structures have brought composing programs to a level of sophistication which equals or surpasses human capabilities for performing well-defined tasks such as the ones described in this paper. The potential inherent in this technology -- both as an extension of human creative processes and as a tool for understanding these processes -- is enormous.

NOTES


2. In fact, the language for all these programs was FORTRAN 77 running under the VAX-VMS operating system.

3. A comprehensive explanation of how linked structures are implemented and maintained may be found in Donald Knuth's "The Art of Computer Programming", 2nd ed. (Addison-Wesley, 1973).

4. An additional requirement of linear lists is that no node can succeed itself, either directly or indirectly. This requirement eliminates lists which "wrap around" upon themselves. Similar restrictions apply to trees and networks.

5. Modifying sequential arrays, by contrast, requires wholesale transfers of information.

6. "Registration channel" means the same as "contrapuntal part"; the term "channel" comes from Robert Erickson's Sound Structure in Music (University of California Press, 1976), pages 117-119.

7. cf. Stephen Holtzman, "A Generative Grammar Definition Language for Music", Interface, volume 9, number 1, page 1 (1980). The Mix or Match productions were not recursively implemented; with hindsight, such an implementation would not have been difficult.

8. Networks are also called "directed graphs".
