Final published version of 1984 ICMC Presentation

An On-Line Algorithm for Real-Time Accompaniment

Roger B. Dannenberg

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Real-time accompaniment solves most of the synchronization problems inherent in taped accompaniment; however, this new approach requires the computer to have the ability to follow the soloist. Three subproblems arise: detecting and processing input from the live performer, matching this input against a score of expected input, and generating the timing information necessary to control the generation of the accompaniment. It is expected that the live solo performance will contain mistakes or be imperfectly detected by the computer, so it necessary to allow for performance mistakes when matching the actual solo against the score. An efficient dynamic programming algorithm for finding the best match between solo performance and the score is presented. In producing the accompaniment, it is necessary to generate a time reference that varies in speed according to the soloist. The notion of *virtual time* is proposed as a solution to this problem. Finally, experience with two computer systems that produce real-time accompaniment is summarized.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-81-K-1539.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

Copyright (C) 1985 Roger B. Dannenberg

1. Introduction

Real-time digital music synthesis offers unprecedented flexibility to composers and performers of computer music. Applications have ranged from simple "player piano" systems to interactive improvisation involving both human and computer performers. An interesting possibility is that of a computer-generated accompaniment to a human soloist or ensemble. In this case, the computer is given a score containing parts for the soloist and for the corresponding accompaniment. Because the computer follows a human performer, most of the problems of synchronization between performers and computer-generated tapes is avoided. In fact, new artistic possibilities emerge because the accompaniment can respond to subtleties of the soloist's performance.

The problem of following a soloist can be divided into three subproblems. The first subproblem is to detect what the soloist is doing.[1] The second subproblem is to match the detected input against a score, which contains the expected input. This matching process must be tolerant of performance mistakes and of errors in the detection process. The third subproblem is to produce an accompaniment that follows the soloist. Techniques are needed here to perform the accompaniment at the appropriate rate between detected solo events, and to recover smoothly when it is determined that the accompaniment and soloist have lost their synchronization.

In this paper, I will concentrate on the second subproblem by describing a technique for following a live performer. The technique allows the computer to recover

[1] Alternatively, one might wish to detect indications from a conductor.

from errors in the performance or the detection of the performance. This technique has been implemented successfully on two computer music systems at Carnegie-Mellon University and is currently undergoing further experimentation and development.

Previous work in this area has concentrated on computer input devices, including the sequential drum [7], sonar sensors for conducting [6], pitch detection [4]; and others [8, 2]. Chafe, Renaud, and Rush [3], and Foster, Schloss, and Rockmore [5] considered the problem of following a melodic line in order to produce a transcription of it, but their results are not directly applicable because most of their algorithms require knowledge of the future and therefore cannot be realized in real-time.

At least two problems outside of the realm of music are closely related to that of matching a performance against a score. The problem of finding the difference between two computer data files requires the examination of two sequences to find the best correspondence between them. The UNIX[2] *diff* program is based on an algorithm by Stone [9], but this algorithm would take time proportional to the length of the score for each match in the musical context, and is therefore ruled out as a good real-time algorithm. Another related problem is that of matching computer-detected speech features against known templates for speech recognition. The dynamic programming technique [10, 11] used for "time-warping" in speech research motivated the algorithm described in this paper. A similar application is described by Bloom [1], who uses dynamic programming to synchronize the dialogue for a film soundtrack with a reference dialogue recorded during filming.

In the remainder of this paper, I will first describe the problem in abstract terms that make it easier to reason about. Then, an algorithm that that solves the problem is described, and an efficient implementation of the algorithm is presented. Finally, I will describe my initial application of the algorithm to computer music.

2. The Model

While the human accompanist relies on visual and auditory cues of many types, let us assume that the computer accompanist detects a limited class of performance events. These may be, for example, keys struck by a pianist, pitches sounded by a cellist, or the valve positions of a trumpet player. In any case, it must be possible to compare events from the performance to those in the score to determine whether they are the same or different. A solo performance can be regarded as a stream of events, and a score as an ordered list of expected events. The score stored by the computer is not a conventional specification for the performer, but rather the list of events that would be detected in a correct performance of the solo. (See Figure 2-1).

The problem is then to find the "best" match between the two streams. While "best" could be defined many ways, I chose the following definition: The *best* match is the longest common subsequence of the two streams. In other words, we want to match as many events of the score to the performance as possible, and vice-versa. Figure 2-2 illustrates the best match for two event streams. In this example, the match omits the D from the performance and the A from the score. Events dropped from the performance correspond to extra (wrong) notes, and events dropped from the score correspond to notes that the performer omits.

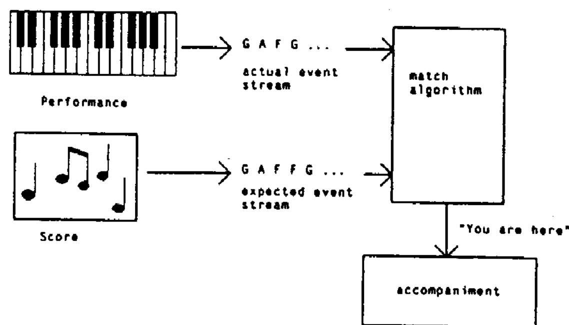

Figure 2-1: The actual event stream from the performance is compared to the expected event stream from the score to determine the present location.

In summary, we can model a solo performance and score as two streams, or sequences, of events. The problem of following the score as the solo progresses is

[2] UNIX is a registered trademark of AT&T Bell Laboratories

```
performance:   A G E D G B C
               | | |  / | |
best match:    A G E  G  B C
               | | | /   | |
score:         A G E G A B C
```

Figure 2-2: Two event steams and the best match.

equivalent to finding the best match between the performance and the score. In the next section, I describe an algorithm that searches for the best match.

3. The Algorithm

To find the best match, an integer matrix is computed where each row corresponds to an event in the score and each column corresponds to a detected performance event. At each detected performance event, a new column is computed. (A more efficient form of this algorithm will be given below.)

The integer computed for row r and column c answers the following question: If we were currently at the rth score event and the cth performance event, what would be the length of the best match up to the current time? The answer to this question can be computed from the answers for the previous column (the previous performance event) and from the previous row of the current column.

The best match up to score event r, performance event c will be at least as long as the one up to $r - 1, c$ because considering one more score event cannot make the best match any shorter. Similarly, the best match up to r, c will be at least as long as the one up to $r, c - 1$, where one less performance event is considered. Furthermore, if score event r matches performance event c, then the length of the match will be at least one greater than the length of the one up to $r - 1, \ c - 1$.

These rules can be applied to compute the length of the longest (best) match, as shown in the algorithm in Figure 3-1. As each performance event is detected, the algorithm computes one more column in the *bestlength* matrix. Figure 3-2 illustrates a matrix for the score A G E G A B C after performance events A G E D.

```
forall i, bestlength[i, -1] ← 0;
forall j, bestlength[-1, j] ← 0;
for each new performance event p[c] do
  begin
    for each score event s[r] do
      begin
        bestlength[r, c] ← max(bestlength[r − 1, c],
                               bestlength[r, c − 1]);
        if p[c] matches s[r] then
          bestlength[r, c] ← max(bestlength[r, c],
                                 1+bestlength[r − 1, c − 1]);
      end
  end
```

Figure 3-1: Basic algorithm to find the best match between the detected performance and the score.

```
           performance:

               A G E D G B C
score:     A   1 1 1 1
           G   1 2 2 2
           E   1 2 3 3
           G   1 2 3 3
           A   1 2 3 3
           B   1 2 3 3
           C   1 2 3 3
```

Figure 3-2: Intermediate state of the computation after the first five events have been performed.

At this point, the algorithm tells us the *length* of the best match, but it does not tell us what events must be matched to obtain this length. This information is required by the accompaniment process. Furthermore, this must be an on-line algorithm, that is, one that gives us results incrementally as the input becomes available. Therefore, we must augment the algorithm to report the position in the score of the current performance event. This is accomplished by remembering the length of the best match up to the current event. This is the largest value in the matrix yet computed. Whenever a match results in a larger value, we assume that a new performance event has matched a score event and report that the performer is at the corresponding location in the score. In Figure 3-3, the matches that cause reports are underlined. Notice that the D, which is performed, but which is not in the score, does not give rise to a report of a score location. Also, when the B is performed, it becomes apparent that the soloist has skipped an A. The algorithm correctly reports the new location in the score that corresponds to the B.

This algorithm can be made more efficient in both space and time. To save space, notice that only the previous and the current columns of the matrix are ever

```
          performance:

                A G E D G B C
score:    A     1 1 1 1 1 1 1
          G     1 2 2 2 2 2 2
          E     1 2 3 3 3 3 3
          G     1 2 3 3 4 4 4
          A     1 2 3 3 4 4 4
          B     1 2 3 3 4 5 5
          C     1 2 3 3 4 5 6
```

Figure 3-3: Completed computation of the best match. Points at which score locations are reported are underlined.

needed. There is no need to store the entire matrix. To save execution time, I make the assumption that the performance will be a close match to the score; that is, the performer will never omit or add more than a few events to the expected score. If this is true, then it is only necessary to look at a small *window* of score events centered around the expected match. However, if the detected performance deviates radically from the score, windows will prevent the algorithm from finding the best match. Figure 3-4 illustrates the values computed for the same performance and score as before, but with a window size of 3. The center of the window is determined as follows: if a match was found and reported in the previous column, then center the window on the row after that match. On the other hand, if no match was found, then move the window down one row. This heuristic causes the window to follow the score if the number of performed events corresponds to the number of score events, even if the events do not match. The window size should be at least one more than double the number of consecutive errors to be tolerated. Alternatively, the window could be adjusted in size to always encompass a given time interval.

```
          performance:

                A G E D G B C
score:    A     1 1
          G     1 2 2
          E     1 2 3 3
          G         3 3 4 4
          A           3 4 4 4
          B             4 5 5
          C                 6
```

Figure 3-4: Computation of the best match using only a small window centered around the current score location.

4. More Heuristics

A problem with the algorithm as described is that it tends to jump ahead in the score when an extra (wrong) event is detected. This happens whenever the extra event matches, by coincidence, a score event in the future. The information is ambiguous: did the soloist produce a wrong event, or did he skip one or more events? Since the goal of the algorithm is to produce the longest match, it jumps ahead in the score, matching the event in the future. This is often the wrong choice.

Of course, the algorithm continues to keep track of the possibility that the event was spurious and not part of the score, so eventually, the correct interpretation should be discovered. This actually happens if, say, the next two detected events only match the score at the correct locations. For example, consider the score B G E F C D and suppose the first three detected events are F B G. The algorithm will initially match the F to the fourth event of the score, but after the B and G are detected, it will be discovered that a better match can be made at the beginning of the score (two events match instead of one).

In practice, the score may be more like the following: B G E F C B D G. In this case, the algorithm would again match F to the fourth score event, but this time, B would match the sixth event, and G would match the eighth! Clearly, some heuristics are needed to discourage (but not prevent) the algorithm from skipping ahead in the score. Three heuristics have been implemented.

The first heuristic limits the amount by which the center of the window can change, as this ultimately limits the rate at which score events can be skipped. The center of each window is computed as before, except the center is only allowed to increase by two rows per detected event (per column).

The second heuristic associates a penalty with matches that skip score events. This is done by counting one point for each pair of matching events and deducting one point for each event skipped in the score. The matrix is constructed in terms of these scores rather than simple match lengths. The effect on the implementation is minimal: line 7 of Figure 3-1 becomes

$$bestlength[r, c] \leftarrow \max(bestlength[r-1, c] - 1, bestlength[r, c-1]);$$

Finally, it is possible for the highest value in the matrix to

occur in several rows of a column, corresponding to several locations in the score. The third heuristic says to always report the match as occurring at the earliest event in the score whenever a choice arises.

5. Accompaniment

The matching process can be used to report the temporal location of the live performer with respect to the score. In this section, I will describe one way to use this information to produce an accompaniment. It is assumed that the accompaniment is already composed and stored in memory, so the only problem is deciding when to perform each event of the accompaniment.

Consider a conventional real-time system (one that does not follow live performers) for realizing a precomposed score. Ordinarily, a starting time would be attached to each event of the score, and these times would be compared to a real-time clock to decide when to start each event.

Now, suppose that one could vary the speed of the "real-time" clock. This would allow the stored score to be performed faster or slower than otherwise. In addition, if the clock could be reset, the score performance could be made to jump forward or backward. These are exactly the properties required for real-time accompaniment. Assuming that the (hardware) real-time clock cannot change speed or be reset, a software *virtual* clock is implemented using the hardware clock as a reference. Virtual time is defined to be:

$$(R - R_{ref}) * S + V_{ref}$$

where R is real time, R_{ref} is the (real) time at which the virtual clock was last set, S is the speed of the virtual clock, and V_{ref} is the virtual time at which the clock was last set. At any time, the virtual clock can be set by assigning the current real time to R_{ref} and the desired virtual clock time is assigned to V_{ref}. In addition, one could set S to establish the "speed" of virtual time, that is, the rate at which virtual time passes relative to real time.

In the current implementation, the virtual clock is set each time a match is reported by the matching process in response to a detected input event. The virtual time is determined from the corresponding event in the solo score, and the real time is determined from a hardware clock. In addition, the speed S is increased slightly whenever the virtual clock is set forward, and S is decreased slightly whenever the virtual clock is set backward.

6. Implementation

An experimental system was constructed using an AGO keyboard for input and a digital synthesizer for output. This system supports a single-voice accompaniment of the solo played on the keyboard. The scores for the solo and accompaniment are read from a file, after which the user can perform on the keyboard and listen to both solo and accompaniment. The program was reimplemented on a small 8-bit microcomputer system that performs real-time pitch detection of trumpet sound as solo input, and generates a single voice accompaniment output. Pitch detection and synthesis are assisted by hardware.

7. Limitations

As mentioned above, the present set of algorithms make no attempt to adjust tempos in a particularly musical manner. Experience has shown that this is not a serious problem as long as only slight adjustments are necessary. Furthermore, no effort has been made to respond to the soloist in any way other than temporally. For example, a human accompanist is expected to respond to loudness, articulation, and other nuances in addition to temporal cues. This is more an omission than a limitation, and there is much room for improvement here.

A more fundamental limitation is the assumption that the input from the soloist is a totally ordered set of events. This assumption is not valid in at least two interesting cases. First, one might use multiple sensors to capture the solo performance. For example, pitch and valve position might provide two parallel streams of events from a trumpet. The other case is where multiple events can take place "simultaneously." What is simultaneous to a human performer is *not* simultaneous at the implementation level where decisions are made in microseconds. Furthermore, the order of events at the microsecond scale is not likely to be under control by performers. Unfortunately, the present matching algorithm *always* considers temporal order, and does not allow matching to sets of unordered, nearly simultaneous events. It is therefore not suitable for polyphonic keyboards or, for example, string quartets.

8. Conclusions

As expected, the algorithm can reliably track and accompany a performer even when extra notes are played or when notes are omitted. The algorithm does not require a steady tempo; in fact, it does not use any timing information whatsoever to follow the soloist.

Many variations of this approach are possible; for example, one might want to take time into account by limiting the distance in time that the accompaniment is allowed to jump forward. One could also consider assigning priorities to various events. For example, if a note is known to be performed or detected unreliably, it might be useful to use this information to follow the soloist more accurately.

It is difficult to assess the "quality" of the accompaniment beyond observing that the accompaniment does indeed follow the performer. In practice, the trumpet input system is surprisingly tolerant of slurs, ornamentation, and even sequences of several wrong notes.

No attempt has been made to incorporate knowledge of musical interpretation into the accompaniment process. For example, if the present implementation discovers it is about to play a sixteenth note and discovers it is behind the soloist by that amount, it simply skips over the note. On the other hand, a human accompanist would be more likely to play the note anyway, and accelerate to catch up with the soloist.

In the case of new music, performance practice is not defined by tradition, and the shortcomings of the present implementation may not be apparent. Furthermore, the present system is fairly simple and predictable; thus, it should cause few surprises when confronted with non-traditional musical material.

9. Acknowledgments

I am happy to acknowledge the help and encouragement of Paul McAvinney and Marilyn Thomas. In addition, conversations with Joshua Bloch have led to improvements in the presentation of material in this paper.

References

1. P. J. Bloom. Use of Dynamic Programming for Automatic Synchronization of Two Similar Speech Signals. Proceedings of IEEE International Conference on Acoustics, Speech, and Signal Processing, 1984, pp. 2.6.1-2.6.4.

2. W. Buxton, W. Reeves, G. Fedorkow, K. C. Smith, and R. Baecker. "A Microcomputer-based Conducting System." *Computer Music Journal 4*, 4 (Spring 1980), 8-21.

3. Chris Chafe, Bernard Mont-Reynaud, and Loren Rush. "Toward an Intelligent Editor of Digital Audio: Recognition of Musical Constructs." *Computer Music Journal 6*, 1 (Spring 1982), 30-41.

4. Jane Clendinning and Paul E. Dworak. Computer Pitch Recognition: A New Approach. 1983 ICMC Proceedings, Computer Music Association, 1983.

5. Scott Foster, W. Andrew Schloss, A. Joseph Rockmore. "Toward an Intelligent Editor of Digital Audio: Signal Processing Methods." *Computer Music Journal 6*, 1 (Spring 1982), 42-51.

6. Steven M. Haflich and Mark A. Burns. Following a Conductor: The Engineering of an Input Device. 1983 ICMC Proceedings, Computer Music Association, 1983.

7. Max V. Mathews and Curtis Abbot. "The Sequential Drum." *Computer Music Journal 4*, 4 (Winter 1980), 45-59.

8. John Snell. "The Lucasfilm Real-Time Console for Recording Studios and Performance of Computer Music." *Computer Music Journal 6*, 3 (Fall 1982), 33-45.

9. . diffreg.c. Source code for the 4.1BSD Berkeley Unix™ *diff* program.

10. A. Waibel, N. Krishnan, R. Reddy. Minimizing Computational Cost for Dynamic Programming Algorithms. Tech. Rept. CMU-CS-81-124, Carnegie-Mellon University Department of Computer Science, June, 1981.

11. A. Waibel and B. Yegnanarayana. Comparative Study of Nonlinear Time Warping Techniques in Isolated Word Speech Recognition Systems. Tech. Rept. CMU-CS-81-125, Carnegie-Mellon University Department of Computer Science, June, 1981.