Real-Time Computer Accompaniment of Keyboard Performances

Joshua J. Bloch and Roger B. Dannenberg

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

We have extended the capabilities of real-time computer accompaniment by developing fast methods for matching polyphonic performances against scores. and we have improved the musicality of the accompaniment through the use of a number of accompaniment heuristics and techniques. A set of algorithms has been designed to handle streams of partially ordered events typified by the key-down events of a keyboard performance. The new algorithms can be viewed as points in a space of alternative designs, where the dimensions are (1) the choice of function that evaluates the quality of a performance/score association, (2) the method of determining when to trust the current guess as to score location, and (3) the method of dealing with compound, unordered events in the score. Two or more alternatives along each of these dimensions are described. Substantial progress has been made in the area of controlling the rate of passage of musical time to achieve a musical accompaniment as the soloist speeds up, slows down, and skips around in the score. Several heuristics for musical accompaniment are presented. Accompaniment systems based on this work are operational, and demonstrate the viability of these techniques.

1. Introduction

An interesting and challenging task for a computer music system is to take on the role of accompanist in a live music performance. Specifically, the system should listen to one or more performers, comparing their performance to a stored representation of the score. Assuming a high correlation between performance and score, the system should synthesize an appropriate accompaniment. For example, if the performer changes tempo, the accompaniment should do likewise, and other parameters such as dynamics and articulation could also affect the accompaniment. It is assumed that a complete score is provided for all parts, so no improvisation or real-time composition is involved. We call this task "computer accompaniment".

To date, only two systems for computer accompaniment have been described [2, 5], and both assume that the score consists of a totally ordered sequence of notes or events. In the present study, we consider *polyphonic* music in which multiple notes (or other events) can occur simultaneously.

In the process of designing and implementing systems that deal with polyphonic input, we encountered a number of design decisions for which there was no clear first choice. Several of these choices are independent, leading to a *design space* of alternative implementations of polyphonic computer accompaniment systems. In Sections 3 through 6, we will take the reader on a guided tour of the known design space. In most cases we will describe the implications of design decisions in a formal way, but that will not necessarily tell us which decision is best. Then in Section 7, we describe new techniques for adjusting the relative time and tempo of the accompaniment. Finally, in Section 8, we will describe our implementation of two different designs. We begin the tour in Section 2 with an overview of accompaniment systems.

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.

Bloch, J. B. and Dannenberg, R.B., Real-Time Computer Accompaniment of Keyboard Performances. In *Proceedings of the 1985 International Computer Music Conference*, (1985), International Computer Music Association, 279-289. http://www.cs.cmu.edu/~rbd/bib-accomp.html#icmc85.

2. Accompaniment Overview

For convenience, we will call the input to the system the *solo*, and the output will be called the *accompaniment*. It should be clear that these are simply designations and do not imply traditional musical form. To distinguish what is actually played from what the composer wrote, we will use the terms *performance* and *score*. The solo score is not a written manuscript, but a machine-readable description of the performance, indicating every expected event and the expected time of the event. Similarly, the accompaniment score specifies every accompaniment event and the time at which it should occur. Time in the score is called *virtual* time, which is "warped" [3, 1] into real time as necessary to match tempo deviations in the real-time solo performance.

In general, a system for automated accompaniment has four important parts (see Figure 2-1). The first part, the *Input Preprocessor* extracts information about the solo from hardware input devices. The Input Preprocessor may be a pitch detector or a keyboard scanner, for example. The derived information is fed to the second part, the *Matcher*, which compares the solo to a stored score. The Matcher reports correspondences between the solo and the score to the *Accompanist* part. The Accompanist decides how and when to perform the accompaniment based on timing and location information it receives from the Matcher. The last part consists of synthesis hardware and software to generate sounds according to commands from the Accompanist. In this paper, we will only be concerned with the Matcher and Accompanist.

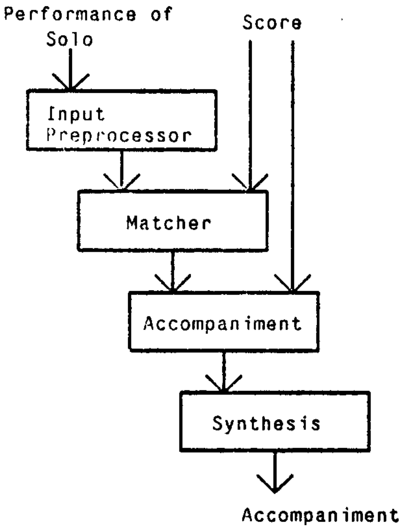

Figure 2-1: Structure of an accompaniment system.

The Matcher has the difficult task of comparing the solo to the score to find the best association between them. The Matcher must tolerate mistakes, and it must produce its output in real-time as the solo is performed. In principle, the Matcher could consider note pitches, starting times, durations, and any other information provided by the Input Preprocessor. In practice, however, we have had great success limiting input information to pitch sequences only for monophonic accompaniment [2]. One of our polyphonic matchers uses additional timing information to group pitches into compound events.

Figure 2-2 illustrates a partial solo performance, its corresponding score, and the best association between them (illustrated by connecting lines). In this monophonic case, the events in the score are totally ordered, and this limits the number of valid associations that must be considered. For example, Figure 2-3 shows an invalid association where the order of the score events does not correspond to the order of the associated performance events. This association would not be allowed (or even considered) by our monophonic matcher. The problem finding the best association between a performance and a score has been solved by using variations of a technique called *dynamic programming* [4].

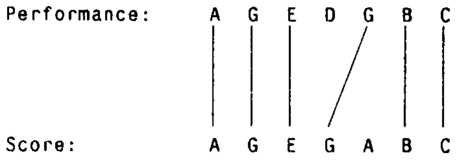

Figure 2-2: A performance and score association.

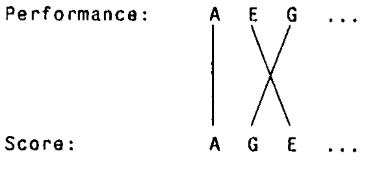

Figure 2-3: An invalid association between a performance and score.

2.1. A Monophonic Matcher

To illustrate the dynamic programming technique for matching, we will describe a simple approach for the monophonic case. This is a slightly edited version of Section 3 in Dannenberg [2]. Our goal will be to find the association with the most matches (lines between equal notes or events). We will call this number the *rating* of the association. An integer matrix is computed where each row corresponds to an event in the score and each column corresponds to an event in the performance. A new column is computed for each performed event.

The integer computed for row r and column c answers the following question: If we were currently at the r^{th} score event and the c^{th} performance event, what would be the highest rating of any association 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 maximum rating up to score event r, performance event c will be at least as great as the one up to $r - 1, c$ because considering one more score event cannot reduce the number of possible matches. Similarly, the maximum rating up to r, c will be at least as great 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 rating will be exactly one greater than the one up to $r - 1, c - 1$.

These rules can be applied to compute the maximum rating obtained by any association as shown in the algorithm in Figure 2-4. As each performance event is detected, the algorithm computes one more column in the *maxrating* matrix. Figure 2-5 illustrates a matrix for the score A G E G A B C after performance events A G E D.

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

Figure 2-4: Basic algorithm to find the maximum rating of any association between the 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 2-5: Intermediate state of the computation after the first five events have been performed.

At this point, the algorithm tells us the maximum *rating*, but it does not tell us what events must be matched to obtain this rating. 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 maximum rating 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 2-6, 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.

In practice, only "windows", or subcolumns centered on the current location are computed, and only the previous column need be saved to compute the current one. Thus storage and computation-per-event are each bounded by constants.

		performance:						
		A	G	E	D	G	B	C
score:	A	$\underline{1}$	1	1	1	1	1	1
	G	1	$\underline{2}$	2	2	2	2	2
	E	1	2	$\underline{3}$	3	3	3	3
	G	1	2	3	3	$\underline{4}$	4	4
	A	1	2	3	3	4	4	4
	B	1	2	3	3	4	$\underline{5}$	5
	C	1	2	3	3	4	5	$\underline{6}$

Figure 2-6: Completed computation of the best match. Points at which score locations are reported are underlined.

2.2. Polyphonic Matching

In contrast to monophonic matchers, a polyphonic matcher must work with a score in which events are only partially ordered. This follows from the fact that whenever events in the score are simultaneous, the actual order of performance is not specified (even though we expect all of the events to occur within a fraction of a second). Thus the chord C E G could be performed G E C, G C E, C E G, C G E, E G C, or E C G.

In general, we can describe a score as a sequence of sets of symbols as illustrated at the bottom of Figure 2-7. Here, symbols within the same set have been circled. In practice, these sets are derived from a computer-readable musical score and indicate the expected sequence of performed events, except the order of events within a set is of no significance. The solo performance is shown at the top of Figure 2-7, and the best association of this performance to the score is indicated by lines. Notice that lines are allowed to cross when they lead to score symbols within the same set, thus the Matcher must consider every possible ordering of each set!

Within this framework, there are at least three design decisions to be made. The first design decision defines the meaning of "best" association. The best association is conveniently defined as one that maximizes a rating function, and various functions are possible.

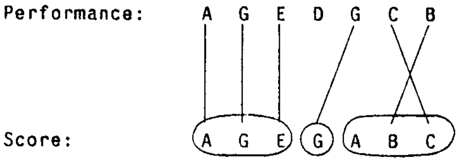

Figure 2-7: An association between a performance and a polyphonic score.

The second design decision tells us when to report the score location to the Accompanist. Intuitively, the Matcher must implement the notion of confidence. If the Matcher is confident that the current best association correctly locates the performance in the score, then this information should be passed to the Accompanist. Otherwise, the Matcher should remain silent and collect more data.

The third design decision is concerned with how to go about grouping the performance events into sets. One method, called *static grouping*, uses timing information to group nearly-simultaneous performance events into sets, and then matches the performance set sequence to the score set sequence. Notice that this reduces the problem to one very similar to the monophonic matching problem: we just substitute event sets for single events. Another method, called *dynamic grouping*, considers all possible groupings of performance events independent of timing in order to achieve the best association. In the following sections, we consider each design decision in turn.

3. Association Rating Functions

Returning to the question of what makes a "good" association, we define the "best" association to be one that maximizes a rating function, which is just a function that takes an association and gives us back a number. For the accompaniment algorithm to work, this matching function must determine the best score prefix for a given performance prefix. We use the notation $p[i]$ for the i^{th} event in the performance, and $p[1:i]$ for the performance prefix consisting of the first i events. Similarly, $s[j]$ refers to the j^{th} event in the score and $s[1:j]$ is the score prefix consisting of the first j events in the score. Then, for a given i, a value of j for which *match-rating*($p[1:i]$, $s[1:j]$) is a maximum must actually correspond to the score prefix which $p[1:i]$ most closely matches. This j value is called *best-match*($p[1:i]$, s). The corresponding match rating is called *best-match-rating*($p[1:i]$, s). In general, there can be several values of j at which *match-rating*($p[1:i]$, $s[1:j]$) obtains a maximum, and some form of tie-breaker must be used to determine which of these values is *best-match*($p[1:i]$, s). We take the smallest value, which corresponds to the shortest score prefix, in the case of a tie.

In constructing an association, we effectively label each performance symbol as either "extra," meaning there is no corresponding symbol in the score, "wrong," meaning the corresponding symbol in the score does not match, or "right," meaning the corresponding symbol does match. In addition, events in the score that do not correspond to performed events are called "missing." A class of functions that seem to give appropriate ratings are of the form:

length of score prefix –
(c_w number of wrong notes +
c_m number of missing notes +
c_e number of extra notes),

where the c's are "cost coefficients" of the three types of errors. In addition to being reasonable from an intuitive musical sense, these functions are amenable to the dynamic programming approach illustrated in Section 2.1, and can be used with either monophonic or polyphonic matchers. Various choices of cost coefficients make sense and will be discussed below.

4. Reporting Matches to the Accompanist

The next design decision determines when to report matches to the Accompanist. This should only happen when the Matcher is reasonably certain that it can report a correct location. If there is significant doubt, it is better to let the Accompanist forge ahead according to the last known location and tempo than to risk giving the Accompanist bad information.

There are two general approaches to formalizing this notion of confidence or certainty, and either approach can be applied to monophonic or polyphonic matching. The first approach is to assume that the best-match-rating will generally increase with the length of the performance, but that it will decrease when errors are made. We report location when the best-match-rating is higher than any previously obtained best-match-rating. This approach (combined with a suitable rating function) has the property that it requires only a few matches to regain "confidence" after a single mistake, but it becomes more cautious, requiring more matches, after a series of mistakes. On the other hand, this approach will only work well for low cost coefficients and low error rates, because high values may lead to a situation where the best-match-rating has a generally downward trend.

The second approach is to base certainty on consistency of the most likely location as predicted by the best-match function. We say that a best-match is good enough to report to the Accompanist if and only if the last note of the performance prefix is consistent with the score position which was most likely on the basis of the previous performance prefix. With our match functions, this occurs at performance event i when *best-match-rating*($p[1:i]$, s) is greater than *best-match-rating*($p[1:i\text{-}1]$, s). This is because the only way to increase the best-match-rating is to extend the previous best association with a new match.

5. The Static Matching Algorithm

The third design decision involves a choice of method to deal with sets of events in the score. The simplest approach to polyphonic matching is called static grouping. The basic idea behind static grouping is to reduce the problem of polyphonic matching to monophonic matching, which we already know how to do. This entails breaking the score up into a (totally ordered) sequence of compound musical events and "parsing" the solo performance into these events in real time. Once this has been accomplished, we can, in principle, apply our monophonic matching algorithm.

The compound events (cevts) into which the polyphonic score is broken consist of all of the note-on events that happen simultaneously, at any given time. A cevt may contain just one note if no other notes occur at the same time. It is clearly trivial break up a printed score into cevts: all of the notes printed at a given horizontal position on each staff make up a cevt.

It is not as easy to break up a performance into cevts because timing information is less exact. When a chord is played on a keyboard, not all of the keys making up the chord will be pressed simultaneously. Furthermore, most real keyboard input systems will serialize the notes that make up the chord, and may end up thinking that some of the notes were pressed later than they actually were due to processing delays from the other notes. But even in a slow system, with a time resolution of 1/100 second, the observed times between consecutive notes in the same chord was never more than 9/100 second (90 ms). Now let us consider how close the times of two consecutive cevts might be. A reasonable upper limit for the speed at which notes are played might be 16th notes played at 120 beats/minute. This comes out to 8 notes per second, or 125 ms. between notes. And most real music is much slower than this. Thus, when a note-on event in a solo performance is detected, it can be determined if it is in the same cevt as the previous note by comparing the times for the two note-on events. If they are from different cevts, they will almost certainly not be within 125 ms, and if they are from the same cevt they will generally be within 90 ms. We arbitrarily chose 100 ms. as a cutoff point. We call this constant *epsilon*.

Unfortunately, it is not quite as easy to break a performance into cevts as the previous paragraph would suggest. The aforementioned method for grouping notes into cevts is not guaranteed to be error-free. While the matching algorithm itself is very robust, and can tolerate errors in the grouping of events into cevts, the performance of the system will be better if these errors are minimized. The primary source of incorrect cevt grouping from the above criterion is rolled chords. If a chord is rolled, the time between constituent notes can easily be greater than epsilon. But it is generally easy to tell when a note belongs to a rolled chord, as it falls much closer to the previous note in the same chord than it does to the first note in the next chord. While the "epsilon test" can be used pretty reliably to say that two notes are in the same cevt, a looser "relative" test should be applied before judging them to be in different cevts. If a note is much closer to the previous note than it is to the predicted time of the next solo event, it is declared to be in the same cevt as the first note, even if they are not within epsilon. Specifically, if the time between two note-on events is less then some fraction of the predicted time from the first event to the next cevt, the second note is grouped with the first. We use 1/4 as the value for this fraction, which we call *epsilon-fraction*. This solves the chord rolling problem in practice. At first glance, 1/4 might appear to be a bit conservative, but there is a good reason epsilon-fraction should not be too high. It sets a limit on how much faster the soloist can play than the Accompanist thinks he is playing. If the soloist exceeds this limit, the Matcher will start grouping together notes that actually belong in different cevts. In fact, a little analysis shows that this maximum tolerable ratio of soloist tempo to Accompanist tempo is the reciprocal of epsilon-fraction.

As each note in the solo performance comes in, we immediately and irrevocably group it into a cevt. Either the note is grouped into the same cevt as the previous note, or the note is grouped into a new cevt, depending on the results of the epsilon/epsilon-fraction test. Since the grouping is fixed and alternatives are not considered, we call the algorithm *static grouping*.

Several more issues must be addressed before we can use the above ideas to build a matcher. First, we must preprocess the solo score into compound events to be matched. If it is written out, the task is trivial, and if it is input at a piano keyboard, we just apply the epsilon/epsilon-fraction test to each note in the piece.

We must also decide how and when the performance cevts under construction are to be compared with the score cevts. In the monophonic matching algorithm, we compute the best-match function each time we process a new solo performance event. In the polyphonic version, we compute a tentative best-match each time we process a solo event, and update the calculation each time we get another note in the same cevt. A report can be made to the Accompanist on the basis of this tentative best-match. When a solo event comes in that is not part of the previous cevt (by the epsilon/epsilon-fraction test), the last tentative best-match calculation is declared correct. This method has the advantage of allowing the Accompanist to get location and timing information as soon as the first note in a chord is hit. Generally, successive notes in the chord will only serve to increase its confidence in the location, but once in a while it will discover that the tentative best-match location was wrong, and send a new location to the Accompanist.

One detail remains to be specified. When the monophonic matcher evaluates best-match, it compares performance events to score events, and there is no difficulty deciding whether a performance event matches a score event. The polyphonic matcher must match performance cevts under construction to score cevts. Thus we need to design a function which, given a partial performance cevt and a score cevt, decides whether they match. Let us call this function *cevt-match*. In order to construct this boolean function, we first design a real-valued function, *cevt-match-rating*, which assigns a match rating to a partial performance cevt and a score cevt. Clearly the rating should go up for each performed note that is in the score cevt and down for each performed note that is not in the score cevt. It seems reasonable that each note should effect the rating in proportion to its representation in the performed cevt. Thus we use the function:

(# performed events in score cevt −
# performed events not in score cevt) /
# performed events.

Cevt-match is true if *cevt-match-rating* is greater than or equal to some threshold value[1]. We use 0.5 as this threshold value, which corresponds to one wrong note in a four-note chord. It works fine, but we have no indication that this value is optimal.

There is a problem with the fact that we accept the last tentative match-rating as correct as soon as the first note in the next performance cevt comes in. If a performed cevt started with the first note in a five note chord, it would be correct to say that it tentatively matched the chord. But if no more notes were played in the cevt, it would not be reasonable to say that they still matched. Thus it would make sense to do one more calculation of best-match with a different cevt-match function before processing the first note of a new cevt. We have not yet tested this idea.

[1] One nice thing about our *cevt-match-rating* is that the numerator of the fractional expression for the function can be updated incrementally each time a new note in the performed cevt comes in. By recasting the comparison test, the numerator can be compared directly rather than calculating the fraction. Thus the function is very cheap to calculate incrementally and involves no multiplication or division.

6. The Dynamic Matching Algorithm

The dynamic grouping algorithm is also similar to the monophonic matching algorithm. The main difference is that dynamic grouping matches a sequence of symbols (notes) against a sequence of symbol sets (chords), while the previous monophonic algorithm matches a sequence against another sequence of symbols. The goal in either case is to find an association between the two sequences that maximizes a rating function as discussed in Section 3.

The primary data structure is a matrix where columns are associated with performance symbols and rows are associated with score sets (see Figure 6-1). Each matrix element consists of an integer called *value*, and a set called *used*. The *value* at row *r*, column *c*, will be the value of the rating function in the best association up to and including score set A_r and performance symbol a_c. The *used* set at row *r*, column *c*, will contain the symbols matched in score set *r* in order to achieve the corresponding *value*. This extra bookkeeping allows us to avoid matching two performance symbols to the same score symbol.

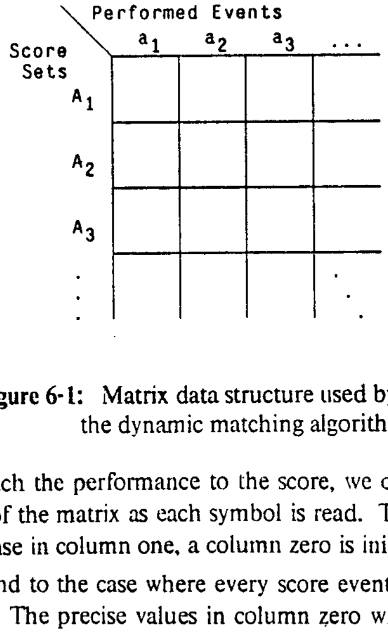

Figure 6-1: Matrix data structure used by the dynamic matching algorithm.

To match the performance to the score, we compute a column of the matrix as each symbol is read. To avoid a special case in column one, a column zero is initialized to correspond to the case where every score event has been skipped. The precise values in column zero will depend upon the choice of rating function. Each other cell is computed in terms of the previous cell in the same row, the cell in the previous row and previous column, and the cell in same column but previous rows. Figure 6-2 labels four cells and we will describe how to compute *z* from *w*, *x*, and *y*. We will label the components of these cells as *w.used*, *x.value*, etc. Also, let *b* be the performance symbol corresponding to the column of *z*, let *A* be the score set corresponding to the row of *w* and *x*, and let *B* be the score set corresponding to the row of *y* and *z*. The algorithm to compute *z* follows. The particular rating function in this instance is the number of correct notes minus the number of skipped or wrong notes. The rating is neither increased nor decreased for extra (performance) notes. The set cardinality operator is "#".

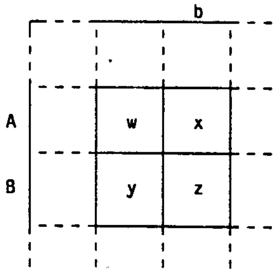

Figure 6-2: Orientation and labels for four matrix cells.

```
$z.value := y.value$;
$z.used := y.used$;
if $b$ in $B$ then
    if $b$ not in $y.used$ then
        $z.value := y.value + 1$;
        $z.used := y.used$ union $\{ b \}$
    endif;
    $d := w.value + 1 - \#(A - w.used)$;
    if $d >= z.value$ then
        $z.value := d$;
        $z.used := \{ b \}$
    endif
endif;
$v := x.value - \#(A - x.used)$;
if $v >= z.value$ then
    $z.value := v$;
    $z.used := \{\}$;
endif
```

By embedding this little program in a loop that iterates down each row of the current column, and by computing a new column for each performance event, we can calculate the entire matrix.

We will now give an intuitive explanation of the algorithm. The basic idea is that we can compute the best association up to position *z* by extending previously computed best associations up to positions *w*, *x* and *y*.

Lines 1 and 2 handle the case where *b* is matched to nothing and symbols before *b* are matched as for *y*. Lines 3 through 7 handle the case where *b* matches a previously unmatched symbol in *B* and other symbols are matched as in *y*. Lines 8 through 12 test to see if a better association could be had by matching *b* with *B* and matching other symbols as in *w*. Finally, lines 14 through 18 consider matching *b* to nothing and matching other symbols as in *x*.

When *z* is in the first row, we must come up with values for *w* and *x*. If we consider the matrix to have a row zero, with initial *value*'s of minus infinity and empty *used* sets, then the tests in lines 9 and 14 will fail and the correct *z* will be computed in row zero.

After many hours of study, it is still not obvious to us that this algorithm covers enough possibilities that we always end up with the maximum *z.value* for all possible associations. A working implementation has demonstrated that the algorithm gives acceptable results in practical situations, but we have no proof at this time that the algorithm *always* gives correct results.

7. The Accompanist

The Input Preprocessor passes each new performance event to the Matcher. Whenever the Matcher thinks it knows which event in the solo score corresponds to the performance event, it reports a match to the Accompanist. Since the solo score indicates a virtual time for each event, the Accompanist can speed up or slow down the accompaniment in order to match the timing of the solo. It is up to the Accompanist to generate a musically acceptable accompaniment on the basis of this information.

In general, the Accompanist synthesizes a virtual time as a linear function of real time; the slope of this function gives us the relative rate of "score time" or virtual time with respect to real time. As new information arrives from the Matcher, the Accompanist must adjust this function to maintain a good fit between computed virtual time and reported virtual times of solo events.

Producing a musical accompaniment as the "virtual clock" is constantly adjusted is a non-trivial task. When the clock is reset, it is not sufficient to merely jump to the new virtual time in the accompaniment score and start playing from there. We must decide what to do about notes that were sounding when the match occurred and notes that should be sounding at the (virtual) time of the match. A specific problem with a naive approach is that it can result in notes with two or more attacks or decays. If a note has just started playing and the Accompanist moves the virtual time to a point just before the note starts playing, it will stop and start again. An analogous effect will occur if a note has just stopped and the virtual time is moved to a point just before the note stops. A real musician will speed up or hold back to come back into synch with other musicians and then resume playing at the speed of the other musicians. Our program should mimic this behavior. We must also address the problem of how and when to change the tempo of the accompaniment.

In fact, solving the accompaniment generation problem in its full generality is a difficult problem in artificial intelligence. However, our program is able to generate adequate accompaniment under most circumstances using fairly straightforward methods which we describe below.

The accompaniment score is stored as a sequence of note-on and note-off-events. In the absence of matches, each accompaniment event is performed when the virtual clock reaches the time of the event. A pointer into the score is maintained in order to know which events have already been performed. When a match occurs, the Matcher passes three pieces of information to the Accompanist: the virtual time of the matched solo event, a flag indicating whether or not the matched event was the next event in sequence after the last matched solo event, and the virtual time of the next solo event in the score. The third piece of information could be easily derived from the first, but it is computationally less expensive for the Matcher to return it.

When a match occurs, the Accompanist invokes two routines. The first, called change_virt_time, has the dual function of correcting the setting of the virtual clock and changing the notes currently sounding. The second, called adjust_clock_speed, has the function of changing the tempo of the accompaniment performance to reflect that of the solo performance. In other words, the two routines adjust respectively the position and speed of the virtual clock, and these tasks are handled completely independently.

7.1. Changing Virtual Time

The first thing change_virt_time (CVT) does is to check the magnitude of the change in virtual time dictated by the match. If the difference between the current virtual time and the time of the matched solo event is less than some minimum value (we use 100 ms.), the time change request is completely ignored. This has the effect of allowing the performer to introduce subtleties into his phrasing (e.g. pushing a note before the beat) without having the accompaniment jump blindly into synch with him. It also results in a "smoother" reading of the accompaniment. If the soloist actually wants to play slower or faster than the accompaniment, the difference between the current time and the time of a matched solo event will quickly grow bigger than the minimum value required to actually process the time change.

When a time change request is actually processed, one of two things happen. If the match was "in sequence" then the accompaniment was merely playing too slow or too fast for the soloist. Rather than jumping around in the accompaniment score, which entails the risk of skipping notes or playing them more than once, the accompaniment merely continues performing the accompaniment events in sequence. If it has lagged behind, it quickly performs the skipped events to catch up. If it has jumped ahead, it continues playing the note that it is playing (if any) until the virtual time for the note to be turned off is reached. At that time, the soloist will presumably have caught up. This behavior approximates the response of a human accompanist.

In some cases, the "in sequence" heuristic described above can lead to rather odd behavior. The problem arises when the soloist makes a drastic change in time, say by miscounting a long rest. When the time change is greater than 2 seconds, it is better to skip to the right place in the accompaniment than to race to catch up or to stop and wait. Our threshold of 2 seconds should really be replaced with a variable threshold that decreases when the density of notes in the accompaniment increases.

If the match resulting in a time change was out of sequence, then the Accompanist's prior notion of the soloist's position in the score was probably wrong. Thus it makes sense to immediately synchronize with him, now that the Accompanist thinks it knows where he is. This is done by making sure that the chord sounding is the chord that should be sounding at the indicated virtual time. The accompaniment score is preprocessed to make it easy to determine this chord. Associated with each event is the number of notes sounding after this event is performed. To determine the chord sounding at any virtual time, the Accompanist merely looks back in the accompaniment score until it sees enough note on-events to account for the number of notes sounding at the last event occurring before or at that virtual time. The synthesizer module itself keeps track of what chord it is currently sounding, and can easily change this chord to the desired chord by turning off any undesired notes and turning on any missing notes. Of course the setting of the virtual clock and the pointer to the next event to be performed in the accompaniment score are updated to reflect the time change.

In summary, CVT employs a two-part approach to virtual clock adjustment which treats tempo mismatches differently from actual location changes. This approach solves, for the most part, the multiple attack/decay problem referred to above. Small time changes of the sort that would cause multiple attacks or decays are generally the result of tempo mismatches, and CVT's method of handling tempo mismatches ensures that the events in the accompaniment score are performed sequentially, temporarily speeding up or slowing down to produce this behavior. This roughly mimics the behavior of a real musician under similar circumstances.

7.2. Adjusting Clock Speed

The second routine invoked by the Accompanist when a match is detected is adjust_clock_speed (ACS). Its function is to maintain the value of the virtual clock speed factor (s). Recall that s is the speed at which the accompaniment is played relative to the speed at which it is written in the score. Clearly, s should be maintained at the speed at which the solo is being played relative to the speed at which it is written.

When the Matcher reports a match, it provides the time of the solo event matched, and this time becomes the current virtual time. By querying the real-time clock, ACS gets a point on the virtual-time vs. real-time graph. The rate at which virtual time is progressing is the slope of this graph. This is the value to which s should be set.

If the soloist were playing his part exactly as written, but changed tempos every once in a while, the virtual-time vs. real-time graph would be a sequence of connected straight lines. In this case, ACS could recompute the slope (change in virtual time / change in real time) after each match and reset s to this value. This would produce nearly instantaneous responses to changes in tempo by the soloist. However, real performances contain local variations in speed that do not indicate actual tempo differences, but merely personal variation in phrasing. In order to filter this out and smooth out the performance of the accompaniment, ACS keeps a table of the last few "match points". (We keep four points in the table.) Each time ACS is called, a new point is added to the table. If the table is already full, the oldest point is shifted out of the table to make room for the new point. If the table is full after the point is added, ACS resets s to the slope of the graph over the range of the points in the table. A more sophisticated approach would be to fit a line to these points and set s to its slope. This would use all points in the table rather than just the first and last, and would be less susceptible to the influence of a small number of "off-time" notes. The only potential disadvantage is the computational expense, which should not be significant in practice.

If a match is detected out of sequence, it makes no sense to consider this point to be on the same line as those prior to it. This could easily result in ludicrous results, like negative tempos. Therefore, the ACS will only accumulate points that represent consecutive events from the solo score in its table. If a point comes in out of sequence, the table is emptied out except for this point.

Two changes were made to the basic clock speed adjustment algorithm to improve its performance. Recall that when a match is detected out of sequence (and at the beginning of the performance) the table of points on the virtual-time vs. real-time graph is emptied out. Recall also that the tempo is not adjusted until the table is full. If the soloist is playing slowly, or it is a slow section of the piece, this could take some time. In the meantime, the accompaniment might run ahead of or lag behind the soloist. To prevent this from happening, ACS will reset the clock speed even if the table is not full, if the real time difference between the first and last point is greater than some minimum value. (We use 1 second.)

The second modification to the algorithm is, in a sense, dual to the first. When a musician plays a fast group of notes, the rhythm of the group is not necessarily a good indication of the tempo at which he is playing the piece as a whole. Therefore, ACS does not bother entering a point in its table if the real time of the point is too close to that of the previous point in the table. (In our program, within 200 ms.) This has the effect of forcing the sample of music on which a tempo adjustment is made to be of at least a certain minimum duration. In practice, this should cause the sample to extend beyond a single fast group of notes.

The clock speed adjustment algorithm will successfully track the tempo of the soloist's performance. But if his tempo is very different from the initial tempo of the accompaniment, the first few seconds of music can sound quite uncoordinated as the Accompanist figures out how fast the soloist is playing. This effect is especially pronounced if the soloist is playing much slower than the accompaniment, because he is not providing as many match points to bring the accompaniment in to synch. To counteract this problem, s should be initialized to the approximate speed at which the solo will be played relative to the speed in the score.

The virtual time and clock speed adjustment algorithms keep the accompaniment synchronized with the solo performance as long as they are called frequently. But if the Matcher loses track of the soloist, or if the soloist stops playing, the Matcher stops reporting matches to the Accompanist. If no special action were taken, the virtual clock would keep ticking away, and the accompaniment would keep playing at the last calculated tempo, oblivious to the fact that the soloist was not playing along. In order to counteract this behavior, the Accompanist is equipped with runaway detection.

Each time a match is reported, the Matcher tells the Accompanist the virtual time of the expected next event. If a sufficient amount of virtual time (in our program, 2 seconds) elapses after the predicted time for the next event and no match is reported, a runaway is declared. Once this happens, any notes currently sounding are turned off and the virtual clock stops moving forward. The runaway is kept in effect until a match is detected, at which point the virtual clock is reset and accompaniment continues. This has the effect of causing accompaniment to cease when the Accompanist knows it is not playing along with the soloist. The accompaniment resumes as soon as the Matcher reports a new location. If the soloist has stopped playing, this will happen as soon as he starts again.

8. Experimental Results

Thus far, we have implemented two polyphonic accompaniment systems, one using static and one using dynamic grouping. We have been able to modularize the software so that new Accompanists and Matchers can be "plugged in" to an experimental system complete with Input Processor, Synthesis module, debugging tools and a user interface. Both systems work well, but neither is clearly better. The static grouping system is more susceptible to errors when the timing of the performance is greatly distorted. On the other hand, the dynamic grouping system never takes any account of timing and therefore may not always model the musician's concept of a good match. In practice, both systems work very well, and only fail on contrived pathological cases..

One interesting feature of the current implementation of static grouping is that because it ignores repeated notes in the same cevt, it is possible to handle trills and other ornamentations with a small amount of score preprocessing. The dynamic grouping system as currently implemented would need some help from the Input Preprocessor to handle trills.

Both systems use the following costs for the rating function:

$$c_w = 2, c_m = 2, c_e = 0.$$

Note that static grouping applies these costs at the level of cevts, whereas dynamic grouping applies these costs at the individual note level. Further studies have led us to consider other choices, particularly

$$c_w = c_m = c_e = 2,$$

but we have no experimental data yet.

Both systems also use the first approach in Section 4 to decide when to report matches to the Accompanist. Again, this approach works well, but may not be optimal. The second approach works well on paper, and we intend to try it in our experimental system.

9. Concluding Remarks

For reasons that are unrelated to our accompaniment software, our experimental systems are too slow for serious performance studies. However, we feel confident that the techniques outlined in this paper can be used to construct a system suitable for live performance of a wide range of scores. Many points in the design space we have described still need to be evaluated. In addition, there are many other avenues left to explore.

Our primary model has been keyboard performance, but there are other possibilities. In ensemble performance, it is certainly possible to merge all of the performed events into one performance and apply our accompaniment techniques; however, more information is available if each performer is tracked individually with his own Input Preprocessor and Matcher. A new module would then be needed to decide how to combine information from Matchers and pass it on to the Accompanist. Ensemble accompaniment is an interesting area for future research.

Another interesting area is the accompaniment of voice. The main problem here is that pitch accuracy is not nearly as high as with most instruments. This leads to a high percentage of spurious pitches at the lowest input level, and greater sophistication of either the Input Preprocessor or the Matcher (or both) is likely to be necessary to achieve satisfactory performance.

Finally, we have only dealt with music in which events are completely planned and notated. This omits a large body of works where improvisation is employed.

10. Summary

We have constructed successful polyphonic computer accompaniment systems, using a piano-like keyboard as the input device. The systems accompany a real-time performance with a stored score, and can handle changes in tempo, wrong notes, extra notes, and skipped notes. The principle technique used to follow the solo performance is a variation of dynamic programming, and we have described a design space of implementation alternatives.

11. Acknowledgments

The authors would like to thank the Carnegie-Mellon University Department of Music for use of the computer music system on which all of our accompaniment systems were constructed and tested

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. Roger B. Dannenberg. An On-Line Algorithm for Real-Time Accompaniment. Proceedings of the 1984 International Computer Music Conference, 1984.

3. D. A. Jaffe. Ensemble possibilities and problems in computer music. Proceedings of the 1984 International Computer Music Conference, 1984.

4. David Sankoff and Joseph B. Kruskal, editors. *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, Reading, Mass., 1983.

5. Barry Vercoe. The synthetic performer in the context of live performance. Proceedings of the 1984 International Computer Music Conference, 1984.