Appendix

Computational analysis of jazz chord sequences

Romain VERSAEVEL, M1 Informatique Fondamentale, ENS de Lyon

Tutored by David MEREDITH, Associate professor at Aalborg University

August 27, 2015

Contents

1 List and definitions of all similarity measures

Equality, root notes, transposition

Let us consider two chords $C_1 = R_1H_1$ and $C_2 = R_2H_2$, where R_i denotes the root note and H_i the harmonic label. The first three similarity measures are then defined by:

$$\mathfrak{M}_{root}(C_1, C_2) = 1 \Leftrightarrow R_1 = R_2$$

$$\mathfrak{M}_{transposition}(C_1, C_2) = 1 \Leftrightarrow H_1 = H_2$$

$$\begin{aligned}\mathfrak{M}_{eq}(C_1, C_2) = 1 &\Leftrightarrow C_1 = C_2 \\ &\Leftrightarrow \mathfrak{M}_{root}(C_1, C_2) = \mathfrak{M}_{transposition}(C_1, C_2) = 1\end{aligned}$$

PCS-prime equivalence

Formally, $\mathfrak{M}_{PCS}(C_1, C_2) = 1$ if and only if C_1 can be obtained from C_2 by combining the following two operations:

$$\begin{aligned}\text{transposition} : \mathcal{P}([0,12]) \times \mathbb{N} &\longrightarrow \mathcal{P}([0,12]) \\ (\{x_i\}_i, t) &\longmapsto \{(x_i + t) \mod 12\}_i \\ \text{inversion} : \mathcal{P}([0,12]) &\longrightarrow \mathcal{P}([0,12]) \\ \{x_i\}_i &\longmapsto \{(12 - x_i)\}_i\end{aligned}$$

The algorithmical way to compute $\mathfrak{M}_{PCS}(C_1, C_2)$ is a bit technical but an excellent description can be found on http://composertools.com/Theory/PCSets.pdf.

Morris', Rahn's, Lewin's, Teitelbaum's and Isaacson's measures

All the following measures imply the use of *interval vectors*, as defined in the main report. Let us now for more clarity denote the two chords to compare by X and Y, and their respective interval vectors by $IV_X = (x_1, \ldots, x_6)$ and $IV_Y = (y_1, \ldots, y_6)$. We also define $D(X, Y)$, the difference vector $((y_1 - x_1), \ldots, (y_6 - x_6))$.

Morris' measure is the sum of absolute values of the coordinates of D (city-block distance applied to the interval vector):

$$\mathfrak{M}_{Morris}(X, Y) = \sum_{i=1}^{6} |D(X, Y)| = \sum_{i=1}^{6} |y_i - x_i|$$

Rahn's measure is the sum of the entries of the interval vectors of X and Y, but only for the coordinates i such that both $IV(X)_i$ and $IV(Y)_i$ are non-zero (if for instance X has intervals of length 4 but Y does not, these are not taken into account). It is then scaled in order to get values in $[0, 1]$:

$$\mathfrak{M}_{Rahn}(X, Y) = \sum_{i=1}^{6} \left[(IV(X)_i + IV(Y)_i) \cdot \mathbf{1}_{IV(X)_i \geq 1 \text{ and } IV(Y)_i \geq 1} \right] \cdot \frac{2}{|X|(|X| - 1) + |Y|(|Y| - 1)}$$

Lewin's measure is the sum of the square roots of products of the coordinates of $IV(X)$ and $IV(Y)$ (like for Rahn's measure, a single coordinate equal to 0 leads to a zero term). It is also properly scaled:

$$\mathfrak{M}_{Lewin}(X, Y) = \sum_{i=1}^{6} \left[\sqrt{IV(X)_i \cdot IV(Y)_i} \right] \cdot \frac{2}{\sqrt{|X|(|X| - 1)|Y|(|Y| - 1)}}$$

Teitelbaum's measure is the euclidean distance applied to the interval vectors:

$$\mathfrak{M}_{Teitebaum}(X, Y) = \sqrt{\sum_{i=1}^{6} [(IV(X)_i - IV(Y)_i)^2]}$$

Finally, and as described in the main report, Isaacson's similarity index is the standard deviation function applied to the interval vectors. If $\bar{D}$ denotes the mean of the $(y_i - x_i)$s, it is:

$$\mathfrak{M}_{Isaacson}(X,Y) = \sqrt{\frac{1}{6}\left(\sum_{i=1}^{6}\left(D_i - \bar{D}\right)^2\right)}$$

Similarity of sequences

For all measures, two chord sequences $(C_i)_{1\leq i\leq n}$ and $(C'_i)_{1\leq i\leq n}$ of the same length n are considered similar according to a certain threshold t if and only if $\mathfrak{M}(C_i, C'_i) \geq t$ for all $i \in [0,n]$[1].

The only exception is $\mathfrak{M}_{transposition}$, which allows a more precise extension to sequences: the *difference* (in the mapping to $\mathbb{Z}/12\mathbb{Z}$) between corresponding notes should all be the same: $\exists d \in \mathbb{Z}/12\mathbb{Z}, \forall i \in [0,n], \mathfrak{M}_{transposition}(C_i, C'_i) = 1 \wedge (C'_i - C_i) \mod 12 = d$.

For instance, if we consider the sequences:

$$S = Am;\ B;\ G7;\ A$$
$$S' = Dm;\ E;\ C7;\ D$$
$$S'' = Am;\ E;\ F7;\ B$$
$$S''' = A9;\ E;\ FM7;\ B7\flat5$$

S and S' are similar ($d = -7 \mod 12 = 5$), but S and S'' are not (the difference between A and A is 0, and $-7 \mod 12 = 5$ between B and E); and sequences S'' and S''' are dissimilar because despite a constant difference (0) between root notes, the harmonic labels are different.

[1] I kept the definitions from the original papers, so no homogenization has been made, and for some measures a better similarity means being closer to 0, so it would be "$\leq$" instead of "$\geq$".

2 LZ77: a full example

In this section I give an illustrative example of an execution of the LZ77 algorithm (without restriction on the buffer and preview sizes). The pseudo-code of the algorithm is recalled in Algorithm 1.

Algorithm 1: LZ77

Input: Queue of Chords $\texttt{I} = (C_1, \dots, C_n)$.

Output: Queue of triples $\texttt{L} = (a_j, b_j, C_{i_j})_j$.

Begin

- `buffer` $\leftarrow$ empty queue
- **While** `I` *is not empty* **do**
  - $\pi \leftarrow$ longest prefix of `I` in $(\texttt{buffer} \cdot \texttt{I})$, beginning in `buffer`
  - $\mathsf{a} \leftarrow \texttt{size(buffer)} -$ (beginning index of π (in `buffer`)) (0 if none)
  - $\mathsf{b} \leftarrow$ length of π (0 if none)
  - **For** j **from** 1 **to** b **do**
    - `buffer.push(front(I))`
    - `I.pop()`
  - `L.push`$(\mathsf{a}, \mathsf{b}, \texttt{front(I)})$
  - `buffer.push(front(I))`
  - `I.pop()`
- **Return** `L`

Let us consider the input data $\texttt{I} = ABCABCABD$. These letters could represent chords with no particular harmonic indication (regular major chords — this is for the clarity of the example, but of course it works exactly the same way with harmonic labels). However they could obviously represent anything else: the LZ77 algorithm is a general algorithm for any stream of symbols.

During the execution, this input will be gradually transferred to the `buffer`, which is originally empty:

Step	**Buffer**								**Input** ("preview")								
0									A	B	C	A	B	C	A	B	D

We are looking for the longest prefix of $ABCABCABD$ beginning in `buffer`: `buffer` is empty and so is the longest prefix. So we add to the result `L` the triple $(0, 0, A)$ (A being the first symbol

of `I`) and we transfer the front of `I` to the `buffer`. We obtain:

Step	**Buffer**								**Input** ("preview")								
0									A	B	C	A	B	C	A	B	D
1								A	B	C	A	B	C	A	B	D	

Again, the longest prefix of $BCABCABD$ beginning in `buffer` is empty. We add to the result `L` the triple $(0, 0, B)$ and we transfer the front of `I` to the `buffer`. The same happens once again (no prefix of $CABCABD$ beginning in `buffer`$= AB$). We obtain:

Step	**Buffer**								**Input** ("preview")								
0									A	B	C	A	B	C	A	B	D
1								A	B	C	A	B	C	A	B	D	
2							A	B	C	A	B	C	A	B	D		
3						A	B	C	A	B	C	A	B	D			

Now, we have the prefix $ABCAB$ of $ABCABD$ which begins in the `buffer`. We can add $(3 - 0, 5, D) = (3, 5, D)$ to `L`. This step could be intriguing because $(3, 5, D)$ means "go 3 symbols back and rewrite the 5 next". What will happen is that we will indeed rewrite the 3 last symbols ABC and the first 2 we just added: AB.

`I` is then empty: the algorithm returns `L` = $(0, 0, A)$, $(0, 0, B)$, $(0, 0, C)$, $(3, 5, D)$.

Considering that chords have the same size as integers, the input sequence had a weight of 9 and the output of $4 \cdot 3 = 12$. The compression factor is then $\frac{9}{12} = 0.75$, which is bad (it is lower than 1: the "compressed" sequence is in fact heavier than the input); this often happens on on small instances.

3 Diagonal pattern decomposition: all algorithms

In this section I give pseudo-codes for all the methods used by the diagonal patterns algorithm, and a complexity analysis.

Listing the patterns

Algorithm 2 shows how the diagonal patterns are selected.

Algorithm 2: Pattern listing

Input: Sequence of chords $\mathtt{I} = (C_1, \ldots, C_n)$.

Output: List of patterns $\Pi = (\pi_1, \ldots, \pi_m)$.

Begin

- **For** i **from** 2 **to** $|\mathtt{I}|$ **do**
  - $\pi \longleftarrow \emptyset$
  - **For** j **from** 1 **to** $(|\mathtt{I}| - \mathsf{i})$ **do**
    - **If** $\mathtt{similar}(C_{i+j}, C_j)$ **then**
      - $\pi \longleftarrow \pi + C_j$
    - **Else**
      - $\Pi.\mathtt{insert}(\pi)$
      - $\pi \longleftarrow \emptyset$
- **Return** Π

Where `insert` creates a new pattern in Π if π has not been seen yet, or adds an occurrence to it if π is already in Π. With a hash table, this operation is performed in $\mathcal{O}(1)$. In my implementation, I could not use a hash table, because it is not robust to the use of similarity measures. So, it is necessary to search for a pattern possibly similar to π in Π. Comparing π with a pattern already in Π is $\mathcal{O}(1)$ if they have different lengths and $\mathcal{O}(|\pi|)$ if they have the same length. There can be at most $\mathcal{O}\left(|\mathtt{I}|^2\right)$ patterns in Π, among them at most $|\mathtt{I}|$ of size $|\pi|$, and $|\pi|$ is at most $|\mathtt{I}|$. Hence this step has a cost of $\mathcal{O}\left(|\mathtt{I}|^2 + |\mathtt{I}| \cdot |\mathtt{I}|\right) = \mathcal{O}\left(|\mathtt{I}|^2\right)$.

Overall complexity is then $\mathcal{O}\left(|\mathtt{I}|^2\right)$ with a hash table and $\mathcal{O}\left(|\mathtt{I}|^4\right)$ in my current implementation.

Set cover

Algorithms 3 and 4 describe the two heuristics I use for the set cover problem.

Algorithm 3: Set cover 1

Input: List of patterns Π.

Output: Cover Π' (initially empty).

Begin

  While *true* **do**

    forall the $\pi \in \Pi$ **do**

      $\texttt{gain}[\pi] \leftarrow (\#\{\text{uncovered elements covered by } \pi\} - |\pi|)$

    $\pi^* \leftarrow$ pattern of maximal `gain`

    $\Pi' \leftarrow \Pi' \cup \{\pi^*\}$; $\Pi \leftarrow \Pi - \{\pi^*\}$

    If Π' *covers the whole piece* **then**

      Return Π'

    forall the $\pi \in \Pi$ **do**

      Remove every useless occurrence of π

For Algorithm 3, computing `gain` or determining the useless occurrences of a pattern can be $\mathcal{O}\left(|\texttt{I}|^2\right)$ (in the –extreme– worst case of a pattern of length $|\texttt{I}|/2$ occurring $|\texttt{I}|/2$ times, for instance), `I` being the original input. There can be $\mathcal{O}\left(|\texttt{I}|^2\right)$ patterns in Π and at most $\mathcal{O}\left(|\texttt{I}|\right)$ steps are necessary. Overall worst-case complexity is then $\mathcal{O}\left(|\texttt{I}|^5\right)$. Yet, the actual running time is very reasonable, and lower than Algorithm 2. Maybe a more precise or amortized analysis could be carried (especially about $|\Pi|$) to obtain a tighter bound.

Algorithm 4: Set cover 2

Input: List of patterns Π.

Output: Cover Π' (initially equal to Π).

Begin

  `sort` Π' by decreasing length

  For i **from** 1 **to** $|\Pi'|$ **do**

    Remove every useless occurrence of Π'[i]

  Return Π'

For Algorithm 4, browsing all the occurrences of all patterns has complexity $\mathcal{O}\left(|\texttt{I}|^2\right)$ (the set of all the patterns is included in the upper triangle of the binary matrix). Sorting costing only $\mathcal{O}\left(|\texttt{I}| \cdot \log(|\texttt{I}|)\right)$, overall worst-case complexity is $\mathcal{O}\left(|\texttt{I}|^2\right)$.

Maximizing the recovery factor

As for the computation of the recovery factor, it is simply performed by Algorithm 5. The decompression is built in the order the patterns are given (the information in the first ones may be erased by the content of further ones). So this order is important. Hence the patterns are sorted beforehand by increasing number of occurrences. The reason for this sorting criterion is not obvious; I conducted short tests to choose the best way of sorting and empirically this one was the best; however I think more complete tests and a theoretical analysis should be conducted.

The sort is done at the beginning of Algorithm 6. Then, for each pattern, the sequence similar to it in the input maximizing the recovery is chosen. When computing again the recovery factor, only the chords finally taken into account will be impacted, so this indeed results in a configuration of the patterns maximizing the recovery factor.

Algorithm 5: Recovery factor

Input: Covering list of patterns Π, chord sequence $\mathtt{I}$.

Output: Recovery factor.

Begin

- **For all** $\pi \in \Pi$ **do**
  - **For all** i *occurrence of* π **do**
    - **For** j **from** 1 **to** $|\pi|$ **do**
      - decompression[i + j] $\longleftarrow \pi$[j]
- rec_factor $\longleftarrow 0$
- **For** i **from** 1 **to** $|\mathtt{I}|$ **do**
  - **If** decompression[i] $= \mathtt{I}$[i] **then**
    - rec_factor ++
- **Return** (rec_factor/$|\mathtt{I}|$)

The number of patterns in Π is at most $|\mathtt{I}|$, since Pi covers $|\mathtt{I}|$ with no useless patterns (this is the case after both 3 and 4). Moreover, for any $\pi \in \Pi$, the sum of its number of occurrences and its length is $\mathcal{O}(|\mathtt{I}|)$, again since there is no redundant occurrence after 3 and 4. So the complexity of Algorithm 5 is $\mathcal{O}\left(|\mathtt{I}|^2\right)$.

For Algorithm 6, we still have $|\Pi| \leq |\mathtt{I}|$; and the sequences similar to π have already been computed, so doing an actual search is not necessary, and there are at most $\mathcal{O}(|\mathtt{I}|)$ such

sequences to test. Overall complexity is then $\mathcal{O}\left(|\mathtt{I}|^4\right)$. Of course, this algorithm could be optimised, especially with more pre-computation (for instance, knowing exactly for each pattern what positions it affects in the decompression would reduce the overall complexity to $\mathcal{O}\left(|\mathtt{I}|^2\right)$); but this is not currently the case in my implementation.

Algorithm 6: Recovery factor maximization

Input: Covering list of patterns Π, chord sequence $\mathtt{I}$.

Side effect Π *is adapted to maximize the recovery factor.*

Begin

- $\mathtt{sort}(\Pi)$
- rec_factor $\longleftarrow$ $\mathtt{recovery\_factor}(\Pi, \mathtt{I})$
- **For all** $\pi \in \Pi$ **do**
  - **For all** π' *similar to* π *in* $\mathtt{I}$ **do**
    - $\pi \longleftarrow \pi'$
    - **If** $\mathtt{recovery\_factor}(\Pi, \mathtt{I}) <$ rec_factor **then**
      - $\pi' \longleftarrow \pi$
- **Return** Π

4 All results

In this section I give all the numerical results I obtained (compression factors and recovery factors, for different measures and different thresholds).

Measures without thresholds

The results of the measures without thresholds are listed in the following array:

	LZ77		**Diagonal patterns**	
	Compression	Recovery	Compression	Recovery
Equality	0.931258	1	1.20672	1
Root notes	1.22421	0.634401	1.46531	0.517409
transposition	1.27305	0.531337	1.39689	0.463788
PCS-prime	1.33706	0.502786	1.42744	0.419916

Measures with thresholds

The curves of compression and recovery factors of the other measures are shown on figures 1, 2, 3, 4 and 5.

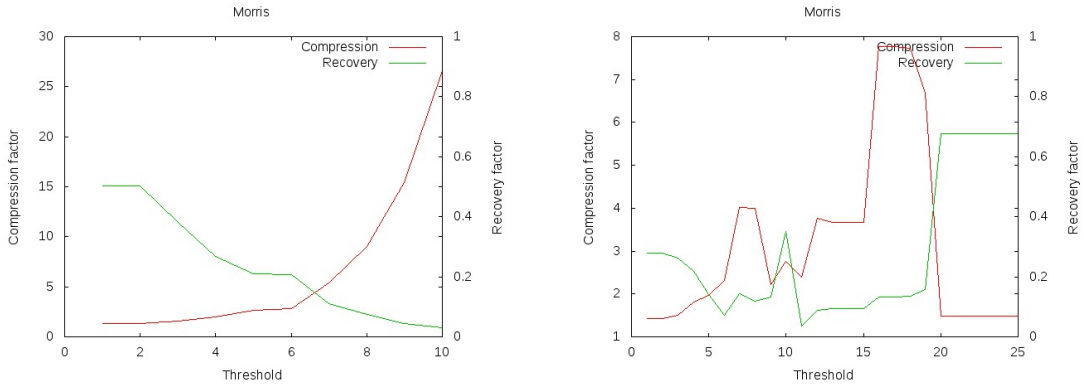

Figure 1: Results with Morris' measure (left: LZ77, right: diagonal patterns).

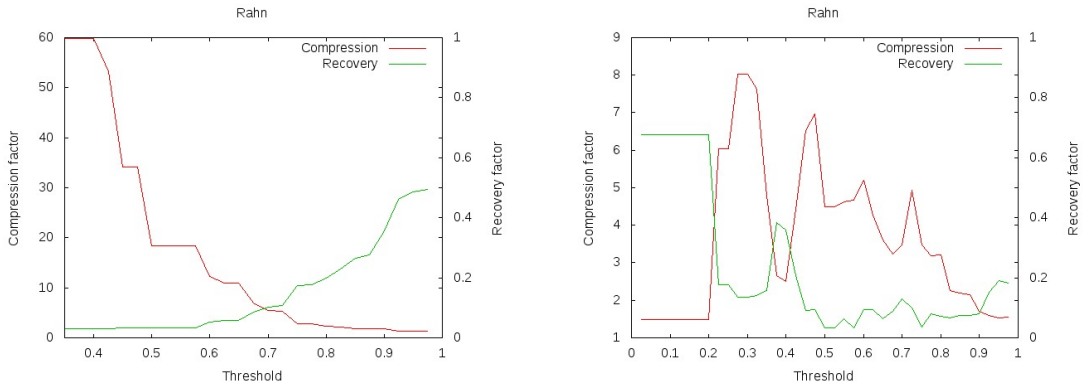

Figure 2: Results with Rahn's measure (left: LZ77, right: diagonal patterns).

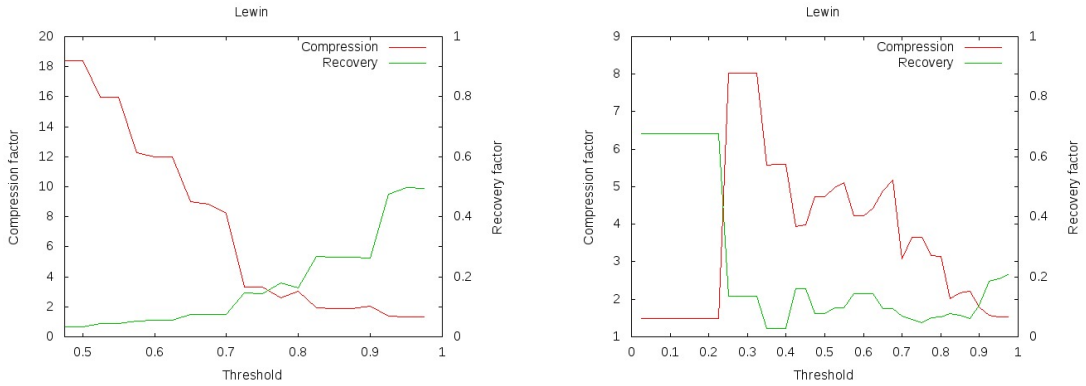

Figure 3: Results with Lewin's measure (left: LZ77, right: diagonal patterns).

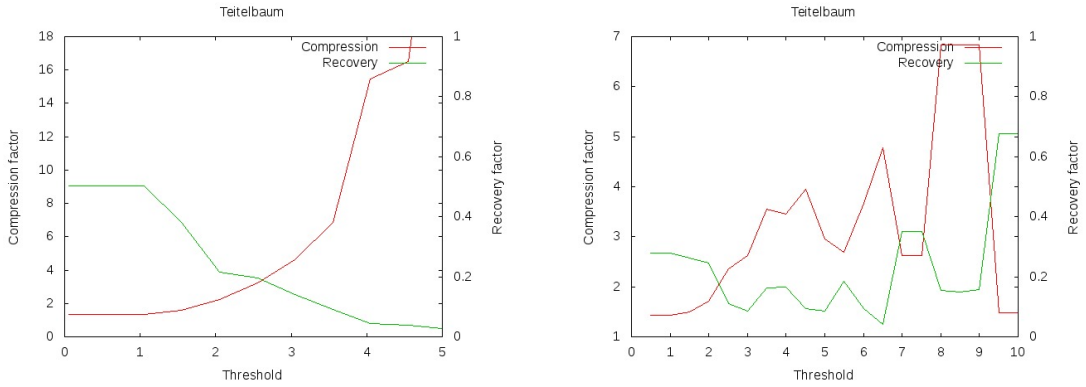

Figure 4: Results with Teitelbaum' measure (left: LZ77, right: diagonal patterns).

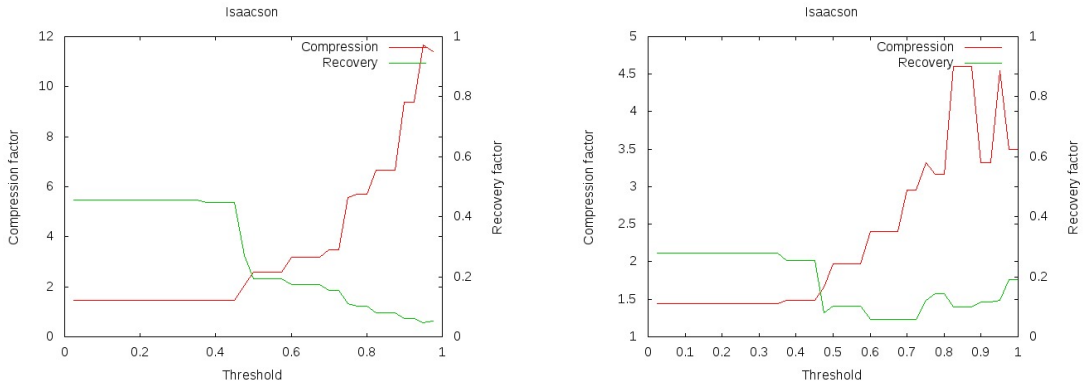

Figure 5: Results with Isaacson' measure (left: LZ77, right: diagonal patterns).

5 Implementation details

This section contains some comments on the coding part of the internship.

I have implemented all the algorithms I mention –be it only because I had no other way to evaluate and improve them than watching the results they produced on the data. I coded in C++, a language that I master and that was adapted to the needs of my experiences, since it is fast and that I did not need to perform particularly complex algorithmic operations.

All my code is stored on a GitHub repository, which also contains the data and the numerical results: https://github.com/Rometach/aalborg.