Here is a brief description of how Synth goes about pattern matching what you're playing to a synthesized source wav file.

1 Signal Processing

The first step is to take the raw waveform and represent the musical information in a way that makes sense. Notes correspond to resonating frequencies, so the Fourier transform provides the correct tool. We take the waveform and divide it into windows, where each window represents an amount of time roughly equal to one-third of a sixteenth note, the shortest note in the demo piece we used. Now we perform the Fourier transform on each window and take absolute values - this gives us a vector of amplitudes whose length is the length of the window we used.

The next step is to identify the frequencies which correspond to actual notes as opposed to other artifacts. The frequencies on a piano are given by $f(n) = 2^{\frac{n-49}{12}} \times 440$ Hz, where n is the key number going from 1 to 88. Note that because of the sampling rate and the finite window, this gives a leeway of about 3 Hz to take into account out-of-tune keys. We pick out the amplitudes of these 88 frequencies.

Finally we take the notes out of this vector of length 88. Notes do not resonate at pure frequencies, even when generated by an electric keyboard; overtones are almost always heard. We thus add up the amplitudes which correspond to the same tone to obtain a vector of length 12 consisting of the twelve possible note letters on a keyboard. We call this the harmonic information.

2 Pattern Matching

Identifying the location corresponds to a minimum cost alignment problem. We can compute a distance between two length-12 vectors by taking their Euclidean distance, but to identify musical phrases, this needs to be done over two time series of length-12 vectors, which are repsented as matrices where one dimension is the 12 notes and the other is the number of samples. The naive way to do this is to extend the Euclidean distance by stacking the matrix into a longer vector, but this approach only works if everything is played in perfect tempo.

To solve this problem we use a modification of Derivative Dynamic Time Warping [KP01], an algorithm originally designed as a similarity measure between time series. Let W be the harmonic information computed from source waveform, of length n, and T be the harmonic information of the input audio, of length m. We define an alignment to be a non-decreasing mapping

$$\phi : \{1, \cdots, m\} \longrightarrow \{1, \cdots, n\}$$

such that the following conditions hold:

$$\phi(i+1) - \phi(i) \leq 2 \tag{1}$$
$$\phi(i+2) - \phi(i) \geq 1 \tag{2}$$

These two conditions require that the input audio T is not more than twice as fast or less than twice as slow as any passage in W. Then the cost of the alignment is defined to be

$$\sum_{i=1}^{m} \|W[\phi(m)] - T[m]\|_2$$

This alignment can be computed efficiently in $O(nm)$ using dynamic programming. We define $D(i,j)$ to be the cost of aligning observations $T[1]$ through $T[j]$ in W such that $\phi(j) = i$. The row $D(:,1)$ is initialized by just taking the Euclidean distance of $T[1]$ with every element of W. The update step is given by

$$D(i,j) = \min\left(D(i-2,j-1), D(i-1,j-1), D(i,j-1)\right) + \|T[j] - W[i]\|_2$$

The indices of the first coordinate in the minimization step enforces condition (1). To enforce condition (2) without significant computational overhead, we copy W and interlace it to form W^* such that length$(W^*) = 2n$ and $W(2k) = W(2k-1)$. We then put the restriction that ϕ is an injective mapping, which takes care of condition (2). Finally to adjust the third condition, since we slowed the piece down by a factor of 2, the ≤ 2 is replaced by ≤ 4. the 2 is replaced by a 2, so the final update step is given by

$$D(i,j) = \min D(i-4:i-1, j-1) + \|T[j] - W^*[i]\|_2$$

where we've used MATLAB notation $i-4:i-1$ to represent the vector of length four given by considering indices $i-4$ through $i-1$.

Finally the optimum alignment is computed by taking the minimum of the vector $D(:,m)$.

2.1 Doing it all in real time

An $O(nm)$ algorithm is not very efficient, but the key observation is that the dependency on the second component moves forward in time. Thus, with each new $T(m+1)$, we can immediately compute the row $D(:,m+1)$ in $O(n)$ time if the previous information is stored.

To store this information we initialize a $(2n) \times (2n)$ matrix at the very beginning. This matrix is cleared everytime the app detects a period of silence, which enables the user to jump to a different area of the piece as well as allowing the application to not have to initialize new matrices, so long as the user does not play the entire song at less than half tempo.

The waveform displayed at the bottom of the program is the vector $D(:,m)$ being updated in real time.

3 References

[KP01] E. Keogh, M. Pazzani (2001) Derivative Dynamic Time Warping. *First SIAM International Conference on Data Mining*