Behavior Research Methods, Instruments, & Computers
1993, 25 (2), 238-241

9. SYMPOSIUM ON DATA VISUALIZATION

Chaired by Frank Marchak, *TASC*

Dynamic programming for the analysis of serial behaviors

EDWARD W. LARGE
Ohio State University, Columbus, Ohio

Studies of breakdowns in music performance can provide rich information about the planning activities required for music performance, as well as offer significant advantages over studies of skilled performance in other domains (Palmer & van de Sande, 1993). Yet despite the potential benefits, documented evidence of errors in music performance is scarce, primarily because of methodological limitations. One important practical problem that arises is how to find a correspondence between the actual performance and the score, or intended performance. When performances are long and complex, with potentially many errors, matching a performance to a musical score becomes a nontrivial task. This paper describes an algorithm for this task, developed in the context of a study of music production errors. The solution to the problem utilizes dynamic programming techniques and runs in polynomial time.

Music provides a rich domain for the study of complex serial behaviors. For example, studies of errors or breakdowns in music performance can provide information about the planning activities required for music performance and also offer significant advantages over other studies of skilled performance (Palmer & van de Sande, 1993). First, the demands of producing many events quickly are so great that errors are fairly common, even in highly skilled performances (Palmer, 1992). Second, the temporal demands of music performance require that musicians continue when an error is made, which provides naturalistic conditions for the study of errors. Third, sampling biases arising from error-collection methods can be avoided by controlling the frequency of occurrence of musical events in a piece. Finally, data collection via computer-monitored musical instruments allows error detection without the perceptual biases that arise in acoustic domains.

Despite the potential benefits of the musical domain, documented evidence of errors in music performance is scarce, primarily because of methodological limitations. In music performance, a score is provided that specifies a sequence of pitch events to be performed, with precisely specified event onset times and durations. The performer is expected to execute precisely those pitch events called for in the score, but deviations in timing and dynamics are expected as part of the performer's artistic license. It can be difficult to distinguish these artistic deviations from actual errors. In addition, there is the practical problem of how to find a correspondence between the actual performance and the score, or intended performance. At first blush, this may seem easy, but when performances are long and complex, with potentially many errors, matching a performance to a musical score becomes a nontrivial task.

This paper addresses this second issue. I describe an algorithm developed in the context of a study of music production errors, in which children and adult pianists learned to play piano pieces. First, the subjects were asked to perform familiar pieces, and then they were asked to learn to play novel pieces. Under both conditions, their performances were recorded on a computer-monitored piano. The goal of the first phase of computer analysis was to code pitch errors according to a scheme similar to that used in speech-error research (Dell, 1986; Garrett, 1975). The problem of determining the correspondence, however, was complicated by two factors. The first was that subject skill levels varied widely, so that although some performances were perfect, others were barely recognizable. The second was that some of the familiar pieces were long and complex, such as Beethoven sonatas, with hundreds to thousands of events. The nature of the data to be analyzed, therefore, required a robust and efficient algorithm.

This research was supported by NIMH Grant 1R29-MH45764 to Caroline Palmer. I thank John Kolen and Caroline Palmer for comments on an earlier version of this paper. Reprint requests should be addressed to E. W. Large, Department of Computer and Information Science, 2036 Neil Avenue, Ohio State University, Columbus, OH 43210.

The first section of this paper describes in some detail the problem of finding the best correspondence between an actual and an intended performance and explains why it presents an interesting computational challenge. The next section gives a solution to the problem using a technique called dynamic programming, and discusses ways in which the basic solution can be improved and refined to yield a robust and efficient computer program. The final section describes some limitations of the technique and the ways in which it can be generalized to the study of performance breakdowns in other domains.

THE PROBLEM

In determining the correspondence between an actual and intended performance (as defined by the musical score), four types of occurrences must be identified. A *match* occurs when the subject performs an intended event correctly. A *substitution* occurs when the subject performs an intended pitch event incorrectly (e.g., plays a wrong note). An *addition* occurs when the subject inserts one or more unintended events between the performance of intended events. Finally, a *deletion* occurs when the subject fails to perform one or more intended events.

More abstractly, let us define a *correspondence* between two sequences of events, $A = \{a_1, a_2, \ldots, a_n\}$ and $I = \{i_1, i_2, \ldots, i_m\}$, the actual and intended performances, as a sequence of pairings $Corr(1,1) = \{a_{i_1} \leftrightarrow i_{j_1}, a_{i_2} \leftrightarrow i_{j_2}, \ldots, a_{i_l} \leftrightarrow i_{j_l}\}$ between events in A and events in I. Each pairing represents either a match or a substitution. An event in A that does not occur in any pairing can be considered to be an addition. An event in I that does not occur in any pairing can be considered to be a deletion. The correspondence must preserve the order of both sequences.

We can create an optimization problem by defining a scoring function for the evaluation of candidate correspondences. For each pairing, a match counts as 2 points and a substitution counts as 1 point. Additions and deletions contribute no points. Finding the best correspondence can then be viewed as optimization of this scoring function.

The naive approach to this problem would be brute-force search: Enumerate all legal correspondences between the actual and intended performance and then pick the best one. But brute-force search is expensive. If n is the number of events in the actual performance, and m is the number of events in the intended performance, the number of legal correspondences is given by[1]

$$\sum_{k=max(n-m,0)}^{n} \binom{n}{k}\binom{m}{n-k} = \frac{(n+m)!}{n!m!}. \qquad (1)$$

This function grows very quickly. For $n = m = 3$, there are only 20 solutions to be checked, but for $n = 16$, $m = 13$ (as in Figure 2), there are 67,863,915 legal correspondences, and most musical performances are much larger. More formally, it can be shown that the series has an exponential lower bound; therefore, brute-force search is out of the question.

A DYNAMIC PROGRAMMING SOLUTION

We have defined this problem, however, so that it satisfies the principle of optimality (Brassard & Bratley, 1988). That is, an optimal solution to this problem will contain within it optimal solutions to subproblems. To see this, imagine an optimal correspondence, $Corr(1,1)$, in which $a_1 \leftrightarrow i_1$ is the first pairing. If $a_1 \leftrightarrow i_1$ is removed, the remainder of the correspondence is an optimal solution to the subproblem $Corr(2,2)$, which is created by removing a_1 from A and i_1 from I. If there were any solution to the subproblem that had a higher score, we could simply add the pairing $a_1 \leftrightarrow i_1$ to that solution, and we would have a better (higher scoring) solution to the original problem. We formalize this notion with the following recurrence relation:

$$score(i,j) = \begin{cases} 0 & \text{, if } i>n, \text{ or } j>m \\ max\left(\begin{array}{ll} 2 + score(i+1,j+1) & \text{(match)} \\ 1 + score(i+1,j+1) & \text{(substitution)} \\ score(i+1,j) & \text{(addition)} \\ score(i,j+1) & \text{(deletion)} \end{array}\right) \end{cases} \qquad (2)$$

It is possible on the basis of this formula to write a recursive algorithm to generate an optimal correspondence, but it, too, would have an exponential running time. The problem now is that this algorithm would generate the solution to the same subproblem more than once.

Fortunately, this overlapping-subproblems property (Cormen, Leiserson, & Rivest, 1990) means that this problem can actually be solved in polynomial time using dynamic programming techniques. Dynamic programming algorithms take advantage of the overlapping-subproblems property by solving each subproblem once and storing the solution in a table where it can be looked up when needed. For this problem, there are in fact only $m \cdot n$ unique subproblems. Figure 1 gives a procedure, ScoreTable, which builds a table containing optimal scores for each of the $m \cdot n$ unique subproblems. The table is constructed bottom-up (from bottom to top, right to left), so that an optimal solution can simply be read from the table.

Here $value(A[i],I[j])$, returns the point value for the pairing $a_i \leftrightarrow i_j$, and rowMax(i,j) returns the maximum score in row i beginning at column j. A table filled ac-

```
ScoreTable(A, I, score)
n ← length(A)
m ← length(I)
for i ← n downto 1
    for j ← m downto 1
        score[i][j] ← value(A[i], I[j]) + rowMax(i+1, j+1, score)
```

Figure 1. Procedure ScoreTable.

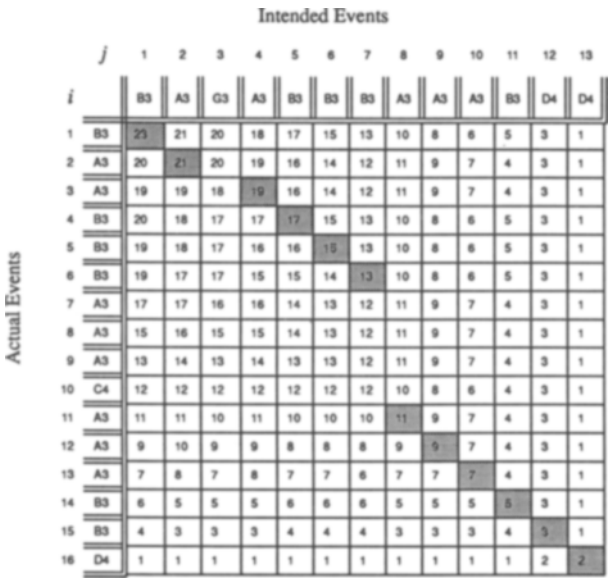

Figure 2. The table created by procedure ScoreTable. Highlighted cells indicate the component pairings of the correspondence printed by procedure ReadSolution (see Figure 3).

```
ReadSolution(A, I, score)
n ← length(A)
m ← length(I)
i ← findColumnMax(1, 1, score)
j ← 1
    repeat
        print('A[i]↔I[j]')
        nexti ← findRowMax(i+1, j+1, score)
        nextj ← findColumnMax(i+1, j+1, score)
        if score[nexti][j+1] > score[i+1][nextj]
            then  i ← nexti
                  j ← j+1
            else  i ← i+1
                  j ← nextj
    until i > n or j > m
```

Figure 3. Procedure ReadSolution.

cording to this algorithm for a sample problem is shown in Figure 2. Each entry in the table, *score*[*i*][*j*], gives the optimal score for subproblem *Corr*(*i*,*j*) according to the recurrence relation given above. Since rowMax must search a row each time it is called, it takes $O(n)$ time, and the entire procedure takes $O(m^2 \cdot n)$ time.

Next, an actual correspondence must be read from the table. Figure 3 shows procedure ReadSolution, which begins with the (bottommost) maximum score in column 1 and steps down and to the right through the table searching for (leftmost, bottommost) maximum subproblem scores. It prints out the constituent pairings of the correspondence as it finds them. The effect of the algorithm can be observed by noting the shaded entries in Figure 2. Here, findRowMax and findColumnMax are similar to rowMax, except these subroutines return the index of the maximum score rather than the score itself. This procedure also takes $O(m^2 \cdot n)$, thus the overall algorithm takes $O(m^2 \cdot n)$ time.

Improving and Refining the Algorithm

The algorithm described above can be improved in an important way. The overall time complexity of $O(m^2 \cdot n)$ is due to the row and column searching operations used in both procedures. The complexity could be reduced to $O(m \cdot n)$ by using auxiliary tables. If the algorithm kept track of the appropriate row and column maxima in the auxiliary tables as the main table is built, the need for searches would be eliminated. The actual program developed for the study described above uses this technique.

The algorithm described in this paper applies only to sequences made up of simple pitch events (i.e., single notes). Most music also contains compound events (i.e., musical chords). It is necessary for the program to identify compound events and apply a slightly more complex scoring function for them.

In addition, the scoring function employed usually admits multiple optimum correspondences, even in situations where human analysts believe that the intent of the performer is unambiguous. The algorithm as presented in this paper resolves the ambiguity simply by taking the maximum score available when doing the row and column searches. In practice, however, this does not always produce satisfactory results. This is because the scoring function fails to take into account information such as event similarity and temporal structure of the performance. The actual program must score event pairings on the basis of similarity (e.g., the number of notes that are the same in two chords, or the pitch height of two sequential notes); it also allows the user to adjust the scoring function depending on the stimulus characteristics.

Finally, the fact that the algorithm gives partial credit for substitutions (which is how it identifies substitutions) enforces a limit on the number of contiguous deletions the algorithm can identify. There is a tradeoff between identifying deletions in order to get an exact match a little farther along in the correspondence and simply considering the events played to be substitutions, which get partial credit and can "overpower" exact matches if there are enough of them. To overcome this difficulty, the scoring function used in the program weights exact matches more heavily than the one presented in this paper.

SUMMARY AND CONCLUSIONS

The computer program based on the algorithm described above is able to analyze even very large performances (over 1,000 events) in a reasonable amount of time. The program can provide output in matrix form and

in a form suitable for input to statistical analysis programs. We estimate error-coding accuracy (by its first-run agreement with human analysts) to be greater than 90%. Users can adjust the scoring function to improve the quality of the correspondence as necessary. Future work will investigate the possibility of representing the output in graphical "piano roll" notation (Palmer, 1989) in order to allow the user an intuitive, graphical means of adjusting the correspondence manually to agree with his/her musical judgment of the performer's intention when automatic analysis fails to give the desired results.

Finally, the algorithm presented in this paper is domain independent. That is, it can be adapted to the analysis of any serial behavior for which an actual performance and an intended performance can be provided. Because of the heuristic nature of the scoring function, however, domain-dependent scoring functions must be developed empirically to suit the more important characteristics of the behavior under study.

REFERENCES

BRASSARD, G., & BRATLEY, P. (1988). *Algorithmics: Theory and practice.* Englewood Cliffs, NJ: Prentice-Hall.

CORMEN, T. H., LEISERSON, C. E., & RIVEST, R. L. (1990). *Introduction to Algorithms.* Cambridge, MA: MIT Press.

DELL, G. S. (1986). A spreading-activation theory of retrieval in sentence production. *Psychological Review*, **93**, 283-321.

GARRETT, M. F. (1975). The analysis of sentence production. In G. H. Bower (Ed.), *The psychology of learning and motivation* (pp. 133-177). San Diego: Academic Press.

PALMER, C. (1989). Computer graphics in music performance research. *Behavior Research Methods, Instruments, & Computers*, **21**, 265-270.

PALMER, C. (1992). The role of interpretive preferences in music performance. In M. R. Jones & S. Holleran (Eds.), *Cognitive bases of musical communication* (pp. 249-262). Washington, DC: American Psychological Association.

PALMER, C., & VAN DE SANDE, C. (1993). Units of knowledge in music performance. *Journal of Experimental Psychology: Learning, Memory, & Cognition*, **19**, 457-470.

NOTE

1. The number of legal correspondences can be counted in the following way. First, decide how many events in A are additions, and throw these away. If $n \geq m$, we must throw away at least $n - m$ events. For each possible number of additions, k, select the k events to throw away. There are $\binom{n}{k}$ ways to do this. Now, match the remaining $(m - k)$ events with the m events in I. Since we must preserve order, all we really need to do is pick $(m - k)$ events of I. There are $\binom{m}{n-k}$ ways to do this.