Optimal guitar tablature with dynamic programming

Ben Sherman

May 1, 2013

I have created a Haskell module that, given a MIDI file containing a guitar piece, and a guitar `Tuning`, produces tablature (`Tab`) for that performance.

1 Overview

The program works by taking as input a Euterpea `Performance`, from which it identifies all the notes (and their start and end points) that are played in a given piece. It then considers all the times at which notes are struck (i.e. the start times of the notes). The program contains a large, procedurally generated list, of size N (currently $N \approx 10,000$), of possible hand positions for fretting a guitar. At each time when new notes are struck, the program determines which positions would be enable the guitarist to strike the new notes, while holding the strings/notes that were previously struck and must continue to be held. Let us represent the set of all fretting positions as P. There are two scoring functions; the individual scoring function, $S_I : P \to (-\infty, 0]$, which rates how comfortable a position is individually, and the transition scoring function, $S_T : P \times P \to (-\infty, 0]$ which rates the difficulty of transitioning from one position to another. Let us represent a guitar tablature by a set of fretting positions $\{p_1, \ldots, p_n\} \subseteq P$ indexed according to the order in which they are played. Then a guitar tablature is assigned a score based on the function

$$S(\{p_1, \ldots, p_n\}) = \sum_{i=1}^{n} S_I(p_i) + \sum_{i=1}^{n-1} \omega(\Delta t_i) S_T(p_i, p_{i+1}),$$

where for each i, Δt_i is the time between the transition from position p_i to p_{i+1}, and where $\omega : [0, \infty) \to [0, \infty)$ is weighting function that more heavily weights transitions that occur in shorter time periods (i.e., it is monotonically decreasing).

To produce "optimal" guitar tablature, we would like to find the tablature that maximizes this scoring function out of all the possible tablatures that would satisfy the constraints (such as playing all the notes, not hitting strings that were previously struck, etc.). However, it would be computationally infeasible to try all the possibilities. For a piece of "size" n, where n is the number of fretting positions in the piece, the scoring function is clearly $O(n)$. However, we could have many as N^n tablatures for such a piece, requiring $O(N^n)$ time to compute the optimal tablature in this way. This combinatorial explosion means that finding an optimal tablature could be very hard.

1.1 Algorithms

It is interesting to note that the entire problem has a "semi-local" nature. The constraint of not hitting a previously struck note is fairly local, because notes are generally only held for short periods of time, and the other constraints are completely local. The scoring function, too, is semi-local, since only individual positions and pairs of neighboring positions are considered. If the problem were completely local, we could determine an optimal fretting position for each point in the piece, and aggregate these together to create an optimal tablature, and then the problem would be $O(nN)$. The problem is also very close to having an "optimal substructure."

If the constraint of not hitting previously struck notes were absent, the problem could be solved by dynamic programming. Suppose we choose to divide the piece into an earlier part and the later part. Then there is only a single "interacting term" that contributes to the score, between the last hand position of the earlier part, and the first fretting position of the later part. Thus, if we know all the optimal tablatures for the earlier part, given a particular end position (at most N tablatures), and if we know all the optimal tabs for the later part, given a certain start position (at most N tablatures), we can piece together an optimal tablature in $O(N^2)$ time.

This yields a strategy for recursively breaking down the optimal tablature problem into smaller subproblems. In the "base case" of a single position, there is no optimization to be done (we simply take all possible positions). However, note that in general (if we're not at the start or end of the piece), we need to keep track of both the start and end positions for the subproblems. This means that we must keep up to N^2 tablatures for each subproblem, and then the merge operation would take $O(N^4)$ time (for each possible start and end, there are up to N^2 tablatures to check, and there are up to N^2 different possible start/end combinations). For a tablature of size n, if we choose to build a balanced binary tree to compute this problem, the problem ends up taking approximately $O(nN^4)$ time.

But we've noted that merging subproblems is easier if either of the subproblems at the start or end, since this means have fewer interaction terms to consider when merging, and so we have less to keep track of. If we simply build the tablature in a linear fashion forwards or backwards, then each time we merge a subproblem, the "outer" one (the one including the start or end) has at most N different tablatures, while the inner has at most N^2 tablatures, and so the merging operation takes $O(N^3)$ time. We still perform $n-1$ merging operations, giving a complexity for the entire algorithm of $O(nN^3)$.

In reality, however, we cannot ignore the constraint that previously struck notes that must be held must not be hit again! My solution to this problem was simply to redefine the base case of the dynamic programming algorithm described above. Instead of letting the base case be a single position, we can partition a guitar performance into "chunks" where each note falls entirely in a single chunk. That is, we partition notes by defining a graph between notes that are played, giving edges between two notes if there is some time at which both notes are audible, and then we let the base cases be the connected components of the graph.

These chunks are solved by brute force, i.e., checking all possible tablatures for the component. This is unfortunate, because this takes time $O(N^\ell)$ for a chunks of size ℓ. The saving grace, of course, is that the possible fretting positions are more constrained by these constraints (i.e.., if a hand position must keep a finger held and a string played, there's going to be very few possibilities for that position), so the "combinatorial explosion" might not be so severe. Also, in most guitar performances I have examined, chunks are often contain a single hand position (i.e., a single time at which notes are struck). Usually, notes are not held across measures, and so chunks are almost always shorter than measures. So if chunks are always of size not greater than ℓ, and there are n distinct chunks, the running time of this algorithm is $O(nN^\ell)$. This algorithm is implemented as the `dynamic` function in the `TabGeneration` module. The "base case" method of producing brute-force solutions to a given chunk (with all possible start and endpoints) is implemented as the `listForChunk` function, while the merging method is implemented by the function `merge`. The merging method utilizes the parallelism afforded by Haskell's `Control.Parallel` module, and could benefit from parallelism of having up to N cores.

However, it could be technically possible for an entire piece to be a single chunk, which would render this method computationally infeasible. For the situations where the chunks are very large, there are algorithms that produce tablature that is not necessarily optimal. For example, the `selfishTab` and `smartFTab` functions in the `TabGeneration` module build tablature by building it forwards in time. `selfishTab` chooses each successive fretting position by simply taking the position that satisfies the constraints (including notes that must be held from previous positions) with the highest individual score. `smartFTab` also considers the "interaction term" between the latest fretting position and the next one to be added.

However, these solutions are not optimal. I have also built an algorithm that searches for optimal solutions using simulated annealing. The function `mcmcAll` in the `SimulatedAnnealing` module takes as input a tablature that has already been produced, the tuning, a function that gives the "inverse temperature" for each step, and a random seed. It then returns an infinite list of tablatures that it generates according to a search algorithm which is something like a Monte Carlo Markov chain. The algorithm randomly mutates the hand position at a certain point in the tab to another suitable fretting position. It favors tabs that would result in a better score for the overall tablature according to a sort of Boltzmann distribution. That is, the probability of changing to a tab ρ is

$$\Pr(\rho) \propto e^{\beta S(\rho)},$$

where β is the inverse temperature, and $S(\rho)$ is the tab's score. Thus if β is low, the mutation is picked from an almost uniform distribution, while if β is high, favorable changes are much more likely to be chosen. In reality, for each proposed change in fretting position to the tab, only a few of the pieces of the tab's score need to be recomputed, and so computing the change in score is $O(1)$. Since we check at most N positions for each mutation, each step of the algorithm is $O(N)$.

Because the score of a tablature is heavily dependent on local effects, it's beneficial for the algorithm to change nearby positions at nearby times. Therefore, the algorithm keeps track of which position was changed last, and it changes the next or previous position in the next tab (each with probability 1/2). For optimal results in simulated annealing, we wish to have a temperature profile that "heats" the tab some of the time, so that it can escape local minima, and that "cools" the tab at other times, so that it can settle into a favorable position.

2 Features, Usage, and Examples

The main module is called `Main`, and so to use this module in GHCi, simply load that module. To compile the module, I have been using the following command, which allows it to run using multiple CPU cores:

```
ghc -O2 --make Main -threaded -rtsopts
```

2.1 Generating tablature

There are four main functions for generating tablature. Three of them generate tablature *de novo*:

```
dynamic, selfishTab, smartFTab :: Tuning -> Performance -> Tab
```

Each of these have been described in the section on algorithms. Notably, there is a list of possible tunings in the `Tuning` module. I have reproduced that list in a readable format in Table 1.

Table 1: Tunings available by default

Variable name	Command line name	Tuning (low string to high string)
`stdTuning`	Standard	E A D G B E
`halfStepDown`	HalfStepDown	D♯G♯C♯F♯A♯D♯
`fullStepDown`	FullStepDown	D G C F A D
`dropDTuning`	DropD	D A D G B E
`doubleDropD`	DoubleDropD	D A D G B D
`openG6Tuning`	OpenG6	D G D G B E
`openCTuning`	OpenC	C G C G C E
`dadgadTuning`	DADGAD	D A D G A D

The fourth main function that generates tablature is the simulated annealing algorithm, which produces an infinite list of `SAState`s; each `SAState` includes a tablature and other information necessary for continuing the simulated annealing.

```
mcmcAll :: Tab -> Tuning -> (Int -> Double) -> StdGen -> [SAState]
mcmcBest :: Int -> [SAState] -> Tab
```

`mcmcAll` takes an input tablature, the tuning that it's in, the temperature profile (temperature at each step), and a random number generator, and produces the infinite list of `SAStates`. The `mcmcBest n` function looks at the first n elements of the list and returns the tab with the best score.

2.2 Outputting Tablature

These two functions are the most important for outputting tablature:

```
annotatedTab ::  Int -> PTime -> PTime -> Tab -> IO ()
tabPositions :: Tab -> IO ()
```

`annotatedTab n acc meas tab` outputs the tablature `tab` in an ASCII format. `acc` gives the amount of time (in seconds) that should be covered by each "unit" in the tab. So higher values of acc will make the tab more condensed, while lower values will make the tab more spread out, and position the notes more accurately in time. `meas` indicates amount of time in a measure, and `n` gives the number of measures to output on the same line in the tab.

`tabPositions` outputs a rough visualization of the hand position for each position played in the tablature, showing a guitar string where numbers indicate pressed frets. Each number gives the number of the finger, e.g., "2" represents the index finger.

2.3 Command line usage

From the command line, the `Main` module allows anyone to convenient generate tablature for a given MIDI file. The usage is as such:

```
./Main filename tuning n acc meas
```

This program will open the MIDI file at `filename`, and will use the `dynamic` function to create a tablature for the piece using the tuning specified in `tuning` (see Table 1 for possible values). It will then output the tablature using `tabOutput`, where `n` is the number of measures on a single line, `meas` is the duration of a measure, and `acc` is the amount of time in a single "unit" of the tab. Most `meas` and `acc` must be input to the command line in the form of rationals, i.e. `m%n` indicates $\frac{m}{n}$ seconds.

2.4 Functions for convenience

The function `midi :: String -> IO Music1` loads a MIDI file in to a Music1 data structure. The function `defPerform1 :: Music1 -> Performance` creates a performance for such a music file. The function `midiToTab :: String -> Tuning -> IO Tab` loads a MIDI file given the filename, and creates a tablature for the piece with the given tuning (using the `dynamic` algorithm).

2.5 Examples

I have included some MIDI files to serve as examples. I am using a dual core computer so I have the parameter "-N2" set, but it should be changed to reflect the number of cores on the machine. For example, running

```
./Main midi/mist.mid DropD 3 1%8 3%1 +RTS -N2
```

will output the tab which is shown in Figure 1. You can also return the tablature for "Mist" in GHCi using the function `mistTab :: IO Tab`. When compiled (and with the cutoffs I have chosen), the dynamic programming algorithm takes about 22 seconds of "real time" (and 40 seconds of CPU core-time) on my computer to generate the tablature for "Mist." It does run almost twice as quickly using two cores as using a single core.

I also have some other examples of working tablature.

```
./Main midi/cove.mid Standard 4 3%20 24%10 +RTS -N2
./Main midi/canon.mid DoubleDropD 2 1%8 4%1 +RTS -N2
./Main midi/im_yours.mid OpenG6 4 1%10 16%10 +RTS -N2
./Main midi/im_yours2.mid Standard 2 15%146 240%73 +RTS -N2
```

2.6 Parameters

Unfortunately, the number of possible fretting positions N is very large, and so actually computing all possibilities for either the dynamic programming algorithm or for the simulated annealing is very computationally expensive. There are several cutoff parameters in the module `Parameters` which limit the computations that are done, usually simply by only investigating a subset of the possible fretting positions (always the subset with the highest individual scores).

There are also a few parameters that affect the scoring of tablature in the `Parameters` module, which essentially allows you to set the time-varying transition weighting function ω that was mentioned earlier.

3 Implementation Details

This section describes some implementation details, particularly with regard to how the information is represented in data structures.

The module makes rather heavy use of the `Data.Map` module, which is imported in the namespace `M`:

```
import qualified Data.Map as M
```

The strings of the guitar are represented by the datatype `GString`,

```
data GString = S1 | S2 | S3 | S4 | S5 | S6
```

Each possible fretting-hand conformation is represented as a `Position`:

```
type Position = M.Map Finger Fingering
data Finger = F1 | F2 | F3 | F4 | F5
type Fret = Int
data Fingering = Bar GString GString Fret | On GString Fret
```

Thus a `Position` gives the fingering for each finger that presses on guitar frets in that hand conformation. `Bar s1 s2 f` means barring the fretboard from string `s1` to `s2` at fret `f`, while `On s f` simply means pressing string `s` at fret `f`.

I have tried to enumerate most of the common positions in the list `allPositions`. To facilitate this, I created two functions:

```
cross :: [Position] -> [Position] -> [Position]
copysOf :: Position -> [Position]
```

```
|------------------------------------------------|------------------------------------------------|------------------------------------------------|
|-----1-3---1------------------------------------|-----1-3---1-------------------------------1----|-----1-3---1------------------------------------|
|-----0-0-----2---0-----0-2-0--------------------|-----0-0-----2---0-----0-2-0---0-2---2-0-------0|-----0-0-----2---0-----0-2-0--------------------|
|-----0-0-----------3---------3-2-0---0-2-------0|-----0-0-----------3---------0-----0------------|-----0-0-----------3---------3-2-0---0-2-------0|
|-----------------------------------0-----0-3----|------------------------------------------------|-----------------------------------0-----0-3----|
|-0-----------------------0----------------------|-0-----------------------0---------------0------|-0-----------------------0----------------------|

...

|-----0-1-0---0-1-0-----------0-1-0---0----------|-----0-1-0---0-1-0-8---------0-1-0---0-----5----|-----0-1-0---0-1-0-1-0-3-1-0---0-1-0-1-0-------8|
|-3-----------------3-----3----------------------|---3-----------------------3--------------------|-3----------------------------------------------|
|-5---------0-------------5---------0------------|-5---------0-------------5---------0------------|-5---------0-----------------0-----------0-7----|
|------------------------------------------------|------------------------------------------------|------------------------------------------------|
|------------------------------------------------|------------------------------------------------|------------------------------------------------|
|-0-----------0-----------0-----------0----------|-0-----------0-----------0-----------0----------|-0-----------0-----------0-----------0-----0----|

|10---8-5---3-5-3---3-5-3-1---3-0-------0-------8|10---8-5---3-5-3---3-5-3-5-3-------------------8|10---8-5---3-5-3---3-5-3-1---3-0-------0-------8|
|-------------------------------------3----------|-------------------------------810-810---810----|-------------------------------------3----------|
|-----------------0---------------0-0-----0------|-----------------0-----------0------------------|-----------------0---------------0-0-----0------|
|------------------------------------------10----|------------------------------------------------|------------------------------------------10----|
|------------------------------------------------|------------------------------------------------|------------------------------------------------|
|-0-----------0-----------0-----------0----------|-0-----------0-----------0-----------0----------|-0-----------0-----------0-----------0----------|

...

|-0-0-0-0-----------0-0-0-0-0-0-0-----------0-0-0|-0-0-0-0-----------0-0-0------------------------|-0-0-0-0-----------0-0-0-0-0-0-0-----------0-0-0|
|-0-0-0-0-----------0-0-0-0-0-0-0-----------0-0-0|-0-0-0-0-----------0-0-0-0-0-0-0-0-0-0-0-0-0-0-0|-0-0-0-0-----------0-0-0-0-0-0-0-----------0-0-0|
|-2-2-2-2-----------2-2-2------------------------|-5-5-5-5-----------5-5-5-------------0-0-0-0-0-0|-2-2-2-2-----------2-2-2------------------------|
|-0-0-0-0---0-0-0-0-0-0-0-0-0-0-0---0-0-0-0-0-0-0|-0-0-0-0---0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0|-0-0-0-0---0-0-0-0-0-0-0-0-0-0-0---0-0-0-0-0-0-0|
|-----------0-0-0-0-----------------0-0-0-0------|-----------0-0-0-0------------------------------|-----------0-0-0-0-----------------0-0-0-0------|
|-----------0-0-0-0-----------------0-0-0-0------|-----------0-0-0-0------------------------------|-----------0-0-0-0-----------------0-0-0-0------|

|-0-0-0-0-----------0-0-0------------------------|-3-3-3-3-----------3-3-3-3-3-3-3-----------3-3-3|-3-3-3-3-----------3-3-3-3-3-3-3-3-3------------|
|-0-0-0-0-----------0-0-0-0-0-0-0-0-0-0-0-0-0-0-0|-0-0-0-0-----------0-0-0-0-0-0-0-----------0-0-0|-0-0-0-0-----------0-0-0-0-0-0-0-0-0-0-0-0-0-0-0|
|-5-5-5-5-----------5-5-5-------------0-0-0-0-0-0|-2-2-2-2-----------2-2-2------------------------|-5-5-5-5-----------5-5-5-------------0-0-0-0-0-0|
|-0-0-0-0---0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0|-0-0-0-0---0-0-0-0-0-0-0-0-0-0-0---0-0-0-0-0-0-0|-0-0-0-0---0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0-0|
|-----------0-0-0-0------------------------------|-----------0-0-0-0-----------------0-0-0-0------|-----------0-0-0-0-----------------------0-0-0-0|
|-----------0-0-0-0------------------------------|-----------0-0-0-0-----------------0-0-0-0------|-----------0-0-0-0---------------------------0-0|

...

|------------------------------------------------|------------------------------------------------|------------------------------------------------|
|-----1-3---1-------------------------------3----|-----1-3---1------------------------------------|------------------------------------------------|
|-----0-0-----2---0-----0-2-0---0-2---2-0-------0|-----0-0-----2---0-----0-2-0--------------------|-------------------------------------------0----|
|-----0-0-----------3---------0-----0------------|-----0-0-----------3---------3-2-0---0-2-------0|---0-0-2-0---0-2-0---------0-0-2-0---0-2-0------|
|------------------------------------------------|-----------------------------------0-----0-3----|-3---------0-------3-----3---------0------------|
|-0-----------------------0---------------0------|-0-----------------------0----------------------|------------------------------------------------|

|------------------------------------------------|------------------------------------------------|------------------------------------------------|
|------------------------------------------------|------------------------------------------------|------------------------------------------------|
|------------------------------------------------|-------------------------------------------0----|------------------------------------------------|
|---0-0-2-0---0-2---3-2-0---0-0-2-0---0-2-0------|---0-0-2-0---0-2-0---------0-0-2-0---0-2-0------|---0-0-2-0---0-2---3-2-0---0-0-2-0---0-2-0-0----|
|-3---------0-----0-------3---------0-------3----|-3---------0-------3-----3---------0------------|-3---------0-----0-------3---------0-------0----|
|------------------------------------------------|------------------------------------------------|-------------------------------------------0----|
```

Figure 1: Excerpts from the sample tablature generated by dynamic programming for John Butler Trio's "Mist."

`cross` implements a sort of Cartesian product: given lists of positions `as` and `bs`, it returns the set of all positions where the fingers press a combination of an `a` from `as` and a `b` from `bs`. The `copysOf` function simply gives all of the translates of the position up and down the fretboard.

One of the main goals of the module is to enumerate all the *suitable* positions for playing a set of notes in a given `Tuning`, where a `Tuning` simply gives the pitch of each string:

```
type Tuning = M.Map GString Pitch
```

However, what a suitable position might be often depends on the context; in many cases, it is desirable to allow a note that was struck earlier to continue playing, and therefore, some fingerings must be held, and some strings must not be struck again. Thus, we also need to know the `PosContext`, a description of the fingerings that must be held and the strings that must be allowed to ring:

```
type PosContext = (Position, StruckNotes)
type StruckNotes = M.Map GString PitchT
```

The datatype `PitchT` simply includes the start and stop times and pitch of a given note. The function `suitablePositions` then takes in all of this information, and returns a list of all the possible `PosContext`s that could result:

```
suitablePositions :: Tuning -> PosContext -> [PitchT] -> [PosContext]
```

A `Tab` is simply defined as a map of all the fretting position contexts for a musical piece, indexed by the time at which those positions are held:

```
type Tab = M.Map PTime PosContext
```

4 Future Directions

There are many possible improvements that could be made to this module, including the following:

1. Currently, I have not generated enough fretting positions. Occasionally, guitar pieces that I try will fail at some point, because they have some odd fretting position that is not in my "bank" of possible positions.

2. Currently, if the module can't find a fretting position, it fails and produces an error, and cannot output the tab. It would be nice to have an ability for the module to perhaps be unable to find positions some portions of the music, but still output tablature for the rest of the music.

3. The individual and transition scoring functions are very rudimentary currently. It would help to have factors that take into account factors like the angle of the hand, and scrunching or spreading of fingers, and to better understand changes between hand positions. Also, I haven't spent much effort trying to balance the scoring functions, and it's possible that some scoring functions are much too heavily weighted.