Stephen Toub
sht213@nyu.edu
May 1, 2006

System for the Automatic Extraction and Segmentation of MIDI Files

The NYU Query By Humming system is based on whole-sequence comparison of a hummed series of notes against a database of songs. As the system is not capable of doing partial matching, the database does not contain one complete series per song. Instead, it contains multiple segments from each song, where each segment is a piece of the song likely to be hummed as a unit. This allows the system to do whole-sequence comparisons between the hummed sequence and sequences in the database.

A core problem with this system is that it requires the database to be populated with appropriate and useful segments. Up until now, this population has been a manual process, requiring human intellect and interaction to extract the melody from songs and to split up those melodies into appropriate segments. The man hours required for this task place a significant implicit limit on the song database's potential growth and size.

The goal of this project was to come up with a system for the automatic extraction of melodies from MIDI files (a source of parsable music abundant on the internet) and the automatic segmentation of those melodies into sections that can be directly input into the database. This allows for a system to be built that scours the Web for MIDI files, runs them through the preprocess extraction and segmentation system, and inserts the new information into the database.

Architecture

The developed system is built as a pluggable framework of filters, allowing for new algorithms and approaches implemented as filters to be plugged into the system at any time, simply by implementing the appropriate interfaces and adding the location of the developed class to an XML configuration file.

The application, AutoSegmenter, is supplied with command-line arguments for the location of input and an output files. The input location can be either a single MIDI file or a directory of MIDI files which will be searched recursively for all available files. The output location refers to a directory that will contain an output file for each input MIDI file successfully processed; this output file contains all of the melody notes and the segmentation information that can then be fed into the database. As an example, if the directory of files to be processed was at C:\inputFiles\ and the directory in which to store the output files is at c:\outputFiles, the command-line execution would look like the following:

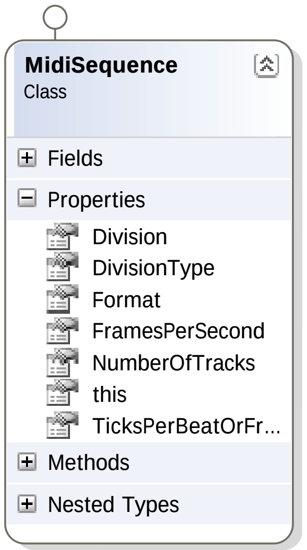
```
AutoSegment /i:"C:\inputFiles\" /o:"C:\outputFiles\"
```

Each MIDI file is processed individually and sequentially. While they could be processed concurrently, the system is predomanintly CPU bound and thus would only benefit from threading support on a multiprocessor system. If a MIDI file is processed successfully, an output file named the same as that MIDI file (though with a .txt extension) is output into the output directory. If any errors occurred during processing, error information is output to the console and no output file is rendered.

MIDI defines a structured file format made up of one or more tracks of one or more events. Each track is a linear sequence of events, meaning that every event has a delta time associated with it that is the time to wait since the previous event before processing should occur. If the delta time is 0, the event occurs at the same time as the previous event. Negative deltas are not possible. Positive delta times are interpreted based on data stored at the MIDI sequence level.

Every action derived from a MIDI file is implemented through an event, and thus there are several kinds of events. For example, a change of instrument is signaled through a program change event. The system starts playing a note in response to a note on event, and turns off a note in response to a note off event. Text can be stored in a MIDI file and triggered to display in response to text or lyric meta events. And so forth. For this project, I've implemented an object-oriented library for the parsing, manipulation, and outputting of MIDI files. This library is contained in Toub.Sound.Midi.dll.

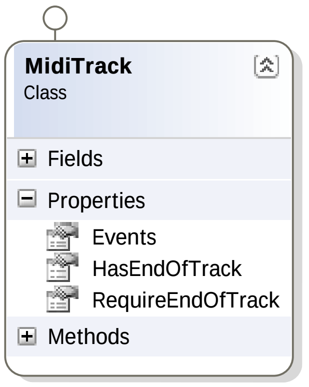
A MIDI file is parsed and represented by the Toub.Sound.Midi.MidiSequence class:



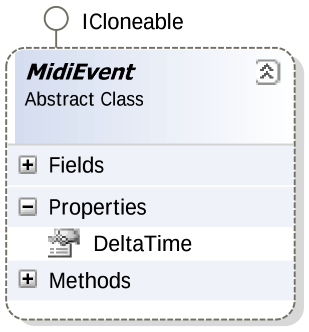
Given a MIDI file at C:\song.mid, it can be opened and parsed into a MidiSequence with the following code:

```
MidiSequence sequence = MidiSequence.Open("C:\\song.mid");
... \\ use strongly-typed sequence here
```

A sequence contains several pieces of information about the MIDI file as a whole, including its format (MIDI files can be in one of three formats, and these formats describe how events are distributed across tracks) and its division information (which controls the speed at which events are processed). Primarily, however, MidiSequence is a container for a collection of Toub.Sound.Midi.MidiTrack instances, an ordered collection of tracks that exist in the MIDI file:



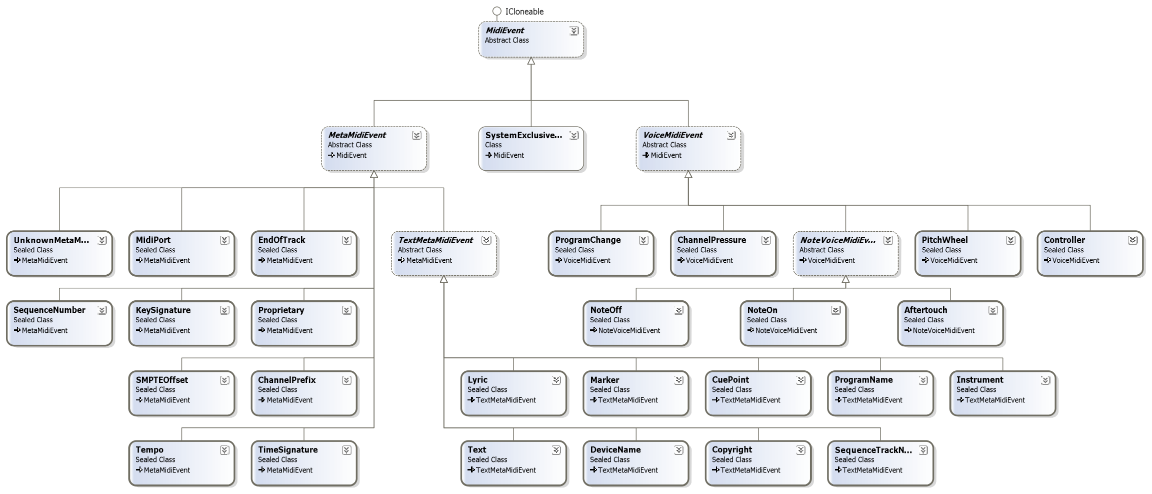
A MidiTrack serves primarily as a container for an ordered collection of the events that make up the track. Each event is represented by a type derived from the abstract type Toub.Sound.Midi.MidiEvent:



Notice that MidiEvent exposes a DeltaTime property: this is the only information common to every event, and thus it's the only information exposed from the base MidiEvent class. The following code shows how we output all delta times for all events in a MIDI file:

```
MidiSequence sequence = MidiSequence.Open("C:\\song.mid");
foreach(MidiTrack track in sequence)
{
      foreach(MidiEvent ev in track.Events)
      {
            Console.WriteLine(ev.DeltaTime);
      }
}
```

Concrete types for all kinds of events that can exist in a MIDI file have been implemented in a hierarchy of types all stemming from the base MidiEvent type:



ICloneable
MidiEvent Abstract Class
MetaMidiEvent Abstract Class → MidiEvent
SystemExclusive... Class → MidiEvent
VoiceMidiEvent Abstract Class → MidiEvent
UnknownMetaM... Sealed Class → MetaMidiEvent
MidiPort Sealed Class → MetaMidiEvent
EndOfTrack Sealed Class → MetaMidiEvent
TextMetaMidiEvent Abstract Class → MetaMidiEvent
ProgramChange Sealed Class → VoiceMidiEvent
ChannelPressure Sealed Class → VoiceMidiEvent
NoteVoiceMidiEv... Abstract Class → VoiceMidiEvent
PitchWheel Sealed Class → VoiceMidiEvent
Controller Sealed Class → VoiceMidiEvent
SequenceNumber Sealed Class → MetaMidiEvent
KeySignature Sealed Class → MetaMidiEvent
Proprietary Sealed Class → MetaMidiEvent
NoteOff Sealed Class → NoteVoiceMidiEvent
NoteOn Sealed Class → NoteVoiceMidiEvent
Aftertouch Sealed Class → NoteVoiceMidiEvent
SMPTEOffset Sealed Class → MetaMidiEvent
ChannelPrefix Sealed Class → MetaMidiEvent
Lyric Sealed Class → TextMetaMidiEvent
Marker Sealed Class → TextMetaMidiEvent
CuePoint Sealed Class → TextMetaMidiEvent
ProgramName Sealed Class → TextMetaMidiEvent
Instrument Sealed Class → TextMetaMidiEvent
Tempo Sealed Class → MetaMidiEvent
TimeSignature Sealed Class → MetaMidiEvent
Text Sealed Class → TextMetaMidiEvent
DeviceName Sealed Class → TextMetaMidiEvent
Copyright Sealed Class → TextMetaMidiEvent
SequenceTrackN... Sealed Class → TextMetaMidiEvent

MIDI events are split into three primary categories. SystemExclusive events are the least interesting, as they're proprietary events with ties to specific devices, and thus there are no specializations of the type. The rest of the events are split into two categories, MetaMidiEvent types and VoiceMidiEvent types. Types derived from MetaMidiEvent include KeySignature, Tempo, ChannelPrefix, TextMetaMidiEvent, and other types that provide meta information for the file. Types derived from VoiceMidiEvent include types such as ProgramChange and NoteVoiceMidiEvent (from which both NoteOn and NoteOff derive). So, if we wanted to list all pitches played in a MIDI file, the code might look like the following:

```
MidiSequence sequence = MidiSequence.Open("C:\\song.mid");
foreach(MidiTrack track in sequence)
{
      foreach(MidiEvent ev in track.Events)
      {
            NoteOn n = ev as NoteOn;
            if (n != null) Console.WriteLine(n.Note);
      }
}
```

The automatic extraction and segmentation system is built around this MIDI library.

Once a MIDI file is loaded into a MidiSequence, it goes through four stages of processing, three of which are implemented through a system of pluggable filters. The first stage deals with the extraction of the melody from a MIDI file. Since a MIDI file can contain an arbitrary number of tracks with an arbitrary number of events, and since there's no set format for how songs should be translated into MIDI files, given an arbitrary MIDI file there is no easy algorithm for extracting the events specific to the melody, ignoring all others. As such, the system allows for custom melody extraction filters to be implemented and easily plugged into the system (and combined with other extraction filters, if so desired). This makes it possible to easily try out new algorithms for extracting the melody from an arbitrary MIDI file. Implementing a custom melody extraction filter is done by deriving from the abstract MelodyFilter class:

```
public abstract class MelodyFilter
{
      public abstract MidiSequence Execute(
            MidiSequence sequence,
            Dictionary<string, object> context);
}
```

To be a melody filter, a class muss derive from MelodyFilter and implement the Execute method. Execute accepts two parameters, the MidiSequence to be processed and a state bag that can be used to pass arbitrary state between filters and out to the main program (most filters will simply ignore the context parameter). The Execute method then returns a new MidiSequence with fewer tracks and/or fewer events than the original sequence (a filter could return the exact sequence provided to it, but that would be a relatively useless filter, unless its purpose was for logging or something orthogonal to the task at hand). If a particular melody filter is the only one registered, the system expects that the returned MidiSequence contains only the events that comprise the melody. If there are multiple melody filters registered, in the order that they're registered the system passes the output MidiSequence from one filter as the input MidiSequence to the next filter. In this fashion, a chain of melody filters can be built up, each of which removes some portion of the events until the final filter outputs a MidiSequence containing just the melody events. For example, a melody filter that eliminates all percussion events from the sequence might look like the following:

```
class PercussionMelodyFilter : MelodyFilter
{
      public override MidiSequence Execute(
            MidiSequence sequence,
            Dictionary<string, object> context)
      {
            // Create a new sequence to store output
            MidiSequence newSequence = new MidiSequence(
                  sequence.Format, sequence.Division);

            // For each track
            foreach (MidiTrack track in sequence)
            {
                  // Create a new output track
                  MidiTrack newTrack = newSequence.AddTrack();
                  long timeSinceLastUsedEvent = 0;

                  // For each event
                  foreach (MidiEvent ev in track.Events)
                  {
                        // If it's a voice event
                        // and if it's on the percussion
                        // channel, ignore it.  Otherwise,
                        // store it to the output track.
                        VoiceMidiEvent voiceEvent =
                              ev as VoiceMidiEvent;
                        if (voiceEvent != null &&
                        voiceEvent.Channel ==
                        (byte)SpecialChannels.Percussion)
                        {
                              timeSinceLastUsedEvent +=
                                    ev.DeltaTime;
                        }
                        else
                        {
                              MidiEvent newEv = ev.Clone();
                              newEv.DeltaTime +=
                                    timeSinceLastUsedEvent;
                              newTrack.Events.Add(newEv);
                              timeSinceLastUsedEvent = 0;
                        }
                  }
            }
            return newSequence;
      }
}
```

This filter is then registered for use by the system by adding an entry for it to the appropriate section of the AutoSegment application's XML configuration file:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
      <configSections>
            <section name="AutoSegmenter"
                  type="AutoSegmenterConfigurationSectionHandler,
                        AutoSegmenter" />
      </configSections>

      <AutoSegmenter>
            <melodyFilters>
                  <filter type="PercussionMelodyFilter,
                                AutoSegmenter" />
            </melodyFilters>
            ...
      </AutoSegmenter>
</configuration>
```

The system dynamically finds and loads at runtime all filters registered in the configuration file, and it maintains them in an ordered collection based on the top-to-bottom order in which they were defined in the configuration file.

Once all melody filters have been run, the system takes the resulting MidiSequence and pulls out all remaining events into a MidiEventCollection, a ordered list of events that make up the melody (note, though, that there can be more than just NoteOn and NoteOff events in this collection). This MidiEventCollection is then provided to any registered segmentation filters for the next stage of processing.

Just as melody filters provide the core logic for extracting the melody from a MidiSequence, segmentation filters provide the core logic for parsing that sequence into a set of segments most likely to be hummed by a user using the query by humming system.

A segmentation filter must derive from the abstract SegmentationFilter class and implement its Execute method:

```
public abstract class SegmentationFilter
{
      public abstract void Execute(
            MidiSequence original,
            MidiEventCollection melody,
            double[] segmentationWeightings);
      public double Weight { ... }
      ...
}
```

Unlike melody filters, where the order of execution is important (since the output from one is fed as the input into the next), the order in which segmentation filters are executed doesn't matter. The Execute method is provided with three parameters: the original MidiSequence (for reference purposes only), the collection of melody events, and an array of floating point numbers to be filled in by the filter. In essence, a segmentation filter works by voting on how likely each event in the melody is to be a segment boundary. The segmentationWeightings array passed to the Execute method initially contains one entry for every event, each of which is initialized to zero. The filter can then do whatever processing is necessary and set various weights into the array. For example, a segmentation filter processing a common-time (4/4) song that segmented based on measure breaks might set a value of 0.5 for every event that starts a measure, and a value of 1.0 for every event that starts every fourth measure; every other event would remain at 0. After filters run, the output weightings from each filter are normalized by the system into a [0.0, 1.0] range. Thus, the actual values used by a filter are unimportant; the important thing is the relative values between all weights supplied. The outputs from a filter are then multipled by a predetermined weight per filter (as supplied in the configuration file) before being combined with the outputs from all other registered segmentation filters. So, for example, a user might choose to specify that a meter-based segmentation filter is half as important as a repetition-based segmentation filter, and thus the meter filter would be given a weight of .5 and the repetition filter given a weight of 1.0. Once all filter outputs have been weighted and combined, the system again normalizes the values into a [0.0, 1.0] range and creates segments from the weightings based on a preconfigured threshold. So, for example, all combined and normalized weightings greater than 0.5 could be treated as a segment boundary, causing the melody events to be split into segments on all events with a weighting of 0.5 or greater. Segmentation filters are registered in the application's configuration file in a manner similar to melody filters (note the additional "weight" attribute that configures the relative important of this filter; if no weight is specified, it defaults to 1):

```
<segmentationFilters>
      <filter weight="1" type="KaraokeTextSegmentationFilter,
                              AutoSegmenter"  />
</segmentationFilters>
```

With the melody event collection and the segment boundary information in hand, the system proceeds to extract the actual notes and segments from the collected information.

It loops through all events in order searching for NoteOn events that have a Velocity greater than 0 (from my testing, it appears that many MIDI files use NoteOn events with a Velocity equal to 0 to represent note off events instead of using true NoteOff events). When a note is found, it is added to the current segment, and the previous note (if there is one) is ended (this results in notes absorbing any rests in the file, which for our purposes is fine). When a segment boundary is reached, the current segment is completed and added to the list of segments, and the current note is added to a new segment. If a segment boundary is found before a note is completed (i.e. after seeing its note on event but before seeing its note off event), the segment boundary is moved until after the completion of the note. In addition, if multiple notes are found to start at the same point in time, the higher pitch note is selected and the lower pitch note ignored.

At this point, we now have a list of all notes in the melody, and a list of segments that index into that list of notes. Each note is represented as an integral pair: a MIDI pitch value which ranges between 0 and 128, and a duration. Durations can optionally be normalized such that the shortest length note has a duration of 1; all other note's durations are modified to retain the same relative length.

The third step in the process is configurable, but does not involve plug-in filters. Short segments can optionally be combined into new, longer segments that are added to the list of segments but that do not replace the shorter segments. For example, consider four sequential segments each comprised of 5 notes, and a system configured to combine segments shorter than 15 notes. In this case, we start with four segments:

```
1 to 5
6 to 10
10 to 15
15 to 20
```

and we end up with 9 segments:

```
1 to 5
1 to 10
1 to 15
6 to 10
6 to 15
6 to 20
10 to 15
10 to 20
15 to 20
```

This allows for segmentation filters to be aggressive in generating smaller segments, while still allowing users to hum larger segments. Taken to an extreme, this would allow a special segmentation filter to generate a segment for every note. The segments would then be combined until a configured maximum length, in a sense allowing the matching system to support partial-sequence matching, since every possible subsequence from the song's notes (up to a predefined length limit) would be in the database of segments.

Of course, the more segments there are in the database, the slower the system will operate, and potentially the less accurate the system will be, resulting in slower results and more false positive matches. As such, the last stage in the process is the filtering and ranking of segments. This stage also supports pluggable filters. Filters for this stage derive from the abstract SegmentPrioritizationFilter class and implement its Prioritize method.

```
      public abstract class SegmentPrioritizationFilter
      {
            public abstract void Prioritize(
                  MelodyWithSegments mws,
                  double[] priorities,
                  bool[] remove);
      }
```

Prioritize accepts three parameters: a MelodyWithSegments, which contains a list of the notes and a list of the segments that index into that list of notes, an array of floating-point values representing the priority of each segment, and an array of Booleans that dictate whether each segment should remain in the list. The filter is provided with all three arguments, and based on the first it sets values in the latter two. Smaller priority values signify more importance. If the filter wants to a remove a segment from the list, it simply sets the corresponding item in the remove array to true, and the system will remove the segment from the list before outputting them. Like segmentation filters, for determining priorities the order in which segment prioritization filters are run does not matter, as the system sums up the priorities from each filter and ranks them based on the totals. However, for determining segment removal, the order does matter, as a filter can choose to override the removal of a segment as registered by a previous filter by changing the value in the array back to false (though it’s difficult to come up with a scenario where this is useful or a good idea). Segment prioritization filters are registered in the application’s configuration file in a manner very much like other types of filters:

```
      <prioritizationFilters>
            <filter type="SimilaritySegmentPrioritizationFilter,
                        AutoSegmenter" />
      </prioritizationFilters>
```

At this point, all stages of processing are complete, and we’ve (hopefully) successfully extracted the melody and computed the segments and their ranking from the MIDI file. Metadata is extracted from the MIDI file as best as possible (this includes title information and lyrics), and all information is written out to a single output file. It is this file that can then be imported in whatever way appropriate into the song database. Here is an example output for a song (“Africa”, by Toto); I’ve ommitted pieces of it (represented with the elipses) for documentation’s sake:

```
      66, 1
      66, 1
      66, 1
      66, 3
      66, 1
      66, 2
      68, 2
```

```
70, 1
71, 6
...
81, 1
80, 2
80, 5
78, 1
76, 2
80, 1
81, 1
80, 2
78, 1

94, 105
121, 129
106, 120
113, 120
121, 135
106, 112
113, 129
317, 334
136, 149
...
94, 120
17, 38
33, 55
56, 79
48, 71
9, 32
0, 23
150, 173
72, 93
80, 105
157, 181
121, 149
274, 301
201, 229
182, 208
39, 65
174, 200
164, 190
191, 217
66, 93
260, 289
```

AFRICA

I hear the drums echoing tonight.
She hears only whispers of some
quiet conversation.
She is coming in twelve thirty flight.
Moonlit wings reflect the stars
that guide me towards salvation.
I stopped an old man along the way
hoping to find some long forgotten words
or ancient melodies.
He turned to me as if to say,

```
"Hurry boy, it's waiting there for you."
It's gonna take a lot to drag me away from you.
There's nothing that a hundred men
or more could ever do.
I bless the rains down in Africa.
Gonna take some time to do
the things we never had.
Wild dogs cry out in the night
as they grow restless longing
for some solitary company.
I know that I must do what's right
sure as Kilimanjaro rises
like Olympus above the Serengeti.
I seem to cure what's deep inside,
frightened of this thing that I've become.
It's gonna take a lot to drag me away from you.
There's nothing that a hundred men
or more could ever do.
I bless the rains down in Africa.
Gonna take some time to do
the things we never had.
"Hurry boy, she is waiting there for you."
It's gonna take a lot to drag me away from you.
There's nothing that a hundred men
or more could ever do.
I bless the rains down in Africa.
I bless the rains down in Africa.
I bless the rains down in Africa.
I bless the rains down in Africa.
I bless the rains down in Africa.
Gonna take some time to do
the things we never had.
```

The first section contains the notes for the song; each line represents one note, with the first value representing the pitch and the second the relative duration. The pitch values are the same values the MIDI specification uses to define pitch values. There are standard formulas for converting between MIDI pitch values and frequencies.

The second section (after the first blank line) contains the segments. Each line represents a single segment, where the first value is the starting index into the notes list (0-based index) and the second value is the ending index into the notes list (inclusive). The order of the segments in the file is their priority order, so the first segment is the most important, the second segment is the second most important, etc.

After the next blank line following the segments is the name of the song. Depending on how the MIDI file was constructed, the “name” might include additional meta information, such as the composer or band. And if available, following a blank line following the title line are the lyrics from the song.

This whole process is repeated for every input MIDI file, resulting in one of these output files for every input MIDI file successfully processed.

Implemented Filters

In addition to the overall framework, several concrete melody, segmentation, and segment prioritization filters have been implemented.

ChannelMelodyFilter – This filter was implemented under the assumption that a particular channel contains all of the events for the melody. As such, it extracts all events from a channel, regardless of the containing track, into the output sequence. The specific channel to extract can be specified in the configuration file, or it can be selected automatically by the filter. The automatic selection can only be used for MIDI files that contain Lyric events, which based on my examination of a large quantity of MIDI files on the internet, seems to be somewhere in the vicinity of 10% of MIDI files available. The automatic selection is based on the premise that the channel containing the melody will have NoteOn events close to Lyric events, and that Lyric events will be close to NoteOn events in that channel. While these two concepts may sound the same, they're not. As such, the filter looks for all Lyric events. For each one found, it accumulates the distance to the nearest NoteOn event in each channel that contains NoteOn events. The filter then finds all NoteOn events, and accumulates the distance for each to the nearest lyric. After this process, the channel that contains NoteOn events and that has the smallest accumulated distance is ruled the melody channel by the automatic selection engine. This automatic selection is based on the premise that Lyric events should occur simultaneouosly or at least in close proximity to notes, since the lyrics must be vocalized with notes. It also presumes that few notes in the melody will be sung unaccompanied by lyrics. This is why both directions (lyrics to notes, and notes to lyrics) are used in determining the channel. While this filter works decently for MIDI files in which the melody is contained in a single channel and in which that channel houses only melody events, this pattern unfortunately isn't the case for a significant number of MIDI files available on the Web.

EntropyMelodyFilter – This filter was implemented under the assumption that most background material in a song is predictable. Guitars frequently strum the same chord progression over and over. Percussion is repetitious. Etc. Specifically, this filter assumes that the track containing the melody is the least predictable of all tracks. As such, this filter loops over all events in each track, maintaining a form of Markov chain that can be used to determine how likely a note is to occur based on the previous series of notes found. The predictabilities for all notes in a track are summed, and the track with the least predictability is returns as the track containing the melody. Unfortunately, the results from this filter are currently disappointing, rarely producing the right answer. With some additional work and logic added, it's possible it could be improved, but for now it proves relatively useless.

PercussionMelodyFilter – This filter is meant to be used in a chain with other filters, and simply filters out all percussion events. Channel 10 (1-based) in a MIDI file is a special channel that contains only percussion events, and thus percussion is easy to isolate. The code for this filter was shown earlier in this report. A segmentation version of this filter that I did not have time to implement, but that could prove fruitful, would be a rhythmic analyzer for the percussion. Significant events in popular songs often have significant changes in the accompanying percussion as a precursor, so such changes could be found and used to weight those events immediately following the changes.

FlattenToOneTrackFilter – This filter does exactly what its name implies, combining all events from all tracks into a single output track (this, in effect, converts format 1 and format 2 MIDI files into format 0). Delta times for all events are recomputed to allow for this merging. Like the percussion filter, this filter is not meant to be used alone, but can instead be used early on in a sequence of filters in order to allow for easier implementation of filters later in the sequence.

OnlyLyricTrackFilter – This filter removes any tracks that don't have lyrics, the idea being that MIDI files with lyrics typically have those lyrics in the same track as the melody events. Unlike the previous two filters, this filter can be used by itself, assuming that the track containing the lyrics and the melody only contains events for the lyrics and melody. Unfortunately, given the wide variety in construction of MIDI files available for consumption, this filter may not be very useful (though for cases where the construction matches the assumptions of this filter, it's very useful and accurate).

NotesForLyricsFilter – This filter is also based on lyrics, accept it doesn't make any assumptions about tracks or channels. Instead, it finds the NoteOn event closest to each lyric and extracts those notes as those composing the melody. For many MIDI files containing lyrics, this filter has good success.

KaraokeMelodyFilter – A file format derived from MIDI is the Karaoke file format. This file format, signified by a .kar extension, follows all the file format specifications for MIDI files, but in addition typically follows a much more stringent pattern of construction, making it possible for Karaoke machines to correctly render the containing lyrics along with a bouncing ball for the associated melodic notes. Given this, an automatic system is able to much more easily extract the melody and to do so with higher accuracy from Karaoke files than from MIDI files. In most Karaoke files, the name of the track containing the melody has a name beginning with either "vocal" or "melody" and the track containing the lyrics has a name beginning with "word" or "lyrics". This filter looks for tracks with a SequenceTrackName event beginning with those phrases, and pulls one of each out into the returned sequence. This is almost always the correct answer, and when provided with a valid .kar file, this filter has a very high success rate.

CueMarkerSegmentationFilter – MIDI files can contain special meta events describing what's going on in the file. These typically come in the form of cues or markers, represented in the Toub.Sound.Midi.dll library as CuePoint and Marker events. Frequently, these events signal the start of something interesting in the MIDI file, such as a new section or the chorus or the bridge. The start of anything interesting should also be the start of a segment, so this filter adds weight to notes in the melody that coincide with any CuePoint and Marker events found in any track in the original file. The filter does the same for other events that frequently occur at the start of something interesting, including Tempo, TimeSignature, KeySignature, Instrument, and ProgramChange. This filter is best when combined with other filters, as most MIDI/Karaoke files use these sparingly, if at all.

MeterSegmentationFilter – This filter adds weight to the first note in each measure. As with CueMarkerSegmentationFilter, it's best used in conjunction with other filters, although for many pop songs it produces relatively decent results by itself as long as the "combine segments" option is enabled.

RestSegmentationFilter – This filter operates under the assumption that long pauses signify a significant moment in a song, and thus are a good place to insert a segment boundary. As such, the filter catalogs all rests, finds the average rest length and the standard deviation, and then inserts segment boundaries at all rests greater than half a standard deviation above the average (a value selected based on subjective testing). This filter produces relatively decent results, on par with those produced by the MeterSegmentationFilter.

RepetitionSegmentationFilter – This filter operates under the assumption that people remember sections (and thus hum sections) that contain sequences of notes that repeatedly appear in a song. This filter looks at every sub-sequence of notes in the melody, and keeps track of how many times each sequence of pitch values occur. Every hit on every sub-sequence of a minimum length (5 notes by default) is tallied. For every sub-sequence that occurs at least a minimum number of times (also 5 by default), tallies are added to the starting and ending events of the sub-sequence. After all tallies have been added, the average and standard deviation of these tallies is computed, and final weights are set for those events with a tally above a standard deviation above the mean. Like the MeterSegmentationFilter and the RestSegmentationFilter, for certain subsets of MIDI files, this filter performs very well, but for others it performs poorly.

KaraokeTextSegmentationFilter – This filter is currently the best implemented segmentation filter, and it works for a large subset of Karaoke files (though not for standard MIDI files). The filter works simply by inserting segment boundaries at the beginning of lines in the lyrics, and optionally at punctuation marks in the lyrics. This produces very good segments, as they correspond to sections of songs that are typically remembered as a unit (individual phrases or segments of lyrics).

SimilaritySegmentPrioritizationFilter – Like the RepetitionSegmentationFilter, this prioritization filter is based on the premise that frequently repeated sections are most easily remembered. As such, this filter ranks segments based on how similar they are to the other segments. Similarity is judged based on an edit distance algorithm between the sequence of notes in each segment. Every segment is compared to every other segment, and a total distance is kept for each segment. Segments are then ranked from low to high based on this distance, where the lower the distance the earlier the segment is in the final list. In addition, the filter removes duplicate segments, where the edit distance between two segments is 0. This frequently happens when a segmentation filter pulls out the same sequence from two different verses or occurrences of the chorus in a song.

Thus far, the system produces the best results when supplied with .kar files and when using the KaraokyMelodyFilter to extract the events composing the melody, the KaraokeTextSegmentationFilter to create segments from these events based on the lyrics in the Karaoke file, and the SimilaritySegmentPrioritizationFilter to rank the segments. Unfortunately, with the current filters implemented, good results from MIDI files (as opposed to the more constrained Karaoke files) are produced only for a medium-sized subset of the MIDI files I've obtained from the internet, and a manual step is necessary in order to verify the output for each file, thus nullifying some of the value of this system. For Karaoke files, however, I believe the system does a good job of extracting the melody and producing viable segments for a large number of the Karaoke files I was able to obtain from the internet. Further work could involve implementing and testing more filters. Algorithms like adaboost could also be used to better combine the results from segmentation filters.