Optimization of audio graphs by resampling
Pierre Donat-Bouillud, Jean-Louis Giavitto, Florent Jacquemard

To cite this version:
Pierre Donat-Bouillud, Jean-Louis Giavitto, Florent Jacquemard. Optimization of audio graphs by resampling. DAFx-19 - 22nd International Conference on Digital Audio Effects, Sep 2019, Birmingham, United Kingdom. hal-02284258

HAL Id: hal-02284258
https://inria.hal.science/hal-02284258
Submitted on 11 Sep 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
OPTIMIZATION OF AUDIO GRAPHS BY RESAMPLING

1. INTRODUCTION

Interactive music systems (IMS) are programmable systems that combine audio signal processing with control in real-time. At run time, during a concert, they process audio signals in real-time, using various audio effects. For that purpose, they periodically fill audio buffers and send them to the soundcard. They also make it possible to control the sound processing tasks, with aperiodic control (such as changes in a graphical interface) or periodic control (for instance, with a low frequency oscillator). Audio signals and controls are dealt with by an audio graph whose nodes represent audio processing tasks (filters, oscillators, synthesizers,...) and edges represent dependencies between these audio processing tasks.

Puredata and Max/MSP are examples of IMSs. They graphically display the audio graph, but modifying it at run time as a result of a computation can be complicated. Other IMSs, such as ChucK or SuperCollider are textual programming languages. They are also more dynamic. In Antescofo, human musicians and a computer can interact on stage during a concert, using sophisticated synchronization strategies specified in an augmented score, programmed with a dedicated language, that can also specify dynamic audio graphs.

Real-time constraints for audio: Audio samples must be written into the input buffer of the soundcard periodically. The buffer size can range from 32 samples for dedicated audio workstations to 2048 samples for some smartphones running Android, depending on the target latency and the resources of the host system. For a sample rate of 44.1kHz, such as in a CD, and a buffer size of 64 samples, the audio period is 1.45 ms. It means that the audio processing tasks in the audio graph are not activated for each sample but for a buffer of samples.

IMSs are not safety critical systems: a failure during a performance is not life-critical, it will not generally result in damages or injuries. However, audio real-time processing has strong real-time constraints. Missing the deadline for an audio task, i.e. the time allotted by the audio driver to fill its buffer, is immediately audible:

- Buffer underflow: The audio driver uses a circular buffer the size of which is a multiple of the size of the soundcard buffer. If the task misses a deadline, it does not fill the buffer quickly enough. Depending on the implementation, previous buffers will be replayed (gun machine effect) or silence will be played, which entails cracks or clicks due to discontinuities in the signal. A larger buffer decreases deadline misses but increases latency.

- Buffer overflow: In some implementations, filling the buffer too quickly can also lead to discontinuities in the audio signal, if audio samples cannot be stored to be consumed later. On the contrary, in video processing, missing a frame among 24 images per second does not entail a visible decrease in quality so that lots of streaming protocols accept to drop a frame. Therefore, real-time audio constraints are more stringent than for video. Yet, they have not been investigated as much as real-time video processing.

Composers and musicians use IMS on mainstream operating systems such as Windows, macOS or Linux, where a reliable and tight estimation of the worst case execution time (WCET) is difficult to obtain, because of the complex hierarchy of caches of the CPU. Therefore, running audio and video real-time streams on the same computer is challenging.

Figure 1: Histogram of relative deadlines for the audio callback on a MacBookPro with macOS. We execute a C++ test program generating a sawtooth signal for 10s. The time budgets range from 3.64 ms to 3.89 ms, i.e. a 254 µs jitter, with a mean of 3.78 ms.
Figure 2: Histogram of the execution time for the same code generating a saw signal for 10 s. Here, we show the execution time at each cycle in the audio callback for generating a sawtooth signal on a MacBookPro with macOS. The execution times range from 4.74 µs to 25.20 µs with an average of 9.17 µs. The standard deviation is 1.59 µs.

processor, because there are usually no real-time scheduler or temporal isolation among tasks, and because it is difficult to predict which tasks will be executed at a given time. Mainstream operating systems are not real-time systems and do not offer any strong guarantees on deadlines for audio processing (see Fig. 1) or on the execution times (see Fig. 2). Applications that perform audio computations have to compete for CPU and memory resources with other applications during a typical execution. Those platforms are also often on batteries and often change the frequency of the CPUs in order to save energy, thus totally changing execution times. It follows that we cannot assume that we know the WCET of tasks. Furthermore, IMSs are more and more ported to embedded cards such as Raspberry Pi and have to adapt to limited computation resources on these platforms. On the other hand, composers and sound designers create more and more complex musical pieces, with lots of dynamically interconnected nodes, using a sampling rate up to 96 kHz.

Hard real-time scheduling algorithms that depend on knowing the WCET cannot be applied here, as in practice, IMSs are executed on those operating systems without strong real-time guarantees.

When real-time constraints are not critical, modifying the quality of service (QoS) by partially executing some tasks or even discarding them is an option to consider. In the case of IMSs, tasks are dependent, with dependencies defined with the edges of the audio graph. The quality of a task is itself position-dependent. It depends on the position of the audio processing task in a path going from an audio input to an audio output. It means we cannot merely discard or degrade any task in the audio graph, to achieve an optimal QoS adaptation.

To deal with these challenges, IMSs have rather chosen to increase the available computation resources, in particular by increasing the parallelism of the audio graphs in scores to take advantage of the multicore processors. For example, an alternative scheduler for SuperCollider, Supernova \cite{10}, can use several cores to synthesize sounds. However, these new schedulers do not automatically parallelize the audio graph but require explicit instructions, such as ParGroup for SuperCollider, or poly for Max/MSP, and so are difficult to program.

In this paper, we propose an alternative solution where we explore how an audio graph can be optimized by degrading audio processing units without perceiving the degradations (or minimizing the degradation perception). The idea is to generate an equivalent but degraded audio graph where the execution time of the graph is decreased while decreasing the quality of the audio processing tasks.

Because of the dependencies, we do not degrade only a single task here, but we have to choose a whole set of dependent nodes in an audio graph to degrade. We aim at finding paths to degrade in the audio graph, where the quality of a task is position-dependent. The nodes are seen as blackboxes and degradations are made by resampling the signal flowing between nodes.

We describe a model of an audio graph based on the dataflow paradigm \cite{11} with models of execution time and quality. We present an offline algorithm based on the models that explores exhaustively or randomly the possible degraded versions of an audio graph with constraints on execution time and quality. Heuristics help to select subpaths to degrade in the audio graph at execution time. We evaluate our optimization strategy on three different sets of audio graphs: an exhaustive enumeration of all possible graphs with few nodes, a random sampling of graph with many nodes, and graphs structurally generated from Puredata patches.

2. RELATED WORK

Degrading computations to achieve to respect some time criterion has been dealt with in the approximate programming paradigm or in real-time system theory, with mixed criticality.

Approximate computing \cite{12} is a paradigm in which errors are allowed in exchange of an improvement of performance. The concept of correctness is relaxed to a correctness with a quantitative error. In \cite{13}, a graph is used to represent a map-reduce program, with map nodes which compute and reduce nodes which aggregate data. Offline, they generate approximate versions of the graph with a given precision. For that purpose, two kinds of transformations are considered: substitution transformations and sampling transformations, where the input is randomly downsampled. However, this model is aimed at batch-processing, not real-time audio graphs.

In multimedia systems, a basic strategy \cite{14} related to mixed criticality consists of dividing tasks between a mandatory and an optional part, which can be discarded in case of overload of the processor. Real-time scheduling with this strategy does not entail too much overhead but does not handle dependencies between tasks, in particular for quality estimation. Some mixed criticality approaches address graph-based tasks for mixed-criticality systems, such as in \cite{15}. However, the dependencies between tasks are functional dependencies: all tasks in a graph have the same criticity but it is possible to switch to other graphs with another criticity. Criticity levels are not like qualities that would depend on the topology of the graph.

\footnote{There is an earliest deadline first scheduler \cite{9} on Linux, but it’s typically not activated on mainstream distributions.}
3. MODEL OF AN AUDIO GRAPH

3.1. Model of an audio graph

In an audio graph, audio streams flow between signal processing nodes at various rates, depending on what the nodes require. The dataflow model [11] is well suited to describe these kinds of dependent tasks. In the usual dataflow model, no time information is provided, so we enrich this dynamic dataflow model with temporal information, such as start times or average execution times (ACET), as in the Time-Triggered Dataflow mode [17]. We also define a quality measure on the graph.

3.2. Dataflow model

The dataflow model is data-oriented: when there are enough data, seen as a sequence of tokens, on a node, the node is fired, consequently, yielding some tokens. More formally, we use the port graph formalism, as in [15]. A dataflow quadruplet $G = (V, P, E, \mu)$ where the vertices in $V$, represent the signal processing nodes and are pairs $(I, O)$ with $I \subset P$ and $O \subset P$, the input and output ports. The edges in $E \subset P \times P$ represent the data flowing between vertices and connect an output port of a vertex to an input port of another vertex. We note $p_1 \rightarrow p_2$ the edge between ports $p_1$ and $p_2$; $v_1 \rightarrow v_2$ would denote any edge $v_1.p \rightarrow v_2.p'$ between $v_1$ and $v_2$. We note $v.p$ the port $p$ of vertex $v$. The function $\mu : P \rightarrow \mathbb{N}$ maps a port to the number of audio samples it consumes or produces.

**Distinguished nodes** The nodes without input ports are called inputs or sources. They are typically audio stream generators. The nodes without output ports are the outputs or sinks, as shown in Fig. 3. They are audio sinks to the soundcard for instance. Nodes that are neither inputs nor outputs are called effects.

Figure 3: A simple synchronous dataflow graph with three nodes $v_1$, $v_2$ and $v_3$ with only one input port and one output port for each of them. Data flows from $v_1$ to $v_2$, from $v_1$ to $v_2$ and from $v_2$ to $v_3$. $v_1$ yields 1 token per firing, and $v_3$ requires 2 tokens to be fired. $v_1$ is a source node, $v_3$ a sink node.

3.3. Timed dataflow

A dataflow graph only describes the partial ordering of the firing of its nodes, but not their time firing instances. However, dataflow graphs are often used to describe real-time processing, including audio signal real-time processing. In the following, input nodes are given firing dates and output nodes, deadlines. Any node $v$ gets an average case execution time (ACET), $A_v$, and a worst case execution time (WCET), $W_v$. Although we will evaluate the model with the ACET, we could also evaluate with the WCET if we had an accurate WCET. We will note $T_v$ the execution time of a node. We also assume that the graph is acyclic.

The audio processing nodes are periodic and we note $s_v^1 \ldots s_v^n$ with $s_v^i < s_v^{i+1}$ the firing times of node $v$, where $s_v^{i+1} - s_v^i = T_v$ with $T_v$ the period of node $v$. The samplerate $f_p$, on a port $p_v$ of node $v$ is $\frac{1}{T_v}$.

**Execution time of a path** On a path $\pi = v_1 \rightarrow \ldots v_n$, the mean execution time (resp. worst case) is $\sum_{i=1}^n A_{v_i}$ (resp. $\sum_{i=1}^n W_{v_i}$).

**Execution time of the whole graph** For the whole graph $G$, on a parallel system with enough resources, the execution time is the largest execution time of a path between inputs and outputs, i.e. for the ACET, $A_G = \max_{\pi \subset G} \{A_{v_1 \rightarrow v_3}, A_{v_1 \rightarrow v_3} \}$

In a sequential execution mode (for instance on a uniprocessor), it is the sum of the execution times of all the nodes:

$$A_G = \sum_{v \in G} A_v \quad (1)$$

**Estimating $A_v$** We measure the average execution time $A_v$ of all the possible nodes that can be part of the audio graph. We do not care about the exact execution time, but rather of an ordering on the execution times between various versions of an audio graph. In addition, the execution time increases monotonically with the buffer input sizes. Thus, we only measure the average execution of a given buffer size. However, some nodes can have a variable number of inputs and outputs, such as a mixer node. We do not want to measure all possible combinations of inputs and outputs. Experimentally, we find that the average case execution time $A_{\text{mixer}}(n_{\text{inputs}}, n_{\text{outputs}})$ of a mixer with $n_{\text{inputs}}$ and $n_{\text{outputs}}$ is given by:

$$A_{\text{mixer}}(n_{\text{inputs}}, n_{\text{outputs}}) = n_{\text{inputs}} \times (A_{\text{mixer}}(2, 1) - A_{\text{mixer}}(1, 1)) \quad \text{cost of adding}$$

$$+ n_{\text{outputs}} \times (A_{\text{mixer}}(1, 2) - A_{\text{mixer}}(1, 1)) \quad \text{cost of copying one buffer} \quad (2)$$

3.4. Quality

The quality of the audio graph is a subjective matter, and relates to psychoacoustics. It depends on the semantics of the nodes. We aim at designing an a priori quality measure based on parameters of the audio effects, which is more practical to compute in real-time than analysing the audio signal.

The quality measure should be compositional, i.e. the quality of the graph $q_G \in [0, 1]$ must be a function of the quality of its nodes and edges. The worst quality is 0 and the best quality is 1. For each node $v$, we also note $q_v \in [0, 1]$ its quality.

We define the quality $q_{v_1 \rightarrow \ldots \rightarrow v_n}$ on a path $v_1 \rightarrow \ldots v_n$ as $q_{v_1} \otimes \cdots \otimes q_{v_n}$ for an operator $\otimes$ with the following properties:

**Associativity** $q_{v_1} \otimes (q_{v_2} \otimes q_{v_3}) = (q_{v_1} \otimes q_{v_2}) \otimes q_{v_3}$

**Decreasing** $q_v \otimes q_{v'} \leq q_v$. It means that quality never increases on the path, as the information lost by degrading cannot be rebuilt.

**Identity element** There is an identity element $1_\otimes$ such that $1_\otimes \otimes v = v = v \otimes 1_\otimes = v$. Such an element is the quality which preserves for the output the quality of its input.
An obvious choice for an operator fulfilling these desired properties is multiplication on real numbers. For \( v \rightarrow v' \):

\[
q_{v \rightarrow v'} = q_v \otimes q' = q_v \times q'\v
\]

On the path \( v_1 \rightarrow \cdots \rightarrow v_n \), thanks to associativity:

\[
q_{v_1 \rightarrow \cdots \rightarrow v_n} = \prod_{i=1}^{n} q_{v_i} \quad (3)
\]

We also define a join operator \( \oplus \) that models the quality resulting from joining two paths, such as \( v_1 \rightarrow v_3 \) and \( v_1 \rightarrow v_2 \rightarrow v_3 \) on Fig. [4]

\[
q_{\oplus} = q_{v_1 \rightarrow v_3} \oplus q_{v_1 \rightarrow v_2 \rightarrow v_3} = (q_{v_1} \otimes q_{v_3}) \oplus (q_{v_1} \otimes q_{v_2} \otimes q_{v_3}) \quad (4)
\]

In practice, we choose \( \sum_{i=1}^{n} q_k = \frac{1}{n} \sum_{i=1}^{n} q_k \) for \( n \) joining paths for mixer-like nodes and \( \oplus = \min \) for the other nodes.

4. OPTIMIZATION BY RESAMPLING

We consider a method of degrading the quality which is agnostic of the actual computation node semantics, as it operates on the signal that flows in-between the nodes by resampling parts of the audio graph. The rationale for resampling is based on the following cognitive observations. Above some frequency threshold, no quality improvement is perceived by human beings, according to psychoacoustics studies [19]. Indeed, the human auditory system cannot perceive frequency above 20 kHz. Shannon’s theorem implies that the sampling rate must be at least double of the maximum frequency, so about the sampling rate of audio CDs of 44.10 kHz. However, resampling makes it possible to better handle rounding errors in the signal processing and that is why we also consider frequencies higher than 44.10 kHz.

4.1. Resampling a signal path

4.1.1. Inserting resampling nodes

In a dataflow graph, nodes are fed with samples. The premise here is that a node which receives less samples will take less time to be executed. Changing the number of samples per time unit is a common operation in signal processing, called resampling: getting less is downsampling, and more, upsampling. In order to resample, we insert resampling nodes in the graph. These nodes have the same attributes and properties as other nodes: they can interoperate between samples, copy samples in audio buffers. They have the same attributes and properties as other nodes: they can interoperate between samples, copy samples in audio buffers. They have the quality measure and temporal characteristics such as a ACET or WCET, so that we can take into account the overhead due to inserting those nodes. Every resampling node has one input port and one output port.

When a downsampling node is inserted on a path, all the following nodes in the path operate on a downsampled signal. We need to insert an upsampling node if there is a node on that path that enforces a specific samplerate, typically, a sink to the soundcard.

We consider a path \( \pi = v_1 \rightarrow \cdots \rightarrow v_n \), where for a node \( v_k \) in path \( \pi, v_k = (p_{k, j}, \{p_{k, j}'\}) \) with \( p_{k, j}^{i,k} \) the input ports and \( p_{k, j}' \) the output ports. Thus, the degraded path \( \pi' = v_1 \rightarrow v_2' \rightarrow \cdots \rightarrow v_n' \rightarrow v_n \). We want to insert downsampling node \( v'_1 \) just after \( v_1 \) and an upsampling node \( v''_n \) just before \( v_n \), as shown in Fig. 4.

![Figure 4: Inserting a downsampling node \( v'_1 \) between nodes \( v_1 \) and \( v_2 \). \( v_1 \) has an output port \( p_1^i \), \( v_2 \) has an input port \( p_2^i \) and \( v'_1 \) has an input port \( p_{1'}^i \) and an output port \( p_{1''}^i \).](image1)

Node \( v \) is on path \( v_1 \rightarrow \cdots \rightarrow v_n \). The resampled signal flows on this path through input port \( p_{1'}^i \) with resampling factor \( q \). Node \( v \) has another input port, \( p_2 \). The signal coming into this port must also be resampled with resampling factor \( r \).

In case a node on the path has several input ports and that one of them receives a resampled signal, we also have to resample by the same resampling factor the other input ports, as shown in Fig. [5] by inserting a resampling node connected to this input port.

Also, if an output port \( p \) of node \( v \) is connected to several input ports \( p_1, \ldots, p_n \), it is more efficient to insert the resampler and then a node with \( n \) outputs, instead of inserting a resampler on each edge \( p \rightarrow p_k \), as shown in Fig. [6].

4.1.2. Execution times

We assume that the computation nodes process all their incoming samples and hence, that the complexity of their computations is at least linear. So we can bound execution times of \( v_k \) for \( k \in [2, n - 1] \) for a downsampler with resampling factor \( 1/t \):

\[
A'_{v_k} \leq \frac{1}{t} \times A_{v_k} \quad (5)
\]

\[
W'_{v_k} \leq \frac{1}{t} \times W_{v_k} \quad (6)
\]

We can deduce a bound on the whole execution times of the degraded path \( \pi' \):

\[
A'_v \leq A_{v_1} + A_{v'_1} + \frac{1}{t} \sum_{k=2}^{n-1} A_{v_k} + A_{v''_n} + A_{v_n} \quad (7)
\]

that is to say for the subpath:

\[
A'_v \leq A_{v_1} + A_{v'_1} + \frac{1}{t} A_{v_2 \rightarrow \cdots \rightarrow v_n} + A_{v''_n} + A_{v_n} \quad (8)
\]
Enumerating all the possible degraded versions of the audio graph is an optimization problem under constraints. The tool is an open source OCaml program that accepts Puredata or MAX/MSP patches, or a custom format as input and can output one optimized version or a set of optimized versions.

Note that we take into account the execution times of the inserted nodes, so that the optimization is overhead-aware. For small graphs with audio processing nodes with an execution time in the same order of magnitude of the resamplers, the execution time of a degraded graph can be actually larger than the non-degraded one.

4.1.3. Quality

The a priori quality measure in the case of resampling is the samplerate: the lower the sampling rate, the lower the quality. In the case of path \( \pi \), we have \( f_\pi = \frac{q \times v(\pi)}{v(\pi)} \). If audio is sent too late to the output buffer, a click can be heard. We consider that it is worse to hear a click because of missing the deadline of the audio driver than to hear a resampled signal. A node that would entail always missing deadlines is given the worst possible quality. The quality \( q_v \) of a downsampled node is such that \( q_v < 1 \). The quality of a non-downsampled node is 1.

4.2. Degraded versions

Choosing the nodes to degrade in the audio graph is an optimization problem under constraints. We can try to maximize the quality of the audio graph given an execution time constraint (a deadline), or minimize the execution time of the audio graph given a minimum target quality.

Our system can enumerate all the possible degraded versions of a graph, or a random sampling of the degraded versions, or use heuristics to compute a useful subset that respects some constraints. The tool is an open source OCaml program that accepts Puredata or MAX/MSP patches, or a custom format as input and can output one optimized version or a set of optimized versions.

4.2.1. Exhaustively

Enumerating all the possible degraded versions of the audio graph is enumerating all the possible subsets of nodes that can be degraded in the audio graph, i.e. all nodes except sources and sinks.

4.2.2. Heuristics

The idea is to degrade nodes starting from the outputs, as inserting a downsampling node near the end of the branch impact less nodes than at the beginning.

We choose to have at most one resampled subpath per branch. We also try to minimize the number of inserted nodes with respect to the number of degraded nodes on the resampled subpath, in order to minimize the overhead due to resampling nodes. For that, we need to minimize the number of branches where we resample and that is why we explore the graph branch per branch.

We can use this heuristics to find approximate solutions the optimization problem of maximizing the quality with an execution time constraint. For that, we start from one of the output nodes, and we traverse the graph backwards and see how inserting a downsampling node on a path going to this output node would change the estimated remaining execution time, until the estimation of remaining execution time is lower than the remaining time before the deadline. Other branches can be explored if it is not enough. Then the downsampling and upsampling nodes are inserted on the paths chosen to be resampled.

5. EXPERIMENTAL EVALUATION

In order to evaluate our theoretical models, we need to instantiate our models on a large number of graphs. However, there are no reference benchmarks of audio graphs for IMS. We decided to generate a huge number of graphs, compute the theoretical execution time (in Sect. 5.3) and quality (in Sect. 5.4) of each graph, and then measure the execution time and a quality based on the actual audio signal. We then compare the theoretical values and the measured ones.

The resamplers in use for these experiments are the linear resamplers, and we only resample by 2.

![Figure 7: The experimental setup to evaluate the models of quality and execution time.](https://github.com/programlyrique/ims-analysis)
5.1. Measuring execution time and quality

We developed a open source rust program\[[1]\] that can execute the audio graph, monitor execution times of the audio callback, and output the resulting audio. The resampling is handled by libsamplespl\[[2]\]. This is an opensource library that makes it possible to downsample down to a 256 ratio, and to upsample up to a 256 ratio. The sampling rate can be changed in real-time. The library provides five resamplers, classified by decreasing order of quality: best, medium, faster sync resamplers (as in [20], zero order hold resampler, and a linear resampler.

**Measuring execution time** We execute the audio graph for a large number of cycles and drop the first cycles to take into account processor cache warming.

**Measuring quality** We compare the audio signal of a degraded graph and the one of the non-degraded graph. Given the spectrum of each signal, we compute their constant-Q transform\[[21]\] using the Erdős–Rényi\[[23]\] random graph mode, where a graph can be chosen uniformly at random from the graphs with \( n \) nodes and \( M \) edges, or with \( n \) nodes and a given probability of having an edge between two edges.

From Puredata patches Audio graphs tend to exhibit a particular structure: few incoming and outgoing edges per node, creating long chains in the audio graph, with a few nodes with more inputs that typically mix signals. To take into account this structure, we parsed all the Puredata patches of its tutorial and examples (i.e. 133 graphs).

**Picking the actual nodes** We maintain a database of possible audio processing nodes, with their estimated execution time, their numbers of input ports and output ports, and their possible control parameters. The possible parameter values can be annotated with a range, \([0,1]\) for instance for a volume, or a set, for instance \((20,440,1000,2500,6000)\) for a set of frequencies.

5.2. Comparing the model and the measurements

5.2.1. Graph generation

We generate the structure of the graphs first, i.e. vertices and edges, and then pick actual audio processors for each vertex in a node dictionary.

**Exhaustive generation for a given number of nodes** We enumerate all the non-labelled WCDAGs with \( n \) vertices. Non-labelled entails that \( a \rightarrow b \) and \( b \rightarrow a \) are isomorphic, are the same graphs. Given the set of vertices \( V = \{0, \ldots, n-1\} \), we undertake the following steps:

1. Compute the set of all the possible edges \( E \) between distinct vertices in one direction. For that, we remark that pairs \((a_1, \ldots, a_n)\) is isomorphic. The following operations reduct the number of graphs so we keep that upper bound.

Thus the powerset has size \( O(2^n) \). The following operations reduce the number of graphs so we keep that upper bound.

**Random generation** As the increase in the number of graphs is over-exponential, it becomes untractable when \( n > 6 \) in practice. For \( n = 7 \) for instance, there are 3781503 possible DAGs. Hence, for larger number of nodes, we randomly generate graphs using the Erdős–Rényi\[[23]\] random graph mode, where a graph can be chosen uniformly at random from the graphs with \( n \) nodes and \( M \) edges, or with \( n \) nodes and a given probability of having an edge between two edges.

5.2.2. Comparing rankings

After generating the graphs, we can both compute the theoretical qualities and execution times, and measure the actual ones. We obtain two permutations of our set of graphs and we need to know how far the two permutations are.

To compare the similarity of the orderings of the graphs ranked by the theoretical and measured quantities, we use rank correlation. The Kendall rank correlation coefficient (Kendall’s tau coefficient)\[[23]\] is linked to the number of inversions needed to transform one ordering into the other one. The Spearman’s rank correlation coefficient (or Spearman’s rho)\[[25]\] is linked to the distance
We compare the position in the two orderings of the worst quality graphs, the shortest execution time and the longest execution time.

5.2.3. Results

Exhaustive enumeration For Fig. 9, we enumerated all the audio graphs with 5 nodes, that is to say, 838 graphs. The average number of versions of a non-degraded graph (including the non-degraded graph) is $3.657518 \pm 2.066005$ and there are 57 graphs without degraded versions. The theoretical models and the measurements both of execution time and quality are well correlated. There are so many correlations in the $0.9 - 1$ bin because many 5 nodes graphs have few degraded versions.

Only 22, i.e. 2%, degraded graphs are quicker to execute than their degraded versions here when using nodes with execution time in the same order of magnitude as the resamplers. Indeed, in that case, the length of a resampled branch must be large for it to execute quicker than the non-degraded version. However, small graphs do not exhibit long branches. On the contrary, when using nodes an order of magnitude bigger, 275, i.e. 33% degraded graphs are quicker than their non-degraded version.

Large random graphs For Fig. 10, we generated 100 random graphs with 10 nodes and a $0.3$ edge probability, in order to have nodes with not too many inputs and outputs, as in real audio graphs. The execution time model is rather accurate, with most correlations above $0.5$. For the quality, the results are less good, as there are lots of correlations close to 0, but mainly strictly positive. With quick nodes, 15 graphs have degraded versions quicker than their non-degraded versions, whereas with slow nodes, there are 45.

With graphs from Puredata For Fig. 11, we generated 133 audio graphs from 133 Puredata patches. The model of execution time is accurate, which is especially visible with the Spearman correlation coefficient histogram. The model of quality and the measurement of qualities are better correlated than for random graphs. With quick nodes, 48 graphs, i.e. 36%, have degraded versions which are quicker than the non-degraded graph. The structure of Puredata graphs exhibit longer branches and more subsets of the graph that can be resampled with just a few resamplers.

Hence, large graphs without too many ramifications, and with nodes with execution times at least an order of magnitude higher than the execution time of a resampler take advantage the most of the resampling optimization. Actual audio graphs from IMSs, such as the ones from Puredata, have actually these characteristics.

6. CONCLUSION

We proposed a model of an audio graph with execution time and quality, and an optimization algorithm based on the models that can find degraded versions of an audio graph while respecting constraints on execution time or quality. The degradations are based on downsampling parts of the graph.
We evaluated the models and the algorithm on audio graphs that were exhaustively enumerated for graph with few nodes, randomly generated for larger graphs, and generated from Puredata patches. The execution time model is quite accurate. The quality model is well correlated for small graphs (Fig. 9), has to be improved on large random graphs (Fig. 10) and promising results are obtained for graphs (Fig. 11) generated from real Puredata patches. The optimization is particularly effective for actual audio graphs with long audio effects and long branches. Our choice of measures of quality is arbitrary. Another option would be to organize listening tests, but would be impractical considering the huge size of the set of possible graphs and possible degraded graphs.

The optimization program can be integrated to the design of an audio graph, using our custom audio graph format that describes nodes and their connections as input and output and can import Puredata patches but we aim at making it easier to use within MAX/MSP and Puredata. We also want to integrate our optimization as a compiler optimization for Faust programs in the Faust compiler [25]. It would unlock access to many more different audio graphs. As the Faust instructions are both very fine-grained and well-defined, we would be able to refine the execution time and the quality models.

7. ACKNOWLEDGMENTS

This work started during a visit in the group of Prof. Christoph Kirsch in Salzburg, whom we would like to thank for his advice. We also would like to thank the anonymous reviewers for their insights and advice.

8. REFERENCES