An Audio Duplicate Detector in MATLAB

Jay Coggin & Jordan Whitney
EEN 536 Final Project
December 13, 2011

INTRODUCTION

The advent of digital music and large digital music libraries has created the seemingly inevitable problem of digital library clutter. Libraries assembled from various sources and synchronized with multiple devices often accumulate duplicate files. This is especially a problem for large cloud music players such as Grooveshark (www.grooveshark.com) whose database is appended by user uploads, and thus has gathered a fair number of duplicate files. An automatic way of detecting these duplicates is often desired. However, these redundant files might contain different metadata, so text-based matching can only catch a number of the files. Much research has been done in the last decade to develop efficient audio fingerprinting algorithms to alleviate problems like these, and various parties have attempted duplicate detection software in various software languages [7, 8].

For this project, since we have been working in the MATLAB environment throughout the course, we created an audio duplicate detector in MATLAB that searches through a file directory, creates fingerprints based on the audio content, then searches efficiently through them to find duplicate songs. This report discusses background research on the subject of duplicate detection as well as the specifics of our implementation.

RESEARCH AND THEORY

An implementation of this duplicate detector would require two distinct processes. The first would be to create efficient, unique, fingerprints for each track, and the second would be to then look through these fingerprints and decide which tracks are duplicates based on some comparison schema.

For the task of creating fingerprints for each track, various sources ([1, 4, 5, 6]) were consulted to learn what types of fingerprint creation algorithms had been used successfully. We knew from the beginning that these fingerprints would need to be based on frequency domain content and not time domain content, since small amounts of distortion could lead two audibly duplicate tracks to appear completely different in the time domain. We also knew that there would need to be several fingerprints, separated temporally, for each track in order to maximize the uniqueness of each track's identity.

While we knew there would be a spectrogram involved as the first step in the fingerprint creation, we knew little of what would need to be done from this point in order to turn that spectrogram into searchable, identifying fingerprints. We soon found that after computing the spectrogram, many query algorithms concerned with identifying digital signals use a wavelet decomposition on the spectrogram [1, 2]. This is done in order to further compress the data represented in the spectrogram into just the top few rows of a wavelet transform matrix corresponding to the largest scale wavelets. Due to the Haar Transform's ease of computation and successful use in several algorithms [1], we decided to use this in our implementation.

Many papers then discuss various methods of further reducing the information contained in the decomposition matrix. It has been found that by keeping a relatively small number of largest magnitude coefficients in the matrix and setting the remaining coefficients to zero worked well [1]. Further, it was found that by only retaining the sign (+ or -) for each of these selected coefficients, not only did the process speed increase, but so did the performance of the query process [1]. Based on these findings, we implemented this sort of fingerprinting scheme in our program, which will be described further in the "Implementation Details" section.

In order to make these sparse wavelet fingerprints efficient for querying, we used a well-known computer science-based hashing scheme. Min-hashing has become a popular data compression scheme used in query-based systems, and has proven to work well in other audio-related fingerprinting studies [4]. The idea of Min-Hashing (and correspondingly, Locality-Sensitive Hashing) is to take a set of high-dimensional data and create several independent projections of this data which only contain a few dimensions of data. If the same projections are made of every high-dimensional set (each wavelet signature), then two fingerprints can be considered a match if a definable threshold value of these projections match.

The concept is easiest to visualize in terms of points on a sphere. If we have several points located on a sphere in 3-D space, each point will have 3 dimensions of data tied to it, its location. Imagine we want to find points on the sphere that are near each other, but that comparing 3 dimensions of data simultaneously on each point would be too costly on the CPU. Instead, we could compare just two dimensions at a time by making an X-Y projection of each point, an X-Z projection, and a Y-Z projection. If two points are truly close in 3-D space, each of their projections will be close, but if they are not, some projections may be close, while others are not. In this way, we can evaluate more sets of smaller dimensional data to test whether smaller sets of higher dimensional data sets are matches.

This concept can be extended to audio duplicate detection where each bit of the wavelet signature can be considered a dimension, and we pick out several relatively small dimensional set of these to create hash values. Thus, not only are we able to further reduce the dimensionality of each fingerprint, but we are able to create several different "projections" of the data sets which will allow for a query process that is more forgiving towards small differences in fingerprints.

IMPLEMENTATION DETAILS

The following section outlines the functions and process of our implementation. First, the process of fingerprint creation is explained, and the query process is explained after.

Fingerprint Creation

detectDuplicates.m

This is the entry point of the program. This file calls getAllWavsAndMP3.m to retrieve the files to process, then calls processDirectory to do work on them, and then runs the duplicate detection procedure, which calls getTracks.m for each fingerprint of each track, getting all the tracks which are duplicates of the current track. Then, it goes through all sets of duplicate tracks, and eliminates any redundant sets. It then prints out only the sets of duplicate tracks. This file contains definitions for the three main global data structures as well as the following variables: numHashTables, numHashKeys, numTracks, permutations, thresholdTables (the single-fingerprint threshold), thresholdPercentage (the overall threshold).

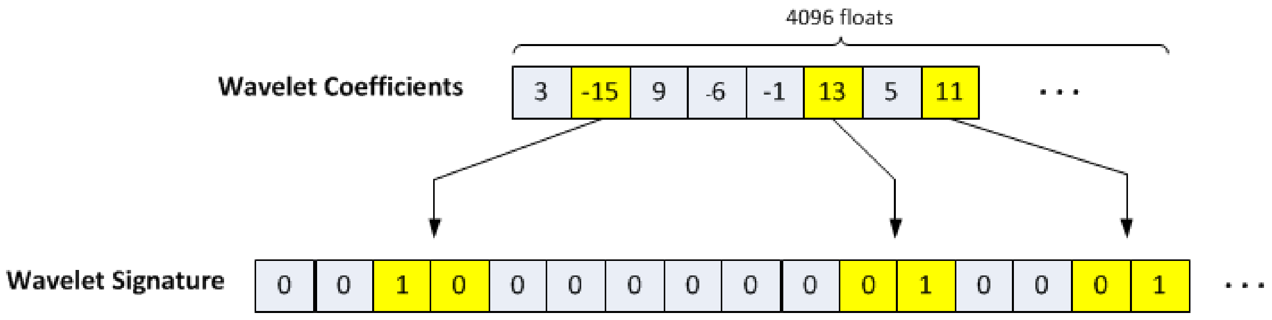
getAllWavsAndMP3.m
This file finds all .wav and .mp3 files in a given directory and its subdirectories. For retrieving these files, we modified a file named getAllFiles from [9].

processDirectory.m
This file generates fingerprints on all files. It calls createFingerprints.m and hashFingerprints.m.

filteranddecimate.m
After retrieving all file data and initializing structures, filteranddecimate.m performs the first audio processing in the chain. filteranddecimate.m takes a single file path as an argument and first opens that file using the appropriate reading function and creates a summed mono version. This is then chopped down to a ten second section beginning five seconds after the start of sound. The clip is then bandpass filtered using a 5th order elliptical filter with a pass band of 250-2,000 Hz. The elliptical filter was chosen due its ability to provide the steepest transition band for a given order. Its sub-par phase distortion was deemed irrelevant since the same distortion would be applied to each file identically. The low frequency cut-off was chosen in order to filter out powerful low frequencies that do little to identify a song. The high frequency cut-off was chosen in order to allow maximum decimation while still retaining enough unique information about each song. Since relevant frequencies for speech are known to be between 1 and 3 kHz, a cut-off of 2 kHz was chosen, allowing for downsampling by a factor of 8 from 44,100 to 5,512.5 Hz. These parameters have also been used with good results in the past [1]. After performing this decimation, filteranddecimate.m returns the vector of samples to a function within createFingerprints.m where it is called.

createFingerprints.m
The function createFingerprints.m first calls filteranddecimate.m as just discussed. The vector returned is first normalized to [-1,1], and is then passed to a series of sub-functions which create the log spectrogram. For creating the log-spectrogram from MATLAB's default specgram() function, we used portions of an m-file named logfsgram.m, from [10]. The stock spectrogram() function is called first, computing a 2048 sample DFT every 64 samples, giving an overlap of 31/32. The 31/32 overlap was used successfully in [1]. Each set is windowed using a Hann window, chosen for its excellent anti-aliasing properties and because its lower resolution would not be an issue since bins would eventually be grouped into 32 logarithmic frequency bins anyway. After this grouping is performed, the matrix of data is chopped into temporal sections corresponding to 128 STFT's in length, or 1.48 seconds of audio each. This creates 6 total spectral sub-images out of the 10 seconds of audio.

Following this, each of the 6 time slices is decomposed using the 2-D Haar decomposition. This matrix of Haar coefficients is then reshaped into a 1-D vector of length 128 x 32 = 4096 coefficients. For each of these vectors, a binary vector of length 4096 x 2 = 8192 bits is created and initialized to zeros. Thus, each coefficient in the coefficient vector maps to 2 bits in the binary vector. The function topwavelets.m finds the 200 largest magnitude coefficients in the coefficient vector, and sets those 200 coefficient's corresponding 2 bits in the binary vector to "01" if the coefficient is positive, and "10" if the coefficient is negative. The diagram below depicts this process.



Now, we have a length 8192 binary vector with mostly zeros and only 200 1's. This sparse bit vector will be referred to as the "wavelet signature". Since each audio file will have 6 spectral sub-images, they will also have 6 wavelet signatures. These 6 wavelet signatures are then combined into a 6 x 8192 matrix and returned to the calling function, processDirectory.m.

<u>twoDhaar.m</u>

The function twoDhaar.m is called within createFingerprints.m and simply computes a textbook 2-D Haar decomposition on the matrix it is passed, and then returns it as a matrix of the same size.

<u>hashFingerprints.m</u>

After creating the wavelet signature matrix, hashFingerprints.m is called and given the matrix. hashFingerprints() calls all the sub-functions needed to turn these wavelet signatures into efficiently sized min-hashed values and return them in a vector.

<u>computeMinHashSignature.m</u>

This sub-function is the first to be called within hashFingerprints.m. This function is responsible for taking in a single 8192 bit signature and creating 100 largely independent projections as required by Min-Hashing. The choice of using 100 permutations is based on statistical analysis of data sets and mutual information. Admittedly, some of the concepts are outside of our understanding of statistics, but 100 permutations have worked for other implementations in the past [7]. To create the .csv file of permutations, we used the output of PermGenerator.cs, from Google's Sound Fingerprinting Library [7]. This uses a pseudo-random Gaussian distributed random number generator to generate 100 sets of 255 integers ranging from 1 to 8192. Each permutation is then interpreted as a list of 255 indices specifying from which index of the wavelet signature each value, a 0 or 1, is taken. The function simply reads in these permutations from the .csv file, then for each permutation, begins getting the values in the wavelet signature indexed by the value in the permutation vector until it gets to the first "1". This index, an unsigned integer between 1 and 256, is then stored for each of the 100 permutations, creating the 100 min-hashed values for each wavelet signature. The process is perhaps easier to understand with the use of Figure 1 below.

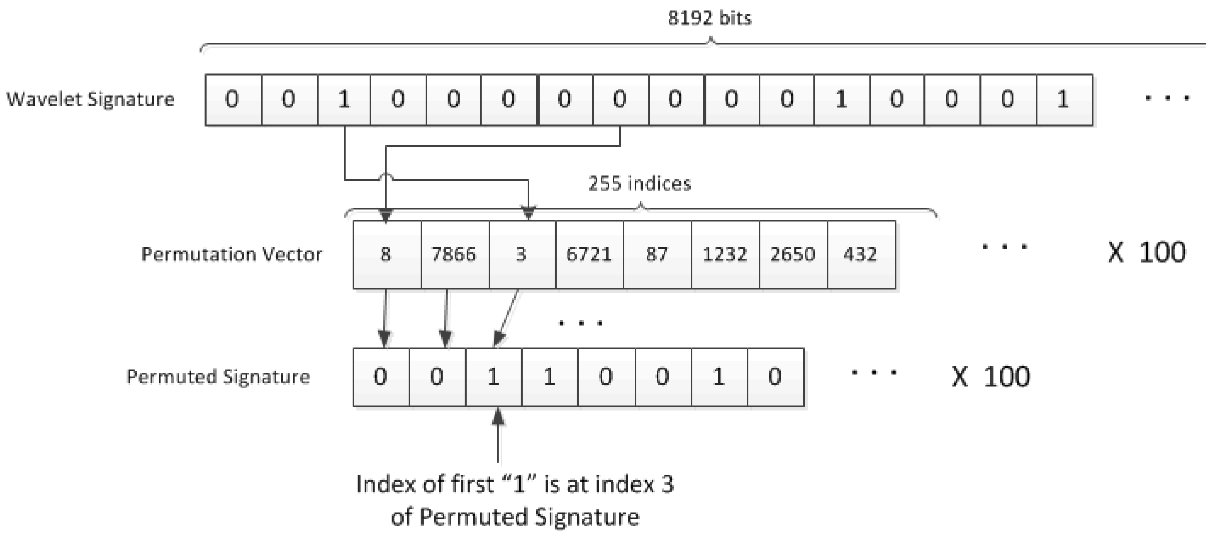


Figure 1: Permuted Signature Creation Scheme

groupMinHashToLSHBuckets.m

This function takes in a single vector at a time of the 100 indices found previously and does the final step in creating the true hashed values which will be used in the query. Since each index is between 1 and 255, each can be represented as an unsigned 8-bit integer. Since usual data types are 32 or 64 bit, these 100 indices are combined in groups of 4 using bit shifting to create 25 32-bit hash values per fingerprint. Then we take the modulus by 2147483659—the smallest prime number larger than half the maximum value representable by a 32 bit unsigned integer. This is used to minimize false collisions when hashing, as used in [7].

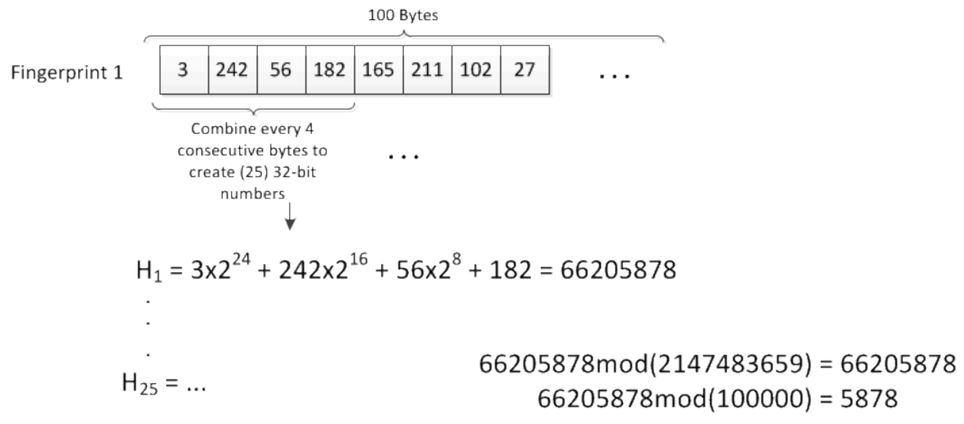

100 Bytes
Fingerprint 1
3 242 56 182 165 211 102 27
Combine every 4 consecutive bytes to create (25) 32-bit numbers
H1 = 3x2^24 + 242x2^16 + 56x2^8 + 182 = 66205878
H25 = ...
66205878mod(2147483659) = 66205878
66205878mod(100000) = 5878

Figure 2: Hash Value Creation Proces

Finally, we mod this value by 100,000, the maximum number of distinct hash values we want in the database. This is tuned manually in order to aid computation speed as it trades off with false collisions [7]. Finally, the function arranges these 25 final hash values in a vector and returns this to hashFingerprints() which in turn, places them in the map of values. This process is demonstrated in Figure 2.

This entire fingerprint creation process is summarized in Figure 3 below.

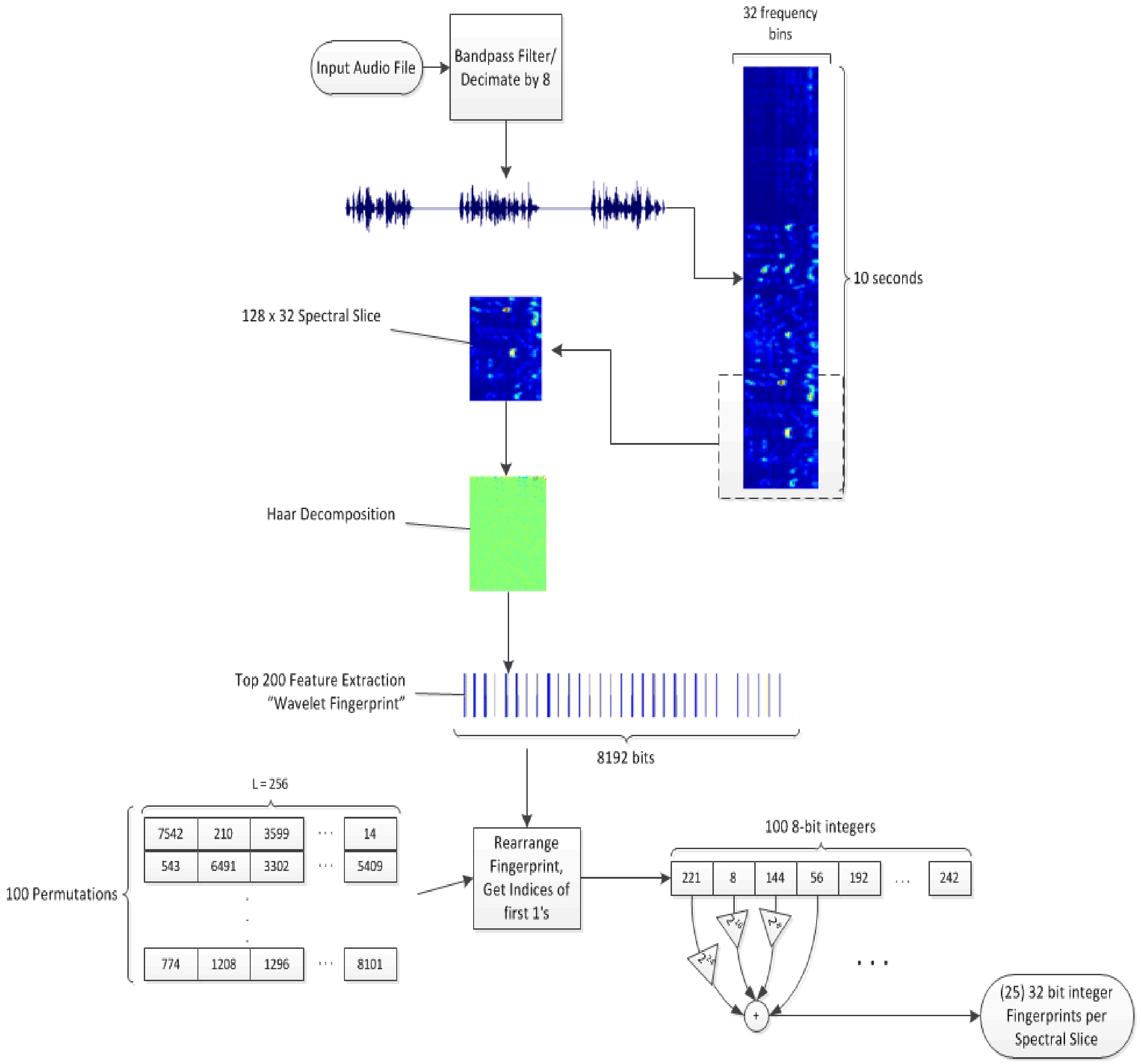


Figure 3: Fingerprint Creation Flowchart

Fingerprint Query Process

Given a music library, we need to keep track of all the audio (.wav and .mp3) files within that directory and its subdirectories, run our fingerprinting algorithm on a suitable portion of each file to generate their fingerprints, and save all of this data in a database. Since we were using MATLAB, we had a few options for data storage. The first was to use a relational database (mySQL in our case), and store three tables: track, fingerprint, and hash. These tables used the InnoDB storage engine and were defined as follows:

track: (<u>trackid:</u> UNSIGNED INT,filepath: VARCHAR(300))
fingerprint: (<u>fingerprintid:</u>UNSIGNED INT,fingerprint: VARCHAR(8192),*trackid*: UNSIGNED INT)
hash: (<u>hashid:</u> UNSIGNED INT,hashtable: UNSIGNED TINYINT,hashvalue: UNSIGNED INT,*fingerprintid*: UNSIGNED INT)

The **fingerprint** table has a foreign key *trackid* to the **track** table, and the **hash** table has a foreign key *fingerprintid* to the **fingerprint** table. With this database schema, we would minimize redundant data storage, and be able to relate one table to another very simply. In order to connect to the mySQL database, we used a file available on Mathworks online MATLABCentral FileExchange called "mym". This m-file allows for database connections to a number of different relational databases, including mySQL. We coded our MATLAB program to insert each track, each fingerprint, and each hashvalue into the database using a SQL INSERT statement, and retrieving them via SQL SELECT statements, with the appropriate WHERE clauses.

This proved to be very time consuming, especially since the length 8192 bit vector for the fingerprint had to be converted to a length 8192 character string upon insertion, and also in retrieval. At this point we decided to make use of mySQL's BULK_INSERT function, which inserts a delimited text file of database rows into the appropriate table at a speed that is orders of magnitude faster than individual INSERT statements. So, we wrote the tracks, fingerprints, and hashes out to text files, and used mym to bulk upload them. While the performance increase was significant, we ran into issues with the mym wrapper, as it had problems doing multiple bulk inserts one after another. After searching online forums, we found that some versions of MATLAB came with the Database Toolbox, which used the mySQL JDBC connector to support any function in the mySQL DBMS. We had access to this toolbox, and we converted all database calls to the Database Toolbox database(), exec(), fastinsert(), and fetch() commands.

In the process of setting up the MATLAB database access functions, we were simultaneously doing research on audio fingerprinting. It came to our attention that to detect duplicates, we would not need the length-8192 fingerprint after we had computed hash values for it. We noticed that even though the Database Toolbox helped with speed and functionality, keeping all of our data in main memory would be the fastest option. And, we would only need to store information for tracks and hashed fingerprints.

MATLAB is a great environment for working with matrices, but has difficulty with complex data structures. A MATLAB *struct* is a data structure which can contain any number of fields, each accessed by its field name. The struct provides easy storage and retrieval of data, and was a good starting point. Unfortunately, it is difficult to link structs together, as we would need to do in order to mimic the relations in a database schema. While MATLAB allows for the use of native Java objects (Collections, etc.), our intention was to create a MATLAB-based program, so instead we made use of the *containers.Map* object. The Map is similar to Java's HashMap, a mapping of (Key → Value), where each Key in the map must be numeric ('int32', 'int16', 'int8', etc.) and more importantly, it must be unique. The Value can be a numeric or string value, or a more complex data type, 'any'. The map structure mimics a table in the database, where the Key represents the foreign key, and the Value represents the rest of the row. Our main global data structures are set up as follows:

global *fileList* - a vector of file paths of all the audio files found within the library. The order in which they are stored is based on the recursive call to getAllWavsAndMP3, which does a breadth-first-search down the directory tree. This ordering is used to assign unique IDs to each of the files, which we take to be the file's index in the vector.

global *trackToHashes* - a containers.Map of ('int32' → 'any'), where the key is the ID of the track (its position in ***fileList***, and the value 'any' is a matrix of size (# fingerprints x 25). For the up to 10 seconds of samples we process, at most 6 fingerprints will be generated, so this matrix is no larger than (6 x 25)

global *hashtableToTracks* - a containers.Map of ('int32' → 'any'), where the key is the ID of the track (its position in ***fileList***, and the value 'any' is another containers.Map of ('int32' → 'int32'), where the key is the hashtable number (1-25), and the value is the hashvalue (an integer between 0 and 100,000)

For our global data structures, we adapted RamStorage.cs from Google's Sound Fingerprinting Library [7]. The C# library used structures such as Dictionary, HashSet, and also used LINQ syntax, which allows for queries on IEnumerable collections. We implemented the functions using these constructs in an equivalent, although slightly less efficient, MATLAB version.

With these global data structures, we now have a setup on which our m-files can operate. A summary of the m-files and their purpose is given below. The files are each individually line-by-line commented for completeness.

TESTING AND CONCLUSIONS

Robustness

Overall, we are very pleased with the algorithm's performance. The robustness to noise, time misalignments, and distortion is indeed evident in the results. The library used for testing contained 5 albums of music, with 8 duplicates added for a total of 60 files. Of the 8 duplicates added, two had 1 to 3 seconds of silence added to the beginnings, five had been converted to lossy formats at various qualities then converted back to .wav files, and the 8th duplicate was an exact copy of a file.

With slight hand-tuning of the threshold variables, we have been able to achieve very satisfactory results. Of the 8 duplicates, the program detects 7 of the pairs and returns no false positives. Upon inspection, the reason for the missed 8th duplicate pair became clear. Not only was the duplicate an extremely low quality (64 kbps) mp3 conversion, but the track began with a very slow, ambient fade-in, and because our program operates on 10 seconds of audio very near the beginning of the track, this non-unique sample of the song is more prone to false matching, especially when being compared to such a low quality conversion. The amount of time to skip before taking in the 10 second clip is easily adjusted in filteranddecimate.m but was set to just 5 seconds in order to accommodate short audio files.

The above results were achieved, as stated, after some manual tuning of threshold values for declaring a match. In particular, we used a 14 table threshold with a matching percentage threshold of 30%. These lower values are needed to cope with the extreme loss of fidelity in the low quality AAC and mp3 conversions. If the format conversions are limited to no less than 256 kbps, thresholds of 21 tables and 75% are sufficient to return all the duplicates.

Parallelization

We found that when operating on 10 seconds of data, the algorithm performed very fast. But, as we increased the length of the section of the track to process, we found the algorithm was slowing to unacceptable speeds. When debugging, we noticed that there were two bottlenecks of repetitive processes: generating fingerprints, and getting the matching tracks for each track in the list. In order to make it faster, we noted that the fingerprints for a particular track are independent of the data for any other track, and we could compute fingerprints in parallel to make our algorithm more efficient. In addition, finding matching tracks required only reading from the data structures, so that could be parallelized as well. Some versions of MATLAB include the Parallel Processing Toolbox, which contain the parallel for-loop construct, *parfor*. The idea behind the parallel for-loop is that it can execute the code on multiple cores of the processor simultaneously. It is a simplification of many thread synchronization concepts, and therefore there are certain restrictions. First, the parallel for-loop can only be used in place of a for-loop where the results of any one iteration do not depend on the results of any other iteration. And second, the parallel for-loop can only operate on “sliced” data. If there are n indexes of a vector, and each index is filled on each iteration of the loop, then this vector can be sliced, and only one index is sent to each thread to operate.

Making some minor changes to the data structures, we replaced the for-loop in processDirectory with the parfor construct, and created fingerprints in parallel. We replaced the main for-loop in detectDuplicates with the parfor construct, and found matches in parallel. The insertion of the data into the data structures and redundant duplicate set elimination was still done serially, as there was no simple mutual-exclusion/lock construct in MATLAB to protect the data structures during simultaneous reading and writing. On a machine with four cores, the parallel version saw a nearly 100% speedup in both fingerprint creation and duplicate detection, when processing 10 and 20 seconds of audio. The results are summarized below:

For 100 tracks in .wav format, processing 10 seconds of data for each track:

	Serial	**Parallel**
Time to create fingerprints	50.996485 s	24.083740 s
Time to detect duplicates	149.327207 s	76.925386 s
Total Time	**200.574802 s**	**101.310491 s**

For 100 tracks in .wav format, processing 20 seconds of data for each track:

	Serial	**Parallel**
Time to create fingerprints	83.416331 s	43.039454 s
Time to detect duplicates	325.684451 s	167.121577 s
Total Time	**409.285146 s**	**210.217278 s**

FUTURE WORK

Overall, we were quite pleased with our progress and the algorithm's robustness to significant format conversion distortion and time misalignments, meaning that the algorithm is a viable foundation for a commercial release. However, as it stands, our implementation isn't breaking new grounds. Most large digital music libraries consist of lossy format files such as mp3, aac, and mp4. Although our implementation will read in mp3 files, it's quite inefficient since the mp3read function must convert the file back to a PCM stream before any processing can be done. To make an efficient detector based on lossy format types without first converting to PCM data requires an entirely revamped fingerprinting algorithm, although the hashing and querying functions would remain largely the same. Very recently, there has been research into doing the fingerprinting straight from the mp3 domain [3], and as such, this is one of our top priorities for future work.

In addition, a bit of manual tuning of the thresholds and other values needs to be performed on a large library to find the values that produce the best compromise of minimal false positives and minimal false negatives. On a large scale test, we might find we need to use greater than 6 fingerprints per track in order to retain unique identities, or we might discover we could use less. In general, there are several values such as the 200 top features extracted, the 32 log-frequency bins created, and the 100 permutations, which were chosen based on research, but could possibly be made smaller in order to speed up the process. This type of testing still needs to be performed.

INCLUDED FILES

We have included a compressed .zip file of our project. The folder DuplicateDetector contains the serial version, and the folder DuplicateDetectorParallel contains the parallel version. To run either of the versions, change your MATLAB folder to the desired project folder, and run "detectDuplicates('/path/to/your/library/')." If your library contains .mp3 files in addition to .wav files, you must also have mp3read, and the associated binaries, which can be downloaded from http://www.mathworks.com/matlabcentral/fileexchange/13852. Note that processing .mp3 files will take longer, because mp3read must first convert the .mp3 file to .wav.

To use the parallel version, you must have MATLAB's Parallel Processing Toolbox. Before running detectDuplicates(), you must first enable the parallel language features by running the command "matlabpool". The number of cores available to do processing will be shown. If you do not have the Parallel Processing Toolbox, this command will not work, and you must use the serial version instead.

The project dependencies and method of operation are also outlined in the README.txt file.

REFERENCES

[1] Haitsma J., Kalker, T.,"A Highly Robust Audio Fingerprinting System", *Proc. International Symposium on Music Information Retrieval (ISMIR)* pp. 107-115, 2002.

[2] Jacobs C., Finkelstein A., and Salesin D., "Fast multiresolution Image Querying", *In Proc.* SIGGRAPH, 1995.

[3] Zhou R., Zhu Y., "A Robust Audio Fingerprinting Algorithm in MP3 Compressed Domain", *WASET Journal,* no. 55, pp. 715-719, July, 2011.

[4] S. Baluja, M. Covell, S. Ioffe, "Permutation grouping: Intelligent Hash-Function Design for Audio and Image Retrieval," *In Proc*. ICASSP, 2008.

[5] Wikipedia. "Discrete Wavelet Transform", http://en.wikipedia.org/wiki/Discrete_wavelet_transform. Accessed 13 November 2011.

[6] Slaney, M., Casey, M., "Locality-Sensitive Hashing for Finding Nearest Neighbors", IEEE Signal Processing Mag., pp. 128 - 131, March 2008.

[7] "Sound Fingerprinting System", *Google Code,* http://code.google.com/p/soundfingerprinting/, Accessed 13 November 2011.

[8] Sergiu, C., "Duplicate Songs Detector via Audio Fingerprinting", *The Code Project,* http://www.codeproject.com/KB/WPF/duplicates.aspx, Accessed 13 November 2011.

[9] "How to Get All Files Under A Specific Directory in MATLAB", *Stack Overflow,* http://stackoverflow.com/questions/2652630/how-to-get-all-files-under-a-specific-directory-in-matlab, Accessed 13 November, 2011.

[10] Ellis, D., "Spectrograms: Constant-Q (Log-frequency) and conventional (linear)", *Columbia University,* http://labrosa.ee.columbia.edu/matlab/sgram/, Accessed 13 November, 2011.