Name That Tune:

Content Searching for Music Databases

Final Report

Digital Communications and Signal Processing Systems Design

18-551 Spring 2004

Group 11

Kort Eckman (kort)

Stephen Mangiat (smangiat)

Leo Soong (lsoong)

Table of Contents

Introduction

Project Description

For our project, we implement a search engine that searches through a database of songs based on a hummed input. Popular engines such as Google rely on text input. However, if you do not know any details about the song, you will have a difficult time finding any more information about the song using a text-based search engine. Our system allows you to hum the melody of a song and it will return the highest matched songs. This is extremely useful if you have a melody stuck in your head and you have no idea what it is.

Prior Work

Both pitch tracking and DTW have been used in past 18-551 projects, though no group has done a "query by humming" project. There is some research currently being conducted by Phillips, although their technology requires an actual sample of the song to be searched [7]. Other groups have recently published papers concerning query by humming. The majority use note segmentation, however Zhu and Shasha have used a method that does not [5]. We based our method on this general idea, though we did not follow their method explicitly, as their paper mostly discussed how to improve performance and speed.

Solution

We solved this problem using existing pitch tracking algorithms to translate the input hum into a series of pitches. From there we create a continuous pitch vector by removing any values where the pitch was unable to be determined. This includes empty sounds, where nothing was being hummed. We then normalize this pitch vector to help eliminate errors caused by the user being in a different key. This new vector is then compared against every entry in our database. The database was pre-processed and also follows the format of the pitch tracked input. This means that each entry does not have an empty note and the pitches of each entry are normalized.

For comparing the series of pitches from the hum with the database, we decided to use a DTW (Dynamic Time Warp) algorithm. We hoped to eliminate errors caused by changing tempos with the dynamic time warp.

Why Hum?

Humming is a natural input for a music search system. Frequently, simply hearing a portion of a song on the radio is enough to get the melody stuck in your head. If the radio DJ didn't mention either the title of the song or the artist, you will be at a loss to identify the tune using traditional text based search techniques unless you can remember a good portion of the lyrics.

So, what is a good way to input a melody into a search engine? One could potentially sing the song. However, one can hum a song without having to recall the lyrics. In addition, instrumental music, which makes up a large portion of the classical catalog, is by nature lyric free. One could potentially play the song on a piano or violin. However, unlike playing a musical instrument, most people can hum without any special training. While whistling could also be used to input a melody, humming comes more easily to most people.

A hum signal is also a good input for signal processing and pitch tracking. The hard consonant sounds, which correspond to bursts of white noise, present in both singing and speech are largely absent from a hum signal. Also, great care must be exercised when whistling into a microphone. Whistling directly into a microphone will create wind noise that will distract the pitch tracker.

We found that humming with an open mouth worked better than humming with a closed mouth. This is probably due to a louder signal reaching the microphone, thereby improving the signal to noise ratio in a busy lab. This would likely prove useful in real life situations as well, such as a loud music store.

What People Do

Not all users of a query by humming system will be trained singers. Even if they are, it is very hard to hum the melody perfectly. Humming perfectly can be defined as having the correct absolute pitch (in the right key), relative pitches (intervals), tempo, and rhythm. Very few users will be able to hum exactly in key with the database entries. Experts estimate as few as 1/10,000 people have “perfect pitch” [5], or the ability to identify a pitch without any frame of reference. Errors in rhythm are also extremely common, as it is unlikely the user will bother to tap his or her foot or listen to a metronome while humming. On the other hand, many users, even with no musical training, will be able to hum the intervals reasonably well. Global tempo, which can easily be accounted for, is of little interest. The real defining feature of any melody is the relative pitches. In equal tempered tuning any melody can be played in every key without any real loss (aside from personal taste). Because of the use of normalization and dynamic time warping, the features of a melody have essentially been narrowed down to relative pitch. If the user can hum these intervals reasonably well, as is common, then the system will be able to correctly identify the melody.

Pitch Contours & Note Segmentation

Most of the pre-existing query by humming systems use pitch contours whereby the user’s humming is converted into a series of discrete notes from which the contour is then extracted. The contour information is represented by an alphabet of letters: “U” for up, “D” for down, “S” for same as previous, “u” slightly higher, “U” significantly higher, etc. The contour database entries consist of similar strings formed from the same alphabet. The edit distance between a pitch contour query and a database entry constitutes their matching score.

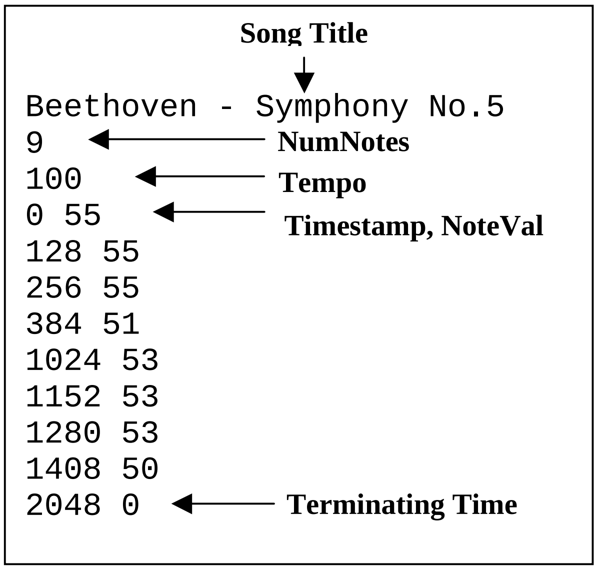
One problem with the contour method is that contour information alone is insufficient to distinguish the entries in a large database [6]. Moreover, the difficulty of segmenting hum input into discrete notes limits the precision of a contour system. At present, there is no good note segmentation algorithm. Thus, the precision of a contour method depends upon an inadequate preprocessing stage [5].

Our approach does not rely upon pitch contours and involves no note segmentation. Matching is accomplished with time warping techniques. It should be noted that the time warping search is more computational expensive than the string matching techniques used in the contour method. On the other hand, it offers the possibility of greater precision. The costs of a search can be reduced using DTW indexing schemes [5].

Algorithms

Database:

The “Name that Tune” system allows the user to search a database of melodies. A significant issue in query by humming searching is that there can be many unique melodies in a single song. There are two solutions to this problem. The first requires scanning through the entire song, essentially comparing the query against all sections of a song – the verses, choruses, bridge, etc. This provides a complete solution for searching a song database, but the runtime of such an approach increases dramatically. The second approach, the one we have chosen, involves comparing the query to recognizable “snippets” of a song’s melody. For example, if the user is searching for Beethoven’s “Symphony No. 5”, it is very likely that he or she will hum the opening eight notes that have made it famous. If this snippet is stored in the database, the query can be matched to the entire entry at once, reducing runtime. Of course the drawback of this method is that the user must hum something similar to what is stored. What if the user happened to hear a later movement of the symphony, and was wondering what it was? This is a likely situation, as the other sections are less recognizable as Beethoven’s Fifth. The solution to this is to store multiple snippets for a single song. Zhu and Shasha did this for their database. They stored 20 snippets of 50 different Beatles songs, amounting to 1000 database entries. Depending on the song, there may need to be more or less snippets. Although 1000 entries seems very large for only 50 songs, this database method is still more scalable then the method that stores the entire song. Entries are small and DTW can be performed on snippets quickly in comparison to complete song scanning. For our database, we used 42 recognizable snippets from 42 different songs. We chose not to store multiple snippets for each song because the process is the same as matching a query to snippets from different songs. As far as the computer is concerned, a



```
Beethoven - Symphony No.5
9
100
0 55
128 55
256 55
384 51
1024 53
1152 53
1280 53
1408 50
2048 0
```

snippet is a snippet. More searchable songs also made for a more interesting demo.

Our database is comprised of text files for each entry. These text files are read and processed by the PC on startup. Each file is very small and contains all necessary information. The total size of the database is 5.8 kB. For 42 songs, this makes the average entry size approximately 140 bytes. The first line of the text file is the song name and its artist/composer. This info is returned to the user. Next is the total number of notes stored. The next value in the file is the MIDI tempo of the melody. Initially, we planned on comparing the length of the query to the length of the database entry. If these lengths were very different, then we could assume that entry was not hummed and return without performing DTW. We decided not to do this method as it limited the user. A very fast melody could then be hummed more slowly and still result in a correct match. The rest of the entry files are the actual MIDI notes associated with each melody. The notes are paired with a timestamp, indicating the start of the note. Rest information is not used, so each note essentially extends to the next without overlapping. A note with a value of 0 consequently indicates the end of the last note.

The database entries were created using MIDI notation software as well as MIDI conversion Matlab code [1]. MIDI files contain information we did not need, such as velocity, instrument information, and note off timestamps. We therefore used the Matlab code to strip away this information and produce a text file containing the notes and their “note on” timestamps. The MIDI notation software Sibelius was first used to delete all chords, accompaniment, and backing tracks, and to extract the melody snippet. About half of our database was created from scratch. The other half used MIDI files from SAIL that were processed as described above [2]. Song names and the rest of the information were then inserted at the top of each file.

Itsy Bitsy Spider	Maria	George Gershwin - Rhapsody in Blue
Tchaikovsky - 1812 Overture	Take Me Out to the Ballgame	Elgar - Pomp and Circumstance
Mary had a Little Lamb	Twinkle Twinkle Little Star	Scott Joplin - The Entertainer
Oh Suzanna	Wedding March	John Williams - Indiana Jones
It's a Small World	Beethoven - Symphony No.5	John Williams - Imperial March
James Bond Theme	Beethoven - Symphony No.9	Nino Rota - Godfather Love Theme
United States National Anthem	Duke Ellington - Take the A Train	Nino Rota - Godfather Theme
Happy Birthday	Thelonious Monk - 'Round Midnight	Mozart - Marriage of Figaro
Offenbach - Can-Can	Strauss - Zapf Zarathrusta	Beethoven - Fur Elise
The Beatles - Hey Jude	Wagner - Flight of the Valkyries	Mussorgsky - Pictures at an Exhibition
If You're Happy and You Know It	Bach - Toccata & Fugue	John Williams - E.T. Theme
Jingle Bells	John Williams - Superman	Mozart - Eine Kleine Nachtmusik
London Bridge	John Williams - Star Wars	John Williams - Close Encounters
Londonderry Aire	Bill Conti - Rocky	Vangelis - Chariots of Fire

Pitch Tracking

There are many different methods for tracking the pitch of sampled audio, each of which could have been employed for our project. Zhu and Shasha tested different methods when applied to humming and ultimately chose a modified version of the classic pitch tracker using Autocorrelation. We therefore decided to go the same route, and the pitch tracking algorithm we used is based on a Fast Autocorrelation Matlab script written by Gareth Middleton [3]. Autocorrelation is an intuitive form of pitch tracking because a peak in energy will occur when shifting by the fundamental period of the input. The frequency can then be calculated by dividing the sampling rate by the number of shifts. The pitch corresponds to a note in equal tempered tuning. These are found by converting into MIDI values using $P = 69 + 12*\log(\text{frequency}/440)$. Concert pitch, or $A_4 = 440$Hz, has a MIDI value of 69. The input to our system is humming, so the fundamental frequency is limited to a few octaves. Unless an opera singer is using the system, this frequency will be under 1kH. The formants of the human voice will also cause peaks to occur at higher harmonics. This effect varies if the user's mouth is closed or if different vowel sounds are used. The vowel "ah", which is very easy for the throat to produce, has less high formants than many other vowels, such as "ee". Therefore, as discussed before, we found that the best pitch tracker results were produced from inputs using the "ah" sound. For the purposes of our project, we could have probably used a sampling rate of 22kHz or less with good results. We chose to use a sampling rate of 44.1kHz for the highest amount of accuracy. This increased the pitch tracking runtime, but because the hum input is limited to a few seconds, this was not an issue.

The pitch tracking is performed on windows of 1000 samples, which corresponds to about 23 ms of sound. This is more than sufficient because it is unlikely the user will hum a note for less than 23 ms. Actually, windows of 1400 samples are used to provide the values for correlation when shifting. To increase speed, these windows are first copied to on chip RAM using DMA transfers. The difference between the start of each window, or jump size, is set to half the window size or 500. This increases the continuity between all of the tracked pitches. Consequently, the resultant pitch track vector is of size inputSize/500.

The pitch tracker does make some assumptions about the pitch in order to increase its speed. If the maximum amplitude of a window is very small, or less than 10% of the maximum amplitude of the entire input, then the tracker assumes there is no pitch to be tracked. The frequency is then set to 0 to indicate this. The frequency is also set to zero if the fundamental frequency of a window can simply not be determined. This happens when the peak of the autocorrelation is less than a certain threshold. It is also important to note that these 0 values are removed from the pitch track vector before being converted to note values. If the frequency of the previous window is known (nonzero), then autocorrelation is performed with a range of shifts between 20 less and 20 more than the shift of the previous window. In terms of pitch, this range corresponds to a different number of semitones depending on the octave. The bounds will be in the range of 3 to 10 semitones. Consequently, if there is a very quick leap in pitch greater than this range then the pitch tracker will not be searching for the correct pitch in the next window. This will then cause the autocorrelation peak to be less than the threshold, resulting in a frequency for the window equal to zero. If the frequency of the previous window is in fact zero, then the autocorrelation is performed with a range of shifts from 10 to 800. This corresponds to a frequency range of 55Hz to 4kHz and a pitch range of approximately equal to that of an entire piano keyboard. This is plenty of range, considering the only input is humming.

As previously stated, the zeros present in the pitch tracker output are removed. More specifically, they are changed to the last known pitch. This is not a problem and is in fact beneficial, because our algorithms do not rely on note segmentation and rest information. Again, the results of note segmentation algorithms are unpredictable and have impaired past query by humming systems. The zeros at the start of the pitch track vector are deleted because the user has not yet started to hum.

Normalization

Matching the hummed melody to the database entry correctly is difficult if the user hummed an octave higher or lower or if the user hummed in a different key. So, normalizing the pitches is key to solving this problem. To normalize, we take the time-weighted mean of the pitch tracked sample. This means that the longer notes are given more weight than the shorter notes. The time-weighted mean is subtracted from each element of the pitch tracked sample. Each database entry is also normalized in this way.

Uniform Time Warp

Each database entry is different in its duration and number of notes. So, we take the database entry and stretch each note, so that it matches the length of the pitch tracked hum. Each database entry becomes matched in terms of length. This helps eliminate the factor of global tempo. The user can hum the melody for ten seconds or he/she can hum the same melody twice as fast for five seconds. Either way, the scores will be adjusted uniformly for variations in the overall tempo of the hum.

Dynamic Time Warp

The dynamic time warp then finds the shortest ‘distance’ between the pitch tracked query in note space and a database entry. The Uniform Time Warp leaves two vectors of the same length to be compared, the hummed query and the database entry. From this, we can simply calculate the amount of error from one element to the next, with the following equation:

$$\sum_{i=0}^{i<L} (Query_i - Dbentry_i)^2 \qquad \text{where L is the number of pitch tracked values}$$

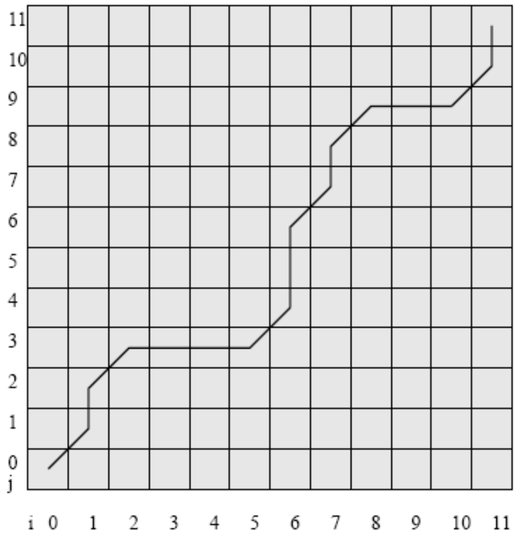
However, this comparison will not be done correctly if the user is not consistent with his/her rhythm. So, if the user speeds up or slows down the tempo as he/she hums, the notes from the query will not be compared with the correct notes from the database entry. For example, if the user holds the first note too long (in comparison to the length of the query), the end of that note may be compared with the second note in the database entry.

But we wanted to have some allowance for the error of the user’s rhythm. So we used the dynamic time warp to accommodate these errors. The dynamic time warp uses a distance matrix (D-matrix). We chose to build a distance matrix where each element is calculated by the following equation:

$$D_{i,j} = (query_i - DEntry_j)^2 + 1$$

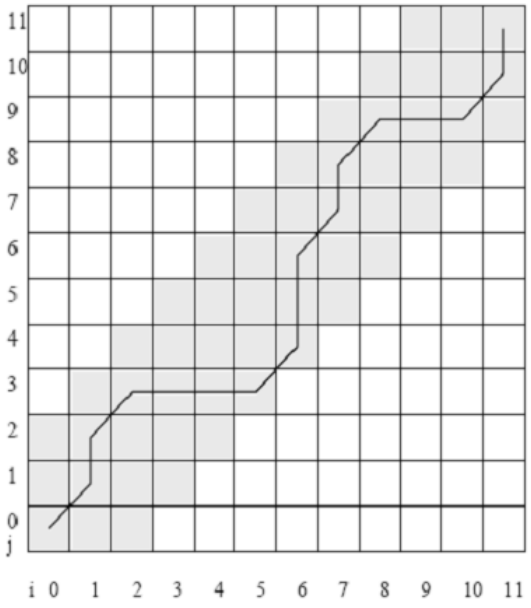
We add the one to each element to incur a small penalty for deviations in local tempo. The DTW allows variations from a straight path and alters the D-matrix by iterating over the following equation:

$$D_{i,j} = Min(D_{i-1,j-1}, D_{i,j-1}, D_{i-1,j}) + D_{i,j}$$



The result of the match of the query with the database entry is located in the top right corner of the D-matrix. The implicit path of the notes that gets compared is illustrated with the graph to the right. This basically minimizes the distance between the notes from the hummed query with the notes from the database entry. We originally used Matlab code by Dan Ellis to perform DTW [4].

Although we can perform a dynamic time warp on the database entry that has not been uniformly time warped, this will create a bias toward shorter database entries. Smaller database entries will have less numbers to add for the result. So, the uniform time warp eliminates this bias and corrects the scores.



k-Local Dynamic Time Warp

The dynamic time warp allows a tremendous amount of freedom for variations in tempo. So, in effect a user can hum the first note for five seconds and the rest of the melody phrase in the next second. Realistically, this will not happen too often, and we can restrict the time warping range by applying a constraint, such that the compared elements must be within 40 elements from the diagonal. The shaded region to the left illustrates the allowed area for the dynamic time warp to follow. This basically restricts the user to be more consistent with his/her tempo. So although the user can hold a few notes too long or too short, he/she cannot completely change tempo. If the user does, his/her hummed notes will not match properly in the DTW with the notes in the database entry.

Scoring

The results from the comparisons of the query with each database entry are then used to calculate a score of how well each match was. The number of elements in the pitch tracked vector is divided by the result of the Dynamic Time Warp and multiplied by 100, giving a result between 0% and 100%. If the user hums a database entry with perfect tempo and pitch, the Dynamic Time Warp will generate a result that is equal to the number of elements in the pitch tracked vector, *L*. The pitch tracked vector should then be exactly the same as the database entry. Remember that the D-matrix is initially set to $D_{i,j} = (query_i - DEntry_j)^2 + 1$, so in this case the distance matrix is filled with ones along the diagonal. So iterating over the Dynamic Time Warp would result in a shortest path along the diagonal, where $D_{i,i} = i + 1$. Since the matrix is *L* by *L*, the DTW feature result is *L*. So the score in this case would be 100%. On the other hand, a badly hummed query would have a very large result from the Dynamic Time Warp, due to the error between the hummed notes and the database notes, in which case, the score would be closer to 0%. Typically, a decent hum will score anywhere between 40% and 80%.

Optimizations

Memory

The large hum query (sampled at 44.1 KHz) is stored in SDRAM. This provides the space for a 10 second sample. Windows of 2,800 bytes from this sample are copied into on-chip memory using a DMA transfer, before we perform any calculations on them. This allows each memory access to take only one cycle rather than 15 cycles. Also, the entire DTW is performed on chip.

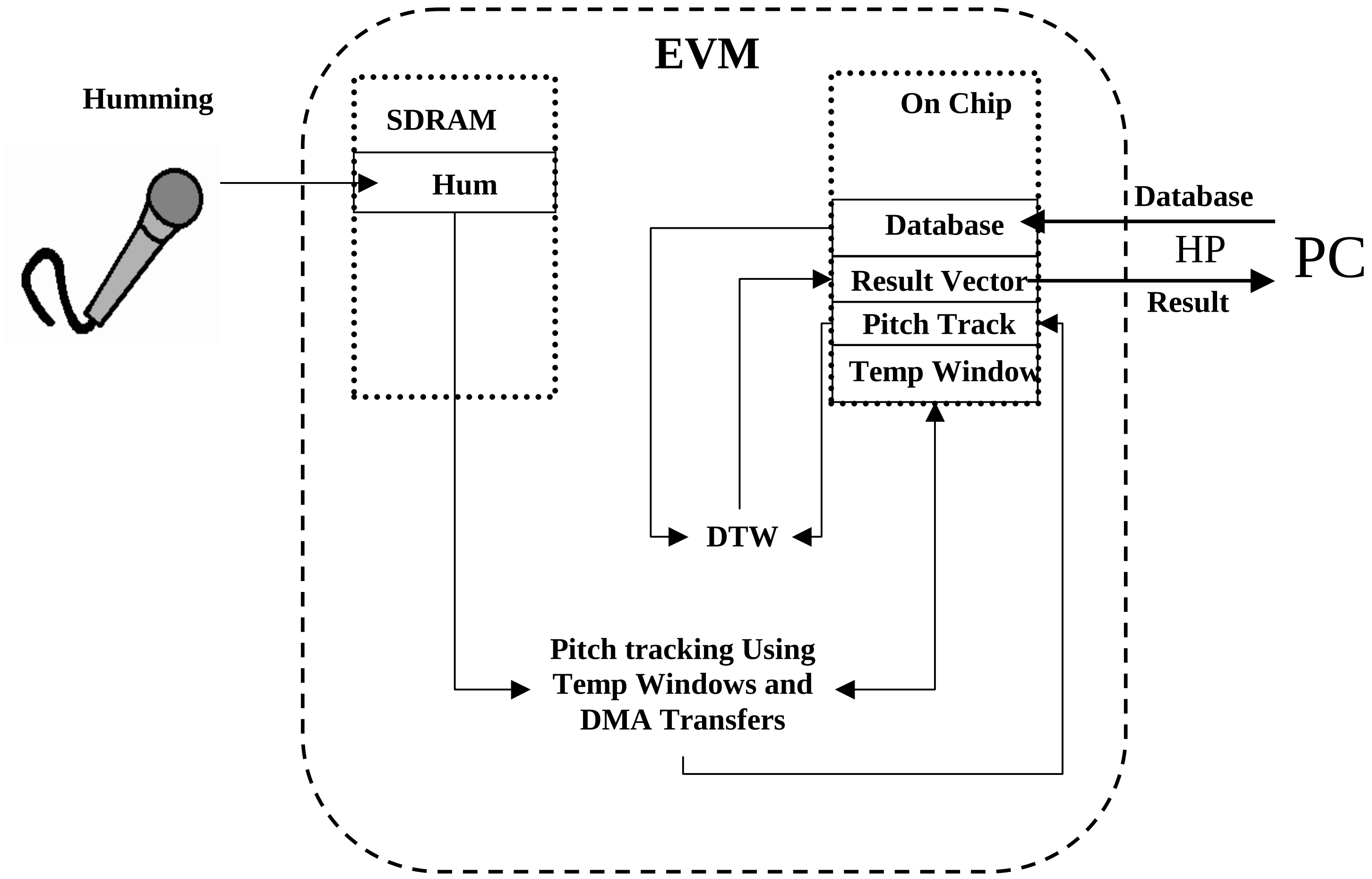
Instead of calculating the D-matrix completely, we generate one row at a time, including the calculations for the DTW. This reduces the number of elements from n^2 to 2 n. This allowed us to keep everything for the DTW on-chip.

Speed

The Matlab pitch track code we found relied on floating point arithmetic. This performed poorly on the EVM. It was possible to convert the pitch track code into integer arithmetic without losing precision. Before we did any optimizations, a typical pitch track of the input query would take several minutes. After all optimizations, the runtime of the pitch track improved by roughly a factor of 10, meaning it would take a matter of seconds to finish computing.

The DTW was also written in fixed point and used k-local dynamic time warping, which reduced the order of the DTW from $O(n^2)$ to $O(n)$. Although this can potentially change the resulting scores returned from the DTW, in some ways it can add accuracy to the matched songs. Although the DTW may have been clearer using recursion, doing so would have been much slower.

Implementation



Overview

We used the Texas Instruments TMS320C6701 Evaluation Module, or EVM, to implement our system. The EVM is our back-end system that does all the work and calculations for our algorithms. The PC handles the front-end of our system and the interface for the user.

Signal Flow

At the onset of running the system, the PC reads our database from the hard drive and sends it along to the EVM, which stores the database on-chip. When the user wishes to begin humming the melody through the microphone, he/she hits the space bar on the GUI, which then tells the EVM to begin recording the input from the microphone and stores it on the off-chip SDRAM. After the user stops humming, he/she hits the space bar again and waits for the results. The GUI instructs the EVM to stop recording and to begin processing the hummed query. This input hum is passed to the pitch tracker, which copies windows of the hum to the on-chip RAM for quicker memory accesses during tracking. After tracking the pitch tracked output is converted to notes, normalized, and sent to the DTW, where we compare the pitch track with the database entries, all of which are kept on chip. The results are sent back to the PC, where the GUI displays the top ten matches.

Memory and Speed

Our system requires the following amounts of memory on the EVM:

Input query:	441 Kbytes (on average of 5 sec recorded at 44.1 KHz)
Database of 42 songs:	8 Kbytes
Code:	88 Kbytes

We used HPI transfers to send the database and results between the PC and the EVM. We also used DMA transfers to transfer blocks of the input query from the off-chip SDRAM to the on-chip RAM, which made memory accesses much quicker.

A 4 second hum input results in a pitch tracking time of roughly 4 seconds and a DTW search time of roughly 1 second for a database of 42 songs. The DTW was used with $k = \pm 40$. The pitch tracked entries usually have a size between 150 and 300 data points. So, using this value for k, the DTW was faster by a factor of 2 to 4.
The pitch tracking is O(n). Therefore, the pitch tracker takes 1.4 seconds for each second of input. The k-local DTW is linear with respect to the size of the input hum.

Runtimes

Both the pitch tracker and the dynamic time warping were profiled using Code Composer. The runtime of each is dependant on the length of the recorded hum sample. If k-local dtw is used, then both algorithms are *O(n)*. Some sample results are below:

CPU clock speed:	133 MHz
Sampling Rate:	44.1 kHz
Database Size:	42

Pitch tracker:

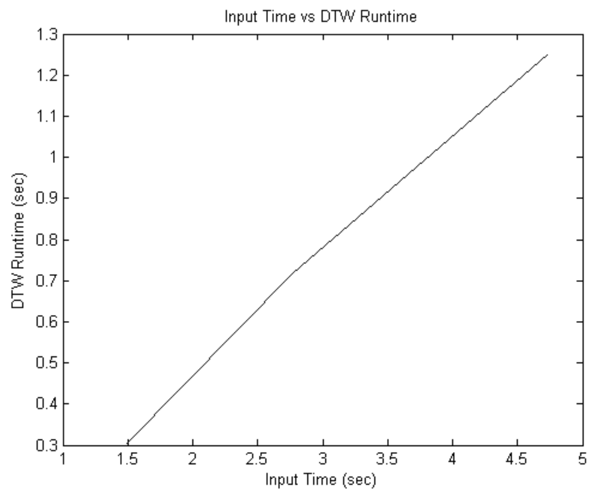
Hum Query Length: 63,196 samples = 1.433 sec
Fast Autocorrelation = 177,750,091 cycles = 1.3365 sec

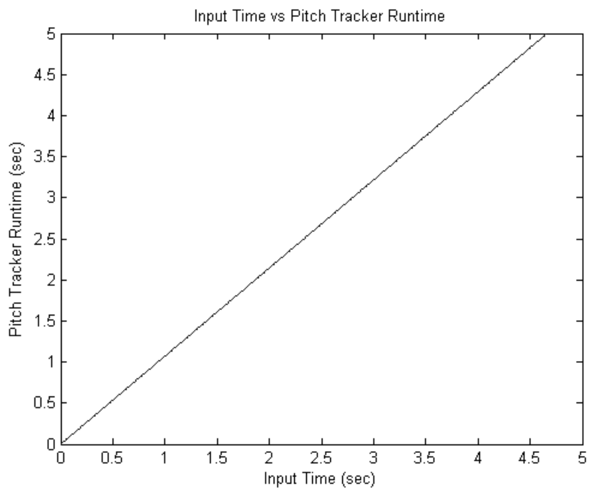
DTW:

Query Length: 65,776 samples = 1.4915 sec
DTW for Entire Database = 40,421,097 cycles = .3039 sec
Time/entry = 7.2 msec

Query Length: 121,877 samples = 2.7637 sec
DTW for Entire Database = 95,162,809 cycles = .7155 sec
Time/entry = 17 msec

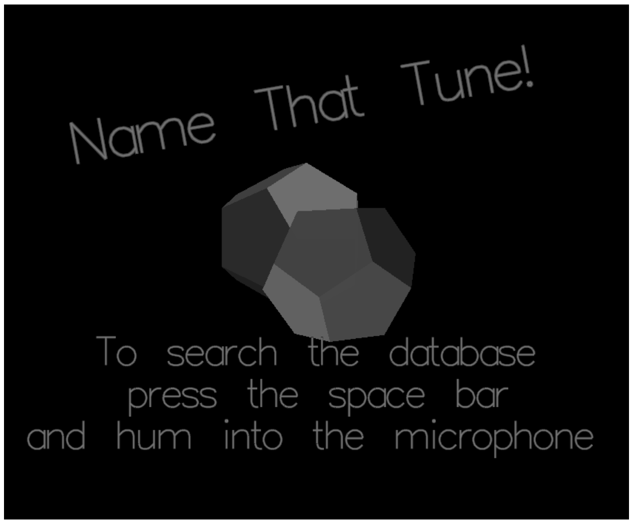
Query Length: 208,955 samples = 4.7382 sec
DTW for Entire Database = 166,643,813 cycles = 1.253 sec
Time/entry = 29.8 msec





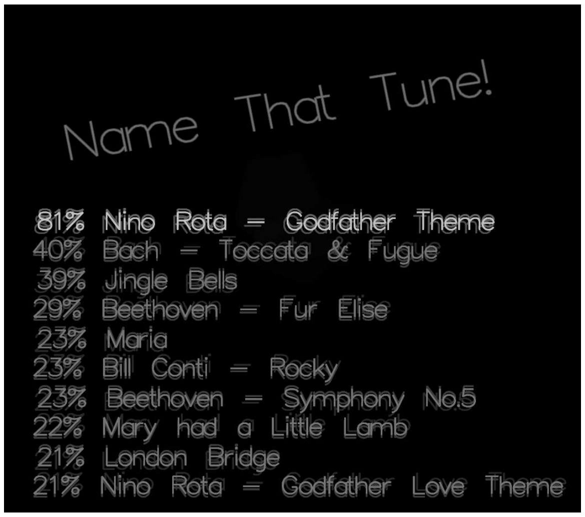
Demonstration

Our demonstration was very straightforward. We implemented a GUI that made our system very user-friendly. When the user was ready to begin humming, he/she would hit the space bar to tell the EVM to begin recording. Once the user was finished, he/she hit the space bar again. After the PC received the results from the EVM, the top ten scores were displayed. We varied the key and tempo of our hums to demonstrate the robustness of our system.



Name That Tune!

To search the database
press the space bar
and hum into the microphone



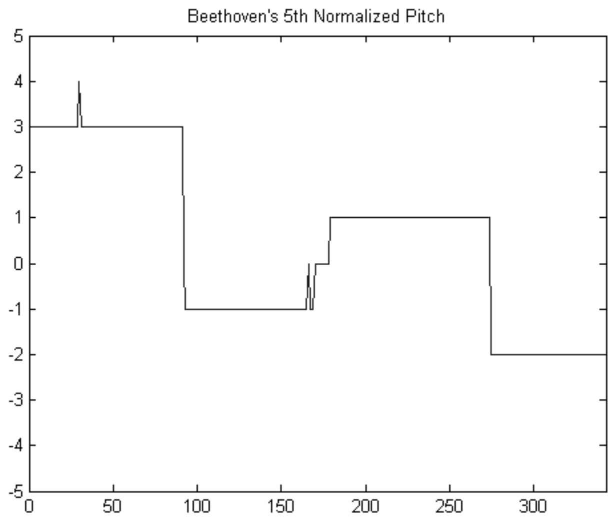
Name That Tune!

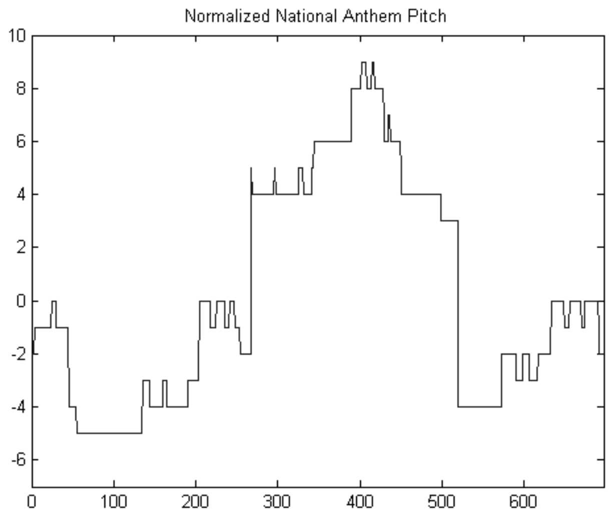
```
81% Nino Rota = Godfather Theme
40% Bach = Toccata & Fugue
39% Jingle Bells
29% Beethoven = Fur Elise
23% Maria
23% Bill Conti = Rocky
23% Beethoven = Symphony No.5
22% Mary had a Little Lamb
21% London Bridge
21% Nino Rota = Godfather Love Theme
```

Results

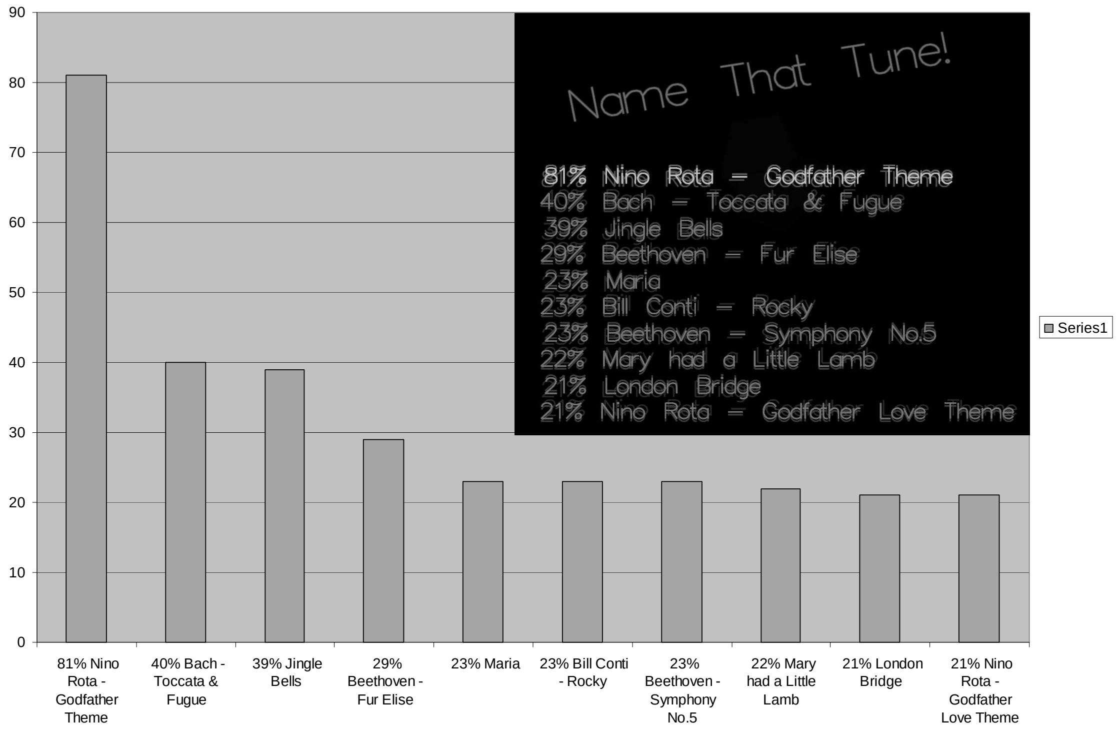
Pitch Track Results

The outputs of the pitch tracker for two hum queries are shown below, with Beethoven's 5th on the left and The National Anthem on the right. Apart from a few spikes, the results of the Beethoven's 5th hum are very accurate. The results for the National Anthem, however, show oscillations around notes. This is a common problem for the pitch tracker. The notes are doubles that have been cast to shorts so there is significant loss of accuracy. If the hum is slightly sharp, then it will register at the correct tone but if it falls below the tone, then it will be rounded down to the next note. The size of the oscillations may be reduced by increasing the resolution of the pitch tracker output. For example, we could track the pitches using a 24-note per octave equal tempered scale instead of a 12 note per octave scale. The other inaccuracies in intervals are due to the user humming, as it is a more difficult song to hum than Beethoven's 5th. Both of these hums returned correct matches. Despite the oscillations of the pitch tracker, the National Anthem still scored the highest at 38%.



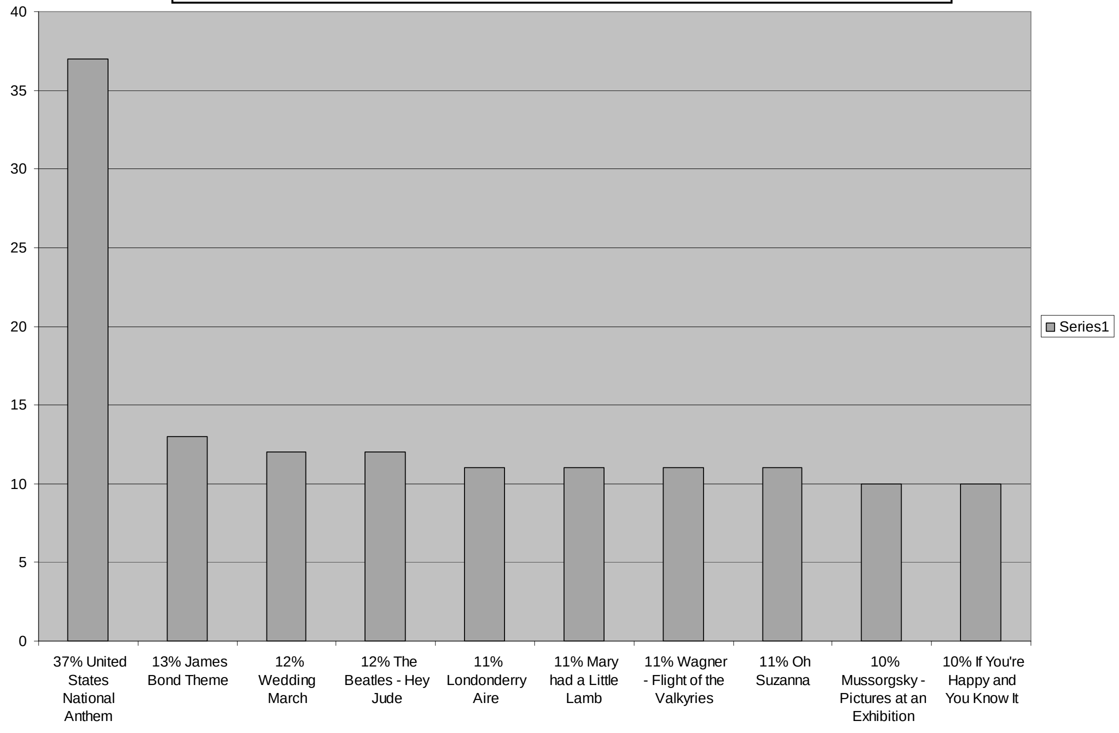


Hum Query: Nino Rota – Godfather Theme



81% Nino Rota – Godfather Theme
40% Bach – Toccata & Fugue
39% Jingle Bells
29% Beethoven – Fur Elise
23% Maria
23% Bill Conti – Rocky
23% Beethoven – Symphony No.5
22% Mary had a Little Lamb
21% London Bridge
21% Nino Rota – Godfather Love Theme

Hum Query: The U.S. National Anthem



Single Query Results

The results from humming *The U.S National Anthem* are clearly different from those of *The Godfather Theme*, even though both were successful matches. First, the top score is 81% for The Godfather and only 37% for the National Anthem. This could be due to several reasons. First, the Godfather Theme has a simpler entry in the database (7 notes) than the National Anthem (12 notes), so the user has more opportunity to hum out of tune in the latter. The National Anthem also has much larger intervals (max of 8 semitones) than the snippet for The Godfather theme (max of 4 semitones). For the amateur singer, larger intervals are much harder to sing or hum in this case. Another thing to notice is that with The Godfather results, the second highest score is approximately half of the highest, whereas in the National Anthem results, the second highest score is a third of the highest. This may be explained by the fact that there may be more entries in the database similar to The Godfather Theme. The National Anthem snippet covers a rather large range of 16 semitones and for a database of this size, this amounts to it being very unique. In contrast, The Godfather Theme has a range of only 5 semitones.

Test Set

To quantitatively test the system, 10 users searched the system for *Jingle Bells*. Two of the subjects were female and eight of the subjects were male. Every subject hummed in a different key or octave. The subjects also hummed at different tempos. Each subject hummed their own interpretation of *Jingle Bells* and interpreted the rhythm of the piece differently.

To further evaluate the system, a single user searched the system for the following 10 songs.

1812 Overture
The Godfather Theme
Jingle Bells
Happy Birthday
Can-Can
The Superman Theme
The E.T. Theme
Toccata & Fugue
The Rocky Theme
The Marriage of Figaro

Test Set Results

For the single-user/multi-song study, 8 of the 10 searches resulted in the queried song coming in first. *Toccata & Fugue* came in second rather than first and *The E.T. Theme* came in third. Although these two tests did not return the correct top match, the correct match was very close to the top, with both scores within 1% of the top score. With the *Toccata & Fugue,* the highest scoring song was *Pomp & Circumstance*, both of which begin with a high-pitched notes and drops approximately the same interval to lower pitched notes. When combined with user variations, this explains why the scores were so close and why our system had trouble distinguishing between the two. For similar reasons, *The E.T. Theme* came in third.

For the single-song/multi-user study, all 10 searches resulted in the correct best match for *Happy Birthday*. These results demonstrate the inter-user robustness of the system. The system works well with both male and female users and accommodates rhythmic and pitch variations.

What’s Good / What’s Not

Due to note normalization, the system is very robust to input that is hummed higher or lower than the actual database entry. For example, consider Mary Had A Little Lamb. The user can hum ‘B A G A B B B’ or ‘E D C D E E E’ and it will match the database just as well as humming the actual database entry of ‘A G F G A A A’. The system works even in the presence of a few sour or wrong notes. However, the system is less robust when it comes to a user who erroneously changes key in the middle of a selection. Thankfully, this is much less likely to happen when humming short snippets. Changing key due to dropping pitch or floating higher is more common when performing longer selections.

In the time domain, the system is very robust to global changes in tempo. By matching the length of each database entry to the pitch tracked input, the system is effectively global tempo invariant. Local variations in tempo are well taken in care of with the DTW matching. When using k-local DTWs the degree of local variations that are acceptable is controlled with k parameter. In the code we demoed, variations of ±½ second are allowed. Of course, this will potentially raise the scores of any snippet that has a similar series of note intervals. Nonetheless, since in our approach the distance matrix is computed as $D_{i,j} = (query_i - dbEntryUTW_j)^2 + \mathbf{1}$, the cost of extra +1s will cause the entry with the correct rhythm to be on top due to its shorter DTW distance. Which is simply to say that, the database entry whose note intervals are correct **and** whose DTW path most closely follows the diagonal of the distance matrix will be chosen as the best match.

The main issue with the method implemented in our project is that the user must hum an actual database entry. If the snippet the user is humming is not explicitly stored in the database, then the system will probably not return the correct song as the best match. For example, our database entry for Jingle Bells consists of the melody associated with the phrase “Jingle bells, Jingle bells, Jingle all the way.” If the user instead hummed the entire chorus, “Jingle bells, Jingle bells, Jingle all the way, Oh what fun it is to ride, in a one horse open sleigh,” then the system is likely to perform poorly.

In general, if a user cleanly hums a snippet that is either a superset as in the example above or a subset of a database entry, then the matching score for that database entry will be reduced. Moreover, the set of all other scores will generally not be reduced as a whole. Therefore, this will increase the likelihood of an incorrect best match.

It is our experience that humming an extra note or two usually will lower the score of the correct song, but it does not prevent the correct song from coming in first. However, this may very well change if more songs are added to the database. Therefore, it is of great importance that all common phrases for a given song be explicitly stored in the database. This will increase database size and using our search algorithm will increase search time by a factor equal to the number of snippets used in each song. For example, if our method takes 1 second to search a hum query on a database of forty songs with one entry per song, then it will take 4 seconds to search the same database if there are 4 snippet entries per song.

Conclusion

We have shown that it is feasible to use pitch tracking and DTW to implement a "Query by Humming" system. Our system was both accurate and fast on the EVM. Using DTW indexing schemes, it will be possible to create a web-based "Query by Humming" search engine. Thus, the time of the hum is now.

References

[1] http://www-mmdb.iai.uni-bonn.de/download/matlabMIDItools/
This site has a library for Matlab that reads and generates MIDIs. We modified the program to generate text files from MIDIs in the format we described earlier.

[2] http://sail.usc.edu/music
Here, we found some sample MIDI files that we used in our database.

[3] http://cnx.rice.edu/content/m11716/latest/
Here, there was Matlab code for an autocorrelation pitch tracker, which we later converted to C for this project.

[4] http://labrosa.ee.columbia.edu/matlab/dtw/dp.m
There was Matlab code for Dynamic Time Warping, which we converted and optimized in C. The modifications we made to the code are described above.

[5] Warping Indexes with Envelope Transforms for Query by Humming
http://www.cs.nyu.edu/cs/faculty/shasha/papers/humming.pdf
Yunyue Zhu, Dennis Shasha
Proc. of SIGMOD, 2003
This paper contained a description of performing a query by humming using a time-series approach that did not require note segmentation. Though the ideas of this paper are what we based our project on, it focused more on efficient warping indexes to optimize performance rather than explaining exactly what they have done in the past.

[6] Manipulation of Music For Melody Matching
http://www.acm.org/sigmm/MM98/electronic_proceedings/uitdenbogerd/
Alexandra. L. Uitdenbogerd, Justin Zobel
ACM Multimedia 98 - Electronic Proceedings
This paper illustrates that contour methods can return a large result set even with a short query hum.

[7] Phillips Electronics
http://www.research.philips.com/InformationCenter/Global/FArticleSummary.asp?lNodeId=986
Audio Fingerprinting