2016

factorOracle: an Extensible Max External for Investigating Applications of the Factor Oracle Automaton in Real-Time Music Improvisation

Adam James Wilson
CUNY New York City College of Technology

Recommended Citation
Wilson, Adam James. "factorOracle: an Extensible Max External for Investigating Applications of the Factor Oracle Automaton in Real-Time Music Improvisation," in Proceedings of the 4th International Workshop on Musical Metacreation (MUME 2016), held at the Seventh International Conference on Computational Creativity (ICCC 2016), Paris, France, June 27 - July 1, 2016. ISBN: 978-0-86491-397-5.

MUME 2016 - The Fourth International Workshop on Musical Metacreation, ISBN #978-0-86491-397-5

factorOracle: an Extensible Max External for Investigating Applications of the Factor Oracle Automaton in Real-Time Music Improvisation

Adam James Wilson
awilson@citytech.cuny.edu
Emerging Media Technology Program, Department of Entertainment Technology
New York City College of Technology, City University of New York
Brooklyn, NY

Abstract

There are several extant software systems designed to generate music in real-time using a factor oracle automaton constructed from the musical input of a human improvisor. The impetus for the design of the *factorOracle* external is neither a desire to supersede these systems nor introduce novel algorithms for traversing the oracle, but rather to provide a fast, canonical interface for the automaton in Cycling74's Max and, in future iterations, the Pure Data programming environment. Technical features of the *factorOracle* software are introduced here.

Background

The factor oracle is a directed acyclic word graph capable of expressing at least all of the substrings of a given word. The oracle can be built incrementally, in *O(m)* with respect to both processing time and memory consumption (Allauzen, Crochemore, and Raffinot 1999). The input can be a chain of pitches, durations, notes, collections of audio samples, or other musical features. The *factorOracle* external (1) continuously builds the oracle for a set of incoming musical data, and (2) generates variations on the stored data by exploiting features of the oracle that identify repeated patterns in the input.

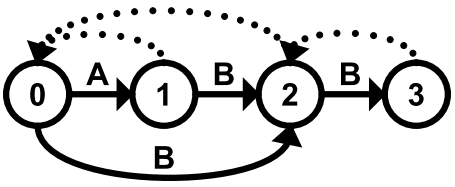

Figure 1: Factor oracle of *ABB*

In constructing the factor oracle, elements of the input word become transitions between states in the graph, as shown in Figure 1. A state may also have forward transitions to one or more non-contiguous states (for instance, *B* connects state 0 to state 2 in Figure 1). Each transition of this kind ends a sequence of transitions (including only the transition itself if the originating state is 0) that shares a suffix with a substring of the original word terminated by the state to which the transition points. Backward links, called suffix links (shown by dotted lines in Figure 1), trace the path of a supply function that determines where state-skipping transitions will be added.

The oracle is initialized in Algorithm 2 by creating state zero and setting its supply function $S_\varepsilon(0)$ to -1. Algorithm 1 is then called for each subsequent element of the input word.

Algorithm 1 Function addLetter(Oracle($p = p_1\ p_2\ ...\ p_m$), σ) (Allauzen, Crochemore, and Raffinot 1999)

Create a new state $m+1$
Create a transition σ from m to $m+1$
$k \leftarrow S_p(m)$
while $k > -1$ and there is no transition σ from k **do**
    Create a new transition σ from k to $m+1$
    $k \leftarrow S_p(k)$
end while
if $k == -1$ **then**
    $s \leftarrow 0$
else
    $s \leftarrow$ the state reached by transition σ from k
end if
$S_{p\sigma}(m+1) \leftarrow s$
return Oracle($p = p_1\ p_2\ ...\ p_m\sigma$)

Algorithm 2 Function buildOracle($p = p_1\ p_2\ ...\ p_m$) (Allauzen, Crochemore, and Raffinot 1999)

Create Oracle(ε) with a single state 0
$S_\varepsilon(0) \leftarrow -1$
for $i \leftarrow 1$ to m **do**
    Oracle($p = p_1\ p_2\ ...\ p_i$) $\leftarrow$ addLetter(Oracle($p = p_1\ p_2\ ...\ p_{i-1}$), p_i)
end for

For each new transition σ and state $m+1$, we follow the suffix link for state m back to an earlier state k (this action describes the supply function $S_p()$ shown in Algorithm 1). If there is no transition σ from the earlier state to the new state, we add it. We continue to follow the suffix links back and add transition σ when needed, until we reach state -1 or a state k_{final} that has a transition σ to some other state. In the

former case, we add a suffix link from $m+1$ to state 0, and in the latter we add a suffix link from $m+1$ to the state pointed to by σ from k_{final}.

It is important to note that all of the systems that use the factor oracle to generate music turn the oracle into a cyclic graph, allowing navigation via suffix links or reverse suffix links (Assayag et al. 2006; Assayag and Bloch 2007; Cont, Assayag, and Dubnov 2007; Dubnov and Surges 2013; Dubnov and Wang 2014; Dubnov, Hsu, and Wang 2015). When both reverse and forward suffix links are used, the graph also becomes partially undirected, violating another constraint of the originally proposed structure. Storage of suffix links for use post-construction increases the memory footprint of the automaton, which was developed in part as a means of data compression.

Suffix links, however, appear to be the most important feature of the oracle for recombination of musical data, since they connect the ends of substrings that have the longest repeated suffix with respect to the substring ended by the origin state of the suffix link. The smoothness of the shift between musical passages depends upon the length of the merged segment (longer == smoother). A method for determining the length of the longest repeated suffix is described in (Lefebvre and Lecroq 2000), and is employed in the generative systems previously cited.

Since the oracle is built incrementally and in real time, it is ideal for alternation between learning structure and generating music (Assayag et al. 2006), and can therefore be developed as an artificial improvisation partner, beginning with the modifications described above. Prior systems have been designed to work with both symbolic music and audio data; these employ various methods for identifying information of interest in the oracle. Instantaneous style reproduction and pattern matching are the primary applications.

Rationale

The software available for generating music with the factor oracle seems to require older or reducible technological dependencies. OMax is a collection of nine externals, which appear to be unsupported for Max versions 6 and above (Levy 2012), and PyOracle's real-time component apparently requires embedding Python in Max (Dubnov and Surges 2013). The *factorOracle* external (see the new alpha-phase source code[1]) came about from a desire for an implementation that is:

- fast—written purely in C, as opposed to Java or an interpreted language
- simple in function and design
- integrated with one or more of the most popular interactive real-time audio environments (Max, Pure Data) as a single external
- usable by a non-expert without modification or extension
- not beholden to any particular generative or analytical model, but extensible by an expert user through patching or source code modification

[1] https://github.com/ajwnycct/factorOracle

Features

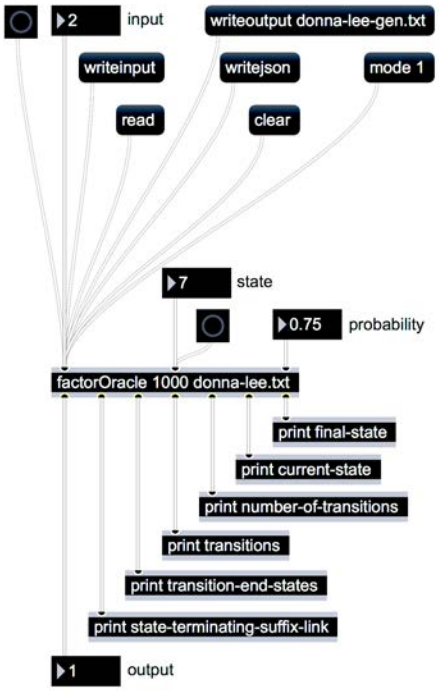

Figure 2: Max patch showing all *factorOracle* I/O options.

Construction

Data used to build the oracle are sent in the left inlet of the external (see Figure 2). Older versions of the software supported multiple data types: symbols, floats, integers, and lists (Wilson 2009; Brent and Wilson 2012). In this version, input is limited to integers, reducing code maintenance overhead and increasing performance. The onus is on the user to provide, when necessary, a suitable encoder/decoder to translate information going into and coming out of the oracle; polyphonic musical data, for instance, must be organized into meaningful structures reconstituted as integers.

In a recent performance incorporating the *factorOracle* external (see Figure 3), MIDI information was mapped to a matrix of 19 [0-18] equal-tempered pitch classes by 24 [0-23] durations. Each pitch/rhythm pair (p,r) was assigned a cross-alphabet letter c by $c = (p*24+r)$, and decoded by $r = (c \bmod 24), p = (\lfloor c/24 \rfloor \bmod 19)$ (Artinian and Wilson 2016). A cross-alphabet of any dimensionality could be similarly expressed, provided the total number of cells in the matrix is less than the largest representable integer. For example, a multidimensional matrix of audio features could be sectioned into a finite number of "geographical" regions, and analysis frames from an input signal could be assigned numbers corresponding to their locations within the matrix. Of course, establishing the resolution and shape of such regions would be an important factor in determining the vaildity of the resulting graph.

Internally, *factorOracle* uses a linked list of state objects to store data. The struct for each state m includes:

- the immediate transition to state $m+1$
- the number of outgoing transitions
- the indices of states terminating the outgoing transitions
- the index of the state terminating the suffix link from state m

Note that it is not necessary to store both the state indices and the transitions themselves, since a transition from state m to state $m+n$ is equal to the transition stored at state $m+n-1$ (Allauzen, Crochemore, and Raffinot 1999). In a subsequent version, each state will also store the longest repeated suffix indicated by its suffix link.

Figure 3: Performance of *Eighteen* at NYC EIS 2016. https://vimeo.com/adamjameswilson/eighteen, http://eis.nyc

Generation

There are two ways to generate output from the *factorOracle* external. (1) A bang in the left inlet will return a transition selected using a weighted random walk of the oracle graph structure, tempered by a few rules to avoid loops. (2) An integer m sent to the middle inlet will return all of the data associated with state object m; this data can then be passed to a custom process designed to choose another state or transition. An external process can be made to accumulate data for applying decisions based on historical choices as well as discovered features. Virtually any conceivable model governing analysis and traversal of the graph can harness the oracle in this way.

Data returned from the external when an integer is sent to the middle inlet include:

- the index of the current final state
- the index of the current state
- the total number of transitions from the current state
- the list of transitions from the current state
- the list of indices of states terminating transitions from the current state (the position of each index in this list can be used to look up its associated transition in the list of transitions above)
- the index of the state terminating the suffix link originating from the current state

When a bang is sent to the left inlet, transitions are chosen using an internal process selectable by sending a "mode" message with an integer argument, specifying the mode of traversal, to any inlet. There is at present only one such mode, which makes use of a probability value sent as a float to the rightmost inlet. This value can vary in real time. The probability of taking a suffix link S is related to the probability of taking a transition T by $P(S) = 1.0 - P(T)$. Uniform weights are used when choosing among transitions along forward links from the current state, including the link pointing to the next state.

When a suffix link is taken, the mode outputs the transition to the state immediately following the state reached by the suffix link, prohibiting forward skips until the next output request. This avoids one of the problems mentioned in (Assayag and Bloch 2007): if we allow the possibility of forward state skips, we can potentially follow a suffix link and then jump right back to the origin state of the suffix link, producing—in this implementation—unwanted repetitions of a single transition. Unlike Assayag and Bloch, we don't prohibit forward skips altogether, though perhaps this approach will be taken with a separate generative mode in a future version.

A second issue, noted in both (Assayag and Bloch 2007) and (Dubnov and Wang 2014), involves the meta-rhythmic "errors"—isolated metric idiom violations—that can arise from taking forward leaps or suffix links based on a very short common suffix. These produce jarring "interruptions" or accents, creating a relationship of output to input analogous to the relationship between a Picasso painting and its subject. If these stylistic anomalies are unacceptable to the user, the input word can be built from a cross-alphabet of metric groups of notes, instead of a cross-alphabet of individual notes, or a custom external ruleset designed to avoid incomplete beats can be programmed for use with the middle inlet.

Note that forcing the generative process to take an immediate forward transition after following a suffix link can lead to an infinite loop at the level of a single transition, in the case where the final state has a suffix link back to the previous state. To remedy this, we force the oracle to follow the chain of suffix links back from the terminal state until we reach a state with a suffix link going back more than one state. This is the only case in which we allow the generative state of the oracle to reach zero; in all other cases, we go back no further than state 1, since there is no suffix prior to state zero to exploit as a shared connection between substrings.

Examples

Figures 4-7 illustrate the variation in output with different probability values P for the internal generative mode. Figure 3 shows the original head for the tune *Donna Lee* by Charlie Parker, labelled with cross-alphabet values calculated from the sets of unique pitch (p) and rhythmic (r) values shown above the score. For example, middle-C, $p = 3$, with a duration of an eighth note, $r = 1$, is encoded as $(r * 25 + p) = 28$.

If a high degree of stylistic coherence is desired, using a higher value for P will produce longer segments of unrecombined material. Note that Figure 5 is stylistically very similar to the original, but recombined in a novel way, with just a few of the rhythmic anomalies discussed in the previous section.

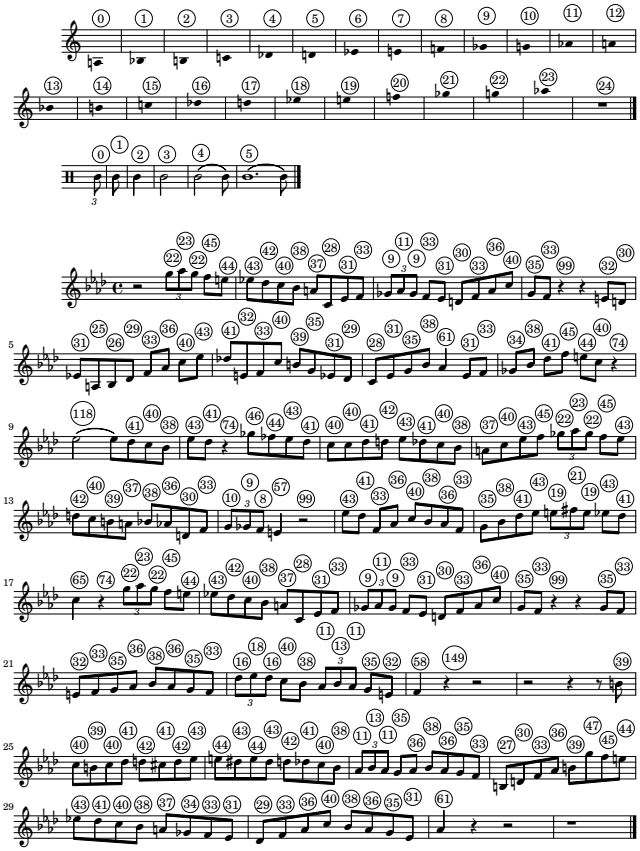

Figure 4: *Donna Lee* with cross-alphabet labels.

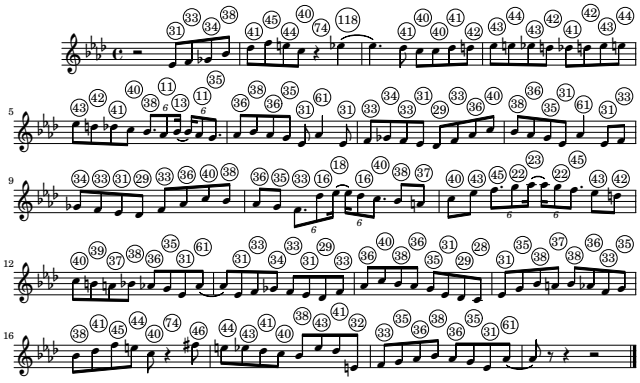

Figure 5: *Donna Lee* recombined with $P = 0.9$.

The anomalies result from the fact that our input alphabet is comprised of single notes, rather than metric groupings of notes; the 1-beat triplet embellishments in the original are occasionally broken up, causing an interpenetration of duple and triple meters.

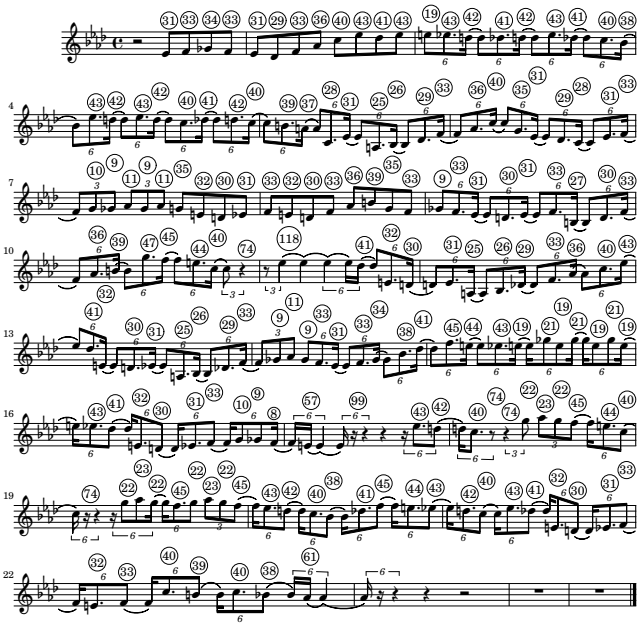

Figure 6: *Donna Lee* recombined with $P = 0.5$.

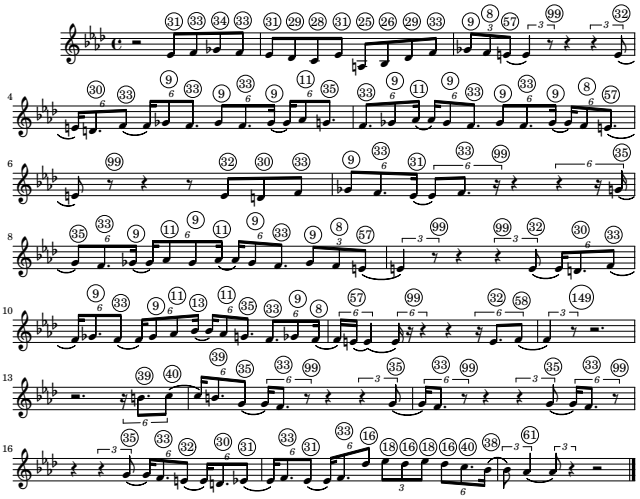

Figure 7: *Donna Lee* recombined with $P = 0.1$.

This effect is more pronounced in Figures 6 and 7, where incomplete triplet figures produce long chains of metric displacement. Using a lower probability in conjunction with an alphabet that ignores metrical grouping can be interesting if the goal is to distort the input phraseology. Again, this is an idiosyncrasy of the one internal generative mode developed thus far, and can be circumvented. A patch could be made, for example, to use the center inlet for choosing the next transition, returning a transition only if it completes the beat and following a suffix link if no such transition exists.

File I/O

The *factorOracle* external can write the input string and the generated output string to disk. These files are simple space-separated lists of integers. The oracle can read these lists of integers in two different ways: the second argument in the body of the object icon can be a read-in file in the Max path, or a "read" message can be sent to any inlet, with or without a file argument.

The first argument in the body of the object specifies the number of states anticipated. Reserving memory for a number of states vastly improves performance by avoiding frequent allocating and copying of memory blocks to resize buffers. The default internal number of states is 10K; if the user specifies both a state number limit and an input file, enough memory is allocated for both the input file and the number of states requested in the first argument. Finally, a "writejson" message in any inlet writes the graph to disk in a custom JSON format.

The top-level hash keys shown in Figure 8 are state indices. The value of each of these keys is an array containing a hash of end-state–transition key-value pairs, followed by the index of the state terminating the suffix link. The motivation behind the JSON hash is to provide a standard way of exporting the oracle graph for use in other programming environments, such as Processing.

```
{"0":[{"1":"0","2":"1"},"-1"],
 "1":[{"2":"1"},"0"],
 "2":[{"3":"1"},"0"],
 "3":[{},"2"]}
```

Figure 8: JSON representation for factor oracle of *ABB*.

Planned Improvements

The most recent version of *factorOracle* is an alpha phase release, verified to work in Max 7 in 32-bit mode on Mac OS Mavericks, and is available in an open-source BSD-licensed version[1]. A more stable release will be available soon.

In terms of new development, longest repeated suffix computation is first in the queue. A trickier problem is how to incorporate data from files that are read in after some performance input, or how to incorporate performance input after a file is read in. One idea is to merge the two based on a longest repeated suffix that preserves the most data from both sets of information, which means waiting for or expecting some minimum accumulation of performance data before fully integrating a read-in file. Performance data and file data are presently concatenated without concern for any discontinuity at the point of connection.

Finally, I would like to produce a series of help files showing how to build patches around the external using many of the analytical and generative methods described in the list of references here. One important caveat to note: although the objective of this software is to provide an interface for both constructing the factor oracle *and* using it to generate music in a real-time audio scheduling environment, the extent to which a patch incorporating the *factorOracle* external can generate music in real time will be contingent on the computational demands of the generative process employed.

References

Allauzen, C.; Crochemore, M.; and Raffinot, M. 1999. Factor oracle: A new structure for pattern matching. *Lecture Notes in Computer Science* 1725:295–310.

Artinian, A., and Wilson, A. 2016. Eighteen. 27 February 2016. New York City Electroacoustic Improvisation Summit, Voorhees Theater, New York City College of Technology, Brooklyn. Performance.

Assayag, G., and Bloch, G. 2007. Navigating the oracle: a heuristic approach. In *Proceedings of the International Conference on Music Computing*.

Assayag, G.; Bloch, G.; Chemellier, M.; Cont, A.; and Dubnov, S. 2006. OMax brothers: a dynamic topology of agents for improvisation learning. In *Workshop on Audio and Music Computing for Multimedia, ACM Multimedia*.

Brent, W., and Wilson, A. 2012. Cross-talk: A shared parameter space for gesturally extended human/machine improvisation. In *Proceedings of the Ammerman Center 13th Biennial Symposium on Arts and Technology*, 22–26.

Cont, A.; Assayag, G.; and Dubnov, S. 2007. Audio oracle: A new algorithm for fast learning of audio structures. In *Proceedings of the International Computer Music Conference*.

Dubnov, S., and Surges, G. 2013. Feature selection and composition using PyOracle. In *Proceedings of the 2nd International Workshop on Musical Metacreation, Ninth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Dubnov, S., and Wang, C. 2014. Guided music synthesis with variable markov oracle. In *Proceedings of the 3rd International Workshop on Musical Metacreation, Tenth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*.

Dubnov, S.; Hsu, J.; and Wang, C. 2015. Music pattern discovery with variable markov oracle: a unified approach to symbolic and audio representations. In *Proceedings of the 16th International Society for Music Information Retrieval Conference*, 176–182.

Lefebvre, A., and Lecroq, T. 2000. Computing repeated factors with a factor oracle. In *Proceedings of the 11th Australasian Workshop on Combinatorial Algorithms*, 145–158.

Levy, B. 2012. OMax: the software improviser, version 4.5.x. http://repmus.ircam.fr/_media/omax/documentation_-omax_4.5.pdf. Accessed: 2016-04-30.

Wilson, A. 2009. *Automatic Improvisation: A Study in Human/Machine Collaboration*. Ph.D. Dissertation, University of California, San Diego.