Supplemental Note on Shazam Audio Fingerprint

Minh N. Do

After computing the spectrogram and finding spectrogram peaks for each audio file with ID i_n we obtain a sequence of time-frequency peak locations, indexed by k, as:

$$\mathcal{P}^{(i_n)} = \left\{ \left(t_k^{(i_n)}, f_k^{(i_n)} \right) \right\}_{k=1}^{K_{i_n}}.$$

Likewise, for an input query audio segment q, we obtain a sequence of time-frequency peak locations as:

$$\mathcal{P}^{(q)} = \left\{ \left(t_l^{(q)}, f_l^{(q)} \right) \right\}_{l=1}^{K_q}.$$

The problem now is to have a fast algorithm that given the query time-frequency peak sequence $\mathcal{P}^{(q)}$, find the audio file ID i_n^* and the time offset $\Delta t^{(i_n^*)}$ that have the best match peak locations at that offset:

$$\left\{ \left(t_k^{(i_n^*)} - \Delta t^{(i_n^*)}, f_k^{(i_n^*)} \right) \right\}_{k=1}^{K_{i_n^*}},$$

with the query peak sequence $\mathcal{P}^{(q)}$. The naive solution that "overlay" $\mathcal{P}^{(q)}$ over each $\mathcal{P}^{(i_n)}$ on the time-frequency plane and then "slide" $\mathcal{P}^{(q)}$ along the time-axis until we find the best match does not scale with big data as the complexity of this solution would grow linearly with the number of audio files in the database (claimed to be hundreds of millions songs by Shazam) times the number of potential matches in each file (would be about 3000 for songs of length 5 minutes and matching step size of 100 milli-seconds).

To understand how the Shazam Audio Fingerprint works let's consider a simpler hashing scheme. We simply use the frequency index of each time-frequency peak as the hash. Note that the frequency index $f_k^{(i_n)}$ from spectrogram is the FFT index, $f_k^{(i_n)} \in \{0, 1, 2, \ldots, W/2\}$ where W is the length of the window used in spectrogram computation (we only retain the positive frequencies or first half of FFT magnitude coefficients due to symmetry). For each computed peak $\left(t_k^{(i_n)}, f_k^{(i_n)} \right)$ from an audio file in the database we store in the hash table an item indexed by $f^{(i_n)}$ the data entry that contains the file ID and the corresponding time $\left(i_n, t_k^{(i_n)} \right)$. Note that there could be zero or more than one data items stored in the hash table with a given index. The latter case is referred to as hash collision.

With the hash table constructed from audio files in the database, given the query peak sequence $\mathcal{P}^{(q)}$ for each peak $\left(t_l^{(q)}, f_l^{(q)} \right)$ we use the frequency index $f_l^{(q)}$ to retrieve the stored data items in the hash table. Note that with the hash table this retrieval takes O(1) operations for each time-frequency peak in the query. Each retrieved data item from the hash table is a pair $\left(i_n, t_k^{(i_n)} \right)$ in which $f_k^{(i_n)} = f_l^{(q)}$. Such a retrieved data item $\left(i_n, t_k^{(i_n)} \right)$ indicates a likelihood that the query audio segment q is matched with the database audio file i_n at the time offset $\Delta t = t_k^{(i_n)} - t_l^{(q)}$. We

tally this hit in a histogram for the audio file ID i_n at the time offset Δt. After parsing through all the peaks in $\mathcal{P}^{(q)}$ the audio file ID and offset with the largest number of hits indicate the best match with the input query segment. The complexity of this hash-based algorithm is linear with the number of peaks in the query $\mathcal{P}^{(q)}$, which is about a thousand for a 10-second query audio segment. This complexity is much smaller than the complexity of the naive matching algorithm above.

One remaining problem with the above hashing algorithm is that since the set of possible frequency index $\{0, 1, 2, \ldots, W/2\}$ is relatively small (for typical choices of $W = 256$ to 2048). Consequently there would be many hash collisions when we store the extracted time-frequency peaks from the audio files in the database. Hash collision implies that for a given time-frequency peak in the query we might have to examine many data items in the hash table with the same hash index. The Shazam system addresses this problem by using a combinatorial hash that is composed of the frequency index of an anchor peak, the frequency index of another fan-out peak, and the the time different between these two peaks. The new combinatorial hash has about 3 times more bits, and hence significantly increase the hash table size and reduce hash collision. If the fan-out size is F then the number combinatorial hashes increases by a factor of F. With a well-chosen value of F, say $F = 10$, we slightly increase the number of hashes while significantly reduce the number of hash collisions and thus speed up the overall searching time.