# CHAPTER 14

## CONSTRAINED SEARCH

We have seen that many problems encountered in automated composition may be resolved into a sequence of elementary decisions, each of which admits a fairly small number of options. However, we know from chapter 12 that the number of potential solutions to such problems grows exponentially with the number of decisions. Even though it is theoretically possible to find an _optimal_ solution to any problem using the methods of comparative search (chapter 12), in practice the requisite computations may go on for months or even years.

As an alternative to the intensive procedures of comparative search, this chapter investigates the strategy of _constrained search_. Of all the decision-making strategies discussed in this book, constrained search undoubtedly comes closest to simulating how human composers actually work. The approach involves specifying a minimum standard above which _any_ solution is acceptable. Evaluative criteria are provided not as formulae for computing relative keys, but rather as constraints. For each decision, the search steps through potential options, testing for violations. Whenever it encounters an option meeting all of the constraints, the search advances to the next decision; should the search exhaust all available options, it backtracks, revises one or more earlier decisions, and tries again.

Because this approach accepts the first complete solution encountered while rejecting any flawed solution immediately upon discovering a fault, constrained searches avoid the extended digressions which are characteristic of comparative searches. Consequently, constrained searches provide a practical mechanism for solving highly complex problems embracing large numbers of decisions. The disadvantage of constrained search versus comparative search is that the solutions produced by constrained searching are merely "acceptable", not optimal. However, it remains possible to impose heurisms affecting the schedule of decisions and the schedule by which the search considers options for each decision. Such heurisms enable the composer/programmer to bias a solution toward qualities which, though desirable, do not ~~merit~~ require the ~~absolute~~ inflexible status of constraints.

## 14.1 APPLICATION: PART-WRITING BY CONSTRAINED SEARCH

In order to illustrate the mechanics of a constrained search, we shall apply this strategy to a problem of traditional harmony: finding a six-part C major triad which resolves a six-part dominant seventh chord on G. Figure 14-1 depicts a schedule of parts in this progression along with schedules of potential resolutions for each pard. The problem divides into six decisions, one for each part; each decision in turn admits up to three options, expressed in Figure 14-1 as melodic motions. Notice that decisions 3 and 6 admit only one acceptable option; the E4 in part 3 is the only pitch in a C major chord which resolves F4 downward by a step, while the C3 in part 6 is necessary to keep the chord in first inversion. In addition to these constraints <u>implicit</u> in the schedules for parts 3 and 6, the search imposes four <u>explicit</u> constraints:

1. no two parts may cross,

2. no two parts may move in consecutive fifths or octaves,

3. the C major chord may contain no more than two G's, and

4. the C major chord must contain exactly one E.

Figure 14-2 chronicles the search for an acceptable C major chord.

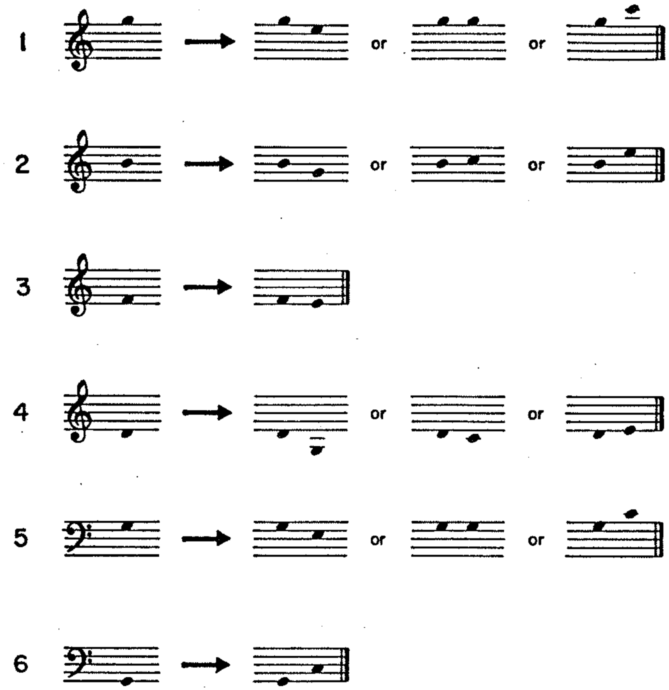


Fig 14-1

14-3b

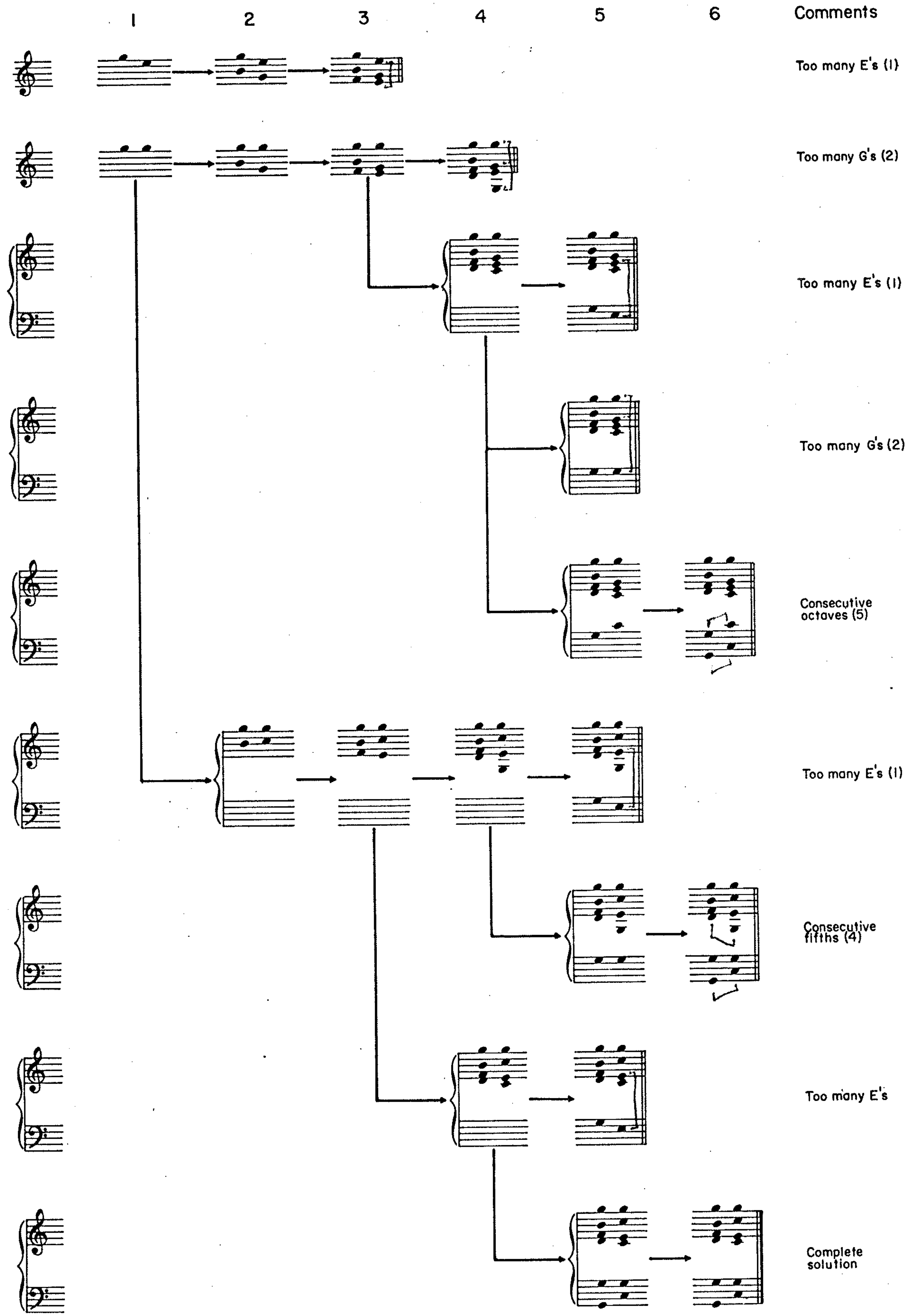
1
2
3
4
5
6
Comments
Too many E's (1)
Too many G's (2)
Too many E's (1)
Too many G's (2)
Consecutive octaves (5)
Too many E's (1)
Consecutive fifths (4)
Too many E's
Complete solution

Fig 14-2

Figure 14-1: Part-leading schedules - The sequence of decisions proceeds from top to bottom, while the schedule ~~sequence~~ of options for each decision proceeds from left to right.

Figure 14-2: Chronicle of search for an acceptable resolution - The numbers at the top of each column refer to the schedules depicted in Figure 14-1. Bold arrows indicate where one decision holds for multiple solutions. The parenthetic number after a comment indicates the source of conflict with an unacceptable decision.

We now consider the effect of heurisms affecting the schedules of decisions and options. Figure 14-3 depicts an alternate set of schedules for the same problem detailed above. It ranks heurisms for scheduling decisions as follows:

1. Number of options - The least flexible decisions (those with the fewest available options) receive greatest priority.

2. <u>Urgency</u>: The traditional "urge" for a dissonance to resolve downward by step is already implicit in the

restriction that F4 may only resolve to E4. However, Figure 14-3 also incorporates the less emphatic "urge" of the leading tone to resolve upward.

3. Prominence: Other factors held equal, Figure 14-3 allots greater priority to the more readily audible outer parts, at the expense of inner parts.

4. In the event that the preceding three heurisms apply equally, scheduling of decisions is random.

The heurisms used to schedule options for each decision were:

1. Tendency: If a part has a tendency (that is, if it involves a dissonance or a leading tone), the pitch which resolves this tendency receives greatest priority.

2. Smoothness of progression: By contrast to Figure 14-1, which simply lists pitches of the C major triad from lowest to highest, Figure 14-3 favors the smallest motions.

3. In the event that the preceding two heurisms apply equally, scheduling of options is random.

Figure 14-4 chronicles a search proceeding according to these revised schedules. It is clear that, while the complete solution selected by both searches are identical, the effort invested in scheduling decisions pays off in much-decreased searching time. Notice, however, that the decision to lead the B4 upward in Figure 14-4 occurs directly. By contrast, it is simply an accident of circumstance that caused the previous search to take this step.

Figure 14-3: Revised part-leading schedules - The sequence of decisions proceeds from top to bottom, while the ~~sequence~~ schedule of options for each decision proceeds from left to right.

Figure 14-4: Chronicle of search for an acceptable resolution - The numbers at the top of each column refer to the schedules depicted in Figure 14-3. Bold arrows indicate where one decision holds for multiple solutions. The parenthetic number after a comment indicates the source of conflict with an unacceptable decision.

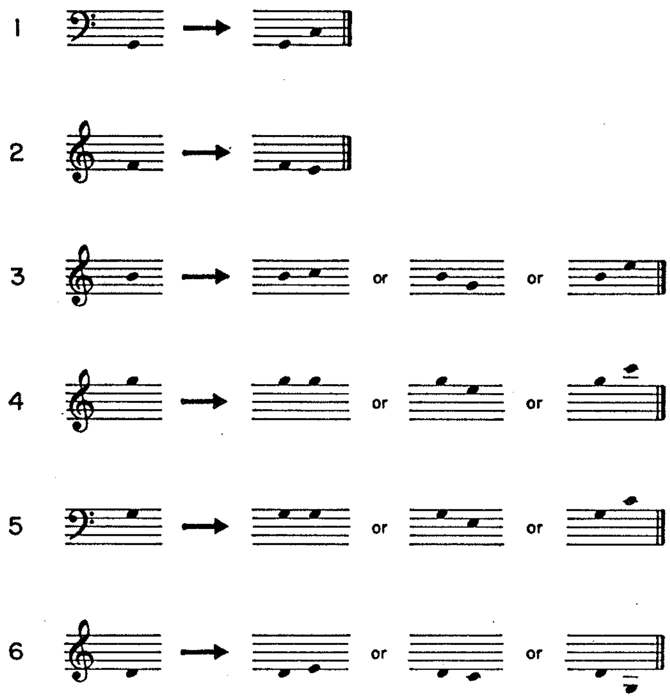


Fig. 14-3

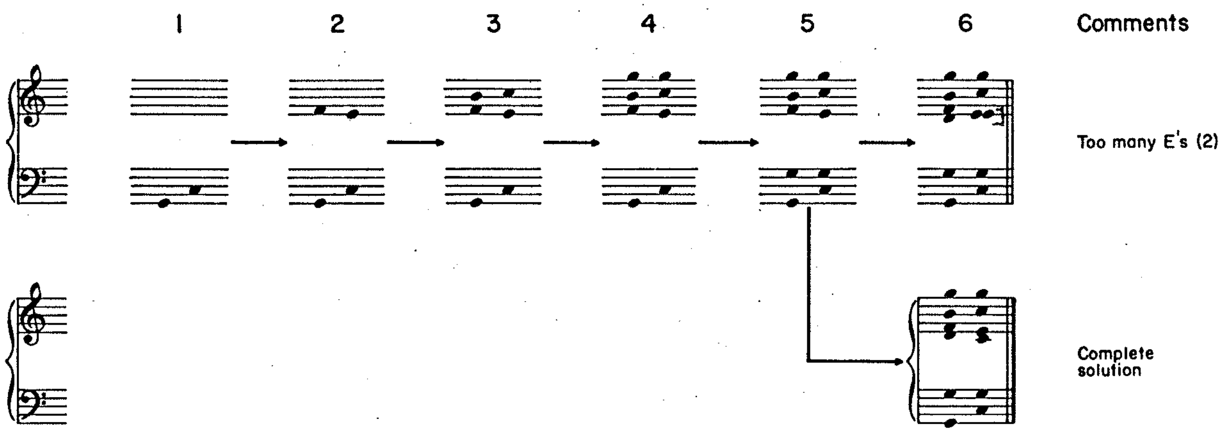


Fig 14-4

## 14.2 IMPLEMENTATION

A fully general constrained search is a paradigm of what chapter 10 has designated "horizontal" recursion. Interpreted in this way, the recursive "level" ~~indicates~~ corresponds the current decision -- either directly or through a schedule -- while the process terminates when it reaches the goal of selecting acceptable options for every decision. The basic strategy generalizes the approach taken by subroutines PARTS, EVAL, and LEGAL of program DEMO7 (heading 9.3.2), which implement most of the relevant procedures with the exception of backtracking. Remember that PARTS failed irrecoverably when none of the eight pitches scheduled by EVAL for any given decision satisfied all of LEGAL's constraints. Backtracking enables a search to recover from such failures.

Since backtracking requires the capability to take up where a search has left off in an earlier schedule, it is necessary to keep track of the following information for each decision:

1. the schedule of options,
2. an index to the current option under consideration, and

3. any ancillary data associated with ~~the current~~ a decision.

Program SEARCH illustrates how ~~one~~ a programmer might implement a constrained search with backtracking. The parameter MDEC ~~gives~~ specifies the number of decisions, ~~whose~~ while the schedule of decisions resides in ~~is provided by~~ the integer array DECIDX. The integer variable IDXDEC determines the level of recursion. Array element DECIDX(IDXDEC) holds the current decision, which SEARCH transfers to the holding variable IDEC for increased efficiency. Array element LIMDEC(I) holds the number of options for the Ith decision. Individual schedules of options reside in the two-dimensional integer array OPTIDX, which allows up to MOPT elements per decision. Array element IDXOPT(IDEC) ~~provides~~ yields an index to the current position in this schedule, while array element OPTIDX(IDXOPT(IDEC),IDEC) holds the option itself. The integer array OPTDEC stores selected options for each decision; to determine the current partial solution, ~~one~~ the program must consult array elements OPTDEC(DECIDX(I)) for I=1,...,IDXDEC.

A call to a hypothetical subroutine ORDER (line 7) establishes the schedule of decisions. Subroutine EVAL (lines 13 and 29) determines individual schedules of options 'on the fly' each time the search advances to a new decision. The nature of ORDER and EVAL will vary with the application, though subroutine EVAL of program DEMO7 is representative; these subroutines may be dispensed with when schedules are ~~provided~~ specified manually. The logical function LEGAL (called from line 20) determines whether

H-8a

```
      program SEARCH
      parameter (MDEC,MOPT)
      integer DECIDX(MDEC),OPTIDX(MOPT),OPTDEC(MDEC)
     :        OPTPRT(MPRT),PCHOPT(MOPT,MPRT)
C
C     Schedule decisions
      call ORDER(DECIDX,MDEC)
C
C     Search for acceptable solution
      IDXDEC = 1
      IDEC = DECIDX(IDXDEC)
C     Schedule options for first decision
      call EVAL(OPTIDX(IDEC),LIMDEC(IDEC))
      IDXOPT(IDEC) = 0
      do
        I = IDXOPT(IDEC) + 1
        if (I.le.LIMDEC(IDEC)) then
          IDXOPT(IDEC) = I
          OPTDEC(IDEC) = OPTIDX(I)
          if (LEGAL(DECIDX,OPTDEC,IDXDEC)) then
            if (IDXDEC.eq.MDEC) then
              print *, (OPTDEC(I),I=1,MDEC)
              stop
            else
C             Advance to next decision
              IDXDEC = IDXDEC + 1
              IDEC = PRTDEC(IDEC)
C             Schedule options for next decision
              call EVAL(OPTIDX(IDEC),LIMDEC(IDEC))
              IDXOPT(IDEC) = 0
            end if
          end if
        else
C         Options exhausted:  Backtrack to preceding decision
          IDXDECEC = IDXDEC - 1
          if (IDXDEC.lt.1) stop 'Unsuccesful search.'
          IDEC = DECIDX(IDXDEC)
        end if
      repeat
      end
```

Ex 14-1

or not a newly-selected option is acceptable; like ORDER and EVAL, LEGAL will vary with the application, function LEGAL of program DEMO7 is representative.

-- Programmine example 14-1: program SEARCH --

### 14.2.1 Dependency-Directed Backtracking

A deficiency in SEARCH arises from the fact that whenever the search reaches an impasse the program simply backtracks to the immediately preceding decision. As a result, SEARCH must grope its way backward along the schedule of decisions until it locates the cause of an impasse. For example, suppose SEARCH had attempted the search illustrated in Figure 14-2. Upon encountering the first conflict depicted in that Figure ("Too many E's", in the uppermost row), SEARCH would determine that all (one) of the options available to decision 3 had been exhausted, and SEARCH would in consequence ~~would~~ backtrack to decision 2. ~~It~~ The program would then substitute a C5 for the G4 in decision 2 and return to decision 3. Since this action would not effect the number of E's in the chord, the impasse at decision 3 would still remain. Back to decision 2 again. SEARCH would now attempt the third option in decision 2's schedule, E5, only to run up against the same constraint, "Too many E's". Only then would SEARCH backtrack to revise decision 1, which caused the problem in the first place by selecting an E.

14.2.1.1 An Expedient Method - The search actually depicted in Figure 14-2 incorporates a feature christened ~~called~~ "dependency-directed backtracking" by Stallman and Sussman, who first describe the problem (1977). A simple though non-rigorous implementation of dependency-directed backtracking involves simply determining the most recent source of conflict for each decision. Program SEARCH can perform such a determination with the following modifications:

1. Declare an integer array called BAKIDX of dimension MDEC in order to keep track of conflicts,

2. Since SEARCH has yet to encounter any sources of conflict at the onset of new decisions (that is, after lines 14 and 30), have it set BAKIDX(IDXDEC) to zero at these points.

3. In place of the logical function LEGAL (line 20),

substitute a new function ISOURC which tests the current option against each constraint and returns either 0 (no conflict) or a positive integer locating the <u>earliest</u> decision which is incompatible with the current option. Store the result of ISOURC in the holding variable IBAK and select whichever of the following branches is applicable:

a. If IBAK is zero, then SEARCH proceeds as if LEGAL had returned .true. (by executing lines 21-31);

b. otherwise, SEARCH sets BAKIDX(IDXDEC) = max0(BAKIDX(IDXDEC),IBAK). This second branch insures that if an impasse arises for the current decision, then SEARCH will backtrack only the minimal number of decisions required to break this impasse.

4. The actual process of backtracking reduces to setting IDXDEC=BAKDEC(IDXDEC). However, if the decision-making process involves cumulative feedback or maintains some other data which is not held 'frozen' for each decision, then it will be necessary to work back decision-by-decision, cancelling out intermediate

computations.

The programs used to generate Demonstration 11 (described later in this chapter) illustrate variations upon this expedient method of backtracking.

14.2.1.2 A Rigorous Method - The mechanism just described is simple to implement and highly effective for most applications. However, if it has to backtrack several times in a row (that is, if upon reaching an impasse, the search backtracks to the most recent source of conflict only to encounter another impasse, and so on), then the mechanism tends to loose track of the original impasse. Consider the sources of conflict indicated below:

| Decision | Sources of Conflict |
|---|---|
| 1 | none |
| 2 | 1 |
| 3 | none |
| 4 | 2 |
| 5 | 4 |
| 6 | 3, 1 |
| 7 | 6, 5 |

Suppose the search reaches an impasse at decision 7. It will then backtrack to decision 6, since this decision is the most recent source of conflict. Suppose, however, that all of the options available to decision 6 have themselves been exhausted. The expedient mechanism described above will then cause the search to backtrack to the most recent conflict with decision 6, which is decision 3. In the process, the mechanism has lost touch with the original impasse, which might also have been broken by revising decision 5, at much less waste of effort.

In order to insure a mechanism rigorous enough to keep track of the original impasse, it is necessary for the program to assemble a complete list of conflicts for each decision. From these lists, the program in turn derives a backtracking schedule as follows: Initially, the schedule is empty. Whenever the search reaches an impasse, the computer merges the current list of conflicts into the backtracking schedule. (Duplicate conflicts are ~~ignored~~ discarded.) The search then backtracks to the most recent conflict on the schedule.

A variety of information structures may be used to implement comprehensive backtracking. If sufficient memory is available, it may be most expedient to store the lists of conflicts in a two-dimensional array indexed by decision and option. Alternately, a one-dimensional array may store pointers to linked lists. This alternative is recommended only when the number of decisions is large and the average number of conflicts per decision remains well under half the number of options, since each node in the linked list requires two elements, a value and a link. A linked-list structure is recommended for the backtracking schedule itself, since this schedule must accomodate frequent insertions and deletions of items.

### 14.2.2 Prescience

This chapter has used the word "impasse" to designate situations in which a search attempts to make a decision, but discovers that all of the available options are in some sense unacceptable. In such situations, an option must fall into one of two categories:

1. Options which are immediately found unacceptable ~~either~~ by the constraints implemented in either function LEGAL or function ISOURC, or

2. Options which currently seem to be acceptable, but which propigate unresolvable conflicts at later points in the search.

An example of the latter category of unacceptable option is the

E5 considered for the first decision (the uppermost part) in Figure 14-2. It ~~seems very clear~~ is intuitive~~ly~~ that if one (implicit) constraint requires the third-from-uppermost part to resolve F4 to E4 while another (explicit) constraint states "the C major chord must contain exactly one E", then the search is going to run into problems if it tries to select E5 for the uppermost part. Unfortunately, this information has not been communicated to the search of Figure 14-2, which blithely attempts to use E5 anyway.

For all the help which backtracking provides in recovering from fruitless digressions, the quickest way out is often to forsee such digressions and avoid them in the first place. For example, rather than saying "the C major chord must contain exactly one E", one could instead say "only the third-from-uppermost part may contain an E".

Such prescience must often be incorporated at the expense of generality, though not always. As another example, consider the simple problem of composing an arpeggio. Assume that there are ten attacks for the program to place within eight consecutive beats given two constraints: 1) each beat should contain at least one attack, and 2) no beat should have more than two (simultaneous) attacks. An unprescient way of coding the first constraint would be wait until all of the attacks had been used up, then check through the arpeggio to see if any holes remain.

A better way would be to keep tallies both of the number of empty beats and the number of unplaced attacks and to ask the following question each time the search attempted to double up attacks: "Will there still be sufficient unplaced attacks left to fill up the remaining empty beats?"

## 14.3 CONSTRAINED SEARCHES IN AUTOMATED COMPOSITION

The strategy of solving compositional problems by searching was used as early as the Illiac Suite (1957). In attempting to process streams of randomly generated notes through a "sieve" of stylistic rules, Hiller and Isaacson very quickly noted that "...with the addition of more rules, the probability of obtaining a successful piece of music would soon become very small". This problem led to their incorporation of a "try-again method" which allowed the Illiac to regenerate a note whenever it was confronted with a violation. This method was limited to retries on specific decisions only. It also ~~and~~ lacked ~~in~~ efficiency, ~~because~~ ~~in that no~~ having no provision ~~was made~~ to restrain the Illiac from retrying notes which the computer ~~it~~ had already rejected.

An article by Stanley Gill (1968) describes an approach used by Gill to compose a short piece entitled Variations on a Theme

<u>by Berg</u>. Though Gill does describe explicit procedures, it is evident from his tree-graph of the decision-making process that Gill's program was capable of backtracking to earlier decisions whenever it ran out of options.

Of Larry Polansky's <u>Four Voice Canons</u>, numbers 2 (1975) and 3 (1976) were both written using computer programs which incorporate backtracking. The <u>Four Voice Canons</u> are based on series of values used to determine musical attributes such as pitch, duration, envelope, and various other aspects of timbre. The number of values in each series varies from canon to canon. Polansky's programs generated lists of permutations of this series conforming to two constraints:

1. any permutation is derived from its predecessor in the list through the exchange of two elements; for example, the first and last elements of the five-note series "ABCDE" may be exchanged to obtain "EBCDA", and

2. every possible permutation occurs exactly three times in the list.

Polansky derived lists for each musical attribute to produce a sequence of notes, and then overlaid the resultant sequence with itself for times to produce his canons.

Kemal Ebcioglu has implemented constrained searches as means for testing the dictums of traditional contrapuntal theory. His 1980 paper describes a program for generating a single counterpoint against a cantus firmus, subject to rules provided by the user. In more recent work, Ebcioglu has developed programs which accept a chorale melody and attempt to compose four-part homophony in the style of J.S. Bach. His results have been impressive, duplicating Bach's own harmonization exactly in more than one instance.

Constrained search has become the primary technique used by Charles Ames. To compose Gradient for solo piano (1982), Ames used constrained searches to compose a progression of six-part chords and subsequently to arpeggiate each chord. With Undulant for seven instruments (1983) Ames implemented constrained searches capable of scheduling options "on the fly" for each decision, based on cumulative feedback. He also introduced a linked information structure capable of representing contrapuntal textures of arbitrary complexity (note 1).

## 14.4 DEMONSTRATION 11: CONSTRAINED SEARCH

Demonstration 11 illustrates the use of constrained searches in a full-fledged composing program. The composition produced by this program is a study in what Robert Erickson (1975) terms "perceptual channeling", that is, the mechanism by which listeners perceive disjoint musical events as components of an ongoing process, or "channel". In Demonstration 11, the major factor contributing to channeling is register, although the fact that each pitch constantly associates with a fixed group of partners also plays an important role.

### 14.4.1 Compositional Directives

The compositional process divides into four stages of production:

1. Stage I: Material - composing the eight 'cells' depicted in Figure 14-6;

2. Stage II: Form - selecting material for each segment in order to determine the compositional profile depicted in Figure 14-7;

3. Stage III: Rhythm - composing rhythm and selecting

cells for each note; and

4. <u>Stage IV: Pitch</u> - selecting inflections for each note.

As happened in Demonstration 10, the present Stage II induces the form of Demonstration 11 ~~is induced~~ from the 'bottom up' ~~in Stage II~~ on the basis of qualities inherent in material composed ~~previously by~~ during Stage I. Information from Stage II enables Stage III to describe all of the notes in the piece to the extent of rhythms and cell numbers; Stage IV completes the process by filling in inflections. The final product appears in Figure 14-9.

14.4.1.1 Stage I: Material - Figure 14-6 depicts the material of the ~~work~~ piece, which consists of eight melodic cells. Each cell consists of three 'inflections' of a register: low, middle and high; these 'inflections' are realized by chromatic pitches spaced no farther apart than a whole tone. The material has the following properties:

1. each cell consists of two melodic steps, where a step may be either a semitone or a whole tone,

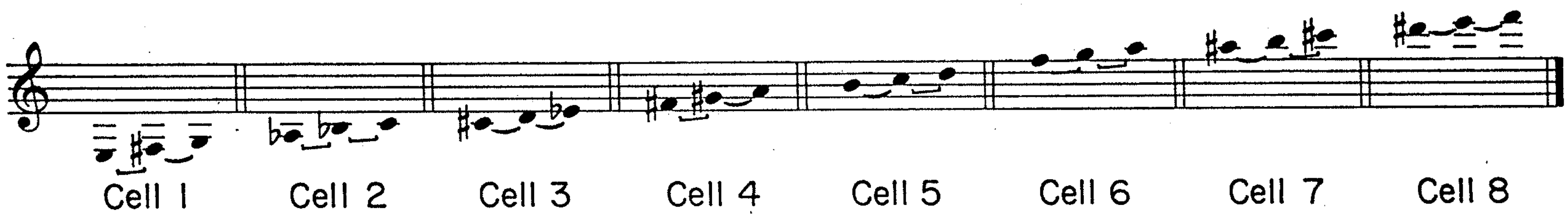


Fig 14-6

2. of the four intervallic structures possible given the preceding constraint, each structure appears exactly twice,

3. no two cells overlap,

4. no degree of the chromatic scale appears more than twice in all the material, and

5. no two cells share more than one common chromatic degree.

Figure 14-6: Material for Demonstration 11 - Curved brackets indicate semitones; square brackets indicate whole tones.

14.4.1.2 Stage II: Form - The piece ~~work~~ consists of 18 segments. Eight segments draw material from one cell only; five segments draw material from two cells simultaneously; three segments draw material from three cells; and the remaining two draw material from four cells at once. Since the effect is of "implied counterpoint", it will be appropriate to use the word 'part' to

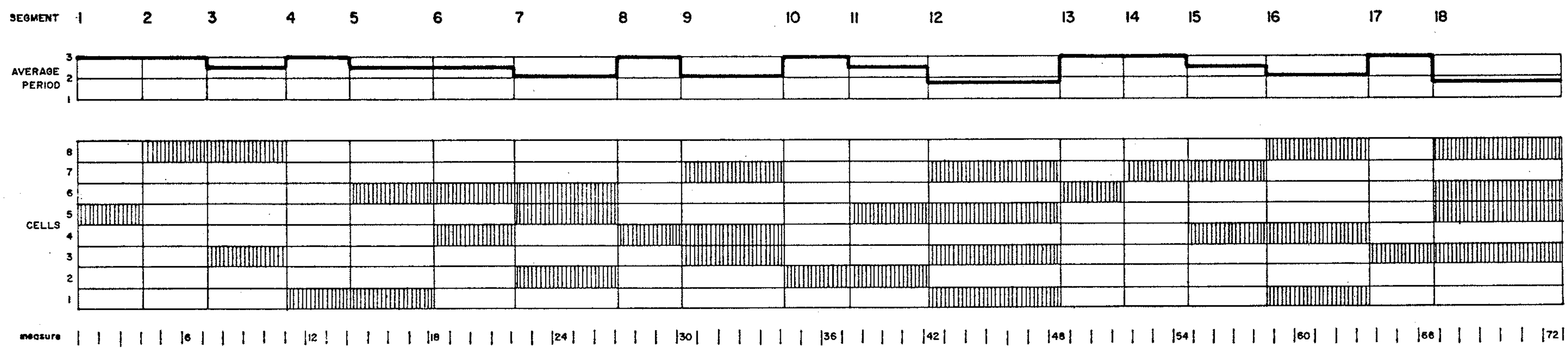
SEGMENT
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
AVERAGE PERIOD
3
2
1
CELLS
8
7
6
5
4
3
2
1
measure
6
12
18
24
30
36
42
48
54
60
66
72

Fig 12-7

K-21a

distinguish between cells simultaneously exploited in a single segment and also to refer to the various segments as 'solos', 'duets', 'trios', and 'quartets', depending on the number of parts involved.

Figure 14-7: Profile of Demonstration 11 - Segment durations, numbers of simultaneous cells, and average periods were specified manually by the author; the cellular content of each segment was composed by computer.

The constraints governing selection of cells for ~~for each~~ segments are:

1. No two solos, duets, trios, or quartets may share an identical configuration of cells; neither may two quartets share more than two cells. This constraint insures a diversity of segments.

2. Two cells in adjacent registers may not occur in the same segment if their closest inflections lie within a minor third. This constraint inhibits 'cross channeling' between registrally adjacent cells.

3. A solo may not exploit any cell appearing in the immediately preceding segment; duets, trios, and quartets must share at least one cell with their immediate predecessor if the number of parts remains the same or increases. These constraints serve to provide a 'dovetailing' effect between consecutive segments.

14.4.1.3 Stage III: Rhythm - Stage III of the composing process selects periods between consecutive attacks by direct random selection using an exponential distribution ~~modified by~~ limited using John Myhill's procedures (heading 4.4.2.1) so that the ratio of maximum to minimum durations is 8.0. Figure 14-7 details the average period between attacks for each segment. Notice that this average decreases (equivalently, the density of notes increases) as amount of available material rises.

The program selects cells using random selection with cumulative feedback (heading 7.2). This procedure allows unpredictable short-term choices while balancing cell-usage balances out over the long-term.

Articulation is sensitive to whether or not a note's successor shares the same cell:

1. If two consecutive notes share the same cell, then the program acknowledges this relationship by indicating either that the pair should be slurred or that the successor should be tongued with no intervening rest. This decision is conducted by Bernoulli trial (heading 4.4.1.1); the more parts occuring in a segment, the greater the likelihood that the pair will be slurred.

2. If two consecutive notes exploit different cells, then program acknowledges this difference by insisting that the successor always be tongued. A Bernoulli trial with 50% probability of success decides whether or not the program inserts an intervening sixteenth rest.

14.4.1.4 Stage IV: Pitch - The final stage of the composing process selects for each note in the composition which of the three registral inflections available to the cell specified in Stage III will provide the pitch. The program attempts to keep these inflections in balance by employing cumulative feedback in order to favor the least-used inflection of any cell. It also forbids any cell from repeating an inflection without in the meantime stating at least one of ~~its~~ the cell's two alternate inflections.

Since the type of harmonic connections suggested by inter-cell consonances would contradict the central ~~idea~~ motivations of the piece, the pitches in Demonstration 11 adhere to a dissonant style. As a minimum precaution against 'cross-channeling', the pitch-selecting program avoids virtual octaves; that is, when one cell plays a chromatic degree shared by a second cell, at least one of the two cells must play a different degree before the second cell may use the shared degree. In addition, consecutive notes must obey the stylistic matrix illustrated in Figure 14-8. Unlike stylistic constraints employed for Demonstrations 6, 7, and 8, the program skips over chromatic identities in order to apply this matrix to the first two <u>distinct</u> degrees immediately preceding the current note.

Figure 14-8: Stylistic matrix for Demonstration 11. Columns indicate 'current' chromatic intervals, given the 'most recent' interval indicated by the row. Non-blank entries show 'acceptable' intervallic sequences.

Figure 14-9: Transcription of Demonstration 11.

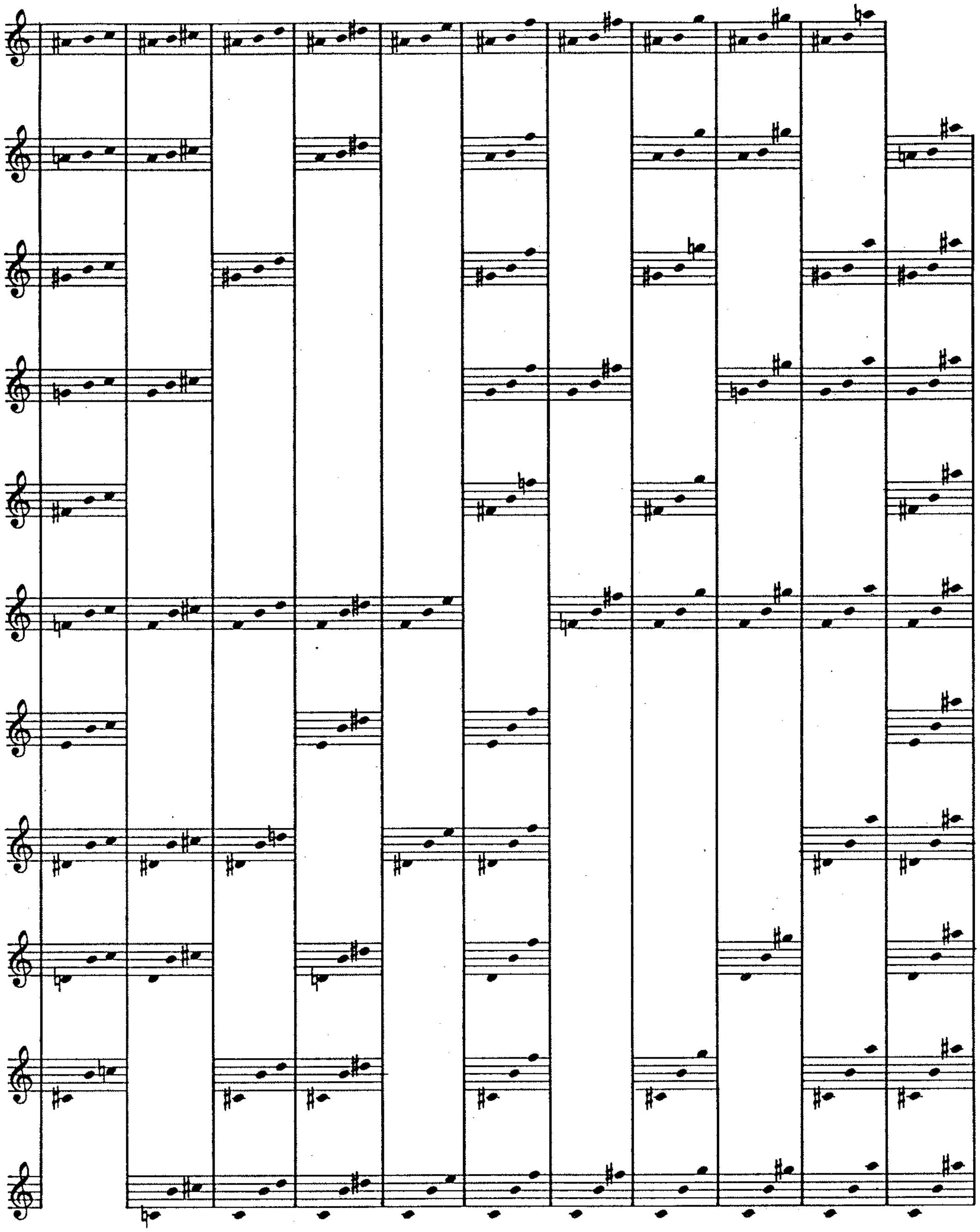

Fig 12-8

14-25b

# Demonstration II

Clarinet

Charles AMES

STRICTLY ♩ = 80




Fig 12-9

```
      program DEMO11
C
C     Demonstration of constrained search
C
      parameter (MCEL=8,MPRT=35,MSEG=18)
      integer NUMSEG(0:MSEG),LIMSEG(0:MSEG),DURSEG(MSEG),CELPRT(MPRT)
      real    CUMCEL(MCEL),INCSEG(MSEG)
      common  CUMCEL,NUMSEG,LIMSEG,DURSEG,INCSEG,CELPRT
C
      IPRT = 0
      LIMSEG(0) = IPRT
      do (ISEG=1,MSEG)
        IPRT = IPRT + NUMSEG(ISEG)
        LIMSEG(ISEG) = IPRT
        INCSEG(ISEG) = float(DURSEG(ISEG))/float(NUMSEG(ISEG))
      repeat
      call FORM
      call RHYTHM
      stop
      end

      subroutine FORM
      parameter (MCEL=8,MPRT=35,MSEG=18)
      integer NUMSEG(0:MSEG),LIMSEG(0:MSEG),DURSEG(MSEG),CELPRT(MPRT)
      integer BAKSEG(MSEG),IDXCEL(MPRT),CELIDX(MCEL,MSEG),
     :        ILGCEL(MCEL,MCEL)
      logical LEGCEL(MCEL,MCEL),OKAY
      real    CUMCEL(MCEL),INCSEG(MSEG)
      real    FUZCEL(MCEL)
      equivalence (ILGCEL,LEGCEL)
      common  CUMCEL,NUMSEG,LIMSEG,DURSEG,INCSEG,CELPRT
      data ILGCEL / 0, 0,-1,-1,-1,-1,-1,-1,
     :              0, 0, 0,-1,-1,-1,-1,-1,
     :             -1, 0, 0,-1,-1,-1,-1,-1,
     :             -1,-1,-1, 0, 0,-1,-1,-1,
     :             -1,-1,-1, 0, 0,-1,-1,-1,
     :             -1,-1,-1,-1,-1, 0, 0,-1,
     :             -1,-1,-1,-1,-1, 0, 0, 0,
     :             -1,-1,-1,-1,-1,-1, 0, 0/
C
C     Initialization
      do (ICEL=1,MCEL)
        CUMCEL(ICEL) = 0.0
        do (ISEG=1,MSEG)
          CELIDX(ICEL,ISEG) = ICEL
        repeat
      repeat
C
C     Search for acceptable arrangement of cells
      ISEG = 1
      NUM = NUMSEG(ISEG)
      LIM0 = LIMSEG(ISEG-1)
      LIM1 = LIMSEG(ISEG)
      IPRT = 1
      LCEL = MCEL - NUM + 1
      BAKSEG(ISEG) = 0
      IDXCEL(IPRT) = 0
C     Schedule cells for first segment
      call FUZZY(CELIDX(1,ISEG),CUMCEL,FUZCEL,1.0,MCEL)
      do
        I = IDXCEL(IPRT) + 1
        if (I.le.LCEL) then
          IDXCEL(IPRT) = I
          ICEL = CELIDX(I,ISEG)
          CELPRT(IPRT) = ICEL
C         Constraints:
          OKAY = .true.
          IBAK = ISEG
C         No duplicate segments;  four-part segments may not share
C         more than two cells
          if (IPRT.eq.LIM1) then
            do (IS=1,ISEG-1)
              if (NUMSEG(IS).eq.NUM) then
                K = 0
                IP = LIM0
                do
                  IP = IP + 1
                  IC = CELPRT(IP)
                  LP = LIMSEG(IS-1)
                  do (NUM times)
                    LP = LP + 1
                    if (IC.eq.CELPRT(LP)) then
                      K = K + 1
                      exit
                    end if
                  repeat
                  if (IP.eq.LIM1) exit
                repeat
```

```
            if (K.eq.NUM .or. K.ge.3) then
              OKAY = .false.
              IBAK = IS
              exit
            end if
          end if
        repeat
      end if
C     Test for unacceptable pair of cells in same segment.
      IP = LIMO
      do
        IP = IP + 1
        if (IP.eq.IPRT) exit
        if (.not.LEGCEL(CELPRT(IP),ICEL)) then
          OKAY = .false.
          exit
        end if
      repeat
C     Count number of cells shared with preceding segment
      if (IPRT.eq.LIM1) then
        N = NUMSEG(ISEG-1)
        K = 0
        IP = LIMO
        do
          IP = IP + 1
          IC = CELPRT(IP)
          LP = LIMSEG(ISEG-2)
          do (N times)
            LP = LP + 1
            if (IC.eq.CELPRT(LP)) then
              K = K + 1
              exit
            end if
          repeat
          if (IP.eq.IPRT) exit
        repeat
C       Solo may not share cell
        if (NUM.eq.1) then
          if (K.gt.0) then
            OKAY = .false.
            IBAK = min0(IBAK,ISEG-1)
          end if
        else if (NUM.ge.NUMSEG(ISEG-1) .and. K.ne.1) then
C         Other segments must share one cell if number of cells stays the
C         same or increases
          OKAY = .false.
          IBAK = min0(IBAK,ISEG-1)
        end if
      end if
C     Accept or reject this cell
      if (OKAY) then
C       Cell is acceptable for this part
        CUMCEL(ICEL) = CUMCEL(ICEL) + INCSEG(ISEG)
C       Advance to next part
        IPRT = IPRT + 1
        if (IPRT.gt.LIM1) then
C         Advance to next segment
          ISEG = ISEG + 1
          if (ISEG.gt.MSEG) return
          NUM = NUMSEG(ISEG)
          LIMO = LIMSEG(ISEG-1)
          LIM1 = LIMSEG(ISEG)
          LCEL = MCEL - NUM + 1
          IDXCEL(IPRT) = 0
          BAKSEG(ISEG) = 0
C         Schedule cells for next segment
          call FUZZY(CELIDX(1,ISEG),CUMCEL,FUZCEL,1.0,MCEL)
        else
          LCEL = LCEL + 1
          IDXCEL(IPRT) = IDXCEL(IPRT-1)
        end if
      else
C       Cell is not acceptable for this part
        BAKSEG(ISEG) = max0(BAKSEG(ISEG),IBAK)
      end if
```

```
              else
C               Cells exhausted:  Backtrack to preceding part
                if (IPRT-1.le.LIMO) then
C                 Combinations exhausted:  Backtrack to most recent conflict
                  IBAK = BAKSEG(ISEG)
                  if (IBAK.lt.1) stop 'Unsuccessful search.'
                  do
                    IPRT = IPRT - 1
                    if (IPRT.le.LIMO) then
                      ISEG = ISEG - 1
                      NUM = NUMSEG(ISEG)
                      LIMO = LIMSEG(ISEG-1)
                      LIM1 = LIMSEG(ISEG)
                    end if
                    ICEL = CELPRT(IPRT)
                    CUMCEL(ICEL) = CUMCEL(ICEL) - INCSEG(ISEG)
                    if (ISEG.eq.IBAK) exit
                  repeat
                  LCEL = MCEL - NUM + 1
                else
                  IPRT = IPRT - 1
                  ICEL = CELPRT(IPRT)
                  CUMCEL(ICEL) = CUMCEL(ICEL) - INCSEG(ISEG)
                  LCEL = LCEL - 1
                end if
              end if
            repeat
            end

            subroutine RHYTHM
            parameter (MCEL=8,MPRT=35,MSEG=18,MNUM=4)
            integer NUMSEG(0:MSEG),LIMSEG(0:MSEG),DURSEG(MSEG),CELPRT(MPRT)
            real    CUMCEL(MCEL),INCSEG(MSEG)
            real    INCCEL(MCEL),AVGNUM(MNUM),ARTIC(MNUM)
            logical SUCCES
            common  CUMCEL,NUMSEG,LIMSEG,DURSEG,INCSEG,CELPRT
            data AVGNUM/3.0,2.5,2.2,1.7/,ARTIC/.5,.66,.8,1./
            data HUGE/10000000.0/
C
C           Initialize
            open (2,file='DEMO11.RHY',status='NEW')
C           Increments for cumulative feedback in selecting cells for notes
C           determined by cell-usages accumulated in FORM;  likelihood of
C           selecting least-used cell is 3 times smallest increment
            SUM = 0.
            OFFSET = HUGE
            do (ICEL=1,MCEL)
              C = CUMCEL(ICEL)
              SUM = SUM + C
              INCCEL(ICEL) = C
              OFFSET = amin1(OFFSET,C)
              CUMCEL(ICEL) = 0.
            repeat
            OFFSET = OFFSET * 3.0
            SUM = SUM / float(MCEL)
C           Compose rhythm and select cell-numbers for each note
            ITIME = 0
            KTIME = 0
            REMAIN = 0.
            LIM1 = 0
            do (ISEG=1,MSEG)
              NUM = NUMSEG(ISEG)
              LIMO = LIM1
              LIM1 = LIMSEG(ISEG)
              KTIME = KTIME + DURSEG(ISEG)
              AVGPER = AVGNUM(NUM)
              do
C               Select current period
                PER = RANX(AVGPER,8.0) + REMAIN
                IPER = max0(1,PER+0.5)
                REMAIN = PER - float(IPER)
C               Determine largest cumulative statistic for cells in this segment
                T = 0.
                CMAX = 0.
                IPRT = LIMO
                do
                  IPRT = IPRT + 1
                  C = CUMCEL(CELPRT(IPRT))
                  T = T + C
                  CMAX = amax1(CMAX,C)
                  if (IPRT.eq.LIM1) exit
                repeat
```

```
C           Select a cell for current note
            R = RANF() * (Float(NUM)*(CMAX+OFFSET)-T)
            IPRT = LIMO
            do
              IPRT = IPRT + 1
              ICEL = CELPRT(IPRT)
              W = CMAX - CUMCEL(ICEL) + OFFSET
              if (R.le.W) exit
              R = R - W
              if (IPRT.eq.LIM1) exit
            repeat
            CUMCEL(ICEL) = CUMCEL(ICEL) + Float(IPER)*INCCEL(ICEL)
C           Articulate preceding note
            if (ITIME.gt.0) then
              IDUR1 = IPER1
              if (ICEL.eq.ICEL1) then
                if (SUCCES(ARTIC(NUM1))) IDUR1 = IDUR1 + 1
              else
                if (IPER1.gt.1 .and. SUCCES(0.5)) IDUR1 = IDUR1 - 1
              end if
              write (2,100) ITIME1,IPER1,IDUR1,ICEL1,ISEG1
  100         format (5I5)
            end if
            ITIME1 = ITIME
            IPER1 = IPER
            ICEL1 = ICEL
            ISEG1 = ISEG
            NUM1 = NUM
C
C           Advance to next note
            ITIME = ITIME + IPER
            if (ITIME.ge.KTIME) exit
          repeat
C         Subtract expected cumulative sum for each cell used in this segment
          IPRT = LIMO
          do (NUM times)
            IPRT = IPRT + 1
            ICEL = CELPRT(IPRT)
            CUMCEL(ICEL) = CUMCEL(ICEL) -  SUM*INCSEG(ISEG)
          repeat
        repeat
        write (2,100) ITIME1,IPER1,IPER1,ICEL1,ISEG1
        write (2,100) -1,-1,-1,-1,-1
        close (2)
        return
        end

        block data
        parameter (MCEL=8,MPRT=35,MSEG=18)
        integer NUMSEG(0:MSEG),LIMSEG(0:MSEG),DURSEG(MSEG),CELPRT(MPRT)
        real    CUMCEL(MCEL),INCSEG(MSEG)
        common  CUMCEL,NUMSEG,LIMSEG,DURSEG,INCSEG,CELPRT
        data NUMSEG/0,1,1,2,1,2,2,3,1,3,1,2,4,1,1,2,3,1,4/
        data DURSEG/25,25,31,25,32,31,40,25,40,
       :            25,31,50,25,25,31,40,25,50/
        end
```

```
      program PITCH
      parameter (MCEL=8,MNFL=3,MQUE=50)
      integer HEAD,TAIL,OLDCEL(MCEL),
     :        TIMQUE(MQUE),PERQUE(MQUE),DURQUE(MQUE),CELQUE(MQUE),
     :        SEGQUE(MQUE),OLDQUE(MQUE),NEWQUE(MQUE),CNTQUE(MQUE),
     :        DEGQUE(MQUE),NFLQUE(MQUE)
      integer BAKQUE(MQUE),IDXNFL(MQUE),NFLIDX(MNFL,MQUE),
     :        DEGNFL(MNFL,MCEL),CUMNFL(MNFL,MCEL),IGLTVL(11,11)
      logical LGLTVL(11,11)
      equivalence (LGLTVL,IGLTVL)
      common  HEAD,TAIL,IQUE,ICNT,LIM,OLDCEL,
     :        TIMQUE,PERQUE,DURQUE,CELQUE,SEGQUE,OLDQUE,NEWQUE,CNTQUE,
     :        DEGQUE,NFLQUE
      data IGLTVL/-1,-1,-1,-1,-1,-1,-1,-1,-1,-1, 0,
     :            -1,-1, 0,-1, 0,-1, 0,-1,-1, 0,-1,
     :            -1, 0,-1, 0, 0,-1, 0,-1, 0,-1,-1,
     :            -1,-1, 0, 0, 0,-1,-1, 0,-1,-1,-1,
     :            -1, 0, 0, 0, 0,-1, 0,-1, 0, 0,-1,
     :            -1,-1,-1,-1,-1, 0,-1,-1,-1,-1,-1,
     :            -1, 0, 0,-1, 0,-1, 0, 0, 0, 0,-1,
     :            -1,-1,-1, 0,-1,-1, 0, 0, 0,-1,-1,
     :            -1,-1, 0,-1, 0, 0, 0, 0,-1, 0,-1,
     :            -1, 0,-1,-1, 0,-1, 0,-1, 0,-1,-1,
     :            -1,-1,-1,-1,-1,-1,-1,-1,-1,-1,-1/
      data DEGNFL/ 5, 7, 8, 9,11, 1, 2, 3, 4, 7, 9,10,
     :            12, 1, 3, 6, 8,10,11,12, 2, 4, 5, 6/
C
      open (2,file='DEMO11.RHY',status='OLD')
      open (3,file='DEMO11.DAT',status='NEW')
C
      do (ICEL=1,MCEL)
        do (INFL=1,MNFL)
          CUMNFL(INFL,ICEL) = 0
        repeat
      repeat
      do (IQUE=1,MQUE)
        do (INFL=1,MNFL)
          NFLIDX(INFL,IQUE) = INFL
        repeat
      repeat
C
      HEAD = MQUE
      TAIL = 1
      IQUE = 1
      ICNT = 1
      LIM = 1
      call RNOTE
      ICEL = CELQUE(IQUE)
      call SHUFLE(NFLIDX(1,IQUE),MNFL)
      call ISORT(NFLIDX(1,IQUE),CUMNFL(1,ICEL),MNFL)
      BAKQUE(IQUE) = 0
      IDXNFL(IQUE) = 0
      do
        I = IDXNFL(IQUE) + 1
        if (I.le.MNFL) then
          IDXNFL(IQUE) = I
          INFL = NFLIDX(I,IQUE)
          NFLQUE(IQUE) = INFL
          IDEG = DEGNFL(INFL,ICEL)
          DEGQUE(IQUE) = IDEG
C         Constraints:
          IBAK = ICNT
C         Cell may not have same pitch twice in succession
          IOLD = OLDQUE(IQUE)
          if (IOLD.gt.0) then
            if (IDEG.eq.DEGQUE(IOLD)) then
              IBAK = min0(IBAK,CNTQUE(IOLD))
            end if
          end if
C         No virtual octaves
          do (LCEL=1,MCEL)
            if (LCEL.ne.ICEL) then
              LOLD = OLDCEL(LCEL)
              if (LOLD.gt.0 .and. DEGQUE(LOLD).eq.IDEG) then
                if (IOLD.eq.0 .or. CNTQUE(IOLD).lt.CNTQUE(LOLD)) then
                  if (SEGQUE(IQUE)-SEGQUE(LOLD).le.1) then
                    IBAK = min0(IBAK,CNTQUE(LOLD))
                  end if
                end if
              end if
            end if
          repeat
```

14-25 h



```
C           Sequence of degrees must conform to stylistic matrix
            IOLD1 = IQUE
            do
              if (IOLD1.eq.TAIL) go to 147
              IOLD1 = IRET(IOLD1,MQUE)
              IDEG1 = DEGQUE(IOLD1)
              if (IDEG1.ne.IDEG) exit
            repeat
            IOLD2 = IOLD1
            do
              if (IOLD2.eq.TAIL) go to 147
              IOLD2 = IRET(IOLD2,MQUE)
              IDEG2 = DEGQUE(IOLD2)
              if (IDEG2.ne.IDEG1 .and. IDEG2.ne.IDEG) exit
            repeat
            ITVL1 = IDEG - IDEG1
            if (ITVL1.lt.0) ITVL1 = ITVL1 + 12
            ITVL2 = IDEG1 - IDEG2
            if (ITVL2.lt.0) ITVL2 = ITVL2 + 12
            if (.not.LGLTVL(ITVL1,ITVL2)) then
              IBAK = min0(IBAK,CNTQUE(IOLD1))
            end if
  147       continue
C           Accept or reject this inflection
            if (IBAK.eq.ICNT) then
              CUMNFL(INFL,ICEL) = CUMNFL(INFL,ICEL) + DURQUE(IQUE)
C             Advance to next note
              ICNT = ICNT + 1
              if (IQUE.eq.HEAD) then
                if (IADV(HEAD,MQUE).eq.TAIL) call WNOTE
                call RNOTE
                if (TIMQUE(HEAD).lt.0) go to 300
              end if
              IQUE = IADV(IQUE,MQUE)
C             Schedule inflections for next note
              ICEL = CELQUE(IQUE)
              OLDCEL(ICEL) = IQUE
              call SHUFLE(NFLIDX(1,IQUE),MNFL)
              call ISORT(NFLIDX(1,IQUE),CUMNFL(1,ICEL),MNFL)
              IDXNFL(IQUE) = 0
              BAKQUE(IQUE) = 0
            else
              BAKQUE(IQUE) = max0(BAKQUE(IQUE),IBAK)
            end if
          else
C           Inflections exhausted:  Backtrack to most recent conflict
            IBAK = BAKQUE(IQUE)
            if (IBAK.eq.0 .and. ICNT.gt.1) then
              IBAK = ICNT - 1
            else if (IBAK.lt.LIM) then
              stop 'Unsuccessful search.'
            end if
            do
              IQUE = IRET(IQUE,MQUE)
              ICNT = ICNT - 1
              INFL = NFLQUE(IQUE)
              ICEL = CELQUE(IQUE)
              OLDCEL(ICEL) = IQUE
              CUMNFL(INFL,ICEL) = CUMNFL(INFL,ICEL) - DURQUE(IQUE)
              if (ICNT.eq.IBAK) exit
            repeat
          end if
        repeat
C
  300   do
          if (TIMQUE(TAIL).lt.0) exit
          call WNOTE
        repeat
        close (2)
        close (3)
        stop
        end
```

Ex 14-2 (page 6 of 7)

```
      subroutine RNOTE
      parameter (MCEL=8,MQUE=50)
      integer HEAD,TAIL,OLDCEL(MCEL),
     :        TIMQUE(MQUE),PERQUE(MQUE),DURQUE(MQUE),CELQUE(MQUE),
     :        SEGQUE(MQUE),OLDQUE(MQUE),NEWQUE(MQUE),CNTQUE(MQUE),
     :        DEGQUE(MQUE),NFLQUE(MQUE)
      common  HEAD,TAIL,IQUE,ICNT,LIM,OLDCEL,
     :        TIMQUE,PERQUE,DURQUE,CELQUE,SEGQUE,OLDQUE,NEWQUE,CNTQUE,
     :        DEGQUE,NFLQUE
C
      HEAD = IADV(HEAD,MQUE)
      read (2,10) TIMQUE(HEAD),PERQUE(HEAD),DURQUE(HEAD),CELQUE(HEAD),
     :            SEGQUE(HEAD)
   10 format (5I5)
      if (TIMQUE(HEAD).lt.0) return
      ICEL = CELQUE(HEAD)
      IOLD = OLDCEL(ICEL)
      OLDQUE(HEAD) = IOLD
      if (IOLD.gt.0) NEWQUE(IOLD) = HEAD
      NEWQUE(HEAD) = 0
      OLDCEL(ICEL) = HEAD
      CNTQUE(HEAD) = ICNT
      return
      end

      subroutine WNOTE
      parameter (MNFL=3,MCEL=8,MQUE=50)
      character*3 MNENFL(MNFL,MCEL)
      integer HEAD,TAIL,OLDCEL(MCEL),
     :        TIMQUE(MQUE),PERQUE(MQUE),DURQUE(MQUE),CELQUE(MQUE),
     :        SEGQUE(MQUE),OLDQUE(MQUE),NEWQUE(MQUE),CNTQUE(MQUE),
     :        DEGQUE(MQUE),NFLQUE(MQUE)
      common  HEAD,TAIL,IQUE,ICNT,LIM,OLDCEL,
     :        TIMQUE,PERQUE,DURQUE,CELQUE,SEGQUE,OLDQUE,NEWQUE,CNTQUE,
     :        DEGQUE,NFLQUE
      data MNENFL/' E3','F#3',' G3', 'Ab3','Bb3',' C4',
     :            'C#4',' D4','Eb4', 'F#4','G#4',' A4',
     :            ' B4',' C5',' D5', ' F5',' G5',' A5',
     :            'A#5',' B5','C#6', 'D#6',' E6',' F6'/
C
      ITIME = TIMQUE(TAIL)
      MEAS = ITIME/8
      IBEAT = ITIME - MEAS*8
      IGAP = PERQUE(TAIL) - DURQUE(TAIL)
      ICEL = CELQUE(TAIL)
      if (IGAP.lt.0) then
        write (3,10) MEAS+1,IBEAT,PERQUE(TAIL),
     :               MNENFL(NFLQUE(TAIL),ICEL)
   10   format (I2,':',I1,I4,2X,A3)
      else if (IGAP.eq.0) then
        type  (3,10) MEAS+1,IBEAT,PERQUE(TAIL),
     :               MNENFL(NFLQUE(TAIL),ICEL)
        write (3,15)
   15   format ('    Break')
      else
        type  (3,10) MEAS+1,IBEAT,DURQUE(TAIL),
     :               MNENFL(NFLQUE(TAIL),ICEL)
        write (3,20)
   20   format ('    Rest')
      end if
      INEW = NEWQUE(TAIL)
      if (INEW.gt.0) then
        OLDQUE(INEW) = 0
      else if (TAIL.eq.OLDCEL(ICEL)) then
        OLDCEL(ICEL) = 0
      end if
      TAIL = IADV(TAIL,MQUE)
      LIM = LIM + 1
      return
      end

      function IADV(I,M)
      IADV = I + 1
      if (IADV.gt.M) IADV = IADV - M
      return
      end

      function IRET(I,M)
      IRET = I - 1
      if (IRET.lt.1) IRET = IRET + M
      return
      end
```

### 14.4.2 Implementation

-- Programming example 14-2: program DEMO11 (7 pages) --

Program DEMO11 proper serves merely as a controlling program for the two major subroutines FORM and RHYTHM. FORM selects material (note 2) for segments (Stage II), while RHYTHM composes all of the notes in the piece to the extent of describing periods, cells, and articulations (Stage III). RYTHM stores its intermediate results in the file DEMO11.RHY for later processing by the independent program PITCH. This last program selects inflections for each note (Stage IV) and creates a mnemonic listing of the final products.

14.4.2.1 Searching for an Acceptable Form - Subroutine FORM implements a constrained search which selects from one to four cells for each segment. The initial data resides in two arrays: array element NUMSEG(I) holds the number of parts in the Ith segment while array element DURSEG(I) holds the segment's duration in sixteenths (data for these two arrays are provided by lines 6-8 of the BLOCK DATA subroutine). FORM stores and accesses cells for each part in each segment by employing arrays CELPRT and LIMSEG along with the following scheme of pointers: cells selected for the Ith segment reside in elements LIMSEG(I-1)+1 through LIMSEG(I) of CELPRT. Program DEMO11 proper automatically computes the relative positions stored in LIMSEG from the cell counts stored in NUMSEG (lines 10-16; excepting line 15).

The varible IPRT serves as the recursive index and as a pointer to the part and segment currently under consideration. Notice that FORM does not reset IPRT to 1 when it advances to a new segment; referring to Figure 14-7 for examples, IPRT=2 for segment 2, part 1; IPRT=7 for segment 5, part 2; and so on.

Since a cell may appear no more than once within any segment, FORM derives one schedule of cells for the whole segment and then proceeds to allot cells to parts by sampling this schedule <u>without replacement</u>. Array element CELIDX(J,I) holds the cell scheduled Jth in line for the Ith segment; array element IDXCEL(K) indicates which position in the schedule is currently being considered for the segment and part accessed by the pointer K; therefore, the actual cell number resides in array element CELIDX(IDXCEL(J),I). Sampling without replacement is effected by maintaining IDXCEL(LIMSEG(I-1)+1) through

IDXCEL(LIMSEG(I)) in strictly increasing order (due to line 137 of FORM).

Prior to the search, program DEMO11 proper computes for each segment the portion of the segment's duration which the program expects to devote to individual parts (line 15 of the loop spanning lines 10-16), storing this value in the real array INCSEG. Array CUMCEL maintains statistics of cumulative usage for each of the eight cells; each time it selects a cell, FORM increments the appropriate element of array CUMCEL by INCSEG(ISEG). Calls to the library subroutine FUZZY (heading 9.2) effect random scheduling with a strong bias -- the offset of 1.0 falls far short of the smallest value stored in INCSEG -- toward the least-used cells (lines 38 and 134 of FORM).

The bulk of subroutine FORM imposes the constraints upon the search. The tests for segments with identical material (lines 50-75) compare the current segment to every preceding segment, counting up the number of common cells in each case. The test for too-close cells (lines 77-85) steps through each cells already selected for the current segment and consults the logical array LGLCEL (initialized in lines 11-18) in order to determine whether or not the cell currently under consideration is compatible with this earlier commitment. Requirements for dovetailing (lines 88-116) are confirmed by first counting up the number of cells shared with the most recent segment, then considering special cases.

Each time FORM encounters a configuration which proves unviable in the light of choices made for an earlier segment, it updates the backtracking variable BAKSEG(ISEG). FORM scrupulously considers every possible configuration of cells for the current segment until either it discovers a workable arrangement or it runs out of combinations. In the latter case, BAKSEG provides the most recent segment responsible for any conflict.

14.4.2.2 Generating Notes - The main body of subroutine RHYTHM consists of an outer loop (lines 32-94) iterating once for each of the 18 segments and an inner loop (lines 38-86) iterating once for each note in a segment.

RHYTHM selects periods between consecutive attacks (lines 40-42) via the library function RANX (heading 4.4.2.1). Average periods reside in array AVGNUM (initialized in line 8) and depend on the number of parts in the current segment, NUMSEG(ISEG).

The subprogram selects cells (lines 44-65) using the methods of the library subroutine DECIDE (heading 7.2). At the end of each segment, RHYTHM steps through the active cells, subtracting each cell's 'expected' cumulative statistic SUM*INCCEL(ICEL) from the actual value CUMCEL(ICEL) (lines 86-91). This procedure insures that if a cell has received its 'fair share' of notes during a segment, it starts out fresh in the next one; however, when cells have been either slighted or overindulged, RHYTHM retains awareness of such imbalances and acts to compensate for them in later segments.

Articulations (lines 67-72) must be selected one step behind in the process since how a note connects to its successor depends on whether the successor exploits the same cell or a different one. RHYTHM selects articulations by asking the library function SUCCES (heading 4.4.1.1) to conduct Bernoulli trials; array ARTIC (initialized in line 8) yields likelihoods that two consecutive notes sharing identical cell numbers will be slurred; as with average this likelihood depends on the number of parts in the current segment. of parts

14.4.2.3 Searching for Acceptable Pitches - The independent program PITCH with its attendent subroutines RNOTE and WNOTE implement a constrained search which selects inflections for each note in the piece. PITCH organizes data pertaining to individual notes in several parallel queues (heading 10.2.1). Each queue is distinguished by the mnemonic 'root' QUE; the following mnemonic prefixes signify information read in by PITCH from the intermediate file DEMO11.RHY:

1. TIM - Starting time in sixteenths,
2. PER - Period to next attack in sixteenths,
3. DUR - Duration of note in sixteenths,
4. CEL - Melodic cell (1-8), and
5. SEG - Segment (1-18).

Subroutine RNOTE creates three additional items of data per note. Two items, a backward link OLDQUE and forward link NEWQUE, enable PITCH to access notes quickly when it needs information pertinant to specific cells. Figure 14-10 illustrates the linked structure derived by RNOTE for an actual sequence of notes read in from DEMO11.RHY. Pointers to the head of the backward list for each cell reside in the auxiliary array OLDCEL. The third item supplied by RNOTE, a decision count CNTQUE, indicates a note's position in the absolute sequence of decisions; this information assists the backtracking mechanism (note 3).

Figure 14-10: Data structure for program PITCH - Each row of numbers signifies a note; the left network of arrows shows backward links while the right network shows forward links. The information depicted here

| Backward Link | Index | Decision Count | Measure & Beat | Period | Duration | Cell | Segment | Forward Link |
|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 51 | 18:3 | 2 | 2 | 4 | 6 | 4 |
| 50 | 2 | 52 | 18:5 | 2 | 3 | 6 | 6 | 3 |
| 2 | 3 | 53 | 18:7 | 1 | 1 | 6 | 6 | 5 |
| 1 | 4 | 54 | 19:0 | 4 | 4 | 4 | 6 | 7 |
| 3 | 5 | 55 | 19:4 | 1 | 1 | 6 | 6 | 6 |
| 5 | 6 | 56 | 19:5 | 4 | 4 | 6 | 6 | 8 |
| 4 | 7 | 57 | 20:1 | 4 | 4 | 4 | 6 | 19 |
| 6 | 8 | 58 | 20:5 | 2 | 2 | 6 | 6 | 10 |
| 7 | 9 | 59 | 20:7 | 2 | 1 | 4 | 6 | 11 |
| 8 | 10 | 60 | 21:1 | 1 | 1 | 6 | 6 | 12 |
| 9 | 11 | 61 | 21:2 | 6 | 5 | 4 | 6 | 30 |
| 10 | 12 | 62 | 22:0 | 2 | 2 | 6 | 6 | 13 |
| 12 | 13 | 63 | 22:2 | 5 | 6 | 6 | 7 | 14 |
| 13 | 14 | 64 | 22:7 | 2 | 3 | 6 | 7 | 15 |
| 14 | 15 | 65 | 23:1 | 2 | 2 | 6 | 7 | 17 |
| 0 | 16 | 66 | 23:3 | 1 | 1 | 2 | 7 | 18 |
| 15 | 17 | 67 | 23:4 | 1 | 1 | 6 | 7 | 22 |
| 16 | 18 | 68 | 23:5 | 3 | 3 | 2 | 7 | 20 |
| 0 | 19 | 69 | 24:0 | 2 | 2 | 5 | 7 | 24 |
| 18 | 20 | 70 | 24:2 | 1 | 2 | 2 | 7 | 21 |
| 20 | 21 | 71 | 24:3 | 4 | 3 | 2 | 7 | 26 |
| 17 | 22 | 72 | 24:7 | 2 | 3 | 6 | 7 | 23 |
| 22 | 23 | 73 | 25:1 | 2 | 2 | 6 | 7 | 27 |
| 19 | 24 | 74 | 25:3 | 2 | 3 | 5 | 7 | 25 |
| 24 | 25 | 75 | 25:5 | 3 | 2 | 5 | 7 | 0 |
| 21 | 26 | 76 | 26:0 | 3 | 3 | 2 | 7 | 28 |
| 23 | 27 | 77 | 26:3 | 4 | 3 | 6 | 7 | 0 |
| 26 | 28 | 78 | 26:7 | 1 | 1 | 2 | 7 | 29 |
| 28 | 29 | 79 | 27:0 | 1 | 1 | 2 | 7 | 0 |
| 11 | 30 | 80 | 27:1 | 2 | 3 | 4 | 8 | 31 |
| 30 | 31 | 81 | 27:3 | 3 | 3 | 4 | 8 | 32 |
| 31 | 32 | 82 | 27:6 | 1 | 2 | 4 | 8 | 33 |
| 32 | 33 | 83 | 27:7 | 2 | 2 | 4 | 8 | 34 |
| 33 | 34 | 84 | 28:1 | 2 | 2 | 4 | 8 | 35 |
| 34 | 35 | 85 | 28:3 | 2 | 2 | 4 | 8 | 0 |
| 0 | 36 | 36 | 14:4 | 1 | 1 | 6 | 5 | 39 |
| 0 | 37 | 37 | 14:5 | 1 | 1 | 1 | 5 | 38 |
| 37 | 38 | 38 | 14:6 | 2 | 1 | 1 | 5 | 40 |
| 36 | 39 | 39 | 15:0 | 5 | 5 | 6 | 5 | 43 |
| 38 | 40 | 40 | 15:5 | 3 | 3 | 1 | 5 | 41 |
| 40 | 41 | 41 | 16:0 | 2 | 3 | 1 | 5 | 42 |
| 41 | 42 | 42 | 16:2 | 3 | 3 | 1 | 5 | 44 |
| 39 | 43 | 43 | 16:5 | 1 | 1 | 6 | 5 | 47 |
| 42 | 44 | 44 | 16:6 | 1 | 2 | 1 | 5 | 45 |
| 44 | 45 | 45 | 16:7 | 2 | 2 | 1 | 5 | 46 |
| 45 | 46 | 46 | 17:1 | 1 | 1 | 1 | 5 | 48 |
| 43 | 47 | 47 | 17:2 | 5 | 4 | 6 | 5 | 50 |
| 46 | 48 | 48 | 17:7 | 2 | 3 | 1 | 5 | 49 |
| 48 | 49 | 49 | 18:1 | 1 | 1 | 1 | 5 | 0 |
| 47 | 50 | 50 | 18:2 | 1 | 1 | 6 | 6 | 2 |

Fig 14-10

describes the portion of Figure 14-9 beginning with the second quarter of measure 14 and ending after the fifth sixteenth of measure 28.

The head and tail of the queue are indicated by the integer variables HEAD and TAIL, respectively. The varible IQUE both serves as the recursive index and locates the note currently under consideration. The integer functions IADV and IRET handle the "wrap-around" arithmetic necessary to keep this and related indices between 1 and MQUE. Array element NFLQUE(IDXNFL(IQUE),IQUE) holds the inflection under scrutiny; PITCH also transfers this value to the holding variable INFL for more efficient acces. Array DEGNFL (initialized in lines 25-26) supplies chromatic degrees for each inflection in each cell; PITCH transfers the current note's degree from array element DEGNFL(INFL,CELQUE(IQUE)) to the holding variable IDEG. Both INFL and IDEG are also stored in queues of their own for easy future reference: NFLQUE and DEGQUE.

Each time PITCH selects an inflection for a note, the subprogram increments the appropriate element of array CUMNFL by the duration the note. Scheduling is first rendered unbiased by random shuffling (heading 5.2), after which a call to the library subroutine ISORT (heading 9.1) strictly favors the least-used inflections (lines 49-50 and 120-121).

The constraints controlling how degrees of the chromatic scale may recur are greatly facilitated by the linked structure illustrated in Figure 14-10. PITCH initiates its test for inflections repeated immediately within the same cell (lines 65-69) by locating the most recent note exploiting the same cell through array OLDCEL. The test then reduces to simply comparing inflection numbers. The test for virtual octaves (lines 71-82) steps through each cell other than the current note's cell, using OLDCEL to locate the other cell's most recent note. If both notes share the same chromatic degree and if the current cell has no intervening note, then the program rejects the current inflection.

By contrast to the tests just described, the test for conformity to the stylistic matrix illustrated in Figure 14-8 (lines 84-105) is unconcerned with cell numbers. The first step is to locate the two most recent notes in the queue whose pitches are chromatically distinct both from the inflection currently being considered and from each other. The program then feeds the resulting chromatic intervals into the logical array LGLTVL (initialized in the DATA statement spanning lines 14-24) in order to determine if the intervallic sequence is suitable.

The backtracking mechanism for program PITCH first consults array BAKQUE in order to determine the source of an immediate conflict. Sometimes the subprogram determines all of the inflections considered for a note suitable, only to propigate impasse at later notes in each instance. PITCH was unable to supply information for dependency-directed backtracking in such cases, so the subprogram was forced to grope back note-by-note in order to pinpoint the source of conflict empirically.

## 14.5 NOTES

1. A similar information structure is used in program PITCH, described under the next heading.

2. The material for Demonstration 11 was itself composed by computer using the techniques described in this chapter.

3. For PITCH'S purposes, array TIMQUE could easily serve this purpose; however, this implementation is designed to handle truly polyphonic applications which allow several notes to start simultaneously.

## 14.6 RECOMMENDED READING

Nilsson, Nils. Problem Solving in Artificial Intelligence (New York: McGraw-Hill, 1971).

Ames, Charles. "Notes on Undulant", Interface, volume 12, number 4 (1983).