Artificial Intelligence (CS182)
Prof. Alexander “Sasha” Rush
Vivek Jayaram, Samarth Singal and Saroj Kandel
December 08, 2015

Auto DJ Mixing

Introduction:

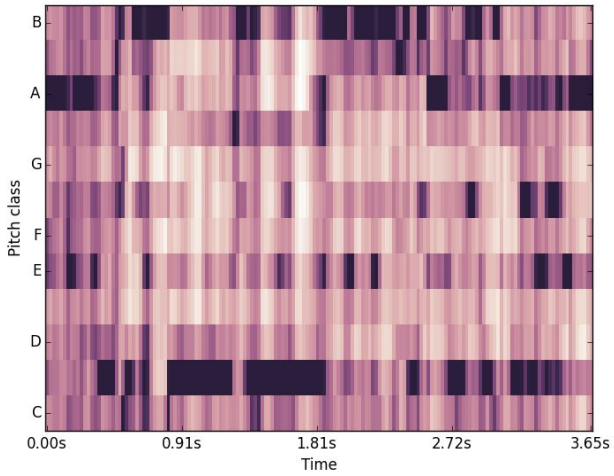
Our goal with the project to build an automated DJ program that can create a mix of several songs. The ability to mix and mash songs together is highly desirable by listeners and is a feature of live parties, radio shows, and online recordings. While the ability to create these mashups takes a certain degree of human intelligence and creativity, the basic properties of good mashups are well defined and easily automated. A high level description of the project is as follows: We will have a song collection of *n* songs and look at the how well each song matches with each other. Then the algorithm will create the best mix of *m* songs where n and m can change and m<n. Before running the algorithm we pre-populate metadata for the songs in a json file by providing aspects like tempo, and the beat numbers where we want to mix in and mix out. Then there are three main parts to the AutoDJ. First we look at all pairs of songs at their mix-in and mix-out points. We use a fourier transform with some other signal processing to determine how well the songs match. Then we model the problem as a CSP to find the best mix of *m* songs. Each song also has a unary factor representing how good the song is, and we run the CSP to ensure that all transitions meet a minimum mashability and that each song played has a sufficiently high unary factor. Finally we use several libraries to help beatmatch and crossfade the songs, as well as time stretch if necessary.

Determining the Mashability of Songs:

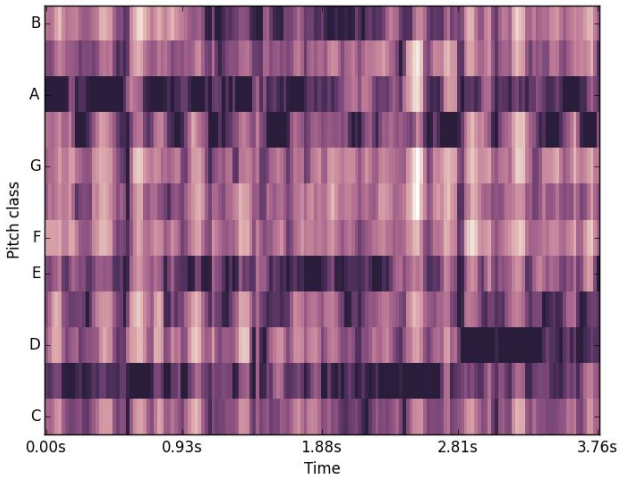
There are several factors that determine whether two songs mash together well - these include key, tempo and timing. Attempting to automatically determine the optimum alignment increases the problem size exponentially, so we choose to manually input a mix-in and mix-out point for every song.

We use aural similarity to determine "mashability" or how well two songs will mash together. To compute the mashability of a pair of songs (A,B), where A is to be mixed-out and B is being mixed-in, we consider 32-beat long samples starting at the mix-in point in A and mix-out point in B. Songs are naturally in a particular "key", corresponding to the dominant frequency ranges in the song. Particular key combinations sound well together because most composition frequencies are in the same ranges called semitones. It is important to compare the power spectrum similarity as classified by semitones because even two sounds close in frequency but in different semitones don't sound good mixed together.

Using Librosa, a python audio manipulation library, we sample the mix-in and mix-out sections at 22 kHz converting it into raw power signal. We then use a stepwise fast-fourier transform (FFT) to bin the power in each power into the 12 semitones. We thus get a 12-dimensional vector for each beat point with the component magnitudes the power in that a semitones. Thus a $12 \times$ *number of sampled points* matrix representation, called chroma, is produced. A visual representation of this matrix is called a chromagram seen in figures below.



Song 1 Chromagram

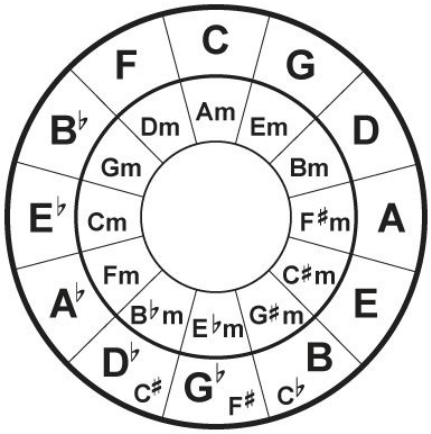


Song 2 Chromagram

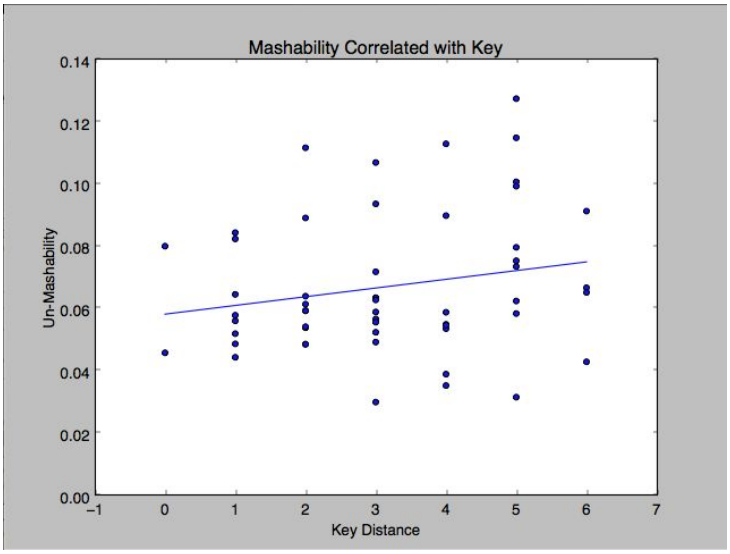
We then use the cosine distance between corresponding chroma vectors of the two songs to come up with a an array of cosine distance between the two songs. We calculate the mean of the cosine distance array. A smaller cosine distance means the songs have close semitone composition and will thus sound well together. A higher cosine distance indicates the opposite. We return this as the "mashability" score. Thus "mashability" is a float between 0 and 1 where lower values indicate better mashability.

Evaluating our Mashability:

One of the difficulties of this project was that mashability and DJing is inherently subjective, and there is no objective metric to compare it against. However we wanted some way to evaluate how good the previous algorithm is at determining how well two songs mix. One such way was to compare it to another measure of mashability. DJs typically use the idea of musical keys to predict how well two songs will sound together. Below is the circle of fifths, a tool commonly used in musical key analysis.



The closeness of two keys on this wheel indicates how many notes two keys share in common. As a result we would expect our algorithm to give better mashability scores for songs that are closer together on the wheel. We created a separate script to plot the pairwise mashability as a function of the distance between the keys of the two songs. Below are the results with a best fit line superimposed.



This shows that in general there is a greater clash in the songs as the key distance increases;which is what we would expect. However there are clearly some outliers, and our algorithm shows that songs with clashing keys may indeed sound good together. One such example is that “Uptown Funk” in F major and “Jealous” in F minor gives the best mashability score at .029 even though the keys are 3 spots away on the wheel. When listening to the mix that was created, “Uptown Funk” started with a lot of percussion and just a single note of F, so the

other conflicting notes in the key were not present. Not only was the mix very good, but it also showed that our algorithm takes into account much more subtlety of the song than the idea of looking at key alone. It also shows why song 1 mixed into song 2 can give such a different score than song 2 mixed into song 1, since "Uptown Funk" mixing into "Jealous" gave a much worse score of .063, as expected. Our algorithm followed the trend of key distance and the outliers were still aurally pleasing, and this made us confident in it's ability to predict mashability. (This example can be found in outfiles/BestMix.mp3. It sounds fantastic as our algorithm predicts, despite having very different keys)

Modeled as Constraint Satisfaction Problem:

We cast the problem of outputting the mix of *m* songs given a list of *n* songs as a Constraint Satisfaction Problem. There are *m* variables, each corresponding to a slot in the mix. Initially each slot has all the songs in its domain. A song's "awesomeness" is constrained by its mashability with other songs, defining how well the song fits with other songs, and its unary factor, determining how good a particular song is. There are |*m-1*| factors each connecting successive variables. A factor between *x1* and *x2*, for example, constraints *(x1, x2)* to values where the mashability is higher than some value, epsilon.

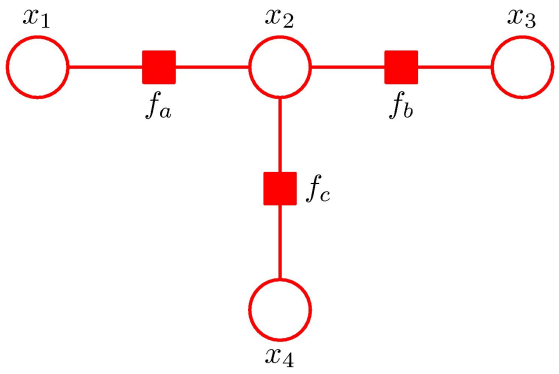


Fig 1: Factor graph with songs constrained to mashability factors

For the implementation of mix generation, we used backtracking algorithm with forward searching. Our 'generateMix' function takes the list of songs from the user, number of song mixes that the user asks for, a dictionary of pairwise mashabilities of all songs, and another

dictionary of unary factors that defines how good a song is in isolation. It then initializes the m variables to all be -1 and places the input songs as initial domain of the variables. Then, it calls the backtracking function.

The recursive backtracking function takes in the variables, their domains, all pairwise mashabilities and unary values. The function starts by picking the first uninitialized variable by calling the 'pickUnassignedVariable' function, and checks the length of its domain to be non-zero. Then it picks the next best song in its domain. Our next best song is defined in two different ways, depending on the position of the unassigned variable slot. Initially when all slots are empty, we check the initial song list and place the first song that has a unary factor greater than a certain threshold in the slot. Because our music library is small, right now all songs are defined to have sufficient unary values, but changing them above the cutoff results in them never getting played. We have currently defined our unary threshold to be 100, an arbitrary scale. The mixability threshold is defined as 0.08, but could change it by altering the constant. Remember that a smaller mashability is actually better, the number is a measure of how far apart the songs are. However, later on where there are some slots already filled, we integrate both unary factors and mashability into the function to pick the next best song in the mix by making sure that both the mashability with the previous song in the mix and the unary factors are better than the threshold value. Also, we add a check to ensure that no song in the mix repeats itself.

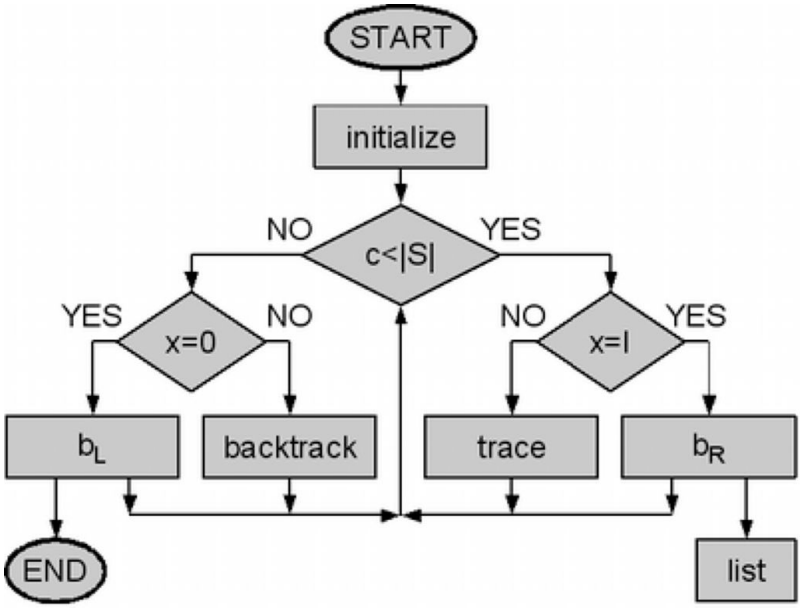


Fig 2: Visualization of our backtracking algorithm

The forward checking happens within the backtracking function, when the next best song returns -1, signifying that there is no any song in the domain with the previous mix assignments that satisfies the unary values and mashability constraints. At this point, we remove the previous slot assignment from the mix, and update the domain of that slot by pruning that particular song. We then recurse the backtracking function with the removed slot.

On the other hand, if our next best song returns true, we just put it on the first uninitialized slot and recurse. This function returns the song mix list when all slots have been filled with mashable songs, or returns false if domain pruning by forward checking results in an empty domain.

Outputting the Final Mix

Once we figure out the best way to mix the songs, we want to create the actual mp3 file. The primary library of use is EchoNest Remix. This looks at a song and gives an array of "beats", so that each song can be sliced by downbeats rather than by time. While this is the most intuitive way to manipulate the track, the EchoNest Remix often detected beats incorrectly. This meant that the transition was not aligned and sounded muddy as a result. An improvement would

be to use a more specialized beat detection algorithm, but this is a very difficult problem that professional DJ software often gets wrong. If the two songs have the same tempo then we can just use a simple Crossfade() to gradually lower the volume of the song mixing out while increasing the volume of the song mixing in. The rest of it is a simple application of the library.

However, if the tempos are different then a more complicated technique is needed. The ideal way to mix songs of different tempo is to gradually transition from tempo1 to tempo2 in a linear way as the volume also changes from song1 to song2. However there were no libraries that did a linear speedup of songs, only a constant tempo change. What we then chose to do is make a step wise tempo change by rendering one beat at a time in the transition section, and shifting the tempo of the two different songs so they matched. Each beat had a constant tempo, but over the entire transition section this tempo went from tempo1 to tempo2. Ther result sounded like a linear speedup/slowdown and an example is in outfiles/ExampleTempoChange.mp3.

Future Work and Conclusion

There are several shortcomings of this Auto DJ algorithm that could be addressed in a more thorough software. The major one is that the mix in and mix out points should be detected automatically. This will allow songs to be played without manually entering metadata beforehand. However this would involve determine macro level statistics about a song such as where the chorus and verse starts, as well as detecting the presence or absence of vocals. The beat detection provided by EchoNest is also not fantastic and as a result certain songs may result in unaligned transitions. Using a better library would fix this problem. Another change we wanted to make was to formulate the mixing as an optimization problem. Right now it only checks if the unary factors are above a certain threshold. However we would ideally try to maximize the sum of unary factors constrained by each transition. This could be done by modeling the problem as a graph with edges between two songs only present if the transition is good enough. While this algorithm would not be too hard to implement, the constaint version of this problem was more closely aligned with the course's goals.

When reflecting on this project, the parts that taught us the most were the low level signal processing. Looking at how humans perceive musical transitions and how that could be

replicated by software made us learn more about music and human perception in general. The idea of taking an intangible aspect of human intelligence and trying to formulate it mathematically is one of the most exciting aspects of artificial intelligence. This project also taught us how to find and evaluate libraries. There were many audio processing libraries for python, but finding the right ones that allowed the easiest integration was difficult. In the end we could have used some better libraries, but changing it would have been more difficult than sticking with the ones we had.

References:

Automatic Multi-Song Mashup System. 2013.
http://telecom.inesctec.pt/~mdavies/pdfs/DaviesEtAl13-ismir.pdf

Mu]ller, Meinard. Signal Processing for Music Analysis. 2011.
https://www.ee.columbia.edu/~dpwe/pubs/MuEKR11-spmus.pdf

Appendix 1

We provided the metadata for 7 different songs of varying tempos in the mp3/ folder an song.json. However the preprocessing for mashability runs in n^2 and may take several minutes for 7 songs. There are many sample outputs in outfiles/, but here's a trace of one. The songs.json has the following songs: Jealous, Outside, Crazy in Love, Safe and Sound, Uptown Funk, and the mash length is 4 songs. It first loads the songs into EchoNest while outputting a line the following for each song:

['en-ffmpeg', '-i', 'mp3/UptownFunk.mp3', '-y', '-ac', '2', '-ar', '44100', '/var/folders/7g/x1924_tn2rn9qt9v1qzmjp9w0000gn/T/tmpAEPZZE.wav']

Then it runs through each pairwise mashability. Because a different library handles time stretching, the audio data has to be encoded to a regular file format (not EchoNest AudioData object) so the other library can read it. this is the lines ['en-ffmpeg', '-i', '/var/folders/7g/x1924_tn2rn9qt9v1qzmjp9w0000gn/T/tmpdFQjWX.wav', '-y', '-ab', '128k', '-ac', '2', '-ar', '44100', 'temp1.mp3'], and the files temp1.mp3 and temp2.mp3. Ideally we would write them to local file buffer in order to avoid going to disk to speed this up, but it still works. Then we get all pairwise mashabilities printed out as follows:

{(Safe and Sound, Uptown Funk): 0.051989431577031016, (Jealous, Wasted): 0.091346177888184513, (Jealous, Safe and Sound): 0.05885145669236605, (Jealous, Uptown Funk): 0.029994323421050774, (Uptown Funk, Wasted): 0.093676670492733288, (Wasted, Safe and Sound): 0.047519903659596421, (Outside, Wasted): 0.058962664327057941, (Crazy in Love, Wasted): 0.054692522124541031, (Uptown Funk, Crazy in Love): 0.064591006566672937, (Wasted, Crazy in Love): 0.053957868010832445, …….. and so on with n^2 - n entries

Next it prints out the songs it chose and the values of those mashabilities:

Outside->Wasted: 0.0589626643271

Wasted->Jealous: 0.0581602506725

Jealous->Safe and Sound: 0.0588514566924

[u'Outside', u'Wasted', u'Jealous', u'Safe and Sound']

Finally it writes those files out. The output of this particular example can be found as outfiles/example2.mp3

Appendix II

First you need to clone the repository from the url listed below: https://github.com/vivjay30/AutoDJ/settings. Make sure the file /mp3s actually contains all the mp3 files. In order to use the system you must also have the following libraries: EchoNest Remix (pip install remix), Numpy, Scipy, dirac, librosa, matplotlib. You can choose the songs you want by selecting the subset in songs.json. allsongs.json has all the songs (8 of them) that we provide support for, but limiting songs.json will make the algorithm run faster. You can also specify the NUM_SONGS argument at the top of main.py (only change that constant). If NUM_SONGS is much less than the number of songs in songs.json, you can change MIX_THRESHOLD to a lower value like .05 to only allow better mixes. The program will complain if no mix could be found so don't be too stringent. running "python main.py" will run the AutoDJ code and "python key_graphs.py" will run the graphs that output which keys the songs are in and how well they mash.

Appendix III

Vivek Jayaram- I worked mainly on outputting a song once the csp returned the mix list. This involved the use of EchoNest to grab and mix the right sections of the song. The tempo shifting part of that was particularly tricky. I also setup all the skeleton code, the songs.json files, and wrote the code to visual mashability as a function of key difference.

Saroj Kandel- I worked on formulating the problem as a CSP and implemented the functions to output the mix as a list of song IDS, given the input list of songs. This involved implementing the backtracking algorithm with forward searching, and all their helper functions such as getting the next unassigned variable in the slot, finding its domain, finding the best song

in the domain to assign to that variable, and to prune its domain if there are no songs available for that particular slot.

Samarth Singal- I worked on determining the “mashability” of 2 songs. This involved understanding music theory and designing a strategy for key matching. I also experimented with about five different audio signal processing libraries to compare results of the FFT and semitone binning. The chromagram graphs were also produced as part of this exercise.