Contents

1 List and definitions of all similarity measures

Equality

$$\mathfrak{M}_{eq}(C_1, C_2) = 1 \Leftrightarrow C_1 = C_2 \quad (1)$$

2 LZ77: a full example example

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

Algorithm 1: LZ77

```
Input: Queue of Chords I = (C_1, ..., C_n).
Output: Queue of triples L = (a_j, b_j, C_{i_j})_j.
Begin
    buffer ← empty queue
    While I is not empty do
        π ← longest prefix of I in (buffer · I), beginning in buffer
        a ← size(buffer) − (beginning index of π (in buffer)) (0 if none)
        b ← length of π (0 if none)
        For j from 1 to b do
            buffer.push(front(I))
            I.pop()
        L.push(a, b, front(I))
        buffer.push(front(I))
        I.pop()
    Return L
```

Let us consider the input data $\mathtt{I} = ABCABCABD$. The letters represent chords with no particular harmonic indication (regular major chords). 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 Chords have the same size as integers, our original 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 is completely normal on small instances.

3 Diagonal pattern decomposition: all algorithms

Listing the patterns

Algorithm 2 shows how the diagonal patterns are selected.

Algorithm 2: Pattern listing

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

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

Begin

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

Where `insert` creates a new pattern in Π if π has not been seen yet, or adds an occurrence to it if it is already in Π. With a hash table, this is performed in $\mathcal{O}(1)$. In my implementation, I did not use a hash table so an exhaustive search for the pattern has to be done. Comparing the possibly new pattern 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 $\mathcal{O}\left(|I|^2\right)$ patterns in Π, among them at most $|I|$ of size $|\pi|$, and $|\pi|$ is at most $|I|$. Hence this step has a cost of $\mathcal{O}\left(|I|^2 + |I| \cdot |I|\right) = \mathcal{O}\left(|I|^2\right)$.

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

Set cover

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

For algorithm 3, computing `gain` can be $\mathcal{O}\left(|I|^2\right)$ (in the –extreme– worst case of a pattern of length $|I|/2$ occurring $|I|/2$, for instance). There can be $\mathcal{O}\left(|I|^2\right)$ in Π and at most $\mathcal{O}(|I|)$ steps are necessary. Overall worst-case complexity is then be $\mathcal{O}\left(|I|^5\right)$. Yet, the actual running time is

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**

      `gain`$[\pi] \leftarrow (\#\{$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'

very reasonable, and lower than algorithm 2; a more precise analysis could be carried to have a better idea of the complexity.

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, there can be $\mathcal{O}\left(|I|^2\right)$ patterns and browsing the occurrences of a pattern to determine if they can be removed or not may take $\mathcal{O}\left(|I|^2\right)$. Sorting costing only $\mathcal{O}\left(|I| \cdot \log(|I|)\right)$, overall worst-case complexity is $\mathcal{O}\left(|I|^4\right)$; but as for algorithm 3 the actual running time is short.

Maximizing the recovery factor

4 PLOTS???

5 Implementation details

6 Bibliography