SEARCHING FOR MUSIC MIXING GRAPHS: A PRUNING APPROACH

Sungho Lee[†*], *Marco A. Martínez-Ramírez*[♮], *Wei-Hsiang Liao*[♮], *Stefan Uhlich*[♯], *Giorgio Fabbro*[♯], *Kyogu Lee*[†], *and Yuki Mitsufuji*[♮♭]

[†]Department of Intelligence and Information, Seoul National University, Seoul, South Korea
[♮]Sony AI, Tokyo, Japan [♯]Sony Europe B.V., Stuttgart, Germany [♭]Sony Group Corporation, Tokyo, Japan

ABSTRACT

Music mixing is *compositional* — experts combine multiple audio processors to achieve a cohesive mix from dry source tracks. We propose a method to reverse engineer this process from the input and output audio. First, we create a mixing console that applies all available processors to every chain. Then, after the initial console parameter optimization, we alternate between removing redundant processors and fine-tuning. We achieve this through differentiable implementation of both processors and pruning. Consequently, we find a sparse mixing graph that achieves nearly identical matching quality of the full mixing console. We apply this procedure to dry-mix pairs from various datasets and collect graphs that also can be used to train neural networks for music mixing applications.

1. INTRODUCTION

Motivation — From a signal processing perspective, modern music is more than the mere sum of source tracks. Mixing engineers combine and control multiple processors to balance the sources in terms of loudness, frequency content, spatialization, and much more. Many attempts have been made to uncover parts of this intricate process. Some have gathered expert knowledge [1, 2] and built rule-based systems [3, 4]. More recent work has adopted data-driven approaches. Neural networks have been trained to map source tracks directly to a mix [5, 6] or to estimate parameters of a fixed processing chain [7]. Yet, efforts to address the compositional aspects of the music mixing, such as which processors to use for each track, are still limited. One possible remedy is to consider a graph representation whose nodes and edges are processors and connections between them, respectively. In other words, each graph contains the essential information about the mixing process. However, other than the dry source and mixed audio, no public dataset provides such mixing graphs or related metadata [8, 9, 10], which hinders this line of research. This is not surprising; besides the cost of crowdsourcing, it is difficult to standardize the mixing data from multiple engineers with different equipment. One recent work [11] sidestepped this issue by creating synthetic graphs and using them for training. However, this approach is not free with downsides. Neural networks would suffer from poor generalization unless the synthetic data distribution matches the real world. Similar data-related issues arise in different domains, e.g., audio effect chain recognition [12, 13] and synthesizer sound matching [14, 15, 16]. Furthermore, real-world multitrack mixes have a much larger number of source tracks and graph sizes, making synthetic data generation more challenging. Therefore, it is desirable to have a systematic, reliable, and scalable method for collecting graphs. All these contexts lead us to ask: *Can we find the mixing graphs solely from audio?*

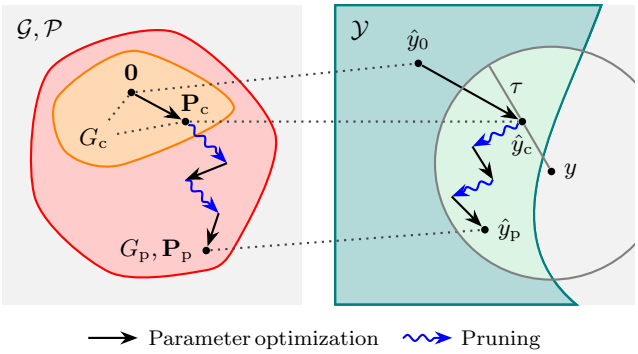


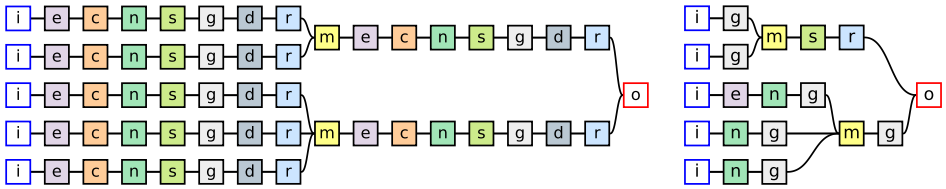
Figure 1: Music mixing graph search via iterative pruning.

Problem definition — Precisely, for each song (piece) whose dry sources $s_1, \cdots, s_K$ and mix y are available, we aim to find an audio processing graph G and its processor parameters $\mathbf{P}$ so that processing the dry sources $s_1, \cdots, s_K$ results in a mix $\hat{y}$ that closely matches the original mix y. With a loss L_a that measures the match quality on the mixture audio domain $\mathcal{Y}$ and regularization L_r, our objective can be written as follows,

$$G^*, \mathbf{P}^* = \arg\min_{G,\mathbf{P}} \big[L_\mathrm{a}(\hat{y}, y) + L_\mathrm{r}(G, \mathbf{P})\big]. \tag{1}$$

Contributions — One might want to explore the candidate graphs without any restriction. However, this makes the problem ill-posed and underdetermined. The graph's combinatorial nature makes the search space $\mathcal{G}$ extremely large. Furthermore, we have to find the processor parameters jointly. As a result, numerous pairs of graphs and parameters can have similar match quality. Therefore, it is desirable to add some restrictions, e.g., preferring structures that are widely used by practitioners. To this end, we resort to the following pruning-based search; see Figure 1 for a visual illustration. Inspired by a recent work [17], we first create a so-called "mixing console" G_c; see Figure 2a for an example. It applies a fixed processing chain to each source. Then, it subgroups the outputs, applies the chain again, and sums the processed subgroups to obtain a final mix $\hat{y}$. This resembles the traditional hybrid mixing console [18]. Each chain comprises 7 processors, including an equalizer, compressor, and multitap delay. We implement all of them in a differentiable manner [19, 20, 21]. This allows end-to-end optimization of all parameters $\mathbf{P}_\mathrm{c}$ with an audio-domain loss L_a via gradient descent. After this initial console training, we proceed to the pruning stage. Here, we search for a maximally pruned graph G_p and its parameters $\mathbf{P}_\mathrm{p}$ while maintaining the match quality of the mixing console up to a certain tolerance τ; this is visualized as

[*] Work done during internship at Sony AI.



(a) Full mixing console (before pruning) (b) Pruned graph

Figure 2: Finding the sparse graph G_{p} from the differentiable mixing console G_{c}. Initial letters in the nodes denote their respective types. `i`: input, `o`: output, `m`: mix, `e`: equalizer, `c`: compressor, `n`: noisegate, `s`: stereo imager, `g`: gain/panning, `r`: reverb, and `d`: multitap delay.

a circle centered at y in Figure 1. Also, see Figure 2b for an example pruned graph. We use iterative pruning, alternating between the pruning and fine-tuning, i.e., optimization of the remaining parameters [22]. To collect graphs from multiple songs, it is crucial to make the entire search process, i.e., parameter optimization and pruning, efficient and fast. To this end, we first introduce batched processing of multiple nodes in a single graph, which speeds up the computation of the mix. Next, we investigate methods for efficient and effective pruning. During the pruning, we need to find a subset of nodes that can be removed. To achieve this, we view each processor's "dry/wet" parameter as an approximate importance score and use it to select the candidate nodes. This approach gives 3 variants of the pruning method with different trade-offs between the computational cost and resulting sparsity. It also draws connections to neural network pruning [23, 24] where the binary pruning operation is relaxed to continuous weights. Note that casting the graph search to pruning is a double-edged sword. The pruning only removes the processors and does not consider all possible signal routings, reducing the search space (shown as shaded regions in Figure 1). As a result, it does not improve the match quality over the mixing consoles. Nevertheless, the pruned graph follows the real-world practice of selectively applying appropriate processors. In other words, the sparsity is crucial for the graph's interpretability. Also, it keeps the search cost in a practical range, which might be challenging with other alternatives [25, 26]. Our method serves as a standalone reverse engineering algorithm [17], but it can also be used for collecting pseudo-label data to train neural networks for music mixing applications. For example, we may extend existing methods for automatic mixing [3, 4, 5, 6, 7, 27] and mixing style transfer [28] to output the graphs. This allows the end users to interpret and control the estimated outputs while leveraging the power of deep neural networks.

Data — We first report a list of datasets to which we can apply our method. For each song, we need a pair of dry sources $s_1, \cdots, s_K$ and a final mixture y. Additionally, we use subgrouping information that describes how dry tracks are grouped together. Therefore, we use the `MedleyDB` dataset [8, 9] as it provides all of them. We also add the `MixingSecrets` library [10]. Since it only provides the audio, we manually subgrouped each track based on its instrument. Finally, we include an `Internal` dataset. The resulting ensemble consists of 1129 songs (188, 472, and 579 songs for each respective dataset). The number of dry tracks ranges from 1 to 133, and the number of subgroups ranges from 1 to 26 (see Figure 6 for the statistics). Except for the final pruned graph collection stage (Section 3.4), we use a random subset for the evaluations (a total of 72 songs, 24 songs for each dataset). Every signal is stereo and resampled to 30kHz sampling rate.

Supplementary materials — Refer to the following link for audio samples, pruned graphs, and appendices that contain technical details: `https://sh-lee97.github.io/grafx-prune`.

2. DIFFERENTIABLE PROCESSING ON GRAPHS

An audio processing graph $G = (V, E)$ is assumed to be directed and acyclic (V and E denote the set of nodes and edges, respectively). Each node $v_i \in V$ is either a processor or an auxiliary module and has its type t_i, e.g., `e` for an equalizer. Each processor takes an audio u_i and a parameter vector p_i as input and outputs a processed signal $f_i(u_i, p_i)$. Then, we further mix the input and this processed result with a "dry/wet" weight $w_i \in [0, 1]$. Hence, the output y_i of the processor v_i is given as follows,

$$y_i = w_i f_i(u_i, p_i) + (1 - w_i)u_i. \tag{2}$$

We have the following 3 auxiliary modules:

- **Input —** It outputs one of the dry sources s_k.
- **Mix —** We ouput the sum of incomming signals.
- **Output —** A sum of its inputs is considered as a final output y.

Each edge $e_{ij} \in E$ represents a "cable" that sends an output signal to another node as input. Throughout the text, we denote an ordered collection from multiple nodes with a boldface letter, e.g., $\mathbf{w}$ for a weight vector, $\mathbf{S}$ for a source tensor, and $\mathbf{P}$ for a dictionary with processor types as keys and their parameter tensors as values. Under this notation, our task is to find G, $\mathbf{P}$, and $\mathbf{w}$ from $\mathbf{S}$ and y.

2.1. Differentiable Implementation

Considering the music mixing context, we use the following 7 processor types. Refer to Appendix B for the full technical details.

- **Gain/panning —** We control both loudness and stereo panning of input audio by multiplying a learnable scalar to each channel.
- **Stereo imager —** We change the stereo width of the input by modifying the loudness of the side channel (left minus right).
- **Equalizer —** We use a finite impulse response (FIR) with a length of 2047 to modify the input's magnitude response. The FIR is parameterized with its log magnitude (thus 1024 parameters). We apply inverse FFT of the magnitude with zero phase, obtain a zero-centered FIR, and multiply it with a Hann window. We apply the same FIR to both the left and right channels.
- **Reverb —** We employ 2 seconds of filtered noise as an impulse response for reverberation. First, we create a 2-channel uniform noise, where these channels represent the mid and side. We filter the noise by multiplying an element-wise 2-channel magnitude mask to its short-time Fourier transform (STFT), where the

FFT sizes and hop lengths are 384 and 192, respectively. This mask is constructed using the reverberation's initial and decaying log magnitudes. After the masking, we obtain the mid/side filtered noise via inverse STFT, convert it to stereo, and perform channel-wise convolutions with input to get an output.

- **Compressor —** We use a slight variant of the recently proposed differentiable dynamic range compressor [21]. First, we obtain the input's smooth energy envelope. The smoothing is typically done with a ballistics filter, but we instead use a one-pole filter for speedup in GPU. Then, we compute the desired gain reduction from the envelope and apply it to the input audio.
- **Noisegate —** Except for the gain computation, its implementation is the same as the compressor.
- **Multitap delay —** For each (left and right) channel, we employ independent 2 seconds of delay effects with a single delay for every 100ms interval. To optimize delay lengths using gradient descent, we employ surrogate complex damped sinusoids [29]. Each sinusoid is converted to a delayed soft impulse via inverse FFT. Its angular frequency represents a continuous relaxation of the discrete delay length. Each delay is filtered with a length-39 FIR equalizer to mimic the filtered echo effect [30].

Batched node processing — It is common to compute the graph output signal by processing each node one by one [15, 19]. However, this severely bottlenecks the computation speed for large mixing graphs. Therefore, we instead batch-process multiple nodes in parallel. For the graph in Figure 2b, we can batch-process 1 equalizer `e`, 3 noisegates `n`, and 5 gain/pannings `g` sequentially. Then, we aggregate the intermediate outputs to 2 subgroup mixes `m` (also in parallel). This part is identical to graph neural networks' "message passing," so we adopt their implementations [31]. We repeat these parallel computations until we reach the output node `o`. By doing so, we obtain the output faster; in this example, the number of sequential processing is reduced from 15 (one-by-one) to 8 (optimal). Also, our benchmark shows that up to $5.8\times$ speedup can be achieved for the pruned graphs with a `RTX3090` GPU. Refer to Appendix C for the implementation and benchmark details.

2.2. Mixing Console

We construct a mixing console G_c as follows (see Figure 2a).

(i) We add an input node `i` for each source track.
(ii) We connect a serial chain (with a fixed order) of an equalizer `e`, compressor `c`, noisegate `n`, stereo imager `s`, gain/panning `g`, multitap delay `d`, and reverb `r` for each input.
(iii) We subgroup the processed tracks with mix nodes `m`.
(iv) We apply the same chain `ecnsgdr` to each mix output, then pass it to the output node `o` (we omit the mix module here).

2.3. Optimization

Before exploring the pruning of each mixing console, as a sanity check, we first evaluate its match performance. To investigate how much each processor type contributes to the match quality, we start with a base graph, a mixing console with no processors, i.e., it simply sums all the inputs. Then, we add each processor type one by one to the processor chain (see the first column of Table 1). We optimize and evaluate all these preliminary graphs for each song. For each graph, we train its parameters and weights simultaneously with an audio-domain loss given as follows,

$$L_\mathrm{a} = \alpha_\mathrm{lr} L_\mathrm{lr} + \alpha_\mathrm{m} L_\mathrm{m} + \alpha_\mathrm{s} L_\mathrm{s} \tag{3}$$

		L_a	L_lr	L_m	L_s
Base graph (sum of dry sources)		19.7	1.52	1.46	74.3
+ Gain/panning	`g`	.689	.686	.634	.752
+ Stereo imager	`sg`	.676	.671	.623	.734
+ Equalizer	`e sg`	.557	.549	.493	.637
+ Reverb	`e sg r`	.481	.471	.457	.523
+ Compressor	`ec sg r`	.423	.407	.385	.492
+ Noisegate	`ecnsg r`	.414	.398	.375	.485
+ Multitap delay (full)	`ecnsgdr`	.409	.395	.375	.469

Table 1: Matching performances of the mixing consoles with different processor type configurations. The strings, e.g., `ecnsgdr`, denote the processors used in the chain and their orders.

where each term L_x is a variant of multi-resolution STFT loss [32] ($\mathrm{x} \in \{\mathrm{lr}, \mathrm{m}, \mathrm{s}\}$, lr: left/right, m: mid, s: side)

$$L_\mathrm{x} = \sum_{i=1}^{I} \left[\frac{\| \log Y_\mathrm{x}^{(i)} - \log \hat{Y}_\mathrm{x}^{(i)} \|_1}{N} + \frac{\| Y_\mathrm{x}^{(i)} - \hat{Y}_\mathrm{x}^{(i)} \|_F}{\| Y_\mathrm{x}^{(i)} \|_F} \right]. \tag{4}$$

Here, $Y_\mathrm{x}^{(i)}$ and $\hat{Y}_\mathrm{x}^{(i)}$ denote the i^th Mel spectrograms of the target and predicted mixture, respectively. N, $\| \cdot \|_1$, and $\| \cdot \|_F$ denote the number of frames, l_1 norm and Frobenius norm, respectively. We use FFT sizes of 512, 1024, and 4096, and hop sizes are $1/4$ of their respective FFT sizes. The number of Mel filterbanks is set to 96 for all scales. We apply A-weighting before each STFT [33]. The per-channel loss weights are set to $\alpha_\mathrm{lr} = 0.5$, $\alpha_\mathrm{m} = 0.25$, and $\alpha_\mathrm{s} = 0.25$. The implementation is based on `auraloss` [34]. We further add a regularization that promotes gain-staging, a common practice of audio engineers that keeps the total energy of input and output roughly the same. This is achieved with the following loss:

$$L_\mathrm{g} = \sum_{v_i \in V_\mathrm{g}} |\log \| f_i(u_i)_\mathrm{m} \|_2 - \log \| u_{i,\mathrm{m}} \|_2 | \tag{5}$$

where $(\cdot)_\mathrm{m}$ and $\| \cdot \|_2$ denote mid channel and l_2 norm, respectively. We apply this regularization to a subset of processors $V_\mathrm{g} \subset V$ that comprises all equalizers, reverbs, and multitap delays. This allows us to (i) eliminate redundant gains that these linear-time invariant (LTI) processors could create and (ii) restrict the parameters to be in a reasonable range. Therefore, the total loss is given as

$$L(\mathbf{P}, \mathbf{w}) = L_\mathrm{a}(\mathbf{P}, \mathbf{w}) + \alpha_\mathrm{g} L_\mathrm{g}(\mathbf{P}) \tag{6}$$

where the gain-staging weight is set to $\alpha_\mathrm{g} = 10^{-3}$. Here, we used a slightly different notation from Equation 1 to emphasize what is optimized. Each console is optimized for 12k steps using AdamW [35] with a 0.01 learning rate. For each step, we random-sample a 3.8s region of dry soruces $\mathbf{S}$ (thus the batch size is 1), compute the mix $\hat{y}$, and compare its last 2.8s with the corresponding ground-truth y. Note that the first second is used only for the "warm-up" of the processors with long states such as compressors and reverbs. The evaluation metrics are calculated over the entire song.

2.4. Results

Table 1 reports the evaluation results. First, the base graph results in an audio loss L_a of 19.7. The side-channel loss L_s is especially large as most source tracks are close to mono (centered) while the target mixes have wide stereo images. With the gain/pannings and

stereo imagers, we can achieve "rough mixes" with a loss of 0.676. Then, we fill in the missing details with the remaining processor types. Every type improves the match, and the full mixing console reports a loss of 0.409. In addition, see the upper $2/3$ parts of Figure 7, which shows mid/side log-magnitude STFTs of the target mixes, matches of the mixing consoles, and their errors. We report the results with 3, 4, 6, and 7 processor types. For the results on every processor configuration and additional songs, see Figure 15-18 in Appendix E. From the spectrogram error plots, we can again confirm that each type improves the match performance. Furthermore, each song benefits more from different processor types. For example, for the song `RockSteady`, the multitap delays improve matching quality more than the reverbs (Figure 7b), which is different from the average trend. Yet, this is expected since the original mix heavily uses the delay effects. Finally, we note that mixes from `MixingSecrets` are more challenging to match than the others; it reports a mean audio loss of 0.545, while `MedleyDB` and `Internal` report 0.296 and 0.385, respectively.

3. MUSIC MIXING GRAPH SEARCH

Considering the full mixing console G_c as an upper bound in terms of the matching performance, we want to find a sparser graph with a similar match quality. We achieve this by pruning the console as much as possible while keeping the loss increase up to a tolerance threshold τ. This objective can be written as

$$\text{minimize } |V_\mathrm{p}| \quad \text{s.t.} \quad \min L_\mathrm{a}(G_\mathrm{p}) \leq \min L_\mathrm{a}(G_\mathrm{c}) + \tau. \quad (7)$$

Here, V_p denotes the pruned graph's node set. We define the pruning as removal of the nodes $V_\mathrm{c} \setminus V_\mathrm{p}$ and re-routing of their edges, in a way that is equivalent to setting every unused node to "bypass mode," i.e., zero weight $w_i = 0$ for $v_i \in V_\mathrm{c} \setminus V_\mathrm{p}$. Also, $\min(\cdot)$ signifies that we are (ideally) interested in the optimized audio loss. We only prune the processors, not the auxiliary modules; thus, we define a pruning ratio ρ as the number of pruned processors over the number of processors in the initial mixing console.

3.1. Iterative Pruning

Finding the optimal (sparsest) solution V_p^* is prohibitively expensive. First, due to the interaction between the processors, we need a combinatorial search. As such, we instead assume their independence and prune the processors in a greedy manner. Following the iterative approach [22], we gradually remove processors whenever the tolerance condition is satisfied. Under this setup, we still need to fine-tune intermediate pruned graphs before evaluating the tolerance condition. For reasonable computational complexity, we simply omit this fine-tuning, paying the cost of possibly missing more removable processors. Our method is summarized in Algorithm 1 (in the following parentheses denote line numbers). We first construct the mixing console G_c, optimize its parameters $\mathbf{P}$ and dry/wet weights $\mathbf{w}$, and evaluate the audio loss (1-3). This validation loss $L_\mathrm{a}^\mathrm{min}$ serves as a pruning threshold with the tolerance τ. Then, we alternate between pruning and fine-tuning, i.e., further optimization of the remaining parameters $\mathbf{P}$ and dry/wet weights $\mathbf{w}$ (5-18). Each stage consists of multiple pruning trials, which sample subsets of candidate processors $\bar{V}_\mathrm{cand}$ from the set of all remaining processors V_cand (8) and check whether they are removable (10). We keep the pruning if the result satisfies the constraint or cancel it otherwise (10-13). We repeat this process until the terminal condition (7) is satisfied. Implementation-wise,

Algorithm 1 Music mixing graph search with iterative pruning.

Input: A mixing console G_c, dry tracks $\mathbf{S}$, and mixture y
Output: Pruned graph G_p, parameters $\mathbf{P}$, and weights $\mathbf{w}$

1: $\mathbf{P}, \mathbf{w} \leftarrow \mathrm{Initialize}(G_\mathrm{c})$
2: $\mathbf{P}, \mathbf{w} \leftarrow \mathrm{Train}(G_\mathrm{c}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
3: $L_\mathrm{a}^\mathrm{min} \leftarrow \mathrm{Evaluate}(G_\mathrm{c}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
4: $G_\mathrm{p} \leftarrow G_\mathrm{c}$
5: **for** $n \leftarrow 1$ to N_iter **do**
6: $\quad V_\mathrm{cand}, \mathbf{m} \leftarrow \mathrm{GetAllProcessors}(V), \mathbf{1}$
7: $\quad$ **while** $\mathrm{TryPrune}(V_\mathrm{cand}, \mathbf{w}, \mathbf{m})$ **do**
8: $\quad\quad \bar{V}_\mathrm{cand}, \bar{\mathbf{m}} \leftarrow \mathrm{SampleCandidate}(V_\mathrm{cand}, \mathbf{w})$
9: $\quad\quad L_\mathrm{a} \leftarrow \mathrm{Evaluate}(G_\mathrm{p}, \mathbf{P}, \mathbf{w} \odot \mathbf{m} \odot \bar{\mathbf{m}}, \mathbf{S}, y)$
10: $\quad\quad$ **if** $L_\mathrm{a} < L_\mathrm{a}^\mathrm{min} + \tau$ **then**
11: $\quad\quad\quad L_\mathrm{a}^\mathrm{min} \leftarrow \min(L_\mathrm{a}^\mathrm{min}, L_\mathrm{a})$
12: $\quad\quad\quad \mathbf{m} \leftarrow \mathbf{m} \odot \bar{\mathbf{m}}$
13: $\quad\quad$ **end if**
14: $\quad\quad V_\mathrm{cand} = \mathrm{UpdatePool}(V_\mathrm{cand}, \bar{V}_\mathrm{cand})$
15: $\quad$ **end while**
16: $\quad G_\mathrm{p}, \mathbf{P}, \mathbf{w} \leftarrow \mathrm{Prune}(G_\mathrm{p}, \mathbf{P}, \mathbf{w}, \mathbf{m})$
17: $\quad \mathbf{P}, \mathbf{w} \leftarrow \mathrm{Train}(G_\mathrm{p}, \mathbf{P}, \mathbf{w}, \mathbf{S}, y)$
18: **end for**
19: **return** $G_\mathrm{p}, \mathbf{P}, \mathbf{w}$

we multiply binary masks, $\mathbf{m}$ and $\bar{\mathbf{m}}$, to the weight vector $\mathbf{w}$ to mimic the pruning during the trial stage (9). After the trials, we actually prune the graph, i.e., updating the graph and removing the pruned processors' parameters and weights, for faster search (16). Sometimes, albeit rare, the pruning can improve the match. In this case, we update the threshold (11).

3.2. Candidate Sampling

The remaining design choices are sampling an appropriate candidate set V_cand (8, 14) and deciding when to terminate the trials (7). We explore the following 3 approaches.

- **Brute-force —** We random-sample every processor one by one, i.e., $|\bar{V}_\mathrm{cand}| = 1$. This granularity could achieve high sparsity, but come with a large computational cost.
- **Dry/wet —** For efficient pruning, we need an informed guess of each node's importance. Intuitively, we can use each dry/wet weight w_i as an approximate importance score. This observation leads to the following method. For each pruning iteration:
  (i) We construct a set of remaining processor types T_cand.
  (ii) For each trial, we sample a type $t \in T_\mathrm{cand}$ and use the smallest-weight processors of that type as candidates. The number of candidates $|V_\mathrm{cand}|$ is set to 10% of the number of type-t processors that existed at the beginning.
  (iii) When the trial fails: if $|V_\mathrm{cand}| > 1$, we reduce the candidate set size to half: $|V_\mathrm{cand}|/2$. If $|V_\mathrm{cand}| = 1$, we finish the search of this type t, i.e., $T_\mathrm{cand} \leftarrow T_\mathrm{cand} \setminus \{t\}$.
  (iv) We iterate above two (ii)-(iii) until $T_\mathrm{cand} = \emptyset$.

  This way, we can skip large-weight nodes and evaluate multiple candidate nodes, reducing the total number of trials.
- **Hybrid —** Solely relying on the weight values could miss some processors that can be pruned but have large weights. We mitigate this by combining the above two, running the brute-force method for every 4^th iteration.

By default, we use the hybrid method with tolerance $\tau = 0.01$.

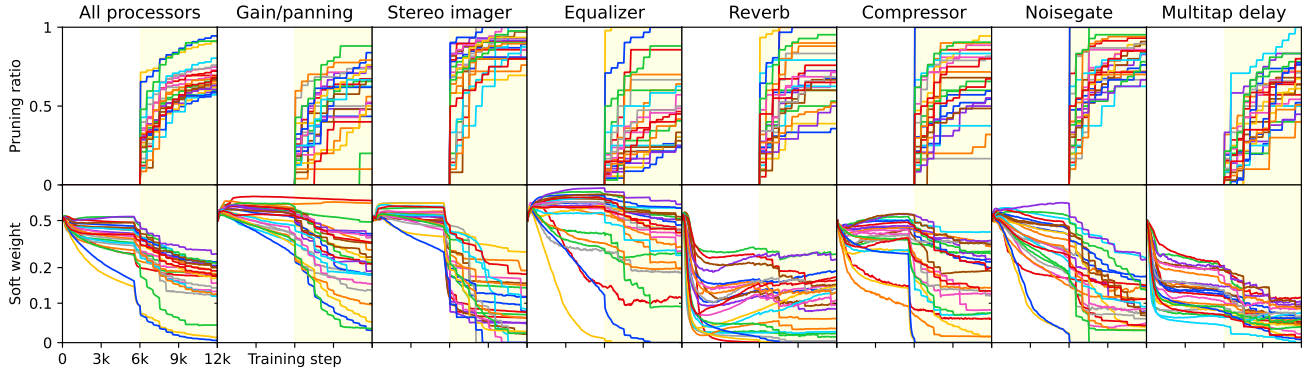


Figure 3: Iterative pruning process (hybrid, $\tau = 0.01$). 24 songs, 8 songs per dataset, are shown; each color represents an individual song. The upper and lower rows show the pruning ratios and mean dry/wet weights. The yellow-shaded regions show the pruning phase.

	τ	L_a	ρ	ρ_g	ρ_s	ρ_e	ρ_r	ρ_c	ρ_n	ρ_d
Mix console	–	.409	–	–	–	–	–	–	–	–
Brute-force	.01	.424	.69	.54	.85	.53	.76	.71	.78	.69
Dry/wet	.01	.420	.62	.51	.84	.38	.69	.66	.76	.53
Hybrid	.001	.411	.49	.35	.76	.27	.53	.57	.62	.34
	.01	.422	.67	.51	.86	.46	.71	.71	.79	.63
	.1	.499	.87	.73	.94	.81	.90	.85	.91	.92

Table 2: Pruning results of mixing consoles with various strategies and tolerance threshold τ.

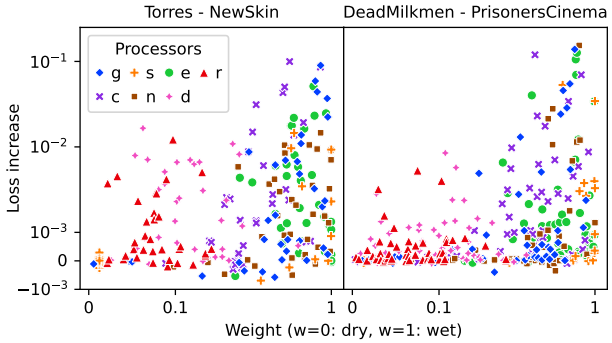


Figure 4: Each node's weight and loss increase when pruned.

3.3. Optimization

We use identical audio loss L_a and gain-staging regularization L_g. To promote sparsity, we add a weight regularization L_p, a l_1 norm of the weight $\mathbf{w}$. Hence, the full objective is as follows,

$$L(\mathbf{P}, \mathbf{w}) = L_\mathrm{a}(\mathbf{P}, \mathbf{w}) + \alpha_\mathrm{g} L_\mathrm{g}(\mathbf{P}) + \alpha_\mathrm{p} L_\mathrm{p}(\mathbf{w}). \quad (8)$$

We first train the console with 6k steps. Then, we repeat $N_\mathrm{iter} = 12$ rounds of pruning, each with 0.5k-step fine-tuning. As a result, the total number of optimization steps is the same as the previous console training. During the first 4k steps of the pruning phase, we linearly increase the sparsity coefficient α_p from 0 to 10^{-2}. While we halved the full console optimization steps, which could lead to increased loss, it is justified due to the tight resource constraints. With a `RTX3090` GPU, each song took about 56m, 29m, and 36m using the brute-force, dry/wet, and hybrid methods, respectively.

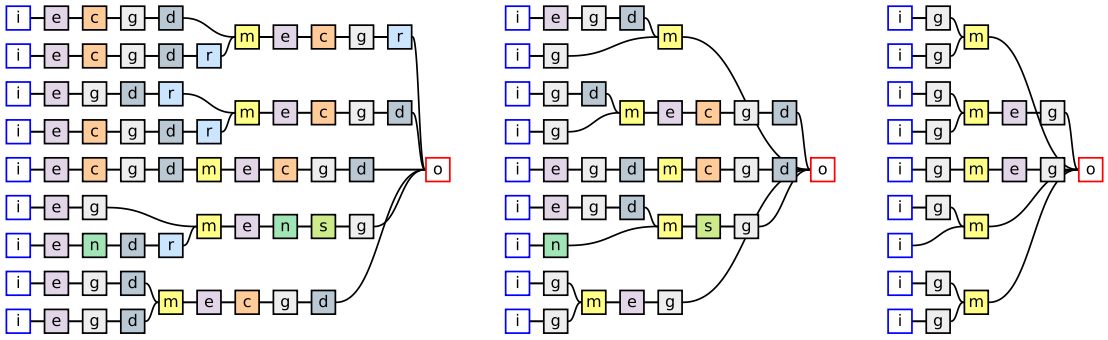
3.4. Results

Pruning process — Figure 2 shows how the pruning progresses. Each graph's sparsity increases gradually while its weights adapt over time. This trend is different for different processor types. The mean objective metrics are reported in Table 2. The default setting reports an average audio loss L_a of 0.422, an 0.013 increase from the full consoles, slightly exceeding the tolerance $\tau = 0.01$. This was expected due to the shorter full console training. The average pruning ratio ρ is 0.67 and the equalizer and stereo imager are the most and least remaining types (0.46 and 0.86), respectively. We note that `MedleyDB` and `MixingSecrets` report similar pruning ratios of 0.61 and 0.62, respectively. However, the `Internal` graphs are more sparse; their average pruning ratio is 0.77.

Sampling method comparison — Here, we fix the tolerance τ to 0.01 and compare the candidate sampling approaches; see Table 2. As expected, the brute-force method achieves the highest sparsity, reporting a pruning ratio of 0.69. Its average audio loss is also the highest, 0.424, an 0.015 increase from the mixing console result. The dry/wet method prunes the least with 0.62, 7% lower than the brute-force method. However, its audio loss is the lowest, 0.420, as more processors remained. We can investigate the cause of this difference in sparsity by analyzing the relationship between each dry/wet weight w_i and the loss increase Δ_i caused by pruning the processor v_i defined as follows,

$$\Delta_i = L_\mathrm{a}(G \setminus \{v_i\}) - L_\mathrm{a}(G). \quad (9)$$

Figure 4 shows scatterplots for 2 random-sampled songs, one for each song. Each point (w_i, Δ_i) corresponds to each processor after the initial console training. To maximize the sparsity using the dry/wet method, a monotonic relationship between the weights w_i and loss increases Δ_i is required, which is clearly not the case. Yet, a positive correlation exists, and this becomes more evident when we analyze the relationship for each type separately, justifying the per-type candidate selection. Still, we cannot completely remove the weakness of the dry/wet method, leading us to the hybrid strategy as a compromise. We note that the pruning methods are not only different in sparsity but also in trade-offs between sparsity and match performance. With more fine-grained analysis, we can observe that the brute-force method finds graphs with better matches even with the same graph size, closely followed by the hybrid method (see Figure 10 in Appendix E).



(a) $\tau = 0.001$
$\rho = 0.45, L_{\mathrm{a}} = 0.525$

(b) $\tau = 0.01$
$\rho = 0.74, L_{\mathrm{a}} = 0.525$

(c) $\tau = 0.1$
$\rho = 0.85, L_{\mathrm{a}} = 0.611$

Figure 5: Pruning results (hybrid method) with various tolerances. Song: `TablaBreakbeatScience_RockSteady`.

Choice of tolerance — Finally, we analyze the effect of the value of tolerance τ. Even with a very low tolerance $\tau = 0.001$, we can nearly halve the number of processors, i.e., $\rho = 0.49$. If we set the value too high, e.g., $\tau = 0.1$, the resulting graphs are highly sparse but degrade their matches ($L_{\mathrm{a}} = 0.499$, i.e., 0.090 increase). The default setting of $\tau = 0.01$ seems "just right," balancing the match performance and graph sparsity. We can verify this with the spectrogram errors (bottom 3 rows for each figure). There is no noticeable degradation from the full consoles to $\tau = 0.001$ and 0.01.

Case study — Here, we report the behavior of the pruning method from observations of the individual results.

- Recall that, for the song `RockSteady`, there was no clear performance improvement when we added the reverbs (Figure 7b). Hence, we can expect those reverbs to be pruned with a moderate tolerance τ. Figure 5 shows that this is indeed the case; only $5/14$ reverbs are left when $\tau = 0.001$ and $0/14$ for $\tau = 0.01$, which is much less than the average statistics (Table 2 and 3). When $\tau = 0.1$, processors for the details get removed; only the gain/pannings and equalizers remain. See captions in Figure 5 for the pruning ratios and audio losses of the pruned graphs (the full console has an audio loss of 0.523).
- The current pruning method fails to detect some redundant processors. In Figure 5b, the bottom 2 sources are processed with 3 gain/pannings. Since there is no nonlinear or time-varying processor between those, at least one can be "absorbed" by the others. While this case can be handled with some post-processing, it hints that we might have missed more sparse graphs.
- Each pruning of the same song yields a slightly different graph. Pruning a mixing console of `GirlOnABridge` multiple times resulted in graphs with the number of processors from 19 to 22 (Figure 11 in Appendix E). This is because our iterative pruning method is stochastic and greedy; candidates that were sampled early were more likely to be pruned.
- The pruning does not necessarily result in graphs that are close to the maximum loss $L_{\mathrm{a}}(G_{\mathrm{c}}) + \tau$. For `RockSteady`, pruning with $\tau = 0.01$ resulted in a loss of 0.525, much lower than the threshold. Interestingly, the $\tau = 0.001$ case achieved the same loss in spite of a much lower pruning ratio (0.56 versus 0.74).
- Processors for sources with short spans and low energy tend to get pruned as their contributions to the audio loss are small. Yet, we found that this could sometimes be perceptually noticeable.

	L_{a}	ρ	$\rho_{\mathtt{g}}$	$\rho_{\mathtt{s}}$	$\rho_{\mathtt{e}}$	$\rho_{\mathtt{r}}$	$\rho_{\mathtt{c}}$	$\rho_{\mathtt{n}}$	$\rho_{\mathtt{d}}$
`MedleyDB`	.431	.63	.37	.84	.44	.69	.74	.77	.57
`MixingSecrets`	.625	.64	.50	.87	.33	.64	.63	.80	.69
`Internal`	.434	.75	.70	.87	.55	.73	.85	.86	.72
`Total`	.506	.69	.57	.87	.45	.69	.75	.82	.69

Table 3: Pruning results with the default setting on the full dataset.

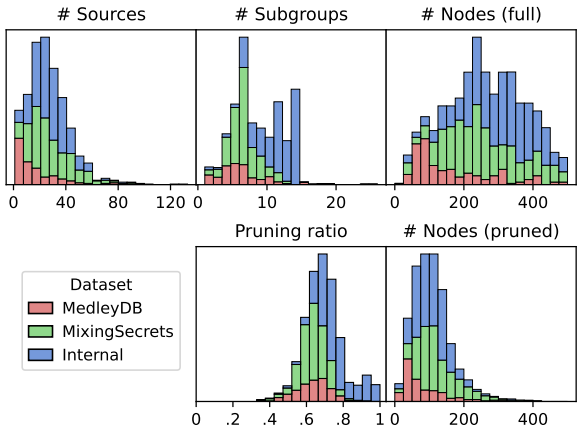


Figure 6: Statistics of the consoles and pruned graphs (full data). Each dataset's results are stacked to form the full histograms.

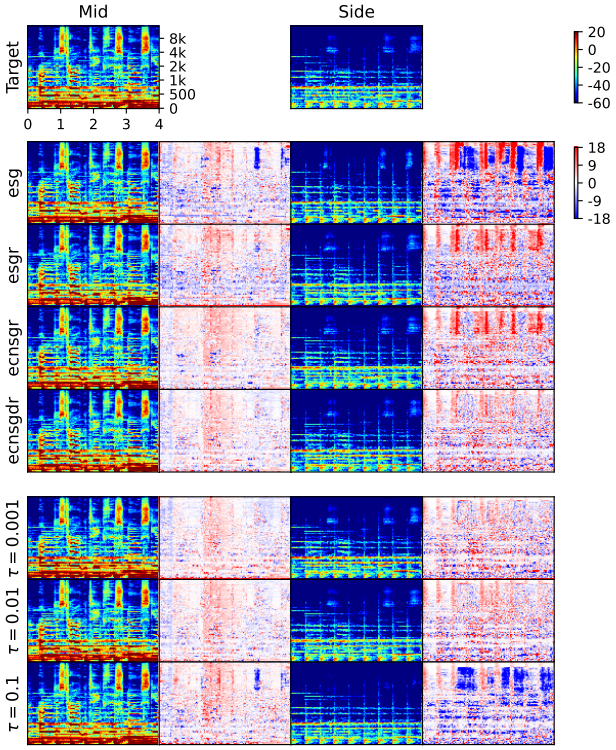
Full results — Finally, we pruned every song in the full dataset ensemble. Table 3 reports the results. The overall trend follows the evaluation subset results but with higher average audio loss (0.509 compared to the previous 0.422). Figure 6 shows statistics of the 3 datasets, initial mixing console graphs, and their pruned versions. `MedleyDB` has the smallest number of source tracks, an average of 17.6. The `Internal` has the largest (28.8), closely followed by the `MixingSecrets` (27.9). The `Internal` dataset also has more subgroups, resulting in even larger mixing consoles. This is one potential cause of the higher sparsity of its pruned graphs; more processors were initially used to match the mix, and many of them were redundant. On average, 72.1 processors (108.5 nodes) were used for each song. Since each full mixing console has 247.6 processors on average, we achieved a pruning ratio of 0.692.

4. DISCUSSION

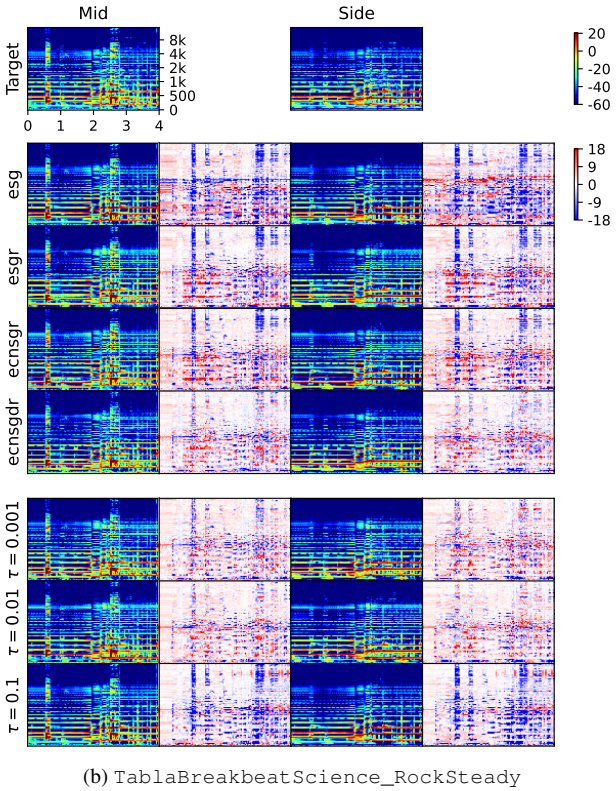
Summary — We started with a general formulation of retrieving mixing graphs from dry sources and mix. Then, we posed restrictions to cast the search to the pruning task, making it computationally feasible and obtaining more interpretable graphs. Next, with additional assumptions, we derived the iterative method that gradually removes negligible processors in a stochastic and greedy manner. As a consequence, instead of finding the exact optimal, our method gives one of close-to-optimal graphs. This can be viewed as posterior sampling, where the prior encodes the inductive bias on graphs that allows the pruning formulation, e.g., processors must follow the fixed order and each should appear at most once in every chain. The likelihood corresponds to the match quality. With the differentiable processors and soft relaxation of the binary pruning with the dry/wet weights, we optimized this objective via gradient descent. We further explored multiple ways to choose pruning candidates, comparing them in terms of their computational cost and the graphs' sparsity. We found that the hybrid method gives a good compromise, so we used it to gather over one thousand graph-audio pairs.

Future works — Before concluding the main text, we list possible variations and extensions of our method.

- The choice of processors and their implementations directly affect the match quality. Our setup, such as the zero-phase FIR equalizer, was motivated by its simplicity and fast computation in GPU. However, we can use other alternatives, especially digital filters such as parametric equalizer [20] and artificial reverb [36], for different parameterizations. Especially, from the spectrogram plots, we can observe that the errors show clear temporal patterns (vertical stripes). This hints that the loudness dynamics are not precisely matched. We suspect it was caused by the ballistics approximation error as recently reported; if so, we might need a more sophisticated implementation [37]. We can also try different processors, e.g., distortion [38] or modulation effects [39]. Finally, the current method does not support time-varying parameters, which might cause audible errors. For example, we could not match the fade-out, i.e., the gradual decrease of track gains, and incorrectly estimated the loudness.
- We note several considerations to improve the current pruning method in terms of sparsity, match quality, and interpretability. First, we can modify the mixing consoles to reflect real-world practices more. For example, we can add send and return loops with additional processor chains and edge weights. Second, to prevent the pruning from harming the perceptual quality, the tolerance condition and the objective function must be appropriately designed. We used a simple multi-resolution STFT loss [32, 34], which has been reported to miss some perceptual features [40, 41]. Hence, we might need an alternative objective as a remedy [42]. The use of average loss to determine the pruning might be inappropriate, as discussed before. Lastly, to increase the sparsity, neural network pruning techniques [23, 24] or domain-specific post-processing can be added.
- We can expand the search space by relaxing the prior assumptions and restrictions on graph structures. If we allow arbitrary processor order, our framework extends to differentiable architecture search [25, 26]. A completely different approach based on reinforcement learning could also be possible [43]. While all of these are promising, balancing flexibility, match quality, and computation cost will be the main challenge.



(a) `Torres_NewSkin`



(b) `TablaBreakbeatScience_RockSteady`

Figure 7: Log-magnitude spectrograms of the matched mixes (odd columns) of the mixing consoles (4 center rows) and pruned graphs (3 bottom rows; in dB). The even columns show the match errors.

5. REFERENCES

[1] P. D. Pestana and J. D. Reiss, "Intelligent audio production strategies informed by best practices," 2014.

[2] F. Everardo, "Towards an automated multitrack mixing tool using answer set programming," in *14th SMC Conf*, 2017.

[3] E. Perez-Gonzalez and J. Reiss, "Automatic gain and fader control for live mixing," in *IEEE WASPAA*, 2009.

[4] B. De Man and J. D. Reiss, "A knowledge-engineered autonomous mixing system," in *AES Convention 135*, 2013.

[5] M. A. Martinez Ramirez *et al.*, "Automatic music mixing with deep learning and out-of-domain data," in *ISMIR*, 2022.

[6] D. Koszewski, T. Görne, G. Korvel, and B. Kostek, "Automatic music signal mixing system based on one-dimensional wave-u-net autoencoders," *EURASIP Journal on Audio, Speech, and Music Processing*, vol. 2023, no. 1, 2023.

[7] C. J. Steinmetz, J. Pons, S. Pascual, and J. Serrà, "Automatic multitrack mixing with a differentiable mixing console of neural audio effects," in *IEEE ICASSP*, 2021.

[8] R. M. Bittner, J. Salamon, M. Tierney, M. Mauch, C. Cannam, and J. P. Bello, "MedleyDB: A multitrack dataset for annotation-intensive mir research." in *ISMIR*, vol. 14, 2014.

[9] R. M. Bittner, J. Wilkins, H. Yip, and J. P. Bello, "MedleyDB 2.0: New data and a system for sustainable data collection," *ISMIR LBD*, 2016.

[10] M. Senior, *Mixing secrets for the small studio*, 2018.

[11] S. Lee, J. Park, S. Paik, and K. Lee, "Blind estimation of audio processing graph," in *IEEE ICASSP*, 2023.

[12] C. Mitcheltree and H. Koike, "SerumRNN: Step by step audio VST effect programming," in *Artificial Intelligence in Music, Sound, Art and Design*, 2021.

[13] J. Guo and B. McFee, "Automatic recognition of cascaded guitar effects," in *DAFx*, 2023.

[14] N. Masuda and D. Saito, "Improving semi-supervised differentiable synthesizer sound matching for practical applications," *IEEE/ACM TASLP*, vol. 31, 2023.

[15] N. Uzrad *et al.*, "DiffMoog: a differentiable modular synthesizer for sound matching," *arXiv:2401.12570*, 2024.

[16] F. Caspe, A. McPherson, and M. Sandler, "DDX7: Differentiable FM synthesis of musical instrument sounds," in *ISMIR*, 2022.

[17] J. Colonel, "Music production behaviour modelling," 2023.

[18] "The mixing console — split, inline and hybrids," https://steemit.com/sound/@jamesub/the-mixing-console-split-inline-and-hybrids, accessed: 2024-02-26.

[19] J. Engel, L. H. Hantrakul, C. Gu, and A. Roberts, "DDSP: differentiable digital signal processing," in *ICLR*, 2020.

[20] S. Nercessian, "Neural parametric equalizer matching using differentiable biquads," in *DAFx*, 2020.

[21] C. J. Steinmetz, N. J. Bryan, and J. D. Reiss, "Style transfer of audio effects with differentiable signal processing," *JAES*, vol. 70, no. 9, 2022.

[22] G. Castellano, A. M. Fanelli, and M. Pelillo, "An iterative pruning algorithm for feedforward neural networks," *IEEE transactions on Neural networks*, vol. 8, no. 3, 1997.

[23] Y. He and L. Xiao, "Structured pruning for deep convolutional neural networks: A survey," *arXiv:2303.00566*, 2023.

[24] H. Cheng, M. Zhang, and J. Q. Shi, "A survey on deep neural network pruning-taxonomy, comparison, analysis, and recommendations," *arXiv:2308.06767*, 2023.

[25] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *ICLR*, 2019.

[26] Z. Ye, W. Xue, X. Tan, Q. Liu, and Y. Guo, "NAS-FM: Neural architecture search for tunable and interpretable sound synthesis based on frequency modulation," *arXiv:2305.12868*, 2023.

[27] C. J. Steinmetz, S. S. Vanka, M. A. Martínez-Ramírez, and G. Bromham, *Deep Learning for Automatic Mixing*. ISMIR, Dec. 2022.

[28] J. Koo *et al.*, "Music mixing style transfer: A contrastive learning approach to disentangle audio effects," in *IEEE ICASSP*, 2023.

[29] B. Hayes, C. Saitis, and G. Fazekas, "Sinusoidal frequency estimation by gradient descent," in *IEEE ICASSP*, 2023.

[30] U. Zölzer, Ed., *DAFX: Digital Audio Effects*, 2nd ed., 2011.

[31] M. Fey and J. E. Lenssen, "Fast graph representation learning with pytorch geometric," *arXiv:1903.02428*, 2019.

[32] R. Yamamoto, E. Song, and J.-M. Kim, "Parallel wavegan: A fast waveform generation model based on generative adversarial networks with multi-resolution spectrogram," in *IEEE ICASSP*, 2020.

[33] A. Wright and V. Välimäki, "Perceptual loss function for neural modeling of audio systems," in *IEEE ICASSP*, 2020.

[34] C. J. Steinmetz and J. D. Reiss, "auraloss: Audio focused loss functions in PyTorch," in *Digital Music Research Network One-day Workshop (DMRN+15)*, 2020.

[35] I. Loshchilov and F. Hutter, "Decoupled weight decay regularization," *arXiv:1711.05101*, 2017.

[36] S. Lee, H.-S. Choi, and K. Lee, "Differentiable artificial reverberation," *IEEE/ACM TASLP*, vol. 30, 2022.

[37] C. J. Steinmetz, T. Walther, and J. D. Reiss, "High-fidelity noise reduction with differentiable signal processing," in *AES Convention 155*, 2023.

[38] J. T. Colonel, M. Comunità, and J. Reiss, "Reverse engineering memoryless distortion effects with differentiable waveshapers," in *AES Convention 153*, 2022.

[39] A. Carson, S. King, C. V. Botinhao, and S. Bilbao, "Differentiable grey-box modelling of phaser effects using frame-based spectral processing," in *DAFx*, 2023.

[40] B. Hayes, J. Shier, G. Fazekas, A. McPherson, and C. Saitis, "A review of differentiable digital signal processing for music & speech synthesis," *Frontiers in Signal Process.*, 2023.

[41] J. Turian and M. Henry, "I'm sorry for your loss: Spectrally-based audio distances are bad at pitch," in *"I Can't Believe It's Not Better!" NeurIPS workshop*, 2020.

[42] C. Vahidi *et al.*, "Mesostructures: Beyond spectrogram loss in differentiable time–frequency analysis," *JAES*, 2023.

[43] J. You, B. Liu, Z. Ying, V. Pande, and J. Leskovec, "Graph convolutional policy network for goal-directed molecular graph generation," *NeurIPS*, 2018.

[44] M. A. Martínez-Ramírez, O. Wang, P. Smaragdis, and N. J. Bryan, "Differentiable signal processing with black-box audio effects," in *IEEE ICASSP*, 2021.

[45] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library," *NeurIPS*, 2019.

[46] X. Serra and J. Smith, "Spectral modeling synthesis: A sound analysis/synthesis based on a deterministic plus stochastic decomposition," *Computer Music Journal*, vol. 14, 1990.

[47] J. T. Colonel and J. Reiss, "Reverse engineering a nonlinear mix of a multitrack recording," *Journal of the AES*, vol. 71, no. 9, 2023.

[48] D. Giannoulis, M. Massberg, and J. D. Reiss, "Digital dynamic range compressor design—a tutorial and analysis," *JAES*, vol. 60, no. 6, 2012.

[49] J. Colonel, J. D. Reiss, *et al.*, "Approximating ballistics in a differentiable dynamic range compressor," in *AES Convention 153*, 2022.

[50] Y. Bengio, N. Léonard, and A. Courville, "Estimating or propagating gradients through stochastic neurons for conditional computation," *arXiv:1308.3432*, 2013.

[51] J. D. Ullman, "NP-complete scheduling problems," *Journal of Computer and System sciences*, vol. 10, no. 3, 1975.

A. RELATED WORKS

A.1. Composition of Audio Processors

Most audio processors are designed to modify some specific properties of their input signals, e.g., magnitude response, loudness dynamics, and stereo width. As such, combining multiple processors is a common practice to achieve the full effect. Following the main text, we will use the terminology "graph" to represent this composition, although some previous works considered simple structures that allow a more compact form, e.g., a sequence. Now, we outline the previous attempts that tried to estimate the processing graph G and/or its parameters $\mathbf{P}$ from reference audio. For example, if the references are one or more dry sources and a wet mixture, this task becomes reverse engineering [11, 12, 13, 17]. In terms of the prediction targets, some works [7, 15, 17, 19, 44] fixed the graph and estimated only the parameters. Others [11, 12, 13, 26, 16] tried also to predict the graph. These works differ in task and domains, processor sets, search space of graphs, and estimation methods. For example, Lee et al. [11] tackled singing voice effects and drum mixing estimation from wet and optional dry source(s). With 33 processor types where some of them were multiple-input multiple-output (MIMO) systems, they generated synthetic directed acyclic graphs (DAGs) and trained a transformer-based graph and parameter predictor in a supervised manner. These are clearly different from our work. We tackled the music mixing domain, which has much larger graph sizes but with a restricted graph structure based on the mixing consoles (more details on the resulting search spaces are in Appendix A.3). We used fewer processor types, but they were all single-input single-output (SISO) and differentiable. We ran joint pruning and parameter optimization of the hybrid mixing consoles via gradient descent (without any neural network). Such differences between the works are summarized and highlighted in Table 4. For a unified viewpoint, we use a probabilistic framework to denote the tasks, e.g., $p(\mathbf{P}|\mathbf{S})$ for automatic mixing, albeit many works do not involve any generative modeling or sampling.

A.2. Differentiable Signal Processing

Differentiable processor — Exact implementation or approximation of processors in an automatic differentiation framework, e.g., `pytorch` [45], enables parameter optimization via gradient descent. Numerous efforts have been focused on converting existing audio processors to their differentiable versions [17, 21, 38, 39, 20, 36]; refer to the recent review [40] and references therein for more details. In many cases, these processors are combined with neural networks, whose computation is done in GPU. Thus, converting the audio processors to be "GPU-friendly" has been an active research topic. For example, for a linear time-invariant (LTI) system with a recurrent structure, we can sample its frequency response to approximate its infinite impulse response (IIR) instead of directly running the recurrence; the former is faster than the latter [20, 36]. However, it is nontrivial to apply a similar trick to nonlinear, recurrent, or time-varying processors. Typically, further simplifications and approximations are employed, e.g., replacing the nonlinear recurrent part with an IIR filter [21] or assuming frame-wise LTI to a sample-level linear time-varying system [39]. Sometimes, we can only access input and output measurements. Then, one can approximate the gradients with finite difference methods [44, 37] or pretraining an auxiliary neural network that mimics the target processors [7]. In the literature, these tricks are also referred to as "differentiable;" hence, it is rather an umbrella term encompassing all possible methods that obtain the output signals or gradients within a reasonable time. Nevertheless, we limit our focus to the implementations in the automatic differentiation framework.

Audio processing graph — Now, consider a composition of multiple differentiable processors; the entire graph remains differentiable due to the chain rule. However, the following practical considerations remain. If we fix the processing graph prior to the optimization and the graph size is relatively small, we can implement the "differentiable graph" following the existing implementations [19, 15]. That is, we compute every processor one by one in a pre-defined topological order. However, we have the following additional requirements. First, the pruning changes the graph during the optimization. Therefore, our implementation must take a graph and its parameters along with the source signals as input arguments for every forward pass. Note that this feature is also necessary when training a neural network that predicts the parameters of any given graph [11]. Second, the size of our graphs is much larger than the ones from previous works [19, 15, 26, 11]. In this case, the one-by-one computation severely bottlenecks the computation speed. Therefore, we derived a flexible and efficient graph computation algorithm (i) that can take different graphs for each forward pass as input and (ii) performs batched processing of multiple processors within a graph, utilizing the parallelism of GPUs. Finally, we note that other than the differentiation with respect to the input signals $\mathbf{S}$ and parameters $\mathbf{P}$, one might be interested in differentiation with respect to the graph structure G. The proposed pruning method performs this to a limited extent; deletion of a node v_i is a binary operation that modifies the graph structure. We relaxed this to a continuous dry/wet weight w_i and optimized it with the audio loss L_{a} and regularization L_{p}.

A.3. Graph Search

Several independent research efforts in various domains exist that aim to find graphs that satisfy certain requirements. First, neural architecture search (NAS) aims to find a neural network architecture that achieves an improved performance. In this case, the search space consists of graphs where the nodes (or edges according to the chosen notation) represent the primitive neural network layers. One very close work to ours is differentiable architecture search (DARTS) [25], which relaxed the choice of layer type to a categorical distribution and optimized it via gradient descent. Theoretically, our method can be naturally extended to follow this approach; we only need to change our 2-way choice (prune or not) to $(K+1)$-way (bypass or select one of K possible processors). While this search is more flexible and general, this greatly increases the computational cost, as we must compute all K processor types for every node. Second, the generation/design of a molecule is another popular topic. One dominant approach is to use reinforcement learning (RL) to generate molecule graphs that achieve the desired properties [43]. However, applying this method to our task has a risk of obtaining nontrivial mixing graphs that is hard to interpret for practitioners.

B. DIFFERENTIABLE AUDIO PROCESSORS

For each processor, $u[n]$ and $y[n]$ denote its stereo input and output, respectively. We write the discrete-time index with n. p with a subscript denotes a real-valued parameter vector, which could be a concatenation of multiple parameters. Each subscript x denotes one of the following channels: left l, right r, mid m, and side s.

[15]	*Task & domain*	Sound matching $p(\mathbf{P}\|x)$. The synthesizer parameters $\mathbf{P}$ were estimated to match the reference (target) audio x.
	Processors	Oscillators, envelope generators, and filters that allow parameter modulation as an optional input.
	Graph	Any pre-defined DAG. For example, a subtractive synthesizer (2 oscillators, 1 amplitude envelope, and 1 lowpass filter were used for the evaluation).
	Optimization	Trained a single neural backbone for the reference encoding, followed by multiple prediction heads for the parameters. Optimized with a parameter loss and spectral loss, where the latter is calculated with every intermediate output.
[16]	*Task & domain*	Sound matching $p(\mathbf{P}\|x)$. An FM synthesizer matches recordings of monophonic instruments (violin, flute, and trumpet). Estimates parameters of an operator graph G that is empirically searched & selected.
	Processors	Differentiable sinusoidal oscillators, each used as a carrier or modulator, pre-defined frequencies. An additional FIR reverb is added to the FM graph output for post-processing.
	Graph	Directed and acyclic (no feedback FM), at most 6 operators. Different graphs for different target instruments.
	Optimization	Trained a convolutional neural network that estimates envelopes from the target loudness and pitch.
[26]	*Task & domain*	Sound matching $p(G, \mathbf{P}\|x)$. Similar setup to the above [16] but with an additional estimation of the operator graph G.
	Processors	Identical to [16], except for the frequency ratio that can be searched.
	Graph	A subgraph of a supergraph, which resembles a multi-layer perceptron (modulator layers followed by a carrier layer).
	Optimization	Trained a parameter estimator for the supergraph and found the appropriate subgraph G with an evolutionary search.
[12]	*Task & domain*	Reverse engineering $p(G, \mathbf{P}\|s, y)$ of an audio effect chain from a subtractive synthesizer (commercial plugin).
	Processors	5 audio effects: compressor, distortion, equalizer, phaser, and reverb. Non-differentiable implementations.
	Graph	Chain of audio effects generated with no duplicate types (therefore 32 possible combinations) and random order.
	Optimization	Trained a next effect predictor and parameter estimator in a supervised (teacher-forcing) manner.
[13]	*Task & domain*	Blind estimation $p(G\|y)$ and reverse engineering $p(G\|s, y)$ of guitar effect chains.
	Processors	13 guitar effects, including non-linear processors, modulation effects, ambience effects, and equalizer filters.
	Graph	A chain of guitar effects. Maximum 5 processors and a total of 221 possible combinations.
	Optimization	Trained a convolutional neural network with synthetic data to predict the correct combination.
[7]	*Task & domain*	Automatic mixing $p(\mathbf{P}\|\mathbf{S})$. Estimated parameters of fixed processing chains from source tracks ($K \leq 16$).
	Processors	7 differentiable SISO processors, where 4 (gain, polarity, fader, and panning) were implemented exactly. The remaining 3 (equalizer, compressor, and reverb) were approximated with a single pretrained neural network.
	Graph	Tree structure: applied a fixed chain of the 7 processors for each track, and then summed the chain outputs altogether.
	Optimization	Trained a parameter estimator (convolutional neural network) with a spectrogram loss end-to-end.
[44]	*Task & domain*	Reverse engineering of music mastering $p(\mathbf{P}\|s, y)$
	Processors	A multi-band compressor, graphic equalizer, and limiter. Gradient approximated with a finite difference method.
	Graph	A serial chain of the processors.
	Optimization	Optimized parameters with gradient descent.
[11]	*Task & domain*	Blind estimation $p(G, \mathbf{P}\|y)$ and reverse engineering $p(G, \mathbf{P}\|\mathbf{S}, y)$. Estimates both graph and its parameters, for singing voice effect ($K = 1$) or drum mixing ($K \leq 6$).
	Processors	A total of 33 processors, including linear filters, nonlinear filters, and control signal generators. Some processors are MIMO systems, e.g., allowing auxiliary modulations. Non-differentiable implementations.
	Graph	Complex DAG; splits (e.g., multi-band processing) and merges (e.g., sum and modulation). 30 processors max.
	Optimization	Trained a convolutional neural network-based reference encoder and a transformer variant for graph decoding and parameter estimation. Both were jointly trained via direct supervision of synthetic graphs (e.g., parameter loss).
[17]	*Task & domain*	Reverse engineering $p(\mathbf{P}\|\mathbf{S}, y)$ of music mixing. Estimated parameters of a fixed chain for each track.
	Processors	6 differentiable SISO processors: gain, equalizer, compressor, distortion, panning, and reverb.
	Graph	A chain of 5 processors (all above types except the reverb) for each dry track (any other DAG can also be used). The reverb is used for the mixed sum.
	Optimization	Parameters were optimized with spectrogram loss end-to-end via gradient descent.
Ours	*Task & domain*	Reverse engineering $p(G, \mathbf{P}\|\mathbf{S}, y)$ of music mixing. Estimated a chain of processors and their parameters for each track and submix where $K \leq 130$.
	Processors	7 differentiable SISO processors: gain/panning, stereo imager, equalizer, reverb, compressor, noisegate, and delay.
	Graph	A tree of processing chains with a subgrouping structure (any other DAG can also be used). Processors can be omitted but should follow the fixed order.
	Optimization	Joint estimation of the soft masks (dry/wet weights) and processor parameters. Optimized with the spectrogram loss (and additional regularizations) end-to-end via gradient descent. Accompanied by hard pruning stages.

Table 4: A brief summary and comparison of previous works on estimation of compositional audio signal processing.

B.1. Gain/panning

We use simple channel-wise constant multiplication. Its parameter vector $p_{\mathrm{g}} \in \mathbb{R}^2$ is in log scale, so we apply exponentiation before multiplying it to the stereo signal.

$$y[n] = \exp(p_{\mathrm{g}}) \cdot u[n]. \tag{10}$$

B.2. Stereo Imager

We multiply the side signal (left minus right) with a gain parameter $p_{\mathrm{s}} \in \mathbb{R}$ to control the stereo width. The mid and side outputs are

$$y_{\mathrm{m}}[n] = u_{\mathrm{l}}[n] + u_{\mathrm{r}}[n], \tag{11a}$$
$$y_{\mathrm{s}}[n] = \exp(p_{\mathrm{s}}) \cdot (u_{\mathrm{l}}[n] - u_{\mathrm{r}}[n]). \tag{11b}$$

Then, we convert the mid/side output to a stereo signal as follows, $y_{\mathrm{l}}[n] = (y_{\mathrm{m}}[n] + y_{\mathrm{s}}[n])/2$ and $y_{\mathrm{r}}[n] = (y_{\mathrm{m}}[n] - y_{\mathrm{s}}[n])/2$.

B.3. Equalizer

We employ a zero-phase FIR filter. Considering its log-magnitude as a parameter p_{e}, we compute inverse FFT (IFFT) of the magnitude response and multiply it with a Hann window $v^{\mathrm{Hann}}[n]$. As a result, the length-N FIR is given as

$$h_{\mathrm{e}}[n] = v^{\mathrm{Hann}}[n] \cdot \frac{1}{N} \sum_{k=0}^{N-1} \exp p_{\mathrm{e}}[k] \cdot w_N^{kn} \tag{12}$$

where $-(N+1)/2 \leq n \leq (N+1)/2$ and $w_N = \exp(j \cdot 2\pi/N)$. We compute the final output by applying the same FIR to both the left and right channels as follows,

$$y_{\mathrm{x}}[n] = u_{\mathrm{x}}[n] * h_{\mathrm{e}}[n] \quad (\mathrm{x} \in \{\mathrm{l}, \mathrm{r}\}). \tag{13}$$

We set the FIR length to $N = 2047$. Therefore, the parameter p_{e} has a size of 1024. Note that we could use a parametric equalizer [20] as an alternative for more compact parameters. However, we used this FIR filter due to its simplicity and fast computation.

B.4. Reverb

We use a variant of filtered noise models [46, 19]. First, we create 2 seconds of uniform noises, $u_{\mathrm{m}}[n]$ and $u_{\mathrm{s}}[n]$. Next, we multiply each noise's STFT $U_{\mathrm{x}}[k, m]$ with a magnitude mask $M_{\mathrm{x}}[k, m]$.

$$H_{\mathrm{x}}[k, m] = U_{\mathrm{x}}[k, m] \odot M_{\mathrm{x}}[k, m] \quad (\mathrm{x} \in \{\mathrm{m}, \mathrm{s}\}). \tag{14}$$

Here, k and m denote frequency and time frame index. Each mask is parameterized with an initial log-magnitude $H_{\mathrm{x}}^0[k]$ and an absorption filter $H_{\mathrm{x}}^{\Delta}[k]$ as follows,

$$M_{\mathrm{x}}[k, m] = \exp(H_{\mathrm{x}}^0[k] + (m-1)H_{\mathrm{x}}^{\Delta}[k]). \tag{15}$$

Next, we convert the masked STFTs to the time-domain responses, $h_{\mathrm{m}}[n]$ and $h_{\mathrm{s}}[n]$. We obtain the desired FIR $h_{\mathrm{r}}[n]$ by converting the mid/side to stereo. We apply channel-wise convolutions to the input $u[n]$ and get the output signal $y[n]$. The FFT and hop lengths are set to 384 and 192, respectively. The parameter vector p_{r} has the size of 768 (2 channels, each with 2 filters, and 192 magnitude values for each filter). We briefly discuss some alternatives to this processor. First, we could use other artificial reverberation models [36]. But again, we chose this STFT-based FIR due to its simplicity and fast computation. Or, we could use a single FIR to represent both reverb and multitap delay since they are linear time-invariant [47]. However, we decided to separate those two for interpretable, controllable, and more compact representations.

B.5. Compressor

We implement the canonical feed-forward digital compressor [48]. First, for a given input audio, we sum the left and right channels to obtain a mid signal $u_{\mathrm{m}}[n]$. Then, we calculate its energy envelope $G_u[n] = \log g_u[n]$ where

$$g_u[n] = \alpha[n] g_u[n-1] + (1 - \alpha[n]) u_{\mathrm{m}}^2[n]. \tag{16}$$

Here, the coefficient $\alpha[n]$ is typically set to a different constant for an "attack" (increasing) and "release" (decreasing) phase:

$$\alpha[n] = \begin{cases} \alpha^{\mathrm{att}} & g_u[n] > g_u[n-1], \\ \alpha^{\mathrm{rel}} & g_u[n] \leq g_u[n-1]. \end{cases} \tag{17}$$

As this part (also known as ballistics) bottlenecks the computation speed in GPU, following the recent work [21], we restrict the constants to the same value: $\alpha = \alpha^{\mathrm{att}} = \alpha^{\mathrm{rel}}$. By doing so, Equation 16 simplifies to a one-pole IIR filter, whose length-N FIR approximation can be computed in parallel using the frequency-sampling method [20] and applying IFFT to the sampled response:

$$h^{\mathrm{env}}[n] \approx \frac{1}{N} \sum_{k=0}^{N-1} \frac{1-\alpha}{1 - \alpha w_N^{-k}} w_N^{kn}. \tag{18}$$

Therefore, the energy envelope can be simply computed as

$$g_u[n] \approx h^{\mathrm{env}}[n] * u^2[n]. \tag{19}$$

Next, we compute the compressed energy envelope $G_y[n]$. We use a quadratic knee, interpolating the compression and the bypass region. For a given threshold T and half of the knee width W,

$$G_y[n] = \begin{cases} G_y^{\mathrm{above}}[n] & G_u[n] \geq T + W, \\ G_y^{\mathrm{mid}}[n] & T - W \leq G_u[n] < T + W, \\ G_y^{\mathrm{below}}[n] & G_u[n] < T - W \end{cases} \tag{20}$$

where, for a given compression ratio R, each term is

$$G_y^{\mathrm{above}}[n] = T + \frac{G_u[n] - T}{R}, \tag{21a}$$
$$G_y^{\mathrm{mid}}[n] = G_u[n] + \left(\frac{1}{R} - 1\right) \frac{(G_u[n] - T + W)^2}{4W}, \tag{21b}$$

and $G_y^{\mathrm{below}}[n] = G_u[n]$. Finally, we can compute the output as

$$y_{\mathrm{x}}[n] = \exp(G_y[n] - G_u[n]) \cdot u_{\mathrm{x}}[n] \quad (\mathrm{x} \in \{\mathrm{l}, \mathrm{r}\}). \tag{22}$$

The scalar parameters introduced above, α, T, W, and R, are concatenated and used as a parameter vector $p_{\mathrm{c}} \in \mathbb{R}^4$. Our implementation is almost identical to the recently introduced differentiable compressor [21]. However, we smoothened to the energy $u_{\mathrm{m}}^2[n]$ instead of the gain reduction $G_y[n] - G_u[n]$ as the former showed a slightly better matching performance in the initial experiments. Since the approximated ballistics (Equation 18) could be one cause of the reduced modeling capability of loudness dynamics, it might be desirable to try alternatives that allow the different attack and release coefficients [37, 49].

B.6. Noisegate

Its implementation is the same as the compressor above, except for the gain computation part. We set $G_y^{\mathrm{above}}[n] = G_u[n]$ and

$$G_y^{\mathrm{mid}}[n] = G_u[n] + (1 - R) \frac{(G_u[n] - T - W)^2}{4W}, \tag{23a}$$
$$G_y^{\mathrm{below}}[n] = T + R(G_u[n] - T). \tag{23b}$$

B.7. Multitap Delay

We consider a 2 seconds of delay effect with at most one delay d_m at every 100ms (hence the number of delay taps is $M = 20$). Each delay is filtered with a 39-tap zero-phase FIR $c_m[n]$ parameterized in the same way as the equalizer. We use independent delays and their filters for the left and right channels, but we ignore this in the following equations for simplicity. Under this setting, the multitap delay's single-channel FIR $h_{\mathrm{d}}[n]$ is given as follows,

$$h_{\mathrm{d}}[n] = \sum_{m=1}^{M} c_m[n] * \delta[n - d_m] \tag{24}$$

where $\delta[n]$ is an unit impulse. Here, we aim to optimize each delay length $d_m \in \mathbb{N}^+$, which is discrete, using gradient descent. To this end, we exploit the fact that each delay $\delta[n - d_m]$ corresponds to a complex sinusoid in the frequency domain. Recent work showed that the sinusoid's angular frequency z_m can be optimized with the gradient descent when we allow it to be inside of the unit disk [29]. In short, for each complex parameter $z_m \in \mathbb{C}$ where $|z_m| \leq 1$, we compute a surrogate damped sinusoid and apply IFFT to it.

$$\delta[n - d_m] \approx \frac{1}{N} \sum_{k=0}^{N-1} z_m^k w_N^{kn}. \tag{25}$$

This relaxed FIR is not exactly the discrete delay signal; precisely, it is a lowpassed version of an aliased sinc kernel. Hence, we use it only for backpropagation with the straight-through technique [50]. Also, we normalize each gradient and regularize the parameter z_m to be closer to the unit circle. We empirically observed that this improves the performance. To summarize, each complex conjugate gradient is modified as follows and used for the optimization:

$$\frac{\partial L}{\partial z_m^*} \leftarrow \mathrm{sgn}\left(\sum_n \frac{\partial L}{\partial h_{\mathrm{d}}[n]} \frac{\partial \tilde{h}_{\mathrm{d}}[n]}{\partial z_m^*} \right) + \gamma(|z_m| - 1)\,\mathrm{sgn}(z_m^*) \tag{26}$$

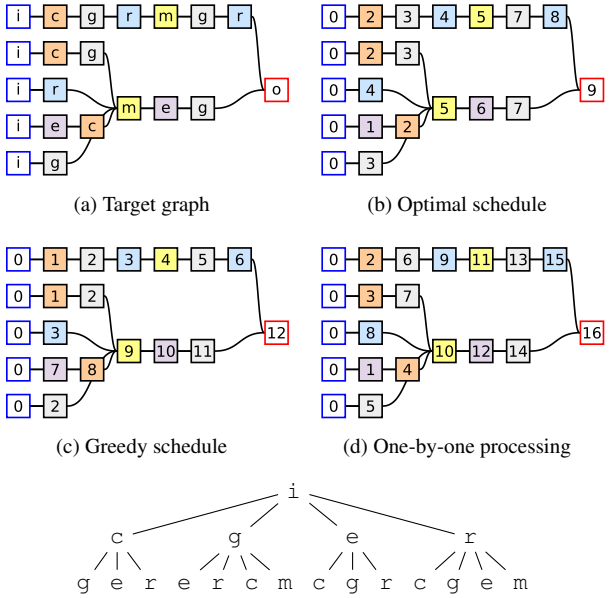
where $\tilde{h}_{\mathrm{d}}[n]$ is the surrogate FIR obtained with the continuous delay, $\mathrm{sgn}(z) = z/|z|$, and γ is a regularization strength set to 0.01. This multitap delay has parameter p_{d} of size 880 (20 delays each for the left and right channel; each delay with a real and imaginary part of the sinusoid's angular frequency and 20 log-magnitude bins of the FIR filter). Note that we restricted the delay lengths to be in separate time intervals; we failed to optimize this processor without this constraint. Resolving this is left as a future work.

C. AUDIO PROCESSING GRAPHS IN GPU

We slightly deviate from the main text and consider a more general setup. We do not distinguish between the processors and auxiliary modules; both are referred to as nodes. We omit all the weights $\mathbf{w}$ for simplicity. Then, each node's output y_i is given as follows,

$$y_i = f_i(u_i, p_i), \quad u_i = \sum_{j \in \mathcal{N}^+(i)} y_j \tag{27}$$

where $\mathcal{N}^+(i)$ denotes a set of source node indices for the node v_i. In short, each output y_i is obtained by gathering the input signals, aggregating those, and processing the sum with the parameters (the latter two can be omitted according to the node type). Furthermore, we allow disconnected graphs and view a batch of multiple graphs as a single large graph. Thus, it is enough to consider only a single graph case. The remaining structural restrictions to graphs are (i) being acyclic and (ii) having only SISO systems.



(a) Target graph (b) Optimal schedule

(c) Greedy schedule (d) One-by-one processing

(e) All possible sequences visualized up to depth 2

Figure 8: Various batched processing schedules. For each schedule 8b-8d, the processing orders are shown inside the nodes.

C.1. Batched Processing

We may obtain the final graph output $\mathbf{Y}$ by simply processing each node one by one. Nevertheless, we can accelerate the computation via parallelism. Consider a subset sequence $V_0, \cdots, V_N \subset V$ that satisfies the followings,

(i) It is a *partition*: $\cup_n V_n = V$ and $V_n \cap V_m = \emptyset$ for $n \neq m$.

(ii) The sequence must be *causal*: there is no path from $u \in V_n$ to $v \in V_m$ if $n \geq m$.

(iii) Each subset V_n is *homogeneous*: it has only one type t_n.

Then, we can compute a batch of output signals $\mathbf{Y}_n$ of each subset V_n sequentially, from $n = 0$ to N. Consequently, we reduce the number of the gather-aggregate-process iterations from $|V|$ to N. Figure 8 shows an example. For a graph with $|V| = 21$ nodes (8a), we can find a sequence with $N = 9$ (8b). Intuitively, this batched processing is effective for a graph with fewer types and a structure that resembles the (pruned version of) mixing console, e.g., having the identical processor order for each track.

C.2. Node Type Scheduling

To maximize the parallelism, we want to find the shortest sequence. This is a variant of the scheduling problem [51]. First, we always choose a maximal subset V_i when the type t_i is fixed. This makes the subset sequence equivalent to a type string, e.g., `iecgrmegro` for 8b. We also choose the initial and the last subset, V_0 and V_N, to collect all input and output nodes, respectively. Since the search tree for the shortest sequence exponentially grows, the brute-force search is too expensive for most graphs (8e). Fortunately, we know the optimal schedule for the graphs in this paper as they follow the fixed ordering. If we do not have such restrictions or prior assumptions about the graph structure, we may try the greedy method that chooses a type with the largest number of computable nodes (8c). However, this usually results in a longer sequence.

Algorithm 2 Batch computation of audio processing graphs.

Input: Node types $\mathbf{T}$, edges $\mathbf{E}$, parameters $\mathbf{P}$, and inputs $\mathbf{S}$
Output: Output signals $\mathbf{Y}$ and (optional) intermediate signals $\mathbf{U}$

1: $\bar{\mathbf{T}}, N \leftarrow \mathrm{ScheduleBatchedProcessing}(\mathbf{T}, \mathbf{E})$
2: $\sigma \leftarrow \mathrm{OptimizeNodeOrder}(\bar{\mathbf{T}}, \mathbf{T}, \mathbf{E})$
3: $\mathbf{T}, \mathbf{E}, \mathbf{P} \leftarrow \mathrm{Reorder}(\sigma, \mathbf{T}, \mathbf{E}, \mathbf{P})$
4: $\mathbf{I}^{\mathrm{G}}, \mathbf{I}^{\mathrm{P}}, \mathbf{I}^{\mathrm{A}}, \mathbf{I}^{\mathrm{S}} \leftarrow \mathrm{GetReadWriteIndex}(\bar{\mathbf{T}}, \mathbf{T}, \mathbf{E}, \mathbf{P})$
5: $\mathbf{U} \leftarrow \mathrm{Initialize}(\mathbf{S}, \mathbf{T})$
6: **for** $n \leftarrow 1$ to N **do**
7: $\quad \bar{\mathbf{U}}_n \leftarrow \mathrm{Gather}(\mathbf{U}, \mathbf{I}^{\mathrm{G}}_n)$ ▷ `index_select`
8: $\quad \mathbf{U}_n \leftarrow \mathrm{Aggregate}(\bar{\mathbf{U}}_n, \mathbf{I}^{\mathrm{A}}_n)$ ▷ `scatter`
9: $\quad \mathbf{P}_n \leftarrow \mathrm{Gather}(\mathbf{P}[\bar{t}_n], \mathbf{I}^{\mathrm{P}}_n)$ ▷ `slice`
10: $\quad \mathbf{Y}_n \leftarrow \mathrm{Process}(\bar{t}_n, \mathbf{U}_n, \mathbf{P}_n)$
11: $\quad \mathbf{U} \leftarrow \mathrm{Store}(\mathbf{U}, \mathbf{Y}_n, \mathbf{I}^{\mathrm{S}}_n)$ ▷ `slice`
12: **end for**
13: $\mathbf{Y} \leftarrow \mathbf{Y}_N$
14: **return** $\mathbf{Y}, \mathbf{U}$

C.3. Implementation Details

Inputs — We represent a graph with a node type vector $\mathbf{T} \in \mathbb{N}^{|V|}$ and an edge index tensor $\mathbf{E} \in \mathbb{N}^{2\times|E|}$. Its parameters are collected in a dictionary $\mathbf{P}$ whose key is a type t and value is its corresponding parameter tensor given as $\mathbf{P}[t] \in \mathbb{R}^{|V_t|\times N_t}$ where $|V_t|$ and N_t are the number of nodes and parameters. All sources are stacked to a single tensor $\mathbf{S} \in \mathbb{R}^{K\times 2\times L}$. We ensure that all the inputs, $\mathbf{T}$, $\mathbf{E}$, $\mathbf{P}$, and $\mathbf{S}$, share a node order. For example, a k^{th} source s_k must correspond to the first k^{th} input `i` in the type list $\mathbf{T}$. Likewise, an l^{th} type-t parameter $\mathbf{P}[t]_l \in \mathbb{R}^{N_t}$ must correspond to the first l^{th} type t in the type list $\mathbf{T}$.

Preprocessing — Algorithm 2 obtains the output $\mathbf{Y} \in \mathbb{R}^{|V_\circ|\times 2\times L}$ from the prescribed inputs. First, we schedule the batched processings, resulting in a type list $\bar{\mathbf{T}} \in \mathbb{N}^N$ (1). Next, as the main batched processing loop (6-12) contains multiple memory read/writes, we calculate the node reordering σ that achieves contiguous memory accesses and improves the computation speed (2). This procedure allows memory accesses via `slice`, as shown in the comments of Algorithm 2. After permuting the graph tensors (3), we retrieve lists of tensor indices, $\mathbf{I}^{\mathrm{G}}$, $\mathbf{I}^{\mathrm{P}}$, $\mathbf{I}^{\mathrm{A}}$, and $\mathbf{I}^{\mathrm{S}}$, used for the read/writes in the main loop (4). Note that all these preprocessing steps (1-4) are done in CPU. Then, we create an intermediate output tensor $\mathbf{U} \in \mathbb{R}^{|V|\times 2\times L}$ (5). As we put all the inputs to be the first partition V_0, it can be initialized with simple concatenation: $\mathbf{U} = \mathbf{S} \oplus \mathbf{0}$.

Main loop — The remainings repeat batched processing and necessary read/writes (6-12). For each n^{th} iteration, we collect the previous outputs $\bar{\mathbf{U}}_n$ that are routed to the current partition nodes. We achieve this by accessing the intermediate tensor $\mathbf{U}$ with the index $\mathbf{I}^{\mathrm{G}}_n$ with `index_select` (7). Then, we aggregate them using `scatter` operation if multiple edges are connected to some nodes (8). We can similarly obtain a parameter tensor $\mathbf{P}_n$ with its corresponding index $\mathbf{I}^{\mathrm{P}}_n$. Especially, our node reordering (3) makes this a simple `slice`, faster than the usual `index_select`. With the obtained input signals $\mathbf{U}_n \in \mathbb{R}^{|V_n|\times 2\times L}$ and parameters $\mathbf{P}_n \in \mathbb{R}^{|V_n|\times N_t}$, we batch-compute the node outputs $\mathbf{Y}_n$ (10). Then, we save them to the intermediate output tensor $\mathbf{U}$ with the `slice` index $\mathbf{I}^{\mathrm{S}}_n$ (11) so that the remaining steps can access them as inputs. After the iteration, we have all node outputs saved in $\mathbf{U}$. The final graph outputs are given as $\mathbf{Y}_N \in \mathbb{R}^{|V_N|\times 2\times L}$ since we set the last node partition V_N to collect all output nodes.

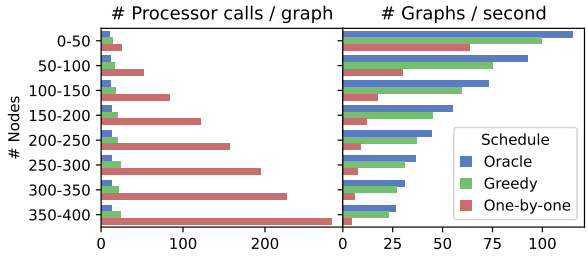


Figure 9: Benchmark results of different processing schedules.

C.4. Benchmark

We measured the efficiency of the schedules with the pruned graphs on a single `RTX3090` GPU. Figure 9 reports the result.

- Our pruned graphs require at most 14 batched processings when scheduled correctly. The optimal schedule achieves 11.8 calls on average, while the greedy and one-by-one schedule report 17.5 and 77.6 calls, respectively. The latter especially increases the processor calls linearly to the graph size.
- As a result, the batched processing improves the speed across all graph sizes (especially for large ones). On average, the one-by-one computes 27.8 graph outputs per second. The greedy and optimal schedule achieve 67.2 and 81.2, respectively, indicating that improved scheduling methods for arbitrary graphs could be desirable for further research.
- Note that the speed-up from the batched processing is sublinear to the graph size. For graphs with 350 to 400 nodes, the optimal calls the processors 21.1 times less than the one-by-one, but the speedup is about 5.8 fold.

C.5. Extension

We can remove the SISO restriction and allow MIMO systems. A processor f_i with M inputs and N outputs processes the incoming signals as follows,

$$[y_{i1}, \cdots, y_{iN}]^T = f_i\big([u_{i1}, \cdots, u_{iM}]^T, p_i\big), \tag{28a}$$

$$u_{il} = \sum_{(j,k)\in\mathcal{N}^+(i,l)} y_{jk}. \tag{28b}$$

Here, $(j,k) \in \mathcal{N}^+(i,l)$ denotes a connection from node j channel k to node i channel l. In a graph viewpoint, this forms a multigraph that allows multiple edges between two nodes. The only remaining restriction (to the graphs) is being acyclic. Implemenetation-wise, we need to introduce an edge channel type tensor, replace the intermediate $\mathbf{U}$ with a node-channel flattened tensor, and reshape the processor input/outputs, $\mathbf{U}_n$ and $\mathbf{Y}_n$, appropriately to gather from and store to the intermediate $\mathbf{U}$.

D. DRY/WET PRUNING ALGORITHM

Algorithm 3 describes the details of the dry/wet method.

- For a simpler description, we modified the initialization part to include per-type node sets and weights, as in line 6-10.
- The termination condition is given in line 11.
- The trial candidate sampling is implemented in line 12-13.
- The candidate pool update is expanded to separately handle the trial successes and failures, shown in line 18 and 20-25.

		MedleyDB				MixingSecrets				Internal			
		L_a	L_{lr}	L_m	L_s	L_a	L_{lr}	L_m	L_s	L_a	L_{lr}	L_m	L_s
Base graph		50.7	1.45	1.42	198	7.30	2.16	2.02	22.9	1.12	.951	.940	1.63
+ Gain/panning	`g`	.550	.583	.485	.550	.876	.856	.819	.973	.642	.619	.597	.734
+ Stereo imager	`sg`	.541	.564	.483	.553	.847	.834	.791	.928	.538	.616	.595	.727
+ Equalizer	`e sg`	.450	.453	.390	.504	.700	.698	.622	.780	.522	.497	.467	.626
+ Reverb	`e sg r`	.368	.361	.360	.390	.614	.601	.579	.674	.463	.451	.432	.517
+ Compressor	`ec sg r`	.315	.304	.297	.356	.558	.542	.512	.637	.396	.377	.347	.482
+ Noisegate	`ecnsg r`	.302	.288	.281	.353	.548	.532	.502	.625	.393	.374	.343	.480
+ Multitap delay (full)	`ecnsgdr`	.296	.288	.284	.324	.545	.529	.502	.618	.385	.369	.338	.465

Table 5: Per-dataset results of the mixing consoles with different processor type configurations.

		MedleyDB									MixingSecrets									Internal								
	τ	L_a	ρ	ρ_g	ρ_s	ρ_e	ρ_r	ρ_c	ρ_n	ρ_d	L_a	ρ	ρ_g	ρ_s	ρ_e	ρ_r	ρ_c	ρ_n	ρ_d	L_a	ρ	ρ_g	ρ_s	ρ_e	ρ_r	ρ_c	ρ_n	ρ_d
`MC`	–	.296	–	–	–	–	–	–	–	–	.545	–	–	–	–	–	–	–	–	.385	–	–	–	–	–	–	–	–
`BF`	.01	.305	.63	.35	.81	.54	.77	.67	.71	.53	.566	.65	.50	.82	.43	.71	.61	.77	.75	.402	.80	.76	.92	.61	.81	.85	.87	.80
`DW`	.01	.302	.56	.32	.79	.42	.74	.57	.56	.43	.561	.57	.47	.82	.22	.62	.54	.74	.55	.397	.74	.74	.82	.49	.70	.87	.86	.61
`H`	.001	.295	.44	.22	.73	.26	.61	.50	.54	.24	.550	.43	.35	.76	.13	.42	.43	.55	.38	.388	.59	.48	.77	.40	.57	.78	.77	.39
`H`	.01	.302	.61	.32	.81	.48	.74	.69	.71	.52	.563	.62	.49	.85	.34	.64	.60	.79	.65	.400	.77	.73	.93	.56	.75	.85	.86	.74
`H`	.1	.375	.83	.59	.93	.83	.92	.80	.84	.90	.648	.84	.70	.91	.75	.86	.83	.93	.91	.474	.93	.90	.99	.84	.92	.93	.95	.96

Table 6: Per-dataset results of the pruning. `MC`: mixing console. `BF`: brute-force, `DW`: dry/wet, and `H`: hybrid.

Algorithm 3 Music mixing graph search (dry/wet method).

Input: A mixing console G_c, dry tracks $\mathbf{S}$, and mixture y
Output: Pruned graph G_p, parameters $\mathbf{P}$, and weights $\mathbf{w}$

```
P, w ← Initialize(G_c)
P, w ← Train(G_c, P, w, S, y)
L_a^min ← Evaluate(G_c, P, w, S, y)
G_p ← G_c
for n ← 1 to N_iter do
    T_cand ← GetProcessorTypeSet(V)
    for t in T_cand do
        V_t, w_t ← Filter(V, t), Filter(w, t)
        N_t, r_t, m_t ← |V_t|, 0.1, 1
    end for
    while T_cand ≠ ∅ do
        t ← SampleType(T_cand)
        V̄_t, m̄ ← GetLeastWeightNodes(V_t, w_t, ⌊N_t r_t⌉)
        L_a ← Evaluate(G_p, P, w ⊙ m ⊙ m̄, S, y)
        if L_a < L_a^min + τ then
            L_a^min ← min(L_a^min, L_a)
            m ← m ⊙ m̄
            V_t ← V_t \ V̄_t
        else
            if N_t > 1 then
                r_t ← r_t/2
            else
                T_cand ← T_cand \ {t}
            end if
        end if
    end while
    G_p, P, w ← Prune(G_p, P, w, m)
    P, w ← Train(G_p, P, w, S, y)
end for
return G_p, P, w
```

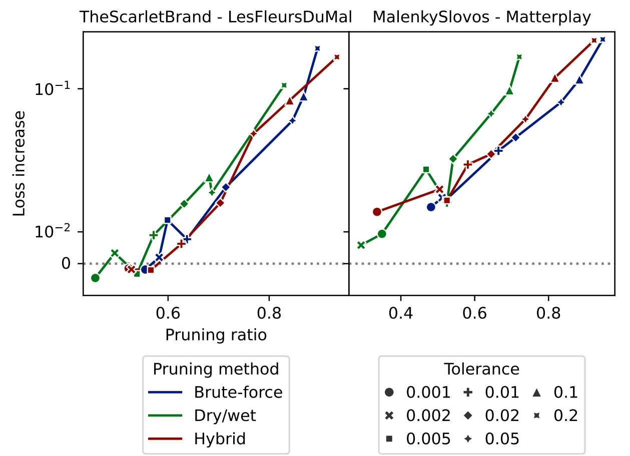


Figure 10: Loss increases from the mixing console and remaining processor ratios for different pruning methods and tolerances.

E. SUPPLEMENTARY RESULTS

- Table 5 and 6 report the per-dataset results on the mixing consoles and graph pruning, respectively.
- Figure 10 compares the pruning methods on 2 random-sampled songs using 7 tolerance settings from 0.001 to 0.2.
- Figure 11 shows multiple graphs obtained by pruning the same console (song) repeatedly.
- Refer to Figure 12-14 for more pruned graphs obtained with the default setting — hybrid method and $\tau = 0.01$.
- Figure 16-18 show more spectrogram plots.

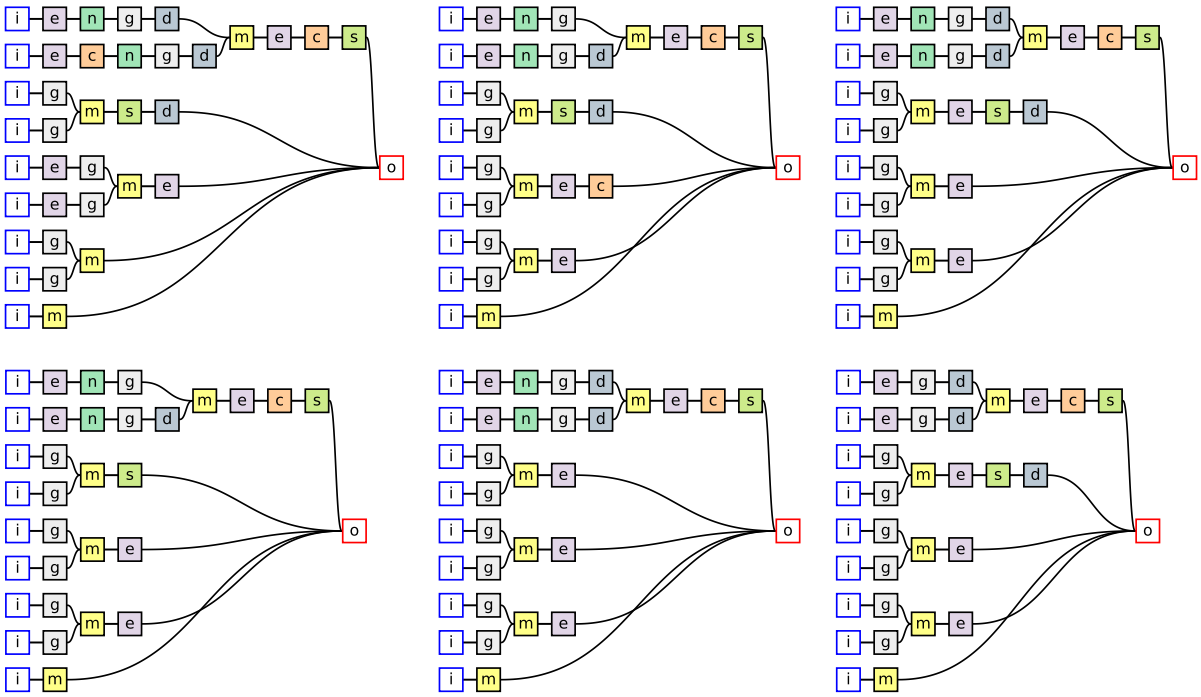
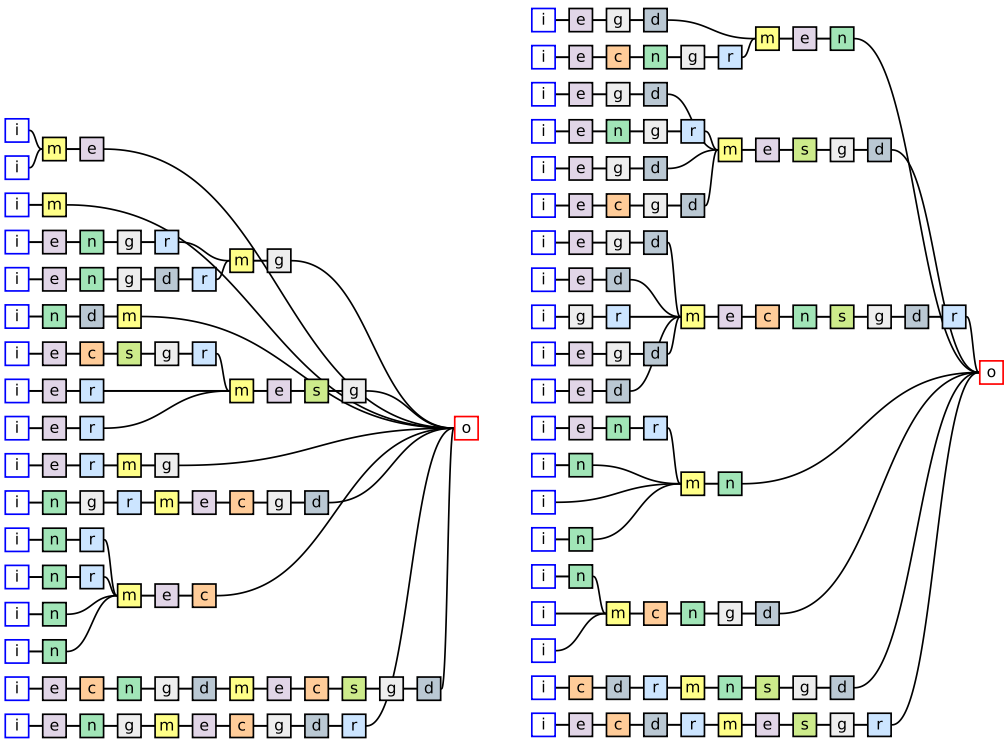
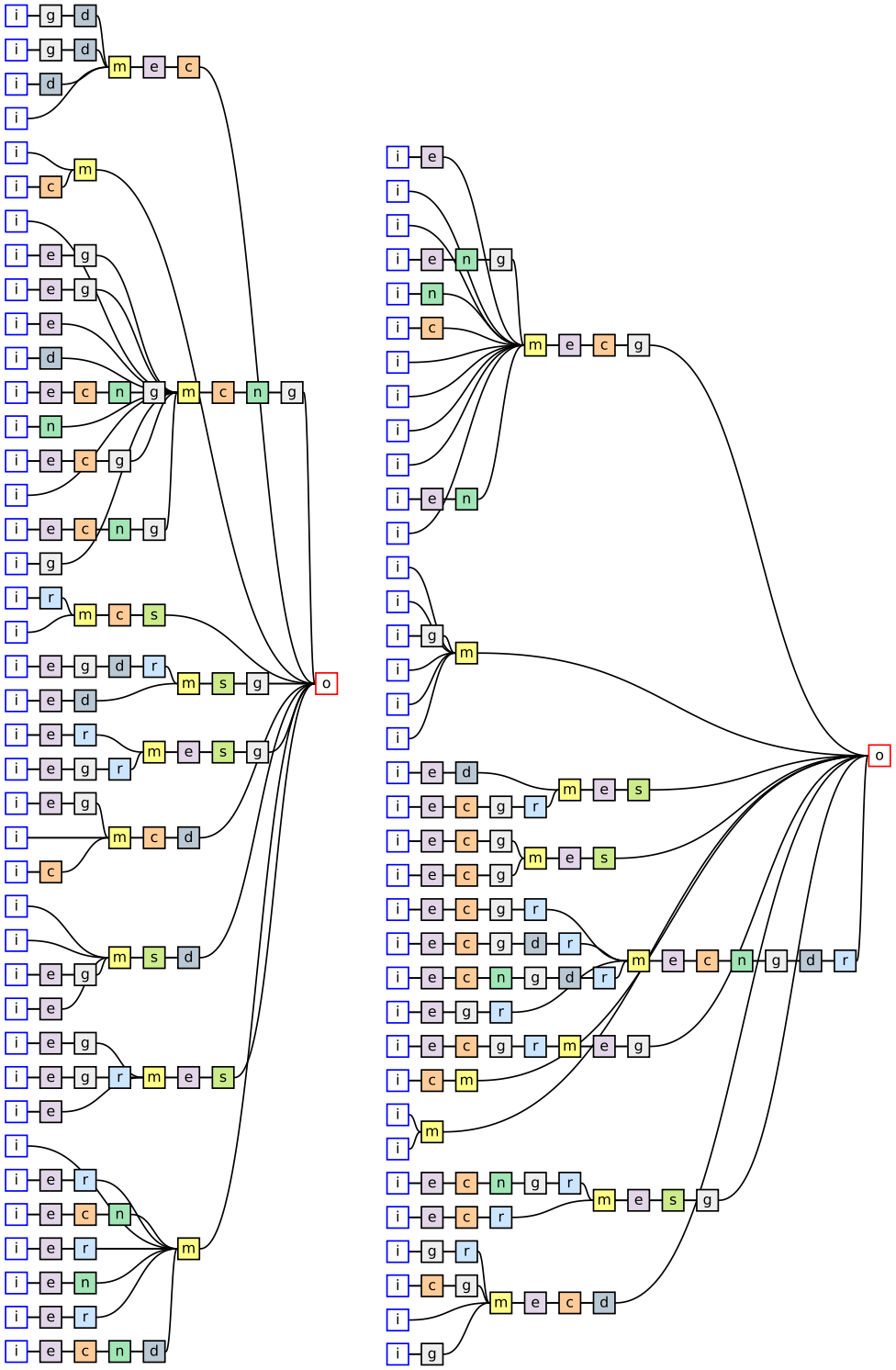


Figure 11: Each pruning run (default setting) yields a slightly different graph. Song: `EthanHein_GirlOnABridge`.



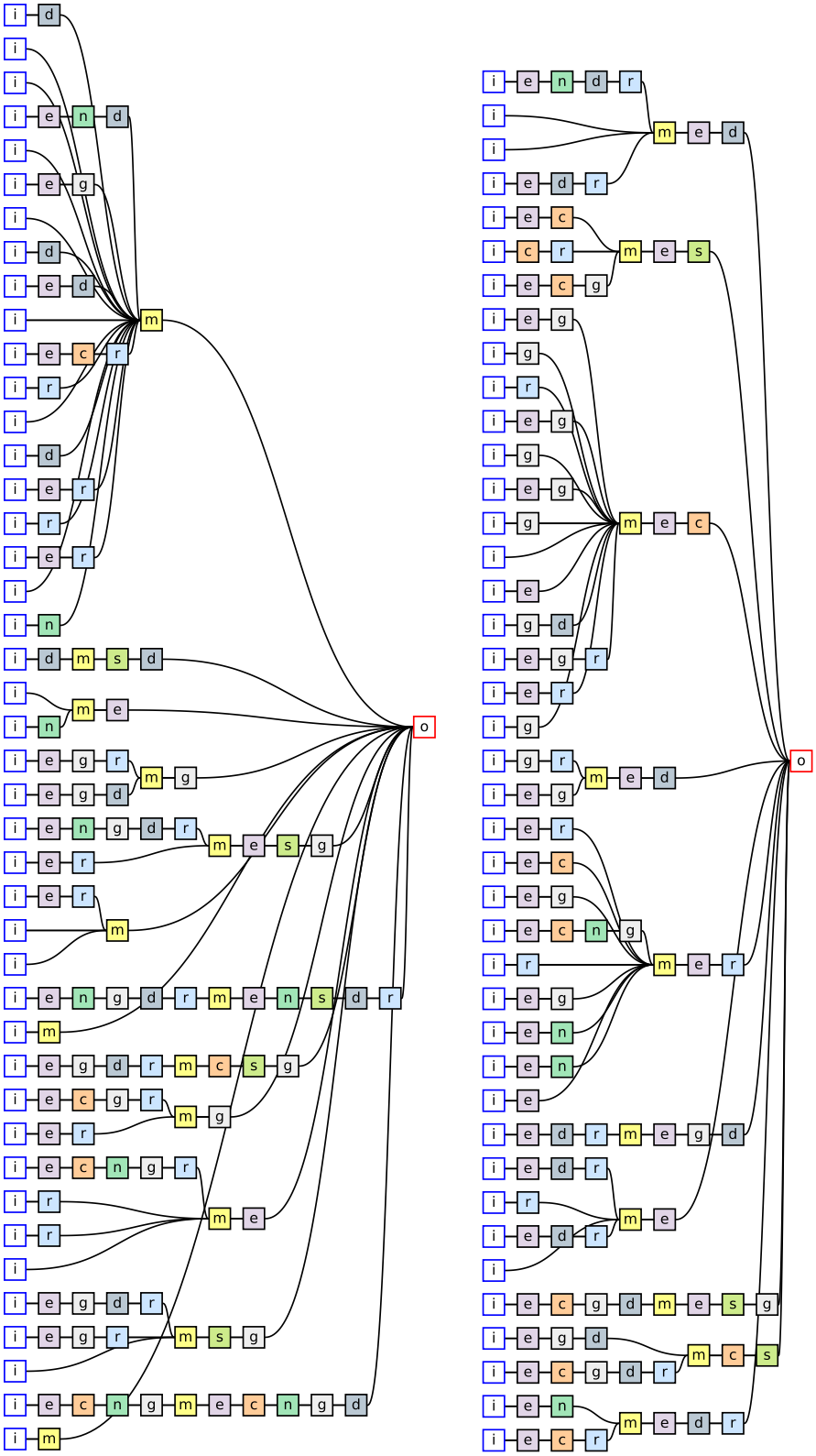
(a) `Internal_part1_65536` (b) `DonCamilloChoir_TrudeTheBumblebee`

Figure 12: Example pruned graphs (default setting). the number of tracks: $K \leq 20$.



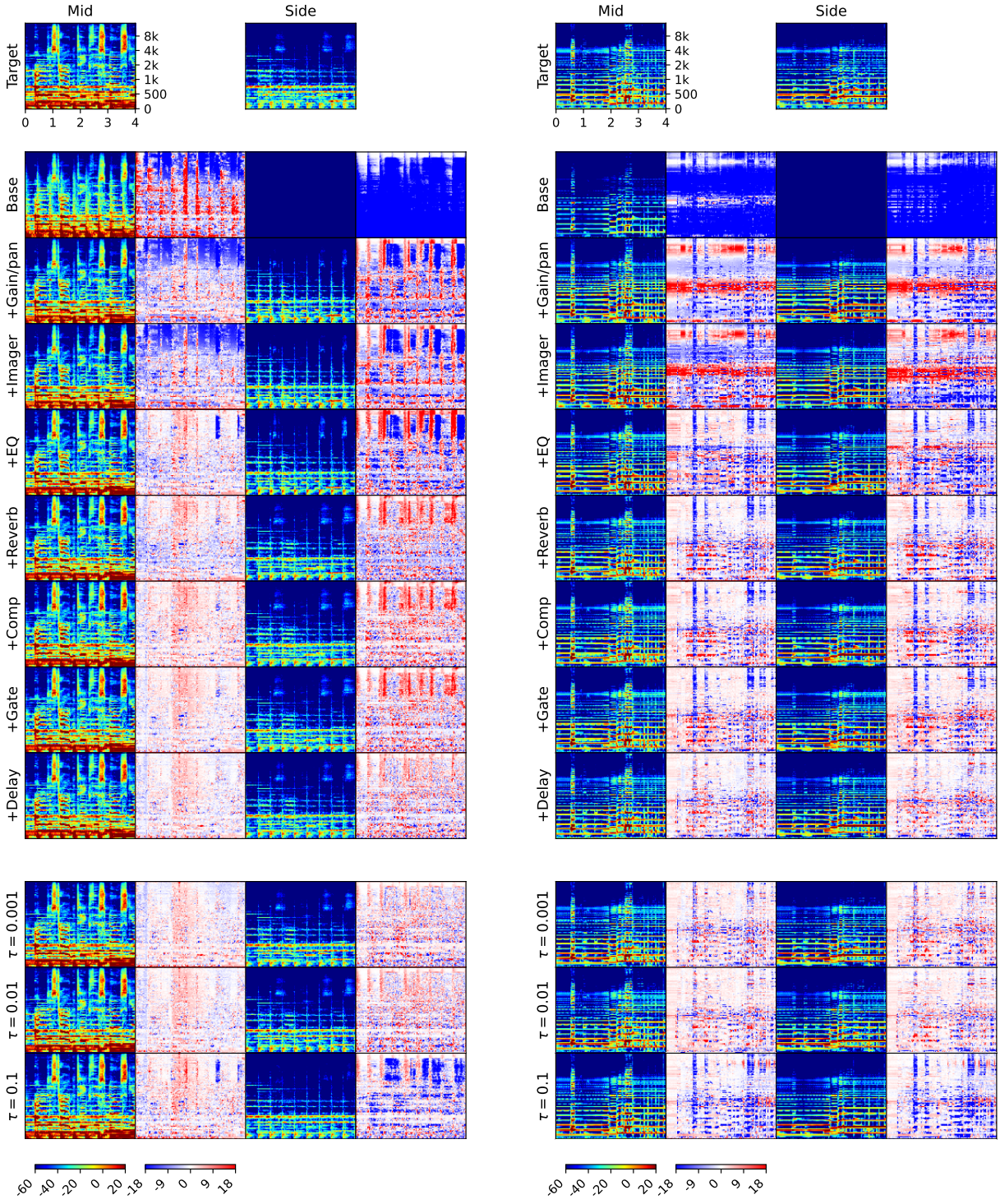
(a) `LittleTybee_TheAlchemist` (b) `RaftMonk_Tiring`

Figure 13: Example pruned graphs (default setting). the number of tracks: $K > 20$.



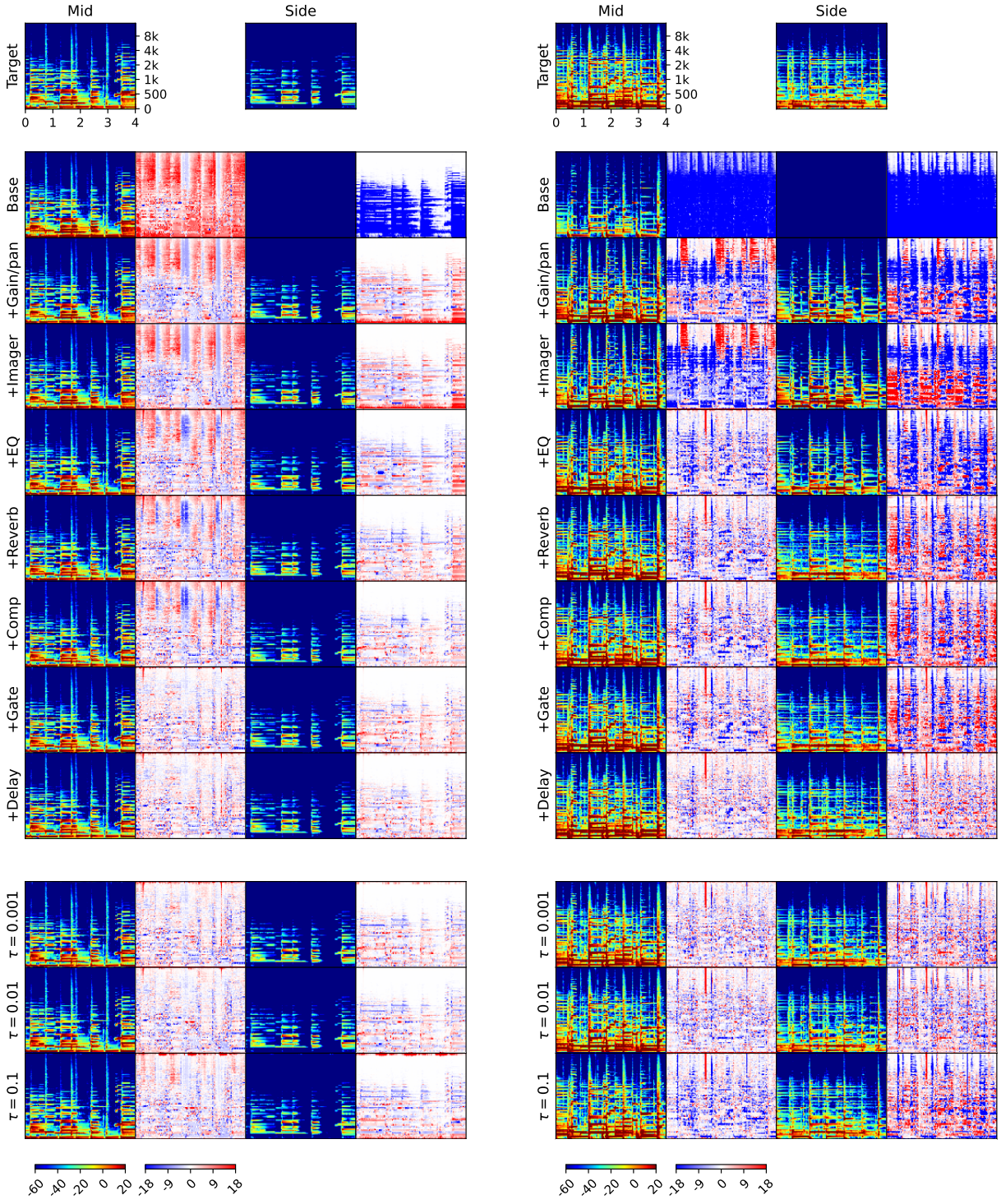
(a) `Internal_part2_67692` (b) `StevenClark_Bounty`

Figure 14: Example pruned graphs (default setting). the number of tracks: $K > 20$ (continued).



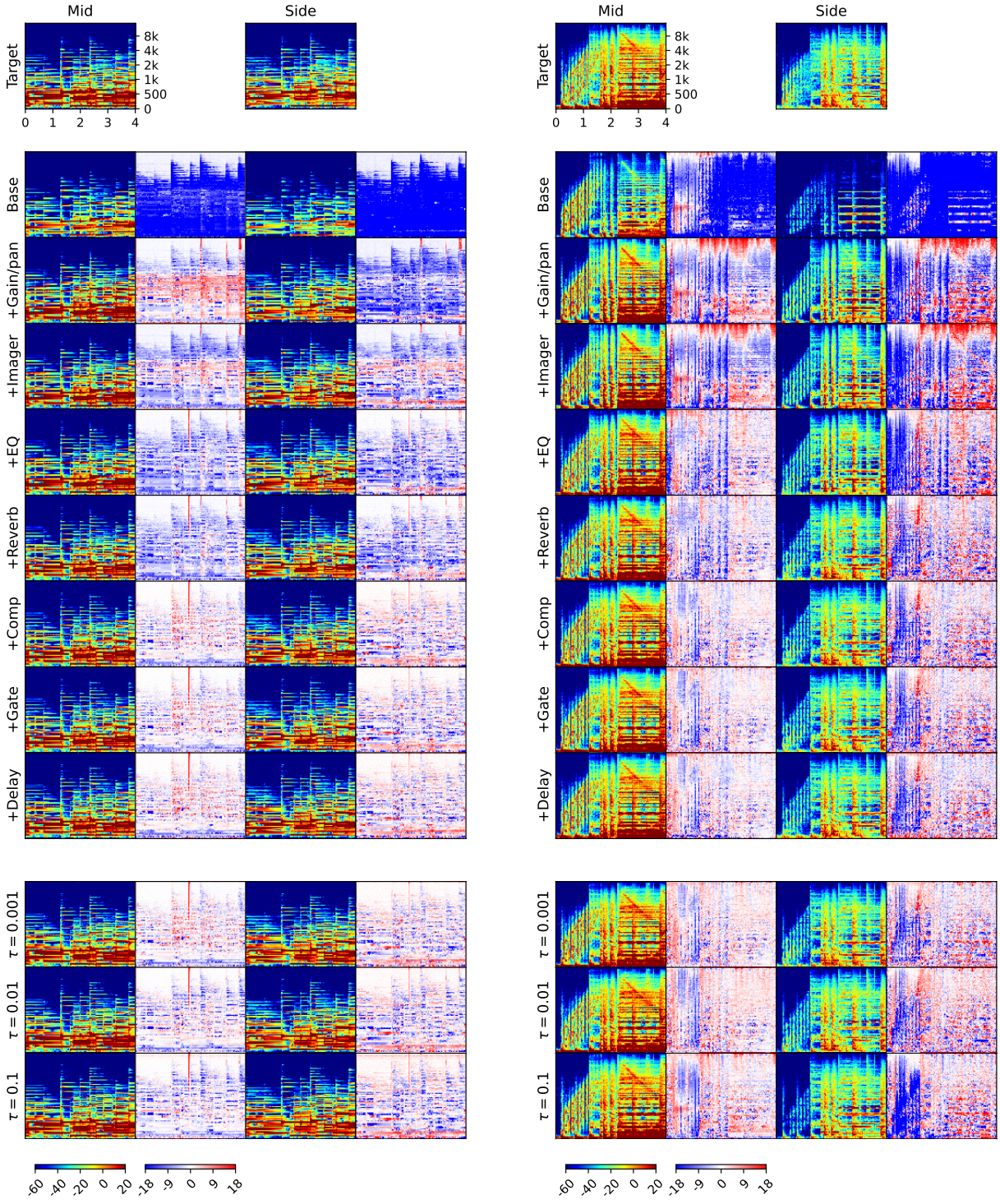
(a) `Torres_NewSkin` (b) `TablaBreakbeatScience_RockSteady`

Figure 15: Matching of target music mixes with mixing consoles and their pruned versions: `MedleyDB` dataset.



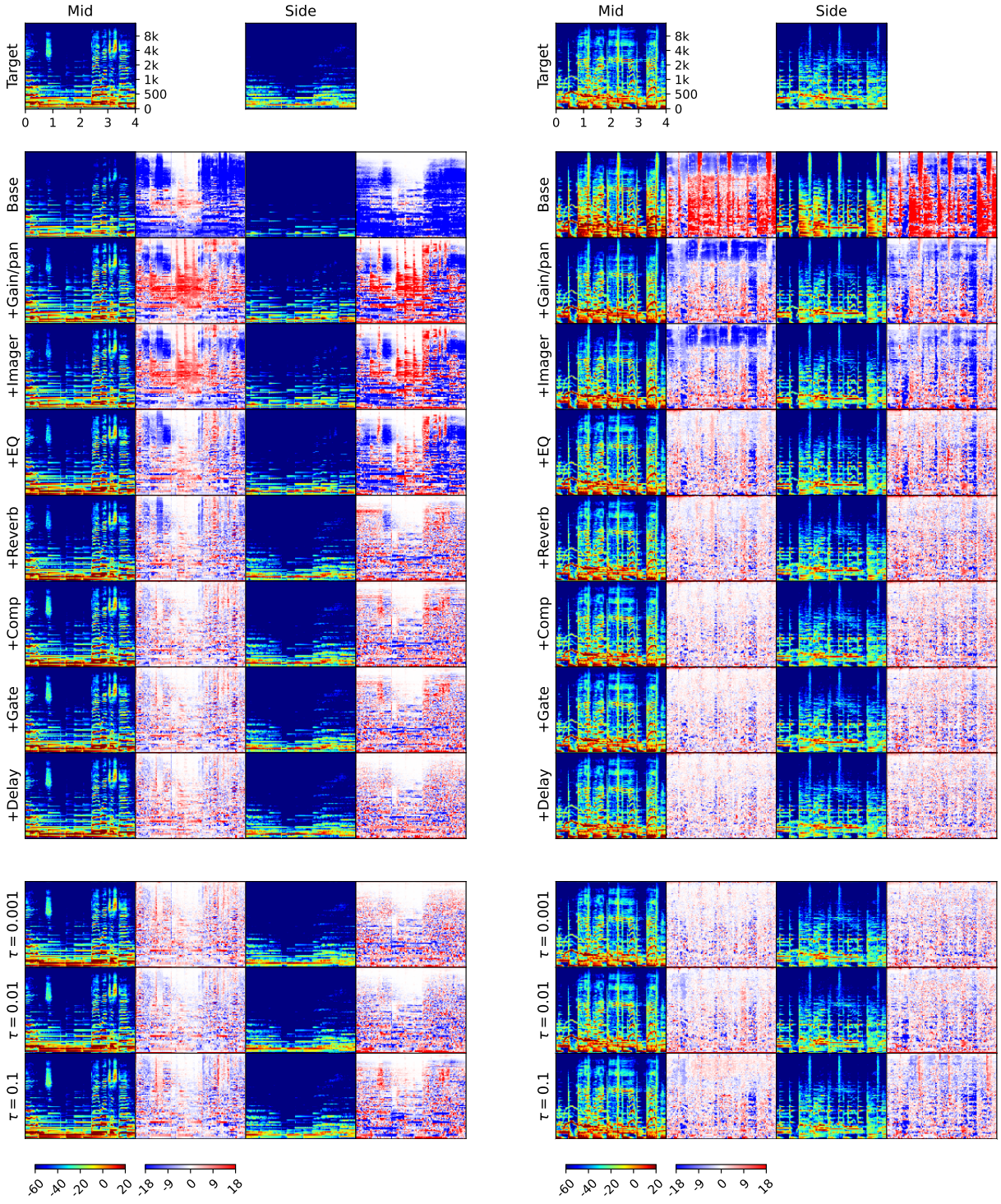
(a) `MusicDelta_SwingJazz` (b) `ChrisJacoby_BoothShotLincoln`

Figure 16: Matching of target music mixes with mixing consoles and their pruned versions: `MedleyDB` dataset.



(a) `HowlProject_IfIWereABell` (b) `IanDearden_TeraniaCreekWalking`

Figure 17: Matching of target music mixes with mixing consoles and their pruned versions: `MixingSecrets` dataset.



(a) `Internal_66680` (b) `Internal_67954`

Figure 18: Matching of target music mixes with mixing consoles and their pruned versions: `Internal` dataset.