QPD: Query by Pitch Dynamics
Indexing tonal music by content

Doug Beeferman
15-829 Course Project

December 4, 1997

1 Introduction

One-dimensional sequences, parameterized by time, are familiar to us from our interaction with the real world: a piece of text, a sound recording, a musical score, and a company's stock performance are all good examples. Where the elements of such a time series are drawn from a meaningfully ordered vocabulary, such as stock prices or musical notes, there is often meaning in the difference between successive values. The ***relative*** values of nearby elements are more important than their ***absolute*** values in shaping our perception of the sequence. A stock is said to crash when its value drops dramatically, independently of its starting value; a musical piece expresses the same theme regardless of which key it is played in or at what tempo.

In this work I'll consider the problem of indexing this sort of time series for efficient and robust subsequence matching, motivated principally by sequenced tonal music. I have developed a search engine for computer music files supporting content-based retrieval. In particular, the user can specify a sequence of notes, or just the rough "shape" of a pitch sequence, and find the files in a music database that contain some semblance of that sequence. The engine also supports nearest neighbor queries, and clustering, of entire songs.

2 Motivation

We begin by motivating the engine with some practical uses of the system for music indexing. Consider a database of sequenced music files. For the purposes of this study we'll consider such a file to be a single discrete stream of pitch values (frequencies or notes), discarding duration information and disallowing multiple voices. We will consider four possible operations on this database.

2.1 Subsequence queries

An obvious and somewhat whimsical application of subsequence querying is an algorithm to play "Name That Tune", the famous game show. Given a melody that's stuck in our head, what song is it from? Given a few successive notes, which songs contain it? If we're tone-deaf or musically inept, we might not be able to articulate the precise note values in the query, but it's pretty likely we know where the notes lie relative to each other. Or if not, we can certainly tell whether each note is higher or lower than its predecessor.

Supposing that there is a single *reference* song that matches what we're seeking, the subsequence query we specify may differ from one contained in the reference in the same ways a speech recognizer may fail: there may be *inserted* notes, *deleted* notes, and *substitution* errors. Insertions and substitutions, however, are not likely to be arbitrary—more likely the user will introduce a similar note transposed up or down from the reference note, or stutter. This argues in favor of using relative pitch structure as a means of fault tolerance, though some degree of "time warping" is also required.

2.2 Spatial joins and nearest neighbor

We may wish to find pairs of songs in the database that share a substantial chunk of melody. Or, given a fixed song, we may want the most similar song with respect to subsequence containment. These correspond to "spatial join" and "nearest neighbor" queries, respectively. If we allow each song in the database to have some fixed number of (perhaps fixed-size) subsequences, the problem of gauging the distance between two songs can be cast as finding the average (or minimum, say) distance between all subsequence pairs drawn from the songs' melodies. Such a distance measure is insensitive to the way a piece is rendered by a musician—to the subtleties of timbre, tempo, and style that give character to a performance—and focuses instead on the mathematical abstraction that is the score itself.

This could lend an empirical tool to intellectual property lawyers puzzling over whether an artist has violated another's copyright. In the mid-1980s Hulex Music won a settlement from Columbia Pictures, contending that the theme from the movie "Ghostbusters" was copied from the then-popular rock song "I want a New Drug" by Huey Lewis and the News. Fans of Brahms have observed that Andrew Lloyd Weber's famous theme from the musical Evita, "Don't Cry For Me Argentina", is conspicuously similar to Brahms' Violin Concerto. Only similar, though—a few notes are added here and there, and a few others changed. What constitutes infringement? As with text plagiarism, creative theft is not as easy to detect as exact duplication.

2.3 Clustering

A less litigious-minded application is automatic clustering—that is, finding clusters of songs in the database with mutually similar melodic phrases or themes. With such a tool we could expose the

similarity or originality of songs by the same artist, or identify emerging genres of music.

Another application is identifying multiple renderings of the same piece. Music files available in the public domain hail from many sources. Some are transcribed by hand from sheet music with composition software. Others are performed live on synthesizers and other MIDI-aware instruments. Performances are prone to error, of course, and an intelligent database should recognize when two performances are expressing the same content in slightly different ways.

3 Previous work

Systems for content-based retrieval of digitized sounds are starting to appear, exemplified by Musclefish [2]. The Musclefish system extracts time-varying properties from sampled sound files (namely *loudness*, *pitch*, *brightness*, and *bandwidth*). For each property, the mean, variance, and autocorrelation over the entire file is recorded. A feature vector consisting of these 12 values plus the sound's duration is used as a final representation of the sound. Using a weighted Euclidean distance measure between these representations, novel sounds are compared with each sound in the database to find the nearest match. While this system is sufficient for comparing noises like scratches, bells, and animal cries, it is far too crude for sounds with time-varying pitch like an entire piece of music.

The sequence of notes cannot be ignored in a content-based representation of music. A musician named Denys Parsons published a "Directory of Tunes and Musical Themes" in the 1970s [9] that indexes thousands of songs by their first few relative pitches. Using a vocabulary of $\{U, D, \star\}$, representing *UP*, *DOWN*, and *REPEAT*, respectively, he maps each song to a short signature according its opening bars, and then alphabetizes these signatures for fast lookup. There are numerous problems with this approach, however, besides that it is a physically cumbersome book: One, it permits no deviation from the reference sequence; two, random access of reference songs is impossible; and three, the ternary code causes more collisions for a fixed prefix size than if the full relative pitch structure were indexed.

Only one past work simultaneously addresses pitch sequence representation and the digital age. Chou *et al.* [3] use a representation based on *chords* to index songs for subsequence queries. Here a chord is a (serially rendered) set of three or more consecutive notes that harmonize well. (The "D minor" chord, for instance, is the set {re, fa, la}.) Chou fixes a set of chords appropriate for the task (his experiments focus on popular music), and devises a decision algorithm for transducing each song in the database from a sequence of notes to a sequence of chords. This algorithm is informed by a few basic principles (for example. longer chords are preferred where there is ambiguity.) He then casts the subsequence query task as a traditional substring matching problem, using PAT-trees [8] to encode every chord suffix in a song. A user's query is likewise converted to a sequence of chords that is used to query the PAT-tree.

This method achieves some fault tolerance despite the exactness of the required match, in that the mapping from note sequences to chord sequences is many-to-one. A stuttered note, for example, does

not change the identity of the chord it appears in, and in many cases eliminating a note or swapping a consecutive note pair doesn't affect the transduced chord. This is a double-edged sword, however, because a lot of information is lost in the mapping that is unrecoverable without consulting the original file, increasing the chance of false alarms without this extra step. Furthermore, it is insufficient as a mechanism of fault tolerance—the user is still required to know the exact tonal difference between successive notes, as the method is blind to "off by one" errors. The algorithm's biggest weakness, then, is that it treats music fragments as strings which may either match or mismatch, with no allowance for near misses except what is inherent in its coarsening of the representation.

Another flaw in Chou's representation is the black magic needed to conjure up a chord set. The ideal chord set is task-dependent, and the performance of the algorithm is dependent on the set chosen. By not focusing on the common denominator of all tonal music, namely individual tones, Chou's system sacrifices portability.

Finally, although Chou provides no space usage figures for realistic-sized databases, the cost of storing a song in the PAT-tree presumably grows quite large as its length increases, since all suffixes are encoded.

Though Chou's algorithm is impractical for the above reasons, we should note that it succeeds in seamlessly permitting queries of varying lengths. While other indexing schemes (such as the method proposed in section 4) require some special casing to handle query sizes greater than and less than a predefined window length, Chou's use of PAT-trees makes random access queries of any length possible. You simply walk down the tree guided by the chords in your query sequence, and if at any point you cannot proceed, the subsequence cannot be extended and there are guaranteed to be no matches.

We now consider a technique for subsequence matching in time-series databases that, despite not having been originally targeted to the problem of music retrieval, has apparent application to our task. Faloutsos *et al.* [7] introduce the concept of feature space "trails". The idea is that we slide a window of fixed length w over the sequence and extract a feature vector (in some space of dimension lower than w, presumably) at each position. The sequence of feature vectors forms a trajectory in the lower-dimensional space, and if the features are well chosen and if the data is right, successive points will be tightly packed together. As an example, Faloutsos considers stock market values, using the 2-dimensional feature space of 0th and 1st order discrete fourier transform coefficients of the sliding window. From one window to the next the first two DFT coefficients change only slightly.

Exploiting the locality of successive points in the feature space trail, Faloutsos proposes algorithms to divide the trail into a number of subtrails, each of which can be approximated by a minimum bounding hyperrectangle in feature space. These rectangles can then be stored using some spatial indexing scheme such as R-trees. A query sequence of size w is mapped into feature space and, as per the indexing scheme used, associated with one of the bounding rectangles and then with one or more database entry.

Faloutsos handles the case of query size mismatch by splitting the query into overlapping chunks of size w, but does not permit queries of size less than w. Still, this method offers a fairly compact

representation—relatively few rectangles need to be stored instead of many times more subsequences. We also have fault tolerance and automatic support for range queries—all of the tools of existing spatial methods, in fact.

While we could translate the problem of indexing music into an instance of this indexing algorithm, the ideas are fundamentally ill-matched. The problem is that we are concerned with a much lower time resolution in music, and we can't afford windows of a size large enough to produce smoothly-flowing feature space trails. For a reasonable-size window (say, 8 notes), the variation in the coefficients of a transform such as DFT or DWT is too high for successive feature vectors to be clusterable without great overlap in the minimum bounding rectangles. The vocabulary of tonal music is discrete, and as such music in the short-term is not a smooth time series like the stock market.

4 An indexing approach based on the Haar wavelet transform

In this section we introduce an indexing scheme that focuses on the relative pitches of notes within a fixed window. For the discussion below, we will use *target pattern* to mean either the user's query sequence (in a subsequence query) or one of the two sequences being compared in a spatial join or nearest neighbor query. We will use *reference pattern* to mean a subsequence contained in the music database.

4.1 Representation

In our simplified, single-voice. durationless model, we assume a music file to be a sequence of integers drawn from $1, 2, \ldots, N$, where N is the number of distinct pitches available ($N = 88$, the number of keys on a standard piano, is a good lower bound.)

Dirst and Weigend [4], in designing a system to predict the end of Bach's unfinished last fugue, map the reference pattern to a *difference representation* in which the intervals between successive note values are recorded rather than the pitches themselves. This holds the representation fixed under transposition of the data (sliding all the pitches up or down by a fixed value) Eliminating dependence on the starting note value is important for reasons stated earlier, but it is insufficient for robustness. We will go one step further and coarsen the difference representation to be simply the *ranking* of note values in the sequence. The following belief motivates this:

> We can't expect the user to know the exact note sequence that he's targeting. In fact, we can't expect the user to know the exact difference representation. However, we *can* expect him to know approximately how the pitches in his target compare to one another.

All we ask is that the user provide a permutation of the notes in the query sequence, with ties permitted—a *packed* version of the tune. I'll term such a pattern a *p-structure* below. This is less

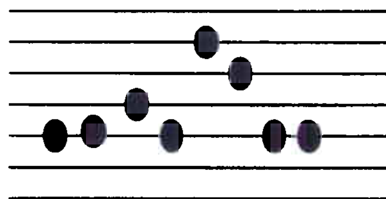

Figure 1: The p-structure for the first 8 notes of "Happy Birthday". (The lines are not meant to suggest a musical staff.)

information than is demanded by Dirst's representation or Chou's method [3], but more information than the ternary sequences of Parsons' index [9]. Figure 1 shows the p-structure of a popular tune.

4.2 Indexing scheme

Underlying our indexing scheme are wavelets and the R-tree spatial access method. Consider a reference music file F. Our approach will be to slide a window of size w over F and record the p-structure of the notes within the window at each step. We will thus extract $|F| - w + 1$ p-structures. Each extracted p-structure will be converted to a point in $w - 1$-dimensional feature space with a transform similar to the Haar wavelet transform; these points will then be indexed in an R-tree T.

We first motivate the Haar wavelet transform by considering a bitstring representation of an arbitrary p-structure. We're for the time being less concerned with space usage and more concerned with the ability to do approximative matching. If a reference p-structure is absent from the database, we'd still like to find similar-sounding sequences quickly, and we'd like "similar-sounding" to correspond in a nice way to "neighboring" in feature space. If we wish to compress our database and don't mind incurring some loss in retrieval accuracy, we'd like to be able to reduce the dimensionality of the space, throwing away some details but retaining the most important information about the melodies in our database. In short, we'd like the earlier bits in our bitstring representation to be the most informative in distinguishing p-structures.

What should the very first bit be? If we had to ask one yes-or-no question about a p-structure, we'd likely want to get some crude estimate of the overall shape of the curve. Does it go up or down or stay the same? We quantify the change by comparing the sum of the ranks in the left half of the p-structure with the sum of the ranks in the right half of the p-structure. If the left sum is larger than or equal to the right, we encode a 0, and if it is smaller we encode a 1.

Now suppose the window size w is of the form 2^k[1]. Then we can proceed by recursively characterizing the shapes of the left and the right halves of the p-structure. The base case is a comparison of two notes yielding a single bit. This recursion leads to a bitstring of length exactly $w - 1$. It makes sense

[1] We will use k=3 for the experiments in this paper.

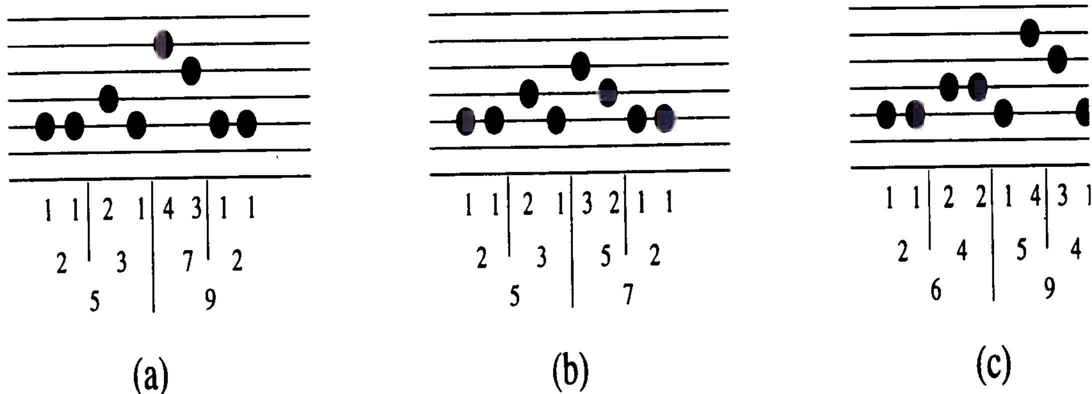

Figure 2: **(a)** The partial sums tree of the "Happy Birthday" example. Its 7-bit primary signature, obtained by reading off the partial sum comparison results in level-order from the root at the bottom, is "1100000". Its wavelet-transformed representation, before weighting, is (-4, 5, -1, 0, 1, 1, 0). **(b)** The effect of changing the sixth note in (a) such that it's equal to the third note. The primary signature doesn't change. The wavelet-transformed vector is (-2, 3, -1, 0, 1, 1, 0). **(c)** The effect of stuttering the third note in (a). (First 8-note window only.) The new primary signature is "1100010". The wavelet-transformed vector is (-3, 1, -2, 2, -3, 0, 0).

to then shuffle the bits such that they are in "level order"—that is, such that we encode all the larger block comparisons before proceeding to a finer granularity. We call the resulting bitstring the *primary signature* for the p-structure. An example of primary signature computations is shown in figure 2.

The primary signature does not uniquely identify a p-structure[2]. We overcome this by beefing up the elements of the signature so that it *does*. Imagine that we replace each bit in the primary signature with the actual value of the difference between the partial-sum subtrees at that point rather than simply the binary comparison result. Equivalently, we can produce the vector of the $2^k - 1$ *differences* between left and right subtree sums that uniquely defines the p-structure. It turns out that this difference vector is equivalent to the Haar wavelet transform [5] on the original p-structure, with a couple of alterations: One, the proportionality constant is $C = 1$ and not $C = \frac{1}{\sqrt{2}}$ as usual, since we're "smoothing" by taking the sum rather than averaging neighboring elements; and two, we can safely discard the first coefficient, the "smoothed" value at the last level (in our case the total sum), since the p-structure is invariant to translation of all the notes up and down.

In order to emphasize the more significant elements of the signature, we scale each component of the resulting $2^k - 1$-element sequence by a weighting factor of 2^h, where h is the height of the parent of the subtrees being differenced to yield that component. The result after weighting each signature is

[2]We can easily verify this: $w - 1$ bits can only distinguish between 2^{w-1} different structures, and there are more than $w!$ different p-structures (there are exactly $w!$ if "ties" in the rankings aren't allowed). Since $2^w = o(w!)$, no conceivable encoding can require at most $w - 1$ bits for all sequences. See the appendix for a discussion of how to count p-structures.

a point in real-valued $2^k - 1$-dimensional space. Viewing p-structures as points in a high-dimensional Euclidean space, we can use a standard spatial access method to store them. We choose to use R-trees with deferred splitting [6] for their efficiency and availability.

4.3 Distance metric

The distance metric between pitch subsequences of the same length is now implicit. Namely, if s_i represents a pitch sequence, and if pack(s_i) represents the "packing" operation that produces a p-structure, and if $h(\cdot)$ represents the abovementioned modified Haar wavelet transform, then the distance between two pitch sequences, each of length 2^k, is

$$D_{\text{euclidean}}(h(\text{pack}(s_1)), h(\text{pack}(s_2)))$$

4.4 Subsequence queries

Subsequence queries proceed as follows: Given a user target note pattern s_t of window size $2^k = w$, it is first converted into its p-structure pack(s_t) and the corresponding wavelet representation $\mathbf{v} = h(\text{pack}(s_t))$ is computed. The R-tree T is consulted with the vector $\mathbf{v}$ to find the identities of songs that either exactly contain or nearly contain the sequence represented by $\mathbf{v}$.

The QPD system currently only allows queries of length w, but minor implementation work can extend the system to handle longer queries. Target patterns longer than w can be processed by taking the intersection of the files returned by processing w-sized windows of the pattern as separate queries. Presumably a user will not enter a sequence longer than an entire song, so this operation cannot be too inefficient.

4.5 Clustering using the document-feature matrix

The algorithms for nearest-neighbor, spatial join, and clustering of *songs* in the database are not immediately clear from the discussion of subsequence querying, as we have not yet defined a distance function for entire songs.

One basic nearest-neighbor approach, given a song A, might be to query the song database with all w-sized windows of notes in A, taking the nearest song B to be the song returned most frequently by those queries. Repeatedly doing this for all target songs A will clearly be inefficient for a spatial join or clustering algorithm, however.

In fact, it seems unnecessary to use the R-tree at all for these kinds of queries, since the task of clustering is not tied in any interesting way to the data structure that happens to store the subsequences. For song-level queries we dispense with the R-tree and process the database offline (see Section 5.3) as a song/feature matrix.

We take clues from traditional approaches to text document clustering and information retrieval, and produce a *vector-space representation* of each song over the *vocabulary* of possible feature vectors. With each song in the database we associate a vector indexed by the distinct features (wavelet-transformed signatures) that are possible. Each element of a song's feature vector is the number of occurrences of that feature in the song.

The task of clustering the database is reduced to the task of clustering the song vectors in this high-dimensional space. The tasks of nearest neighbor and spatial join are reduced to finding minimally-distant pairs of vectors. We use stock algorithms (based on k-means nearest neighbors) for these purposes.

In practice, using the entire wavelet-transformed signature space as a feature set is prohibitively memory- and compute-intensive. We therefore use only the first 3 elements of the signature, corresponding to the top 2 levels of a p-structure's partial sums tree, in composing each feature. This truncation, which is meaningful since the wavelet transform orders coefficients by importance, greatly reduces the number of distinct features that are observed[3].

Even after this feature coarsening, the dimension of the input matrix is still too large for conventional clustering algorithms like KNN. For this reason we perform singular value decomposition on the song/feature matrix to produce a "song/concept" matrix of much lower dimension before clustering. We thereby treat song indexing as an instance of latent semantic indexing, used widely in information retrieval.

5 System implementation

This section discusses the implementation of the QPD system. Development of the system proved to be an interesting exercise in code re-use and modularity. Included in the system is a Web crawler for acquiring MIDI files (written in GAWK); a core database engine that processes, indexes, and performs query operations on the MIDI files (written in C++); code for determining song clusters and nearest neighbors offline (written in Matlab), and a user interface for subsequence queries (written in Java.)

5.1 Offline retrieval

Musical data is readily available in machine readable form: there is a vast library of sound and music available on the Internet. The MIDI sequenced music format has persisted as a standard for nearly ten years, and tens of thousands of files have accumulated. A MIDI file records the times at which each keyboard notes (one of 127 distinct possible values) is turned on and off. Notes can occur in possibly many simultaneously rendered *tracks*.

[3]4233 distinct 3-valued wavelet-vector features were covered by the test set in this study, while in theory there are over 500,000 possible untruncated signatures (see the Appendix.)

(a)

```
%l /net/bobo/usr1/dougb/qpd/data/midi/Elvis/Dontbcr.mid
%f midi
%t Dontbcr.mid
%u http://neburton.simplenet.com/midi/Elvis/Dontbcr.mid
%c Elvis
%end
```

(b)

```
...
Processing sound file /net/bobo/usr1/dougb/qpd/data/midi/Elvis/Dontbcr.mid
  Format: 1
  Tracks: 11
  Delta: 120
  Track 1: (1 notes)   Discarded (0-tone)
  Track 2: (1385 notes)
  Track 3: (2077 notes)
  Track 4: (2243 notes)
  Track 5: (1 notes)   Discarded (0-tone)
  Track 6: (513 notes)   First melody track. (Eliminating prior tracks)
  Track 7: (529 notes)   Discarded (Non-melody)
  Track 8: (513 notes)   Secondary melody
  Track 9: (513 notes)   Discarded (Non-melody)
  Track 10: (1 notes)   Discarded (Non-melody)
  Track 11: (1 notes)   Discarded (Non-melody)
/net/bobo/usr1/dougb/qpd/data/midi/Elvis/Dontbcr.mid : 1027 notes
(1010 p-structures were added to the r-tree.)
...
```

Figure 3: (a) An example entry in the index file produced by the Web crawler. Included in each record is its location on disk (%l), its file format (%f), title (%t), URL (%u), and category (%c). Postprocessing this file with the core engine enriches each record with extra information about the file's length and identification number. (b) The output of the core engine while processing this record in the index file. Note that the (apparent) melody tracks in this file were isolated and all other tracks were discarded.

In order to acquire a large amount of data easily, a simple World Wide Web "crawler" script was implemented to fetch MIDI files from Web pages. The input to the crawler is a set of records, each consisting of a Web page to explore and a category name[4]. Category names are assumed to be provided by the user. but the crawler attempts to infer the title of each song based on the context near the hyperlink to the MIDI file: if it is unsuccessful it uses the filename itself as the title. The output of the crawler is an *index file* for use by the core engine. Figure 3a shows an example entry in an index file. The crawler identifies each MIDI file on each requested page, and downloads the file for analysis. adding an entry to the index file in the process. At the same time it builds output web pages for use by the user interface in section 5.4.

[4]I focused my acquisition efforts on the "Midi File Central" archive available at `http://neburton.simplenet.com/`

5.2 The core engine

The core engine can be run in one of four modes: *Check* mode, in which it simply scans the MIDI files in the index, discarding corrupted files and producing a revised (and enriched) index file (Figure 3b); *Training* mode, in which it initializes a database file and adds each uncorrupted MIDI file in the index to the database; *Query* mode, in which the user can perform exact and approximative matches on sequences in the database; and *Clustering* mode, in which it outputs information relevant to the clustering described in the next section.

All of these modes except *Query* involve processing each MIDI file in the index with the same two phases. The first phase converts each the file into a flat *pitch sequence*, segmented by track dividers. The pitch sequence for a given track is defined to be the sequence of notes observed in the track in the order in which they were *turned on*. Since the focus for this application is *melody*, it is desirable to eliminate tracks which are obviously nonmelodic, *e.g.* the monotonic drone of a baseline or a percussive instrument. These tracks are frequent in the data. Ideally we would like to identify a single melody track for each song, but even if the "principle melody track" for a song were well-defined, it is not practical to try to single it out at the risk of discarding the real melody unless we are quite certain. Two conservative techniques were used to prune apparently unmelodic tracks. One, if a track fails to use a certain threshold number of distinct note values (five, for the experiments run for this paper), it is discarded. Two, text comments that sometimes appear within the tracks of MIDI files are searched for the keyword "MELODY", which conventionally identifies the melody tracks to certain other MIDI clients. If a track contains such a comment, then all tracks in the file not containing such a comment (or not meeting the abovementioned tonality restriction) are discarded.

The second phase extracts the p-structures from the pitch sequence with a sliding window. For each window that is entirely contained within one track, its feature vector is computed as described in Section 4.

The "DR-Tree" package from the University of Maryland, written in C, was modified for inclusion in the engine. This package supports storage and retrieval of spatial data of arbitrarily high dimension using R-trees with deferred splitting [6].

For the *Training* mode, extracted feature vectors are added to a $2^k - 1$ dimensional R-tree. Presently, due to inefficiencies in the storage of the feature vectors (they are stored as zero-volume hyperrectangles, for instance), the size of the resulting R-tree can be several times the size of the original MIDI data. Access is entirely disk-based, however, and queries are quite efficient for the 1022-file test suite studied in this paper.

The *Query* mode entails simply reading the index and then responding to queries from the standard input. Exact match queries and approximative (subsequence-level nearest neighbor) queries are supported. The set of output songs for a nearest neighbor query is defined as *the set of songs that contain*

the top **topn** *nearest subsequence instances, ranked by the minimum distance of any subsequence*, where **topn** is a runtime parameter supplied by the user.

5.3 Clustering and nearest neighbors

Song clustering and (song-level) nearest neighbor queries are operations that can be performed offline; it is of limited utility to allow online clustering operations over arbitrary subsets of the data.

Scripts in the Matlab 5.0 numerical computation language were designed to perform these tasks. The *clustering* mode of the core engine outputs a song-feature matrix as described in Section 4.5. This matrix is outputted sparsely: only nonzero entried are printed. (Furthermore, each feature is identified online by an index into the set of distinct features observed so far.) The sparse, "economy" SVD function in Matlab, **svds**, is applied to this matrix to reduce the dimensionality to a fixed number (20 for this study.) [5]. A simple k-means nearest-neighbors clustering algorithm implemented in Matlab was then applied to the 20-dimensional documents (songs) to cluster them into a fixed number of classes. This number was varied as an experimental parameter, as discussed in Section 6.

Using available routines, the nearest neighbor of a given document (song), as well as spatial join queries on the database, can be formulated as finding minimally-distant pairs in the smaller-dimensional space of documents output by the singular value decomposition.

5.4 User interface

The subsequence matching facility of QPD can be run from the Unix command line; queries are achieved by typing out a command ("Q") followed by a series of integers. This is a cumbersome and unintuitive interface for the task at hand, however. Far better is to allow the user to input a visual representation of the query sequence consistent with standard musical notation, a representation that more transparently communicates the relative pitch structure of the query.

For this reason I designed a graphical user interface as a Java applet that allows the user to move notes in a displayed sequence up and down with the mouse until he is satisfied with the query. The applet is embedded in a page that includes the HTML output of the scripts in Section 5.1. The page for the test suite described in this paper is shown in Figure 4.

Upon submission of the query, the applet transmits the pitch sequence to a Web server by encoding it in a URL directed at a CGI script on the server. The CGI script communicates the sequence to an instance of the core engine that stays running as a daemon on the server. The engine produces HTML output with the top few distinct songs containing subsequences nearest to that of the input in the sense described in Section 4. The output summarizes each entry with a title and category, as well as a link

[5] I would someday like to observe which musical "concepts" were learned by this LSI-like approach to clustering.

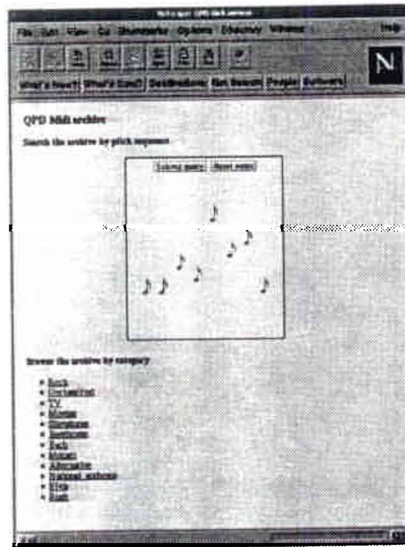

Figure 4: The Web-based user interface to QPD. The user can arrange the notes in the Java applet at the top of the page to construct a subsequence query. Browsing the database manually is also possible.

that when clicked will cause the MIDI file to be downloaded and played on appropriately configured systems.

The user interface will soon be extended to include information about the cluster membership and nearest neighbor information associated with each song, derived offline.

6 Evaluation

This section discusses an evaluation methodology and performance results for the QPD system on a small database of representative music files.

6.1 The test set

In real-world deployment on the Web, this system will need to index tens of thousands of songs in order to be useful. It is therefore hoped that the use of R-trees lends the system some measure of automatic scalability. For the purposes of evaluation, however, we will focus on a small set of about 1000 MIDI files.

The Web crawler script was used to fetch 1137 MIDI files of a wide range of genres, including rock, alternative, classical, broadway, and national anthems. A summary of the test set is provided in figure 5. Of the retrieved songs, 1022 survived the engine's *Check* function[6].

[6] Songs that did not strictly adhere to the MIDI specification were discarded. Unfortunately, as with many document standards, exceptions are sometimes the rule rather than the exception.

Genre or artist	Number of songs	Average number of p-structures/song	Percentage of data
National anthems	142	1444	9.1%
TV themes	163	1270	9.2%
Movie themes	106	2283	10.8%
Show tunes	18	1656	1.3%
Rock	320	2874	40.9%
Alternative	30	2597	3.5%
Rush	40	1541	2.7%
Classical	140	2170	13.5%
Elvis	63	2919	8.2%
ALL	1022	2202	100.0%

Figure 5: A summary of the test set used in this study. In total, 2250803 p-structures were extracted from the data.

6.2 Timing results

One important evaluation criterion is speed. Adding each song to the database takes a couple of seconds, but this is not a critical-path concern. The average time in seconds per subsequence query, over a large range of different subsequence queries, was measured. Subsequence queries were generated randomly by producing a sequence of 8 random integers between 0 and 127. The **topn** setting, the maximum number of sequences to be retrieved by the nearest neighbor query, was varied. This setting is by far the most critical parameter in deciding processing time. The timing results are shown in Figure 6.

6.3 Clustering accuracy results

At least as crucial as speed is the quality of approximative subsequence matches and the clustering behavior. We can formalize an evaluation process that takes advantage of the duplicates present in the database.

Table 7 shows some of the songs that have multiple *instances* (different renditions, or at least different filenames) in the test set. The hope is that the multiple renditions of each song in the table will land in the same group after applying the clustering algorithm, exposing the duplicates; as the chart shows, they often do.

But let's be more precise. Our evaluation metric for clustering is the *pairwise agreement probability* over the set of known duplicates. This performance metric asks, "What is the probability that two uniformly randomly selected songs from the set of duplicates are identified by the system as being in

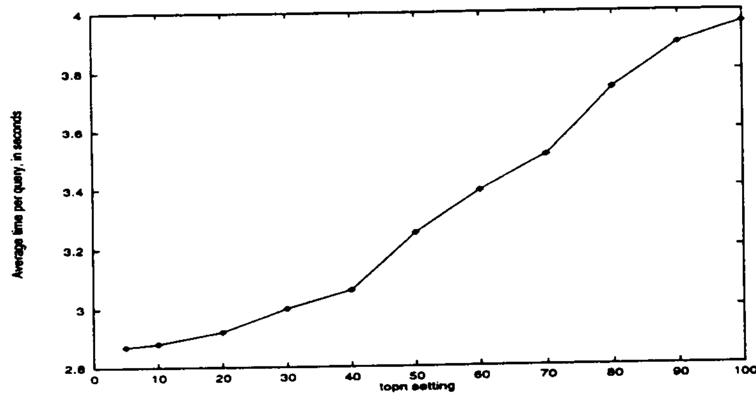

Figure 6: The average time required to process an approximative subsequence match query at various settings of **topn**. 100 randomly generated queries were used. All times are in wall-clock seconds. Experiments were run on a Sun UltraSparc 20 on a single, unburdened processor.

Genre	Title	Instances	Cluster assignments
National Anthems	Brasilian national anthem	5 versions	7,7,18,7,7
National Anthems	Canadian national anthem	3 versions	2,12,2
TV themes	Law and order theme	2 versions	6,6
TV themes	Mission Impossible theme	2 versions	2,6
Pop rock	Spirit of the Radio (Rush)	2 versions	15, 2
Pop rock	Between the the wheels (Rush)	2 versions	4,4
Elvis	Don't Be Cruel (Elvis Presley)	3 versions	2,2,3
Elvis	Hound Dog (Elvis Presley)	3 versions	9,2,9
Show tunes	Phantom of the Opera (A.L. Weber)	2 versions	20,7
Show tunes	All I ask of you	2 versions	6,6

Figure 7: Duplicate songs in the MIDI test set. The clustering shown is the result of 500-way clustering.

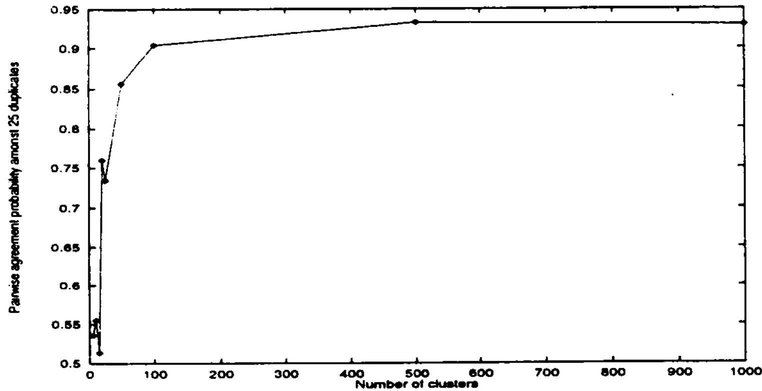

Figure 8: The pairwise agreement probabilities, plotted versus number of clusters induced, for the 25-song task summarized in Figure 7.

the same cluster if indeed they are, or are identified as being in different clusters if they're not?" This is related to the co-occurrence agreement probability for segmentation algorithms discussed in [1]. It is a necessary departure from precision and recall, since the classes induced by the system are unlabeled. Scoring should be based on the structure of the derived classes—in particular on the pairs of songs that are deemed "equal" or "unequal" under the equivalence relation implicitly discovered by the algorithm.

The K nearest neighbors clustering algorithm was run for various numbers of clusters. (KNN initially chooses an arbitrary assignment of the input points into the user-specified number k of clusters, and then refines the clusters iteratively, moving some number of points between clusters at each iteration until finally none are moved.) The pairwise agreement probability is plotted for these configurations in Figure 8. Although a relatively small number of experiments were run, this data seems to indicate that a large number of clusters (around 500) is preferable, with respect to our error metric, for discovering duplicate songs. Note that this is not necessarily the ideal setting for clustering in general—for example, if we wished to automatically classify by genre, a smaller number of clusters would be preferable.

6.4 Nearest neighbors

The performance of the nearest neighbor processing turned out to be a fascinating aspect of the system. Each song in the database was mapped to its nearest neighbor as defined in Section 5.3. The resulting map from songs to songs, and the associated minimum distances, proved very interesting. The pairs with zero distance are exact duplicates that happened to have different filenames, of which there were 8 instances. The pairs with very small distance ($< 10^{-12}$) were near duplicates, of which there were 10 instances. More subtle were pairs such as those shown in Figure 9.

Song 1		Song 2	
Genre	Title	Genre	Title
TV	Theme from "Home Improvement"	Movies	Theme from "Braveheart"
TV	Theme from "American Trilogy"	TV	Theme from "Hill Street Blues"
TV	Theme from "Inspector Gadget"	TV	Theme from "Thirty Something"
TV	Theme from "Fame"	Beethoven	Sonata No. 20 Op 49
Alternative	Nightswimming	Alternative	No Memory
Bach	Symphony No. 4	TV	Theme from "The Monkees"
Bach	Jesu Joy of Man's Desiring	TV	Soundtrack from "Dr. Who"
Movies	Medley from "Dangerous Minds"	Rock	Gangta's Paradise (from Dangerous Minds)
Rock	The Keeper of the Stars	National Anthems	Finnish National Anthem

Figure 9: Some of the nonobvious "mutually nearest" pairs in the data, *i.e.* pairs of songs s_1, s_2 such that s_1's nearest neighbor is s_2 and s_2's nearest neighbor is s_1.

6.5 Subsequence queries

Assesment of the subsequence query function will be achieved by deploying the system to users familiar with the musical genres covered by the test database. Quantitative evaluation can be achieved by extracting queries from an instance from each song class in table 7 and computing the frequency with which other instances in the song class occur in the list of result songs. This evaluation methodology makes the assumption that end-user queries will mismatch with the targeted reference sequence in the same ways that these reference sequences mismatch with each other.

7 Discussion and Future Work

Qualitative evaluation of the QPD system reveals that it excels at finding interesting nearest neighbor pairs and clusters. It is reasonably useful for subsequence queries, but by no means a prize-winning contestant at "Name that Tune". Why not?

- Precision on subsequence queries quickly becomes unacceptable as the database becomes large. The problem is that the query length of 8, which is perhaps the only reasonable setting that is a power of 2, allows for only about 545,835 feature vectors (see the Appendix). This may seem like a large number, but the probability of a collision in our database of 2.25 million p-structures is actually quite high. The result is that even for subsequence queries that exactly match a sequence in the "intended" reference song, many "unintended" songs that happen to contain the sequence somewhere appear too. For example, querying the system with the eight notes comprising the incantation "Don't cry for me Argentina" from Evita, yields the desired MIDI file as well as 20

others that contain the sequence exactly. One solution is to extend the user interface to allow for longer queries and to modify the engine accordingly as suggested in Section 4.4.

- Subsequence queries work best when the music in the database is monophonic or the melody track is isolated, but typically this is not the case. Typically only one of the many tracks in a file can truly be called the "melody" of the song, and this is what motivates the pruning discussed in Section 5.2. It would be interesting to experiment with less conservative approaches to pruning that could eliminate wasteful "distractor" tracks and thereby improve precision. Perhaps the melody track can be guessed using information-theoretic measures like entropy.

The following features are desirable for a full-fledged MIDI database. For this study, I have ignored these issues in favor of refining the accuracy of the core search engine:

- A back-end Web crawler that fetches sound files and dynamically adds them to the index, rather than the static and rather simple-minded supervised approach I've implemented.

- A broader range of search options to supplement pitch-dynamics queries, *e.g.* substring matching on authors and titles and full text searches on lyrics. Servers specializing in lyrics indexing exist on the Web but are not yet integrated with music file search engines.

- Support for *digitized* audio files as well as sequenced music. In my scheme this would require accurate pitch extraction from the signal, still an active area of research. It's worth noting, however, that since we care only about relative pitches we can bypass the musical score as an intermediate representation. Still, segmenting a noisy polyphonous signal into notes is a daunting task.

- A microphone front end that allows the user to hum, whistle, or sing his query. Pitch extraction from this cleaner signal is less a problem than from arbitrary audio, and is implemented in at least one commercial MIDI sequencer.

- Duration indexing. The QPD engine is blind to duration information since its underlying musical representation is based only on relative pitch. Yet a user with a subsequence query may be able to supplement his pitch sequence with valuable information about duration. Furthermore, duration may be useful in clustering songs that have similar rhythmic patterns. It would be trivial to apply precisely the same algorithms described in this paper on the sequence of duration values observed instead of the sequence of pitches. It's not immediately clear whether the indexing method would work well on durations, and even less clear how both dimensions, pitch and duration, could be combined into a single representation.

7.1 Applications to other media

Another potential application of "pitch dynamics" queries is in finding pairs of stocks on the stock market that have similar behaviors over a span of time. Suppose we want to find stocks A and B such that when stock A goes up, so does B; and on days when A goes down, so does B. Two stocks may be correlated but may have gains and losses which in magnitude are distractingly different, eluding other similarity measures. If all we care about is the crude ternary performance (up, down, unchanged) of a stock on a day-to-day basis, a spatial join of the type described above may be suitable for detecting similar relative behavior in two stocks. The architecture of the QPD engine is such that it would be trivial to extend the program to handle arbitrary real-valued time sequences.

8 Acknowledgments

I gratefully acknowledge Christos Faloutsos for useful discussions about this project, and Frank Dellaert for providing Matlab code for k-means clustering.

9 Appendix: Counting the number of p-structures of length n

An interesting combinatorics question is the number of different p-structures, or "rankings with ties allowed", of a window of n notes. The answer lies somewhere between $n!$, the number of different permutations of n elements (with ties disallowed), and n^n, the number of functions from n elements to n values. The lower bound $n!$ can also be thought of as the number of one-to-one and onto (bijective) functions from an n element set to n values; the upper bound represents unrestricted functions. The middle ground we seek is simply the number of functions from an n-element set that are (at least) *onto*. We sum over the number of elements in the range:

$$\sum_{m=1}^{n} \mathrm{ONTO}(n, m),$$

where

$$\mathrm{ONTO}(k, j) = \sum_{i=0}^{j} (-1)^i \binom{j}{i} (j-i)^k$$

is the number of functions from a k-element set onto an j-element set. The formula for the ONTO function can be proved using the principle of inclusion/exclusion.

For $n = 2^3 = 8$, there are 545,835 different p-structures.

References

[1] D. Beeferman, A. Berger, and J. Lafferty. Text segmentation using exponential models. In *Proceedings of the Second Conference on Empirical Natural Language Processing*, 1997.

[2] T. Blum, D. Keislar, J. Wheaton, and E. Wold. Audio databases with content-based retrieval. In M. T. Maybury, editor, *Intelligent Multimedia Information Retrieval*, pages 116–135. AIII Press / M.I.T. Press, 1997.

[3] T. Chou, A. L. P. Chen, and C. Liu. Music databases: Indexing techniques and implementation. In *Proceedings of the International Workshop on Multimedia Database Systems*, pages 46–53, 1996.

[4] M. Dirst and A. S. Weigend. On completing J. S. Bach's last fugue. In A. S. Weigend and N. A. Gershenfeld, editors, *Time Series Prediction: Forecasting the Future and Understanding the Past*, pages 151–172. Addison-Wesley, 1994.

[5] C. Faloutsos. Searching multimedia databases by content. pages 113–120. Kluwer Academic, 1996.

[6] C. Faloutsos. Searching multimedia databases by content. pages 34–37. Kluwer Academic, 1996.

[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proceedings of SIGMOD*, pages 419–429, 1994.

[8] G. Gonnet. Unstructured databases for very efficient text searching. In *Proceedings of ACM PODS*, pages 117–124, 1994.

[9] D. Parsons. *The Directory of Tunes and Musical Themes*. S. Brown, 1991.