NAVIGATING THE ORACLE: A HEURISTIC APPROACH

Gérard Assayag
IRCAM – CNRS UMR 9912,
Paris, France,
assayag@ircam.fr

Georges Bloch
Université de Strasbourg,
Strasbourg, France,
gbloch@umb.u-strasbg.fr

ABSTRACT

We have presented in several papers the musical potentialities of a data structure called the Factor's Oracle (FO) that we have experimented in many situations involving style modeling, musical memory and anticipation models, real-time improvized interaction and performance analysis. We recall briefly the construction and properties of this structure and how we use it in music pattern analysis and generation. Then we describe a new heuristic for navigating the FO in order to significantly ameliorate the musical quality of the output. This heuristic method leads us to a powerful formal structure, derived from the FO, called the Suffix Link Tree (SLT).

1. FACTOR ORACLES

1.1. Oracle Musical Applications

The Factor Oracle structure [1, 16] has been intensively investigated during the past few years as an efficient tool for music pattern analysis and reconstruction. Providing a computation friendly set of algorithms and representations, it has generated numerous fruitful research and application works in computer music, in association with other techniques inspired by statistical sequence modeling [12] and representations for data compression. In an offline approach, such researches have been conducted in : style modeling [6, 8, 11], style simulation [10, 11], improvisation and performance modeling [9], musical memory and anticipation modeling [13, 14], composition [2, 3]. These researches owe a lot to Shlomo Dubnov's early intuition of the benefits brought by sequence models and compression schemes in style modeling problems. As for the on-line, real-time interaction approach, two main systems have been built : Omax [4], by Assayag, Bloch and Chemillier, an OpenMusic/Max based application that builds co-improvisations with human performers using the Midi, audio and video media ; and MiMi [15] by A. François and E. Chew, a multimodal interface for co-improvisation with the computer that gives a visual feed-back to the performer, helping him to anticipate the computer's reactions. Omax has been used as a virtual partner by a lot of professional musicians, in experimental sessions as well as public performances at Ircam and at popular summer jazz festivals.

Recently, we have gotten back to the theory of Factor Oracles in order to set up better generative heuristics than the ones at hand. This paper will focus on these heuristics, showing how a careful understanding of the oracle structure and behaviour may lead to more motivated musical recombination of learned material. In order to get an idea of the power of Omax oracle generation, the reader can check the web site at :

htttp://www.ircam.fr/equipes/repmus/OMax

1.2. Oracle Basics

The Factor Oracle concept comes from research on string patterns indexation. Such research has applications in massive indexation of sequential content databases, pattern discovery in macromolecular chains, and other domains where data are organized sequentially. Generally stated, the problem is to efficiently turn a string of symbols S into a structure that makes it easy to check if a substring s (called a *factor*) belongs to S, and to discover repeated factors (patterns) in S. The relationship between patterns may be complex, because these are generally subpatterns of other patterns; therefore the formal techniques and representations for extracting them, describing their relationships, and navigating in their structure are not obvious. However, these techniques are extremely useful in music research, as music, at a certain level of description, is sequential and symbolic and the pattern level organization of redundancy and variation is central to its understanding.

Among all available representations (e.g. suffix trees, suffix automata, compression schemes), FO's represent an excellent compromise.

1. they compute incrementally and are linear in number of states and transitions
2. they are homogeneous, i.e. all transitions entering a given state are labeled by the same symbol, thus transitions do not have to be labeled, which saves a lot of space
3. they interconnect repeated factors into a convenient structure called SLT (suffix link tree)
4. their construction algorithm is simple to implement, maintain and modify.

We will not detail the construction algorithm, see [1, 16], but just recall the properties of FO's. From a stream of symbols $s = s_1 s_2 .. s_n ..$, the FO algorithm builds a linear automaton with (by convention left-to-right) ordered states S_0, S_1, S_2 .. S_n. Symbol s_i is mapped to state S_i and S_0 is an initial state (source). As said above,

transitions in the FO are implicitly labeled by the symbol mapped to their target state. *Forward* (or *factor*) transitions connect all pairs of states (S_{i-1}, S_i), and some pairs (S_i , S_j) with i < j-1. Starting from the source and following forward transitions one can build factors of s, or one can check if a string s' is a factor of s. We also consider some construction arrows named *suffix links,* used internally by the FO algorithm for optimization purposes, which, however, have to be kept for musical applications. These *backwards* pointing arrows connect pairs of states (S_i , S_j) where j < i. A suffix link connects S_i to S_j iff j is the leftmost position where a longest repeated suffix of s[1..i] is recognized. In that case, the recognized suffix of s[1..i] – call it u – is itself present at the position s[j-|u|+1, j]. Thus suffix links connect *repeated patterns* of s.

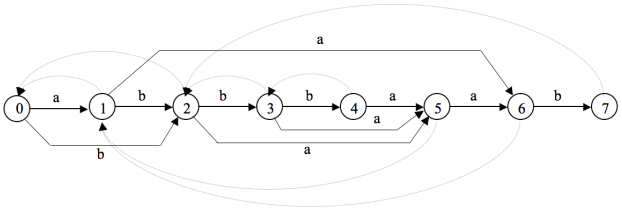

Fig 1 The Factor Oracle for string abbbaab. Suffix links are in dotted lines.

Fig. 1 shows the FO built from the string s=abbbaab. By following forward transitions, starting at the source, one can generate factors, such as bbb or aab. Repeated factors such as ab are connected through suffix links.

However, there is a problem with FO's. As can be checked on fig. 1, the « false positive » factor aba, which is not present in s, can be generated as well. This is because the FO automaton does not *exactly* model the language of all factors of a string. They rather model a language that *contains* it. In other terms, if s' is a factor of s, it will be recognized as such. On the other hand, if s' is recognized, it is *probably* a factor of s.

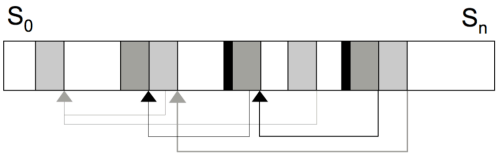

Fig 2 interconnection of repeated factors by suffix links

Fig 2 shows how maximum length repeated factors are interconnected by suffix links. The thickness of the lines represents the length of the repeated factor. This length is computed at no additional cost by the oracle algorithm, and we will see later that it provides a very important clue in the navigation. The color of the lines (gray or black) separates two disjoint substructures in the set of suffix links, each of which forms a tree. The overall suffix links structure is a forest of disjoint trees, whose roots are the smallest and leftmost patterns appearing in the trees (see fig. 3). A fundamental property of these Suffix Link Trees (SLT) is that the pattern at each node is a suffix of the patterns associated to to its descendants (***property 0***). This way, the SLT capture all the redundancy organization inside the sequence.

Factor links also capture redundancy information, because of the following oracle property: let *u* a factor of s appearing at position i (position of its last symbol). There will be a factor link from i to a forward state j labeled by symbol *a* iff (***property 1***):

1. u is the sub-word recognized by a minimal factor link path starting at the source 0 and ending in i.
2. u is present at position j-1, forming the motif ua at position j.
3. the motif ua at position j is the first occurrence of ua in the portion s[i+1, |s|]

So a factor link connects two occurences of a pattern, albeit with some restrictions.

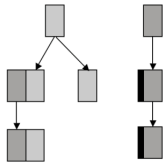

Fig. 3 suffix links form a forest of trees

1.3. Navigating the FO

We have seen that FO's efficiently compute a structure that describes the redundancy organization of a sequence. In order to apply this to music, first of all, the musical stream must be turned into a sequence of atomic units. Previous works show how this can be achieved on arbitrarily complex polyphonic structures [11]. A classification scheme must be provided in order to identify two polyphonic units and reduce them to the same symbol. This is achieved through diverse chord and texture classification schemes [7]. Then the sequence is oraclized (in real-time in the case of interactive applications). Then the oracle is navigated in order to generate new sequences.

We will show the navigation basic principle in an example. Let a,b,c be symbols and u,v, x, y motives, s = xuayvaubxvcuv. A possible navigation would be, starting at the beginning : **xva****ua**ub**x**uay**vc****u**bx.

Each time a motif w is output by the generation, and this motif appears elsewhere in the learned sequence, there is a possible choice to jump to the last symbol of the repeating motif and start again from this location. Such a musical transition is smooth because the location where we're coming from and the location where we jump share a common suffix w. w is called a *context*, and the process *context-switching.* In the previous example, the contexts used for switching are in **bold**.

Of course the transition will be musically valuable only with a significant context length, and if the classification scheme –telling us that context w at position i and context w at position j are the same (although, in the actual music sequence, there could be

superficial differences) – is accurate. We will tell more about this in the following sections.

Context switching can be achieved by the means of factor links or suffix links. In the case of factor links, *property 1* tells us that two positions connected by a factor link share a common context. So, as our purpose is not factor recognition but rather generation, this property can be exploited even though FO is not an exact factor automaton. In the case of suffix links, the SLT structure is precisely a navigation tool inside the contextual organization of patterns. In addition, suffix links give information about context length, and they connect all possible contexts into an exhaustive structure revealing a huge recombination potential.

2. FACTOR ORACLE AS A GENERATING DEVICE : AN HEURISTIC APPROACH

As a difference with most applications using FO's, we use them as *generating device*, so we are interested in specific properties of this representation and are looking for navigation heuristics that make sense in a musical context. The generation process actually *edits* a new sequence out of the original one. The use of the term "editing" is correct since the new sequence is a stream of the old sequence in which connecting jumps are made in context-coherent locations. It allows the creation of a completely new sequence while keeping the stylistic surface features of the original one as shown in [5].

Recent examination of the factor oracle has resulted in quite an improvement of the oracle as a generating device. Several enhancements are just a consequence of a careful analysis of the way the oracle is constructed: for generation applications, context-coherence is mandatory, and it cannot be ensured by factor links. The other observations are somehow less general and depend on the problem at hand, that is, the generation of musical improvisation.

2.1. Not using Factor Links

As stated above, the factor oracle classifies the repetitive patterns in a given stream of data. It does it in two ways: through forward arrows, called "factor links", that generate factors (with false positives), and connect the occurrence of a pattern where it is first recognized with its first repetition in the future; an through backward arrows, called "suffix links", that form a complex tree structure interconnecting repeated patterns and giving information about context length. The initial naive navigation strategy was to follow the factor links with a probability p, and follow suffix links with probability 1-p, with p > 1-p. In the first case, the process was replicating the original in segments whose length would statistically depend on p. From times to times, through a suffix link jump, it would generate a context-based recombination. Listening tests have shown that the quality of the generation, although globally convincing, would in some places dramatically drop, due to some awkward recombination. Careful analysis showed two reasons to explain this problem: suffix recombination would sometimes fail because of a small context length and rhythmical inconsistencies (e.g. a fast figure, expected to continue in a similar way, would collapse by recombining with an unexpected long note); unwanted recombination would sometimes be created by factor link oriented generation.

2.2. The Problem with Factor Links: an Example

In the example shown in fig. 4 – an oracle on the string AAFFBCFCDGGEEEBEEDGGEAFFBA – the problem can easily be seen. The patterns AFFB and DGGE are repeated, and a reasonable generation system would assimilate the point at index 25 to the point at index 5. Therefore, when generating a new sequence, when one arrives at index 25, it may be desirable to allow the possibility to suffix-jump from 25 to 5 considering that the B of 25 is contextually very similar to the B in 5, with the favorable context of 4 elements (AFFB). However, the choice of a factor link at 5, in order to pursue the sequence after the B, gives us three solutions:

1. The obvious one, going on to index 6, therefore getting AFFBC
2. Continuing from where we come from (25=5 to 26), getting AFFBA, but, in that case, we are not generating anything new, and we end up canceling the jump
3. Jumping to 16, a very bad solution: we get AFFBEE, which makes us jump to the unknown with a single symbol context: B.

This problem is acknowledged in the theory of factor oracles [17]. It is related to the fact that the language modeled by FO's is not exactly the language of all factors of a sequence, and to *property 1*, stating that a factor link relates two patterns on behalf of their common context *u*, but only when the first occurrence of *u* corresponds to a minimal length factor path starting at the source. This is exactly the case here: at position 5, the minimum factor-path corresponds to the recognition of 'B', through the factor link (0, 5), so the factor link (5, 16) accounts for the very short context 'B' and not for the more interesting context 'AFFB', thus the wrong generation 'AFFBE'.

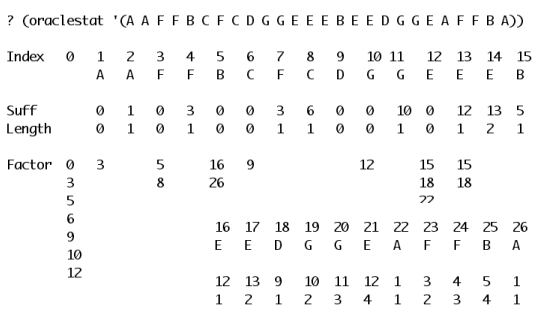

Fig. 4 An oracle showing context-inconsistency with factor links. Lines indicate suffix links, context length, and factor links.

For that reason, we keep up to the first solution: after a suffix-jump to position i, we go to the position immediately on the right, i+1. In this case, we jump from

location 25 to location 6, generating … GE**AFFB**C. Although the motif GEAFFBC was never played in the original, it has a strong markovian justification [10] with a context of length 4. For this same reason, we will avoid factor links altogether, and alternate between sequential reading of the oracle (..i, i+1, i+2,..) and suffix jumps from location j to location i, followed immediately by a sequential reading of i+1.

2.3. How to jump forward without any Factor Link?

However, one senses immediately a problem: if the generation always goes backwards, it will end up being stuck at the beginning of the sequence. This is the reason why we use "reverse suffix link", that is, we are also exploring the arrows that arrive to the current generation location.

By construction, there is, at most, *one* single suffix link out of each state. However, several suffix links can point to the same state. In our above example, the point 13 has two suffix links arriving to it: one coming from 14, with length = 2 (length of the common context), and one coming from 17, also with length = 2. By construction (*property 0*), reverse suffix links are always better than suffix links leaving the same state (that is, with a greater context length). So, given the use of these reverse suffix links, not only can one equate location 25 to 5, but also 5 to 25, which seems logical, and an infinite circulation inside the oracle is now possible.

2.4. Avoiding systematic Connections to the first Occurrence

A suffix link always connects a pattern to a location where this pattern has been recognized *for the first time*. Practically, it means that all identical patterns in the sequence are connected to the leftmost prototype, but they are not connected to one another. There is the risk of monotony, because all these patterns will context-switch to the same leftmost prototype. This implies the following context switching strategy : starting at the edit point (the current location of the generation). We examine:

1. The suffix link leaving the editing point (in the example links 4)
2. The reverse suffix links leaving the editing point.
3. The reverse suffix links leaving the target of the suffix link
4. The suffix link leaving the target of the suffix link, as well as the reverse suffix links leaving it
5. Iteration of (4) down to state 0 or until some optimization criteria are met.

This is the only way of navigating through all the possible occurrences of the same pattern. This examination results in a collection of switch candidates, in the form of triplets (i, j, n) with i the edit point, j the switching target, n the context length. The choice is made in this collection by filtering upon a minimal context length, and a probability distribution favoring long contexts but leaving a chance for smaller ones.

2.5. Continuity Factor

When generating a new sequence, it is interesting to know how much of the original sequence is kept. Will the computed sequence closely resemble the original one? Or, on the contrary, do we want many jumps and editing points in order to get a very surprising sequence compared to the original?

Therefore, one must be able to choose how much continuity from the original sequence is desired: this is called the continuity factor. In theory, it is a very simple parameter: it represents the number of events that are played continuously before looking for a new edit. In a musical example, we could decide to try to edit every 16 notes (or polyphonic-slices, in case of a polyphonic system). In that case, one would hear 16 notes of the original sequence before the system connects (or tries to connect) to another region of the sequence. So the editing points would happen after 16 events.

A very similar factor could be statistically built. One could decide, instead of waiting for 16 events, that for each event there is 1 chance over 16 to jump. Although it may seem very similar, this is not a choice we have taken. There is a good reason for it: when we start a sequence on a choosen point – e.g. a neat phrase boundary - with a high continuity factor, we want the improvisation to replicate a significant length of the original phrase before jumping elsewhere. This would'nt be ensured by the probabilistic approach.

2.6. Optimizing the Search for Editing Points: the best ones, not only the best one

In any case, it is interesting to find a "good" editing point, that is a point with a "good" switching context (up to now, a good context is a context of near-maximal length). A choice too strict for the continuity can be taxing, if one falls one step short of a very good editing point; in this case it would be better to have a continuity factor a little bit longer, and in other cases a bit shorter.

But there is the other side of the coin: if we just choose excellent editing points, there is the danger of a constant looping between the two or three excellent editing points in the original sequence (excellent meaning with a very similar context). There is a trade-off to be found, between a very repetitive generated sequence and bad editing.

As for finding good editing points, the solution is to define an "editing region" around a theoretical continuity point. With a continuity value of N, the research is made between N +/- N/5 (with a maximum of 10 before and after). Therefore, for a continuity of 20, the system searches for editing points between the 16^{th} and the 24^{th} step after the last edit. All the switch candidates for all the editing point are inter-sorted with regard to their context length, and a probability distribution favoring context length is applied, in order to choose one candidate. Due to the filtering out of candidates with a context length smaller than a given threshold, the candidate list can be empty. In that case we reinitiate a

continuity sequence, with the heuristic that it is better to replicate than choose a bad edition.

Why not to take the best solution? Because, if we do so, the process ends up being deterministic with the same looping path into the sequence being visited again and again, and we will miss maximal or near-maximal recombination points located out of this path. For the same reason, if the candidate list contains only one solution, we will decide to take it or not depending on a predetermined probability (e.g. 0.5).

2.7. Repetition and Taboo

Loops are a very important problem, especially in the case of optimized edits (but also without them). Therefore we set a "taboo list" that precludes any former editing point to come back before a certain number of editions steps have occurred. We actually have two strategies for taboos.

We can have a taboo list of a given length N (for example, N=8), consisting in a circular buffer containing the last editing targets. Any edit falling on these targets will be forbidden. This is the easiest way.

However, as seen above, the oracle could "cheat" around these points: frequently, the context length increases when one approaches a very good solution: in the example given fig. 5, we can observe how the repetition of "evierge" creates a good editing zone. Therefore we run the risk of a false loop.

Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
	L	E	V	I	E	R	G	E	L	E	V	I	V	A	C	E	E	T
Suff	0	0	0	0	2	0	0	2	1	2	3	4	3	0	0	2	2	0
Length	0	0	0	0	1	0	0	1	1	2	3	4	1	0	0	1	1	0

	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36
	D	E	V	I	E	R	G	E	A	U	J	O	U	R	D	H	U	I
	0	2	3	4	5	6	7	8	14	0	0	0	28	6	19	0	28	4
	0	1	2	3	4	5	6	7	1	0	0	0	1	1	1	0	1	1

Fig. 5 An oracle subjected to loop problems

The process will first edit from 26, going to 8. If the continuity factor is around 18, it will find again 26, 25, 24 and 23 as very good editing points. Since 26 is taboo, it will chose, for example 24 and jump to 7; later it will chose 25 (26 and 24 being taboo), then 23. After a while, the first edit will be popped out of the taboo list and the whole process will loop. There are two solutions to this problem:

1. When editing, check that the distance at which the jump is performed does not bring us back immediately into the same edit region
2. When feeding the taboo list, push not only the current edit target, but also some of its neighbourhood determined by the pattern of increasing context length values.

2.8. Edit Quality Factor

Up to now, the only edit quality parameter used is the context length. It makes sense, because the classification function that identifies two polyphonic units as identical symbols uses all the relevant musical parameters (pitches, durations, dynamics). However, because of the categorizing nature of this classification, quantization must occur, especially in the domain of rhythm. The classification function has a local scope: during the construction of the oracle, it "sees" only two distant units at a time and decides if they are the same. It has no global view on patterns, that is, it cannot decide that, although two patterns are not identical in all their components, they can be globally identified modulo some acceptable transformation. This is particularly taxing for the rhythm because:

1. two patterns, identified as the same one, can in reality differ significantly, because of an accumulation of quantification error, leading to perceivable rhythmic gaps when switching on these patterns
2. two patterns with the same pitch content and very similar rhythmic motion are not identified as the same one thus no editing will occur, thus we are loosing valuable recombination possibilities

Listening tests have shown that these two problems are perceptively much more prominent in the rhythm domain than any other. Therefore musical parameters have been separated in order to distribute the notion of edit quality over two distinct phases of the process. During the learning process, only pitches are used: they contribute to the construction of the oracle, which is actually a pitch oracle. The quality associated with pitches is the context length as computed and stored into the oracle. The rhythm quality is evaluated at generation time, when the list of edit candidates is collected, by computing and comparing the average rhythmical density of the two occurrences of the context pattern for each edit candidate. The candidates which do not show an acceptable density ratio are filtered out of the candidate list (as well as candidates with a too small context length). This strategy has brought the best musical results we have ever experimented since we use FO's.

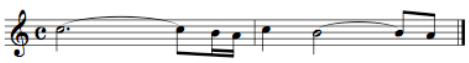

Fig. 6 two rhythmical contexts that should be identified

Fig. 6 shows two rhythmical contexts that are identified by our new quality strategy: as in "real" improvisation, it is the global rhythm structure that is taken into account. Now, phrases giving the same rhythmical feeling than the original are generated, that would never have been created if the classification function had merged the pitch and duration parameters.

3. FORMALIZATION OF THE NAVIGATION STRATEGY

3.1. Suffix Link Trees under the Hood

Being given:

- a sequence of symbols: seq
- an oracle – oracle(seq) – constructed on this sequence,

the set of suffix-links in oracle(seq) constitute a forest (collection of trees). Each tree looks as in fig. 7.

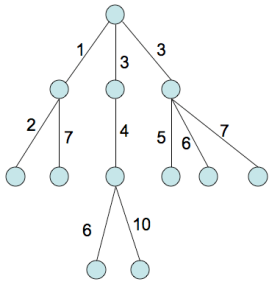

Fig. 7 a suffix link tree

In a suffix link tree (SLT), each child node is connected to its parent through a suffix link. Each node Ni corresponds to a unique position i in the original sequence seq and in oracle(seq) (the reference to i is not indicated in the picture). For the sake of convenience, we will use i to designate both the node and the reference.

The label attached to each parent-child edge is defined by the function rsl(child), that is the length of the context associated to the suffix link (repeated suffix length).

The Suffix Link Tree (called SLT to remember it: Sure, Lisp is Tedious), valued by the function rsl, is associated to a structure of strict partial order between the edges: the edges from a node to its children are all superior to the edge connecting the node to its parent. But there is no particular relation between children; in particular, several edges can share the same label.

A path (a sequence of connected nodes) leading from the root to a leaf is called a *suffix-path* with the following property : call R(i) the context associated to the suffix-link S(i)=j, i and j being nodes and j being the parent of i, if the Suffix-Path from the root to a leaf is (k_1, k_2, …., k_n), with k_1=root and k_n=leaf, then R(k_r) is a suffix of R(k_{r+q}) for all $1 \le r \le n-1$, $1 \le q \le n-r$. This relation is the support of the partial order relation.

While using the oracle as a generating device, the usual situation wit the SLT is:

- we are situated on a node i (position i in the sequence seq)
- we are looking for all the contexts in seq, i.e. all the motives of seq that are a suffix of seq [1…i]. We are looking for contexts of length superior to a given threshold s.

For the time being, we do not consider the quality function unrelated to the oracle (e.g. rhythmic quality). In order to reach all these motives without examining too many positions in the oracle, we need a strategy of navigation inside the SLT. For the sake of convenience, the children from a given parent are sorted left to right according to their ascending edge value.

Let:

- s be the threshold for the context length.
- L= |R(i)| be the length of the context associated to suffix-link (i, S(i))

The upper triangle in fig. 8 represents *one* generation of children below the node S(i) (consequently, i belongs to it). The trapezes in the bottom represent all generations down to the leaves after the first generation. For now, we will admit that L > *s*. The other case will be discussed below.

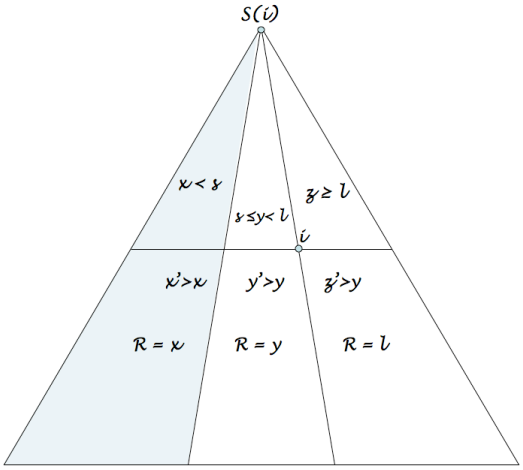

Fig. 8 distribution of context length in a SLT

We break up the direct children of S(i) in three groups: from left to right, there are edges of values $x < s$ (grayed out), edges of values y with $s \le y < L$, and finally those with edges of value $z \ge L$. Identically, the next generations are organized in three groups, with edge values expressed in relation to the edge values in the first generation : x' > x, y' > y and z' > z (*property 0* and strict partial order on the edge values).

Every candidate node j in these sub-trees descends from the root S(i) and thus shares a common suffix with it. The length K of this suffix is given by the last edge encountered when ascending from this node too the root. j will also have a common suffix with i (if two nodes have a common suffix with an ancestor, they do share a common suffix that is the minimum of both suffixes). j will share with i a suffix of length R= min (K, L). We can see that R will take, for j standing in each of the regions below i, the values R=*x*, R=*y* and R=L. There is no need to explore the sub-trees in gray (R is inferior to the threshold), whereas the other two regions have valuable candidates, with a quality $R \ge s$.

If we have a starting value $L < s$, we easily notice that any candidate has a value $R < s$: there is no solution. If L=*s*, the central triangle collapses, and we are left with the right one only and solutions R=L=*s*.

But there is more to do in order to find more edit candidates. We have to take the suffix-path up, starting from S(i): $S^2(i)$, …, $S^n(i)$ – that is the grandparent of i, grand-grandparent, and so on –, up to the ancestor $S^n(i)$ such that the edge leading to it has a value smaller than the threshold *s* (afterwards there is no solution anymore). At each generation, the same algorithm must be applied as we did for i and S(i) , as shown in fig. 9. Again, the grayed parts are not to be explored. L'= $|R(S^{n-1}(i))|$ is the length of the common suffix between $S^{n-1}(i)$ and its parent $S^n(i)$. *x'*denotes the values of edges that respects $s \le x' < L'$. The length of the context for the candidates

will be either *x'* or L', depending on the region of the tree they belong to.

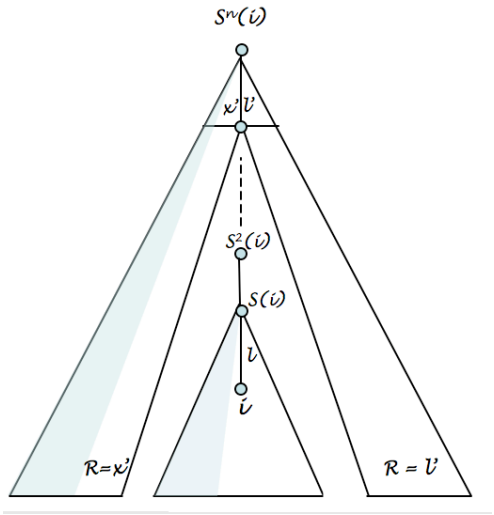

Fig. 9 recurrence over suffix links in a SLT

3.2. The Case of Maximal Suffixes

Given the sequence seq = abaababaaba, the corresponding rsl are:

```
a b a a b a b a a b a
0 0 1 1 2 3 2 3 4 5 6
```

The rsl function has a saw tooth like shape, according to the increasing or shrinking context. In order to minimize the number of vertices in the SLT, we could notice that in the repeated factor baaba (rsl = 6), the last element encodes the maximal repetition; the lesser ones (rsl = 5, 4, 3, 2) can easily be deduced from it; therefore a given value of rsl =n is either inferior or equal to the preceding one or is preceded by a series of decreasing values n-k,…, n-1. The target of the suffix link starting at n-p is just shifted by p units to the left with regard to the target of the suffix link starting at n. Consequently, it would seem practical to construct the SLT by just considering the positions corresponding to the maximal suffixes (just before a drop in context length). These positions are characterized by rsl (i) ≥ rsl (i+1). The size of the trees could be therefore considerably reduced.

Suppose we evaluate the possible edit candidates at position i. By navigating on the tree we find a candidate j with a context of length R. We know that the possible candidates for edit point i-1 include the point j-1 with a context of length R-1.

3.3. Algorithm:

Notations:

- L(i) = |R(i)| length of context attained by suffix-link pointing from i.
- s = minimum threshold for context length
- children(i) yields the list of children k sorted in descending order of L(k).

Details:

- StoreSolution (i j L) stores the candidates obtained by jumping from location i to location j with context length L. The candidate list is incrementally sorted by context length and rhythm quality and it has a fixed length (e. g. 6). So a new candidate is inserted in the list only if pushes a worst solution out.
- SLTCannotImprove (i j L) is true if the candidate list is full and if the context length L for the new candidate is smaller than the minimum context length already registered in the list.

Algorithm SLTSearch: given a state i, compute a list of candidates j such that a context switch (i, j) is acceptable with regard to context length and rhythm quality.

```
Def SLTSearch (i)
For k in children(i)
while L(k) > s
 do
  SLTSearchSubTree (i, k, L(k))
 end
For ii = i then j
For j = father (i) then father (j)
while j != 0 and L(ii) >= s
do
 SLTConsiderSolution (i j L(ii))
 For k in children (j), k != ii
 while min(L(K), L(ii)) >= s
 SLTSearchSubTree(i,k, min(L(k), L(ii)))
end

Def SLTSearchSubTree (i j L)
SLTConsiderSolution (i j L)
For k in children (j)
do
 SLTSearchSubTree (i k L)
end

Def SLTConsiderSolution (i j L)
When SLTCannotImprove (i j L)
  SLTStopSearch()
When RhythmQualityOK (i j L)
  StoreSolution (i j L)
```

4. CONCLUSION

This exploration of Factor Oracles expressivity made us encounter the typical offsetting surprises whoever takes an analysis device backwards in the hope of transforming it into a generative tool might expect. An analysis, chosen for its computer-friendly behavior displays inner devices – namely, the SLT – that allow it to become an efficient context-driven generative device. However, this transformation heavily relies on heuristic choices. Quite important among those, is the desire to be able to reach any relevant part of the Suffix Link Trees, thus to look for optimization in SLT navigation. A related choice is the anti-loop strategy, inducing the creation of a taboo list and a gathering of best solutions into a sorted list where the system will choose with regard to a probability distribution favoring the highly ranked ones (but not excluding the other ones). A continuity factor seems crucial in a generative system that, basically, recombines an existing sequence. However, our choice of "real" vs. "probabilistic" continuity is mostly guided by musical considerations. Finally, the separation / hierarchization of (musical) parameters makes a big difference, as well

as the distribution of these parameters among the learning and the generating processes.

5. ACKNOWLEDGMENT

We wish to thank all the great musicians who have experimented with Omax, the real time interactive environment based on FO designed by G. Assayag, G. Bloch and M. Chemillier (aka the Omax Brother): David Borgo, Mike Garson, Jean-Brice Godet, Philippe Leclerc (in memoriam), Bernard Lubat, Guerino Mazzola, François Nicolas, Hélène Schwartz, Dennis Thurmond. They gave invaluable feedback and expertise on the art of improvisation. We also wish to send a friendly farewell to Vittorio Cafagna, who invented the term Omax Brother, wherever he be now.

6. FO APPLICATIONS AND SOUNDS

http://www.ircam.fr/equipes/repmus/OMax

7. REFERENCES

[1] Allauzen C., Crochemore M., Raffinot M., *Factor oracle: a new structure for pattern matching,* in Proceedings of SOFSEM'99, Theory and Practice of Informatics, J. Pavelka, G. Tel and M. Bartosek ed., Milovy, Czech Republic, Lecture Notes in Computer Science pp. 291--306, Springer-Verlag, Berlin, 1999.

[2] Assayag, G., Bloch, G., Chemillier, M., *OMax-Ofon*, Sound and Music Computing (SMC) 2006, Marseille, 2006

[3] Assayag G., Bloch, G., Chemillier, M., *Improvisation et réinjection stylistique,* Rencontres musicales pluridisciplinaires, GRAME, Lyon, 2006. http://www.grame.fr/RMPD/RMPD2006/

[4] Assayag, G., Bloch, G., Chemillier, M., Cont, A., Dubnov, S., *Omax Brothers: a Dynamic Topology of Agents for Improvization Learning*, Workshop on Audio and Music Computing for Multimedia, ACM Multimedia 2006, Santa Barbara, 2006

[5] Assayag, G., Dubnov, S., *Using Factor Oracles for Machine Improvization*, G. Assayag, V. Cafagna, M. Chemillier (eds.), Formal Systems and Music special issue, Soft Computing 8, pp. 1432-7643, September 2004.

[6] Assayag, G., Dubnov, S., Delerue, O., *Guessing the Composer's Mind: Applying Universal Prediction to Musical Style,* Proc. Int'l Computer Music Conf., Int'l Computer Music Assoc., pp. 496-499, 1999.

[7] Bloch, G., Chabot X., Dannenberg, R., *A Workstation in Live Performance: Composed Improvization*, Proceedings of International Computer Music Conference, The Hague, Netherlands, 1986.

[8] Dubnov, S., Assayag, G., El-Yaniv, R. *Universal Classification Applied to Musical Sequences,* Proc. Int'l Computer Music Conf., Int'l Computer Music Assoc., 1998, pp. 332-340.

[9] Dubnov, S. Assayag, G., *Improvization Planning and Jam Session Design using concepts of Sequence Variation and Flow Experience*, Proceedings of Sound and Music Computing '05, Salerno, Italy, 2005.

[10] Dubnov, S., Assayag, G., *Universal Prediction Applied to Stylistic Music Generation* in Mathematics and Music, A Diderot Mathematical Forum, Assayag, G.; Feichtinger, H.G.; Rodrigues, J.F. (Eds.), pp.147-160, Springer-Verlag, Berlin, 2002.

[11] Dubnov, S., Assayag, G., Lartillot, O., Bejerano, G., *Using Machine-Learning Methods for Musical Style Modeling*, IEEE Computer, Vol. 10, n° 38, p.73-80, October 2003.

[12] Conklin, D. *Music Generation from Statistical Models* , Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences, Aberystwyth, Wales, 30– 35, 2003.

[13] Cont, A., Dubnov, S., Assayag, G., *A framework for Anticipatory Machine Improvisation and Style Imitation*, Proceedings of the Third Workshop on Anticipatory Behavior in Adaptive Learning Systems (ABiALS 2006), Butz M.V., Sigaud O., Pezzulo G., Baldassarre G. (eds.) 2006.

[14] Cont, A., Dubnov, S., Assayag, G., *Anticipatory Model of Musical Style Imitation using Collaborative and Competitive Reinforcement Learning*, in Lecture Notes in Computer Science, Anticipatory Behavior in Adaptive Learning Systems: From Brains to Individual and Social Behavior. Springer Verlag, to appear 2007.

[15] Francois, A.R.J, Chew, E., Thurmond D., *MIMI - A Musical Improvisation System That Provides Visual Feedback to the Performer* , technical report 07-889 PDF, University of Southern California Computer Sciences Dpt, http://www.cs.usc.edu/Research/ReportsList.htm#2006, 2006

[16] Lefebvre, A., Lecroq, T., *Computing repeated factors with a factor oracle.* In L. Brankovic and J. Ryan, editors, Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms, pages 145–158, Hunter Valley, Australia, 2000.

[17] Mancheron, Alban; Moan, Christophe, *Combinatorial Caracterization of the Language recognized by Factor and Suffix Oracle*, International Journal of Foundations of Computer Science, World Scientific 16 (6), 1179--1191, (2005)