Fast Discovery of Motivic Patterns in Symbolic Music via Lossy Compression

Adam James Wilson*
New York City College of Technology, City University of New York
awilson@citytech.cuny.edu

Abstract

Generative systems that react to live musicians require rapid analysis of musical data, which rules out deep learning models: they cannot be trained within the time constraints of live performance. But because analysis results are often transformed before use, we are free to reduce the parameters that undergo transformation to a small set of primitive states. We address this coincidence of constraint and opportunity with an algorithm for online discovery of maximal musical motives that achieves speed through lossy compression: the pitch and inter-onset-interval deltas for all pairs of events in a potential motive are reduced to two-bit values, conceptualized as the edges of a complete graph. Column-major bit-packing of edges into register-sized words makes per-note motive discovery cheap enough to maintain the maximal set incrementally, so that a query reads results rather than computing them.

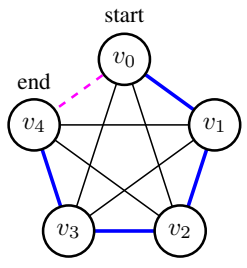
1 INTRODUCTION

The motive discovery algorithm described herein, referred to throughout as MD, is designed as a practical tool for improvisers who deploy real-time generative music systems informed by performance analysis. While there are many approaches to analysis that can yield useful data for generative processes, ours focuses on the most fundamental component of symbolic music structure: the motive. Motivic analysis is nothing new, and has well-documented approaches in the fields of music theory, music composition, and music cognition. Scholars who study the organization of music generally agree that motives represent the shortest meaningful groups of musical events (Auerbach, 2021; Lerdahl and Jackendoff, 1983). Motive length appears to correlate with short-term memory capacity (Snyder, 2000), and repetition of short groups of events, including perceivable transformations of those groups, is fundamental to the identification of motives (Dai et al., 2023; Hsiao et al., 2023; Meredith, 2006). MD treats motives as repeated strings of events, between 3 and 16 in length, and only keeps track of 'maximal' motives, meaning any motive that is a substring of another motive is disregarded.

There are many existing algorithmic approaches to motive discovery, and more broadly, pattern discovery. Suffix automata are frequently used for online pattern discovery, but perform suboptimally with the data structure adopted by MD (see Section 4.2). Systems like SIATEC, COSIATEC, and their derivatives (Björklund, 2022; Meredith, 2013, 2016, 2019; Wang et al., 2025), which are generally geared for precise offline musical analysis of polyphonic music, provide inspiration for this work (though MD is purposed for monophonic input). MD borrows heavily from the field of data compression, particularly the LZ77 family of compression algorithms (Ziv and Lempel, 1977; Gailly and Adler, 2026), for its approach to pattern discovery.

Before discovery can take place, we must decide what information is contained in a motivic pattern. A repetition categorizable as a motive can take many forms, since motives combine several foundational musical elements. For example, we could identify motives using only the sequence of intervals between consecutive pairs of pitches, or between consecutive pairs of pitch-class members of some scale or tuning system (two pitches belong to a pitch class if their ratio is a power of two); or, we could match on both pitch-class intervals and inter-onset intervals (IOIs); or, we could use pitch-class intervals and IOIs quantized to multiples of a short time interval. Our aim is to reduce the data represented by a motive to the strongest perceptually relevant parameters, and then reduce the alphabets of these parameters to a minimum. The more we compress, the likelier we are to find repetition, and the more material we make available for generative processes to develop. Because an improviser may use any set of pitches, up to and including the entire audible spectrum, and employ any imaginable rhythmic lexicon, lossy compression helps us establish a baseline description for motives that works for any type of music.

2 MOTIVE STRUCTURE



Six admissible arcs (lengths 3–5)

length 3: (v_0, v_1, v_2)
(v_1, v_2, v_3)
(v_2, v_3, v_4)

length 4: (v_0, v_1, v_2, v_3)
(v_1, v_2, v_3, v_4)

length 5: $(v_0, v_1, v_2, v_3, v_4)$

Count: $\binom{N-1}{2} = \binom{4}{2} = 6$

perimeter edge (walked)
wrap edge v_4–v_0 (excluded)
chord of K_5

Figure 1: $K_{N=5}$ polygon graph with linear perimeter $v_0 \rightarrow v_4$ (no wrap-around).

A motive can be thought of as a sequence of N events, or notes, each with a pitch, onset time, and duration, that distinguishes itself through repetition. For the moment,

*https://adamjameswilson.info

let's consider pitch alone. If we regard the motive as a cohesive structural unit, we must admit relationships between every pair of pitches in the sequence, not just consecutive pairs. This formal definition lends itself to the geometry of a complete graph, where pitches become vertices and edges become pitch deltas. However, while we wish to exploit all edges of the graph, we do not wish to admit all combinations of edges. We only consider induced complete subgraphs made from the set of all vertices forming contiguous, non-wrapping arcs of length 3 to N (our floor is 3 because it produces the smallest complete graph with a vertex connected to more than one edge). Using all combinations of edges would allow some subgraphs, or subset motives, to "skip" notes. Figure 1 shows the arcs for graph $K_{N=5}$ that delineate the lists of vertices for all considered subgraphs. The number of admitted subgraphs is simply ${}_{(N-1)}C_2$:

$$\binom{N-1}{2} = \frac{(N-1)(N-2)}{2} \quad (1)$$

Going forward, we dispense with polygon configurations of complete graphs in favor of the format shown in Figures 2 to 4, which better captures the sequential nature of the underlying musical data. Vertices in these graphs are ordered left-to-right according to the chronological sequence of associated note events. We use two instances of this structure to represent a sequence: one for pitches and one for inter-onset intervals, or IOIs. Separating them allows us to treat rhythm and pitch independently without an extraction step at the generative stage (for example, we could combine the pitch graph from one match with the IOI graph from another equal-length match), but the principal reason for separation is an implementation detail having to do with the interaction of the encoding schema, the range of motive lengths, the C23 standard (ISO/IEC JTC1/SC22/WG14, 2024), and modern PC architecture.

2.1 Encoding

Precise pitch and time intervals between events are important stylistic markers in music. We can regard musical events as points in a 2-dimensional plane, with pitch on the y-axis and time on the x-axis. A potential motive can be formed by drawing a line through a contiguous collection of points along the x-axis. The function produced is usually referred to in the literature as a contour, and the general shape of the contour is more important for recognizing a sequence of events than the precise disposition of events (Bartlett and Dowling, 1980; Schmuckler and Moranis, 2023). This suggests a rationale for reducing the precision of pitch and time intervals. Additionally, improvisers who use generative systems that act on performance analysis frequently want to introduce transformations that aren't implicit in the relationships between structures returned from analysis. In many cases, such transformations overwrite precise measurements of musical features.

MD compresses contour information to the maximum. Specifically, we reduce deltas between pitch pairs and IOI pairs to three possible states: delta positive, delta negative, and delta unchanged. These mark the edges of our complete graphs. The simplest version of this would be implemented on a ternary computer. Sadly, no commercial ternary architecture exists today, though there are historical examples, such as Setun, which was built in the Soviet Union in 1958 (Brusentsov and Alvarez, 2011). Instead, we use two-bit "crumbs" to represent each value: 0B00

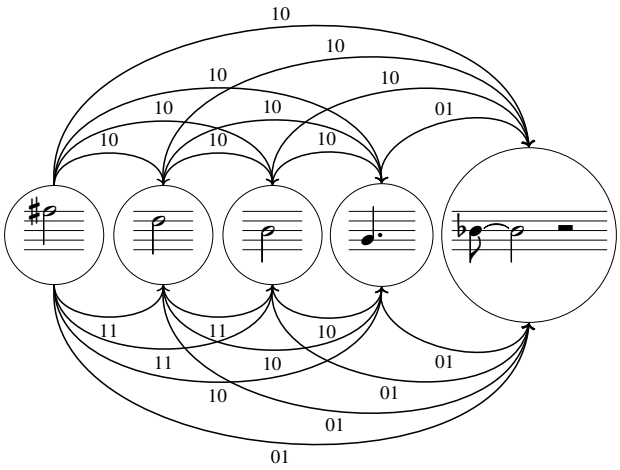


Figure 2: Crumb assignments for pitch deltas (top edges) and inter-onset interval (IOI) deltas (bottom edges).

for no value, 0B01 for delta positive, 0B10 for delta negative, and 0B11 for delta unchanged. See Figure 2, which combines two $K_{N=5}$ graphs, one with pitch-delta crumbs labelling edges (top set of arcs), the other with IOI-delta crumbs labelling edges (bottom set of arcs). The music fragment is taken from the beginning of the theme in John Coltrane's "Giant Steps" (Coltrane, 1959), shown in Figure 5. The number of edges for each graph, expressed in terms of vertices N, is ${}_NC_2$, which can be stored in $N(N-1)$ bits:

$$\binom{N}{2} = \frac{N(N-1)}{2} \text{ edges } \longrightarrow N(N-1) \text{ bits} \quad (2)$$

The software prototype of MD[1] is written in C and packaged to run as an external in the Max real-time audio processing environment (Cycling '74, 2026). The implementation targets modern 64-bit architectures, including arm64 and x86_64. Accordingly, we pack the crumbs for one complete graph into an unsigned integer whose length is set to the lowest multiple of 64 that is greater than or equal to the number of bits required. We start with the most significant bits and pad any unused positions with zeros. We pack bits in column-major order (e.g. vertices 0,1;0,2;1,2;0,3;1,3;2,3). This allows us to do a single XOR on the pitch and IOI graph pairs and then OR the results to attempt a match (see Algorithm 5). The position of the first mismatched bit can be used to determine the size of the largest matching subgraph. (In row-order, subgraph bits are not contiguous, forcing us to loop over candidate sizes, ANDing the graph XOR with a precomputed mask and testing for zero at each step.) An upper bound of 16 was chosen for the number of vertices N by taking into account (1) recommendations from scholars in the fields of music theory, perception, and cognition, and (2) technical limitations of C23 in combination with the hardware constraints of modern 64-bit personal computers.

A good starting point for a limit on motive length is 7 ± 2, which corresponds to the capacity of short-term memory. Short-term memory can also leverage "chunking," which allows multiple groups of 7 ± 2 elements. A good upper

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

limit on chunking, flattened to the list of the individual elements in the chunked groups, is probably around 25 (Snyder, 2000). This is close to 29, the length of the linearized de Bruijn sequence B(3,3) — the longest string over our three-symbol alphabet (0B01, 0B10, 0B11) in which every three-symbol window is distinct, corresponding to a 30-note passage whose consecutive deltas never repeat a three-window.

How close can we get to this limit given our real-time computing requirements? Let's look at a case somewhere in the middle of this range. A 12-note graph requires 132 bits in our schema, which calls for a 192-bit word. We must therefore consider word lengths (128, 192) that exceed the general purpose register widths on modern CPUs. Fortunately we have access to _BitInt() in the C23 standard, which allows us to instantiate and manipulate integers wider than 64 bits. Using the latest GCC and Clang compilers, we can get a common upper word-length limit of $2^{16}-1$ bits. There are caveats to some compiler/architecture combinations; for example, with Clang on arm64, we need to invoke an experimental flag, e.g. `-Xclang -fexperimental-max-bitint-width=256`, to achieve widths greater than 128 bits. More importantly, there are hardware considerations. An unsigned _BitInt(192) takes up three CPU registers. To compare two of these, we need six registers. While arm64 has 31 general-purpose registers, x86_64 has only 16. When we use up too many registers per calculation, the data is in danger of spilling to RAM, which can impact the efficiency of the proposed implementation. Choosing an upper bound of 16 for the number of vertices gives us a maximum word length of 256; comparing two of these takes up only half the register capacity of the lowest-capacity target architecture. Over this cutoff, we are entering territory that involves potential register pressure and spillover.

Recall the decision to separate pitch and IOI graphs. If we combine pitches and IOIs, replacing crumbs with four-bit "nibbles," our previous 12-note example would take 264 bits, which calls for a 320-bit word, and a comparison would take 10 registers, more than 60% of the minimum register space. We stick to the 256-bit independent graphs as an upper bound due to register constraints, but this width is likely more than adequate to cover the maximum-length sequence perceivable as a motive. Though there is some ambiguity regarding the precise boundary in event count between a motive and the second-shortest unit of musical organization, often called a phrase, the act of chunking likely signals a transition to phrase-level attention. While most users seeking discovery of motivic content will likely want to use seven or eight as a maximum value for N, our slightly larger limit allows for the potential to peek into the realm of phrase-length sequences without incurring a substantial performance hit.

2.2 Transitive chain problem

A final encoding consideration requires a step between raw data input and the transformation of pitch and IOI deltas into crumbs. Our implementation accepts fine-grained pitch data in cents (one cent corresponds to a frequency ratio of $2^{1/1200}$). It also looks at time intervals down to the millisecond. Problems become clear when we have pitch or IOI values that are extremely close to one another, e.g. 350¢ and 352¢, or 250ms and 248ms. The deltas between these pairs will be encoded as 0B01 and 0B10, respectively, but both should be encoded as 0B11 because the members of each pair are perceptually equivalent. Human musicians are far from precise, and even computer-generated events, especially those issuing from software running on general purpose computers, are prone to jitter. So, these situations will arise frequently and, if unaddressed, prevent perceptually relevant motivic repetitions from being discovered. The solution is to find the deltas below threshold T between pairs of consecutive raw events in the input buffer and collapse them to zero prior to encoding. However, if we loop over the unsorted list and process each pair as we go, we can introduce distortions. For example, given pitches P=$\{v_0$=300¢, v_1=200¢, v_2=100¢$\}$ and T=200¢, for all i where $\Delta(v_i, v_{i+1}) < T$, we set $v_{i+1} = v_i$. This is problematic because when we snap v_1 to v_0, the delta between v_1 and v_2 becomes viable, whereas we want it collapsed to zero. If we perform the same procedure but set $v_i = v_{i+1}$, snapping v_0 to v_1 and then v_1 to v_2, a delta less than T between the new v_0 and v_1 survives the process. To make sure that no sub-threshold deltas influence our encoded graph, we must sort the raw event values, find clusters where consecutive values are within T, and set the members of each cluster to an anchor value, as follows.

1. Sort values so that $v_{\sigma(0)} \leq v_{\sigma(1)} \leq \cdots \leq v_{\sigma(n-1)}$, where $\sigma(i)$ is the original index of the i-th smallest value.
2. Find each cluster $c_k = \{\sigma(i_k), \sigma(i_k+1), \ldots, \sigma(j_k)\}$, spanning sorted positions i_k through j_k: a maximal run where $v_{\sigma(m+1)} - v_{\sigma(m)} < T, \forall\, m \in \{i_k, \ldots, j_k-1\}$.
3. $\forall\, c_k,\ \forall\, m \in c_k\colon v_m \leftarrow v_{\sigma(i_k)}$

If we were to preserve the remaining raw values, anchoring to the cluster median would reduce contour distortion at cluster boundaries, but since delta magnitudes disappear when we classify them as negative, positive, or unchanged, we simply anchor to the first element of each cluster ($v_{\sigma(i_k)}$).

3 MOTIVE DISCOVERY ALGORITHM

The goal of the motive discovery algorithm is to return the set of maximal unique motives that appear more than once, as well as the number of repetitions per motive. If motive A is matched on a subgraph of motive B, motive A is discarded. If motive A is matched on a supergraph of motive B, motive B is discarded. At the generative phase, we can choose to use any subgraph of a matched motive, so we only care about the longest matches.

3.1 Pre-processing steps

1. Select a maximum motive graph order N.
2. Fill an input buffer of length N for pitch (cents) and another buffer of length N for IOIs (milliseconds). Note that we must capture the N+1-th event to get the last IOI.
3. For each input buffer, sort and look for clusters of consecutive value pairs with deltas less than threshold T (separate thresholds are used for pitches and IOIs). For each cluster, set all members equal to the first and update corresponding values in the associated input buffer.
4. Encode and store each input buffer as a complete graph. Note events (pitches or IOIs) become vertices and their deltas become edges. We encode deltas to one of three values: 0B01 for delta positive, 0B10 for delta negative, and 0B11 for delta unchanged. Delta bits are

packed into a single unsigned integer in the alpha order indicated in Figure 3, graph 0, starting from the most significant bit. For the first full pair of input buffers, we use the procedure shown in Algorithm 1. With each subsequent note, we pop index zero from the input buffers, left-shift, and push the new values. Since $(N-1)(N-2)$ bits of each new buffer are already encoded in the previous buffer, we shift those bits and compute only the bits for the $N-1$ new edges. This procedure is given in Algorithm 2 and illustrated in Figure 3.

Algorithm 1

Require:
B: input buffer of length N (pitches or IOIs)
W: container bit-width (64, 128, 192, or 256)

function INITIAL-BUF-TO-CRUMBS(B, W)
  $bits \leftarrow 0, pair\_idx \leftarrow 0$
  for $j \leftarrow 1$ **to** $N-1$ **do**
    for $i \leftarrow 0$ **to** $j-1$ **do**
      $d \leftarrow B[j] - B[i]$
      $sgn \leftarrow (d > 0) - (d < 0)$
      $lpos \leftarrow (W-1) - 2 \cdot pair\_idx$
      $rpos \leftarrow lpos - 1$
      switch sgn **do**
        case 1
          $bits \mid = (1 \ll rpos)$
        case -1
          $bits \mid = (1 \ll lpos)$
        case 0
          $bits \mid = (1 \ll lpos) \mid (1 \ll rpos)$
      end switch
      $pair\_idx \leftarrow pair\_idx + 1$
    end for
  end for
  return $bits$
end function

Once the input buffer is full, pre-processing steps are triggered on each incoming note. For the second and all subsequent graphs, we execute discovery.

The motive discovery algorithm is similar in mechanics to the LZ77 family of compression algorithms: at each new stream position it looks backward for prior occurrences of the same pattern. But instead of using the brute-force scan of every previous item employed in naive LZ77, we employ a hash-chain lookup keyed on the smallest subgraph ($K_{N=3}$) of a potential match, similar to what is done in the `zlib` compression library. This reduces the per-position cost from $O(M \cdot N)$ in the brute-force scan to $O(N)$ on average, where M is the current stream position and N is the maximum subgraph order. Aggregated over a stream of length L, this brings the average-case complexity from $O(L^2 \cdot N)$ to $O(L \cdot N)$, matching the reduction in complexity `zlib` achieves over naive LZ77. The worst case (all prior positions colliding into a single hash bucket) remains $O(L^2 \cdot N)$, as it does for `zlib`; in practice this is mitigated by the diversity of the keys and, if needed, by capping bucket length (not currently implemented). Because bucket selection reads the low bits of a hash, every key we build is right-aligned before insertion.

Since our method is not purposed for compression, but discovery, we catalog every maximal matching subgraph, which means keeping multiple repeats per position when they are not subgraphs of one another, and discarding a shorter repeat only when it is wholly contained in a longer one already in the catalog. This contrasts with LZ77, which commits to one back-reference per stream position (the longest, or one of two adjacent positions under lazy matching) so it can emit a single distance/length token.

Algorithm 2

Require:
B: input buffer of length N (pitches or IOIs)
$\emptyset$: previously packed graph (vertices $0..N-1$)
W: container bit-width (64, 128, 192, or 256)

function NEXT-BUF-TO-CRUMBS($B, \emptyset, W$)
  $shftd \leftarrow 0$
  for $k \leftarrow 1$ **to** $N-2$ **do**
    $low\_bit \leftarrow W - (k+1) \cdot (k+2)$
    $mask \leftarrow ((1 \ll 2k) - 1) \ll low\_bit$
    $shftd \mid = (\emptyset \;\&\; mask) \ll 2(k+1)$
  end for
  $start\_pair \leftarrow (N-1)(N-2)/2$
  for $i \leftarrow 0$ **to** $N-2$ **do**
    $d \leftarrow B[N-1] - B[i]$
    $sgn \leftarrow (d > 0) - (d < 0)$
    $pair\_idx \leftarrow start\_pair + i$
    $lpos \leftarrow (W-1) - 2 \cdot pair\_idx$
    $rpos \leftarrow lpos - 1$
    switch sgn **do**
      case 1
        $shftd \mid = (1 \ll rpos)$
      case -1
        $shftd \mid = (1 \ll lpos)$
      case 0
        $shftd \mid = (1 \ll lpos) \mid (1 \ll rpos)$
    end switch
  end for
  return $shftd$
end function

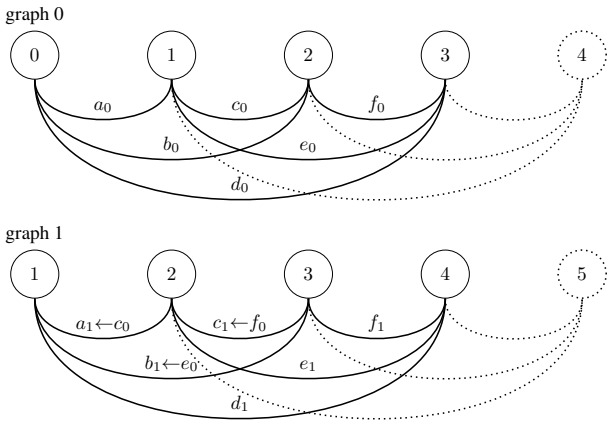


Figure 3: Carrying over bits from the common subgraph derived from the previous input buffer.

Other algorithms specifically designed for pattern discovery, such as Ukkonen's online suffix tree and Blumer's suffix automaton, achieve equal average-case complexity and even better worst-case complexity (Blumer et al., 1985; Ukkonen, 1995), but both are more complicated to implement and both require a post-construction pass to count repeats, whereas our algorithm maintains an incremental

count, which can be an important criterion for motive selection in a parallel generative process (see Section 4.2).

3.2 Discovery steps

1. Store the first complete graphs on P (array of pitch delta graphs) and I (array of IOI delta graphs), and formulate a 12-bit key from the "leftmost" order-three complete subgraphs for P_0 and I_0. Register the key on hashmap H.

With each subsequent complete graph, do the following (see Algorithm 3):

2. Check the minimum subgraph order we are looking for. If it is currently greater than three (the smallest possible), it means that in the last look-back we found a match, and the minimum subgraph order was set to the matched order plus one; if so, we decrement the minimum subgraph order and continue. We do this so that we don't get otherwise inevitable subgraph matches on subsequent passes. As long as there are no new matches, we keep the minimum subgraph order equal to matched subgraph order plus one minus the distance from the last index until we reach three.

Algorithm 3

Require:
P, I: arrays of packed pitch, IOI graphs
N: maximum complete-graph order (3..16)
W: container bit-width (64, 128, 192, or 256)
H: hashmap of arrays of previous matches keyed to "leftmost" N=3 complete subgraphs of P, I
mi: index of most recently packed graphs in P, I
m: minimum subgraph order to match

function FIND-REPEATS
  if $m > 3$ **then**
    $m \leftarrow m - 1$
  end if
  $key \leftarrow$ PREFIX3-FINGERPRINT(mi)
  $bucket \leftarrow H[key]$
  if $bucket =$ **null then**
    return
  end if
  for $k \leftarrow |bucket| - 1$ **down to** 0 **do**
    $i \leftarrow bucket[k]$
    $sg \leftarrow$ COMPARE-GRAPHS-$W(mi, i, m)$
    if $sg \neq -1$ **then**
      REGISTER-MATCH-$W(i, sg)$
      $m \leftarrow sg + 1$
      if $sg = N$ **then**
        return
      end if
    end if
  end for
end function

Note: The function PREFIX-INDEX-REGISTER, which registers the prefix3-fingerprint for each graph on H, is called on every graph, once just after the initial graph is packed, and again after every call to FIND-REPEATS, which executes following the packing of each subsequent graph.

3. Formulate the 12-bit key for the current graph (see Algorithm 4).

Algorithm 4

Require:
P, I: arrays of packed pitch, IOI graphs
N: maximum complete-graph order (3..16)
pos: index into P, I of the graph to fingerprint

function PREFIX3-FINGERPRINT(pos)
  $top\_P \leftarrow 0, top\_I \leftarrow 0$
  switch N **do**
    case $3 \ldots 8$
      $top\_P \leftarrow P[pos]$
      $top\_I \leftarrow I[pos]$
    case $9 \ldots 11$
      $top\_P \leftarrow (\texttt{uint64})(P[pos] \gg 64)$
      $top\_I \leftarrow (\texttt{uint64})(I[pos] \gg 64)$
    case $12 \ldots 14$
      $top\_P \leftarrow (\texttt{uint64})(P[pos] \gg 128)$
      $top\_I \leftarrow (\texttt{uint64})(I[pos] \gg 128)$
    case $15 \ldots 16$
      $top\_P \leftarrow (\texttt{uint64})(P[pos] \gg 192)$
      $top\_I \leftarrow (\texttt{uint64})(I[pos] \gg 192)$
  end switch
  return $((top\_P \gg 58) \ll 6) \mid (top\_I \gg 58)$
end function

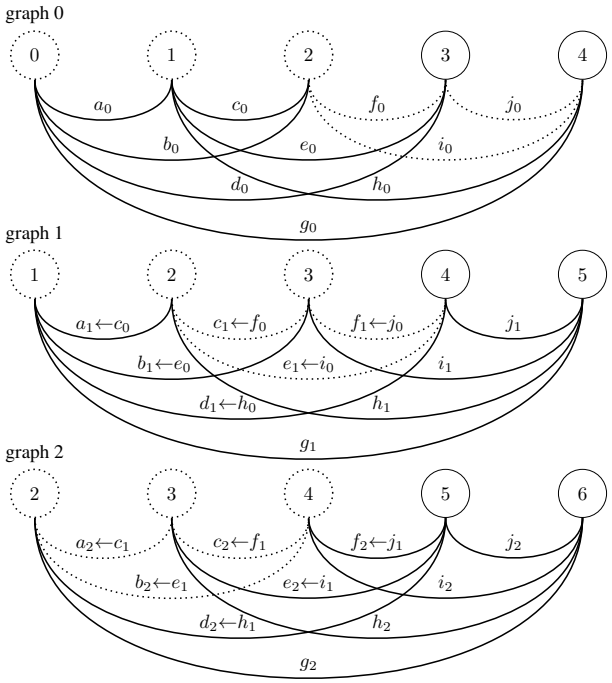


Figure 4: We only need to use the "left" side of each complete graph, including all subgraphs involving the first vertex, to search for matches; here we see edges $\{f_0, i_0, j_0\}$ eventually align with $\{a_0, b_0, c_0\}$ for comparison.

4. Check H for the key. If present, look in reverse order through the indexes of the previously stored graphs that have the same order-three prefix. Compare the candidate graphs using bitwise operations and return the order of the largest subgraph match, if it exists. Note that we only need to look at the complete subgraphs involving the "leftmost" vertex, which corresponds with the oldest captured musical event that is a member of the graph. Due to the overlap between graphs

(one is built starting on every successive event), subgraphs that don't involve the initial vertex eventually get matched (see Figure 4 for an illustration). When a match is discovered, we register the match (after a little post-processing; see the next step), set the discovered subgraph order to itself plus one, and stop looking if the subgraph order is N (we won't find any longer matches). See Algorithm 3 and Algorithm 5.

Algorithm 5

Require:
P, I: arrays of packed pitch, IOI graphs
N: maximum complete-graph order (3..16)
W: container bit-width (64, 128, 192, or 256)
a: index$_n$ of graphs to compare in P, I
b: index$_{n-m}$ of graphs to compare in P, I
min: minimum subgraph order to report a match
CLZ-WIDE-W: counts leading zeros in an unsigned integer
k_from_pairs: maps a count of matching edges to the largest matched subgraph order

function COMPARE-GRAPHS-$W(a, b, min)$
  $diff \leftarrow (P[a] \oplus P[b]) \mid (I[a] \oplus I[b])$
  if $diff = 0$ **then**
    return N
  end if
  $lz \leftarrow$ CLZ-WIDE-$W(diff)$
  $pack\_bits \leftarrow N \cdot (N-1)$
  if $lz \geq pack\_bits$ **then**
    return N
  end if
  $k \leftarrow k\_from\_pairs[\, lz \gg 1\,]$
  if $k < min$ **then**
    return -1
  end if
  return k
end function

5. We record matches in a $catalog$ hashmap keyed by graph content hashed together with matched subgraph order. The value for each key includes a count (number of times matched), the subgraph order, and the indexes of matched graphs in the arrays of bit-packed integers for pitch and IOI. We also maintain a $subsumed$ set containing subgraphs of maximal matches in $catalog$. Before registering a match, we first check if it appears on the $subsumed$ set; if it does, we ignore it (see Figure 6). If the match already exists on $catalog$, we bump its count. If it is a new match, we add it to $catalog$, add all of the subgraphs of the match to the $subsumed$ set, and erase all subgraphs of the match from $catalog$ (see Figure 7). See Algorithm 6.
6. Register the 12-bit key for the current graph on H, appending its index onto the associated array if the key exists.

Figure 5 shows the outcome of discovery with N=5 on the main theme from John Coltrane's "Giant Steps." For N=6, the matched graphs at indexes 10 and 13 accrue one additional note, while the other order-five subgraph matches remain.

3.3 Motive selection and emission

MD is designed to return musically relevant structures for exploitation by generative processes. Preserving maximal

Algorithm 6

Require:
P, I: arrays of packed pitch, IOI graphs
N: maximum complete-graph order (3..16)
W: container bit-width (64, 128, 192, or 256)
mi: index of most recently packed graphs in P, I
$match\_index$: earlier index whose graph matched graph at mi with order sg
sg: subgraph order for the match ($3 \leq sg \leq N$)
$catalog$:
  dict$\{content_key \rightarrow \{count, len, indices[\,]\}\}$
$subsumed$: set$\{content_key\}$
EXTRACT-CONTENT-W: formulates a $content\_key$ combining the prefix subgraphs of a given order from pitch and IOI graphs

function REGISTER-MATCH-$W(match\_index, sg)$
  $key \leftarrow$ EXTRACT-CONTENT-$W(mi, sg)$
  if $key \in subsumed$ **then**
  // Case 1: content already known subsumed by an earlier maximal match.
    return
  end if
  // Case 2: content already in catalog. Bump count, record new index.
  $v \leftarrow catalog[key]$
  if $v \neq$ **null then**
    $v.count \leftarrow v.count + 1$
    $v.indices$.PUSH_BACK(mi)
    return
  end if
  // Case 3: new content. Create catalog entry.
  $catalog[key] \leftarrow \{$
    $count \leftarrow 2,$
    $len \leftarrow sg,$
    $indices \leftarrow [match\_index, mi]$
  $\}$
  // Case 3a: mark every strict sub-arc size below sg as subsumed; erase from catalog if present.
  for $k \leftarrow 3$ **to** $sg - 1$ **do**
    for $o \leftarrow 0$ **to** $sg - k$ **do**
      $src \leftarrow match\_index + o$
      if $src \geq mi$ **then**
        continue
      end if
      $sub \leftarrow$ EXTRACT-CONTENT-$W(src, k)$
      $subsumed$.ADD(sub)
      $catalog$.ERASE(sub)
    end for
  end for
end function

order and motive counts in the $catalog$ hashmap allows us to query and return motives ranked by these properties.

The MD object for Cycling '74 Max features two messages: `maxcounts` and `maxgraphs`. Both messages must be supplied with an integer argument (1..256) indicating the number of desired results. When the user queries the object with one of these messages, the program walks $catalog$, maintaining a min-heap sorted by order when `maxgraphs` is invoked and by count when `maxcounts` is invoked. The min-heap is constrained to the largest 256 values returned from the walk. A tie-break is won by the match with the most recent occurrence. We heap-sort the query results in place and emit them one-at-a-time in im-

mediate succession, as shown in the four-line example below:

```
store 1 0x0000000000055755ULL 0x000000000005ba75ULL 5 19
store 2 0x0000000000069555ULL 0x00000000000ab5b6ULL 5 14
store 3 0x0000000000055a55ULL 0x000000000005ba75ULL 5 13
store 4 0x00000000000aaaa9ULL 0x00000000000fea55ULL 5 7
```

The 'store' prefix allows the values to be captured by the native Max `coll` object. From left-to-right, following 'store,' values represent (a) rank, (b) pitch graph, (c) IOI graph, (d) order, and (e) most recent index (points to an input event as well as the bit-packed graph). Max can only pass signed 64-bit integers between objects, and since our smallest integers are unsigned 64-bit words, we must format them as strings. The small performance expense of converting integers to strings and back could be saved by building some generative mechanism into MD itself, but MD is specifically geared for analysis; any process for expansion back into symbolic music values should be encapsulated in a separate object. A Max object designed for this purpose, MG (*motive generation*), is under development. Figure 8 gives an example of a potential MG expansion.

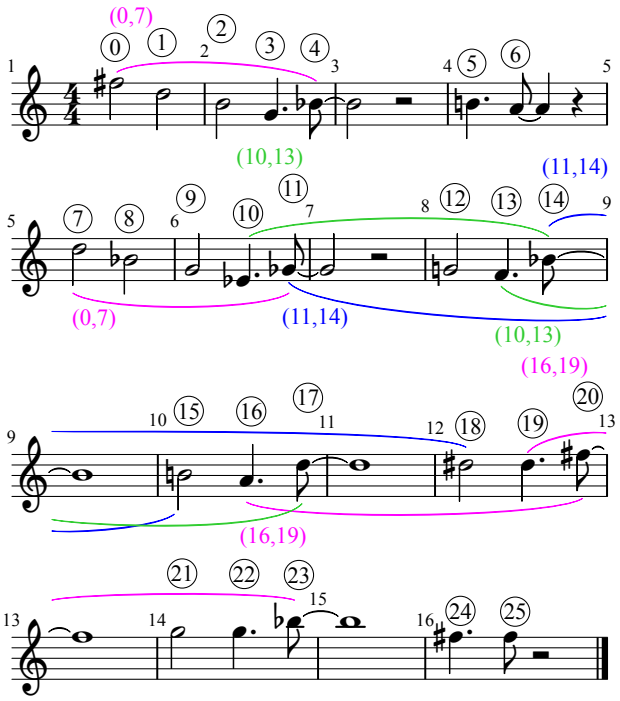


Figure 5: Motives with a maximum graph order N=5 discovered by MD in the theme from "Giant Steps." Identically colored arcs show matched pairs. Each match in a pair is labelled with the starting indices of the matched sequences.

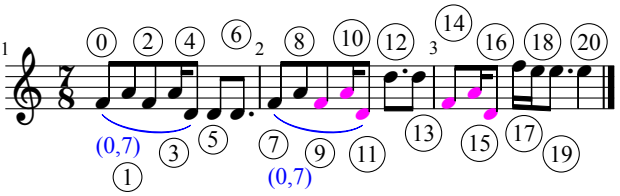


Figure 6: Magenta noteheads indicate subgraphs that are discarded as subsumed by a **prior** match. Blue index pairs indicate a retained N=5 match.

When obtaining motives via `maxcounts`, choosing an appropriate N is critical. For example, with N=16, a single 16-event repeat could contain multiple 3-event motives that repeat dozens of times. Choosing a smaller N mitigates this over-clustering. In fact, multiple instances of MD can be cheaply run simultaneously, so a user can keep track of small motivic cells in parallel with longer phrases.

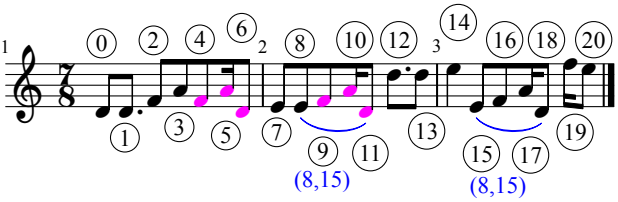


Figure 7: Magenta noteheads indicate subgraphs that are discarded as subsumed by a **newer** match. Blue index pairs indicate a retained N=4 match.

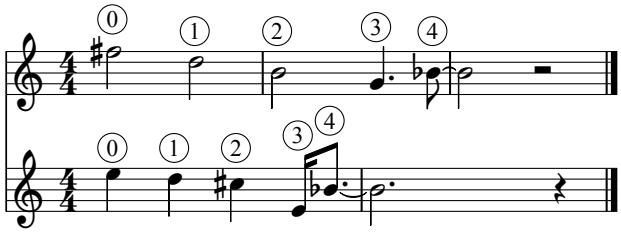


Figure 8: Bottom staff shows a possible decompression from the MD-encoded version of the motive from "Giant Steps," shown in the top staff.

4 PERFORMANCE AND COMPARISON

4.1 Performance

Since the number of hash map entries grows with each note processed, it takes progressively longer for MD to compare potential matches with the most recently packed graph. This suggests a performance benchmark that evaluates runtime for discovery against the last graph of a relatively long input sequence. Two tests were configured for this purpose: one measures the runtime for discovery against the last graph formed by the 18,000th note in a random sequence; the other times discovery against the last graph formed in a musical example, J.S. Bach's Partita in A minor for solo flute, BWV 1013 (Bach, 1723). Both performance tests encompass the full chain of core MD functions: threshold snapping, bit packing, repeat discovery, and registration/deregistration of matches.

For the random sequence, 18,000 was chosen as the number of events to cover the vast majority of solo performances with respect to maximum potential density and upper-bound duration. At an average of 30 notes/second, a blazingly fast performance speed that only virtuoso musicians like guitarist John McLaughlin can hope to sustain for limited periods of time, 18,000 notes equals 10 minutes of music. A more reasonable average speed is 5 notes/second, which equates to an hour of music at 18,000 notes. MD manages an average per-note discovery processing time of less than 0.0022 milliseconds for all N on the last graph of the sequence. See Table 1.

Performance testing with random data simulates MD's handling of sparsely patterned input. To evaluate performance on more patterned data, we turn to a real-world example. J.S. Bach is known for composing highly structured works

containing recurrent musical fragments. For our performance test on Bach's Partita, MD achieves an average per-note discovery processing time of less than 0.0013 milliseconds for all N on the last graph of the final movement. See Table 2.

N	per-note (μs)	catalog	subsumed
3	0.130	102	0
4	0.130	827	88
5	0.748	4682	694
6	1.206	4628	1105
7	1.165	4564	1113
8	1.105	4500	1184
9	1.094	4528	1108
10	1.217	4523	1180
11	0.168	4592	1048
12	1.979	4499	1167
13	2.107	4543	1148
14	1.916	4492	1165
15	2.102	4537	1125
16	2.036	4557	1127

Table 1: Average cost of the 18,000th-note pipeline over 1,000 iterations, per N, with final $catalog$ and $subsumed$ counts (arm64 Mac, Clang, no explicit compiler optimizations).

N	per-note (μs)	catalog	subsumed
3	0.042	61	0
4	0.055	162	57
5	0.093	302	199
6	0.124	406	443
7	0.139	462	755
8	0.127	468	1096
9	0.115	457	1425
10	0.168	440	1728
11	0.144	421	2005
12	0.214	403	2259
13	1.206	386	2486
14	0.586	366	2692
15	0.290	349	2874
16	0.307	335	3036

Table 2: Average cost of the final-note pipeline over 1,000 iterations for BWV 1013 (2,117 notes, four movements concatenated) with final $catalog$ and $subsumed$ counts (arm64 Mac, Clang, no explicit compiler optimizations).

4.2 Comparison

While SIATEC and its derivatives provide inspiration for this work, a direct comparison makes little sense. The SIATEC family includes batch algorithms that run in an offline setting and with much greater runtime complexity than real-time MD. SIATEC family algorithms operate on polyphonic "point sets," while MD works on graphs of compressed interval combinations derived from segments of monophonic music. Finally, the SIATEC algorithms are intended as musicological tools for precise analysis, while MD is geared for live performance and throws away some data irrecoverably.

A comparison between MD and a suffix automaton (SAM) might seem more intuitive, since a SAM indexes every factor of its input and is built online. However, MD's adopted data structure is an awkward fit for a SAM. A SAM consumes a linear sequence of symbols, whereas MD represents each motive as the upper triangle of a pairwise relation matrix over both contiguous and non-contiguous events. Forcing the structure into a SAM would require serializing each pair's pitch-sign and IOI-sign into a single 4-bit token, emitted in column-major order so that the pairs spanning any prefix of k events occupy the leading ${}_kC_2$ tokens contiguously. Since the resulting stream concatenates one such block per event, a bare SAM would match at arbitrary offsets — substrings beginning mid-block correspond to no subgraph — so each block must be prefixed with a reserved symbol and matches accepted only when they begin by consuming it. Traversal then never follows a suffix link or an unanchored transition out of the source, which effectively collapses the automaton to a trie of the anchored strings.

Despite this impediment, we tested MD's performance against both SAM and trie implementations (see Table 3) operating on our Bach and random corpora and preprocessed as described above. Results show that MD is faster than SAM and competitive with trie on construction. MD also maintains a smaller memory footprint than both. (Memory usage is similar for both tests on MD despite $catalog$ sizes being different because data also resides in the $subsumed$ set.) Finally, and most significantly, MD is vastly superior to SAM and trie when it comes to query performance, because MD amortizes the cost of the query process over the construction phase (building and pruning $catalog$), while trie and SAM must traverse their entire structure and filter for maximal substrings on every request.

	build	query	entries	memory
Bach BWV 1013 — 2,117 notes				
MD	2.14 ms	1.53 μs	335 catalog	6.9 MB
trie	2.43 ms	1,833 μs	191,165 nodes	8.8 MB
SAM	9.05 ms	2,014 μs	457,420 states	24.4 MB
uniform random — 18,000 notes				
MD	27.33 ms	13.77 μs	4,546 catalog	7.3 MB
trie	23.92 ms	23,368 μs	2,004,246 nodes	91.7 MB
SAM	239.74 ms	24,812 μs	3,626,264 states	193.7 MB

Table 3: Construction and query costs at N=16 for MD, a trie, and a suffix automaton over the same corpora. Query is top-10 by occurrence count over maximal repeated subgraphs. All three implementations were measured on aarch64 Ubuntu, virtualized on a MacBook M3, with identical Clang compiler settings (optimization flag `-O2`).

5 POTENTIAL IMPROVEMENTS

In addition to the planned MG (*motive generation*) object — essential for realizing the benefit of the real-time analysis provided by MD — it might be worthwhile to add a search for matches on graph transformations. These could include inversion, retrograde, and retrograde-inversion of pitch (P) and IOI (I) graphs, applied independently or in combination. For example, we could look for matches on retrograde(P_n) and inversion(I_n).

Support for polyphonic input could also be incorporated. One possible approach is to separate incoming music into multiple monophonic streams and bit-pack them independently, waiting for each stream to accumulate the Nth

vertex, and then treating the collection of graphs in one window of N-length overlapping streams as if they were a single monophonic sequence.

Finally, query messages returning random, min-count, or the shortest n maximal graphs could be added for convenience.

ACKNOWLEDGEMENTS

Generative AI was used in this project to build unit tests for the MD codebase, refactor some of the author's original code, and draft SAM and trie implementations of graph-based maximal pattern discovery for performance comparison.

The M*LIB project's native C hash table implementation (DICT) is used throughout the MD project for its extremely fast benchmarks in insertion, lookup, and deletion of keys (Allan, 2024; Pellisier, 2026).

REFERENCES

Allan, J. (2024). An extensive benchmark of C and C++ hash tables. Available at https://jacksonallan.github.io/c_cpp_hash_tables_benchmark. Accessed: 2026-08-14.

Auerbach, B. (2021). *Musical motives: A theory and method for analyzing shape in music*. Oxford University Press.

Bach, J. S. (1723). Partita in a minor for solo flute, BWV 1013. Estimated composition date.

Bartlett, J. C. and Dowling, W. J. (1980). Recognition of transposed melodies: A key-distance effect in developmental perspective. *Journal of Experimental Psychology: Human Perception and Performance*, 6(3):501–515.

Björklund, O. (2022). SIATEC-C: Computationally efficient repeated pattern discovery in polyphonic music. In *Proceedings of the 23rd International Society for Music Information Retrieval Conference*, pages 59–66, Bengaluru, India.

Blumer, A., Blumer, J., Haussler, D., Ehrenfeucht, A., Chen, M. T., and Seiferas, J. (1985). The smallest automaton recognizing the subwords of a text. *Theoretical Computer Science*, 40:31–55.

Brusentsov, N. P. and Alvarez, J. R. (2011). Ternary computers: The setun and the setun 70. In Impagliazzo, J. and Proydakov, E., editors, *Perspectives on Soviet and Russian Computing*, volume 357 of *IFIP Advances in Information and Communication Technology*, pages 74–80, Berlin, Heidelberg. Springer.

Coltrane, J. (1959). Giant Steps. Atlantic Records (SD 1311). Recorded May 4 & 5 and December 2, 1959.

Cycling '74 (2026). Max. https://cycling74.com. Computer software.

Dai, S., Yu, H., and Dannenberg, R. B. (2023). BPS-Motif: A dataset for repeated motif detection in symbolic music. In *Proceedings of the 24th International Society for Music Information Retrieval Conference (ISMIR 2023)*, pages 281–288, Milan, Italy.

Gailly, J.-l. and Adler, M. (2026). zlib: A massively spiffy yet delicately unobtrusive compression library. https://zlib.net. Reference implementation; hash-chain longest-match logic in `deflate.c`.

Hsiao, Y.-W., Hung, T.-Y., Chen, T.-P., and Su, L. (2023). Bps-motif: A dataset for repeated pattern discovery of polyphonic symbolic music. In *International Society for Music Information Retrieval Conference*.

ISO/IEC JTC1/SC22/WG14 (2024). ISO/IEC 9899:2024 (en) — Programming languages — C (Working Draft N3220). Technical report, International Organization for Standardization. Available at https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3220.pdf.

Lerdahl, F. and Jackendoff, R. (1983). *A generative theory of tonal music*. MIT Press, Cambridge, MA.

Meredith, D. (2006). Point-set algorithms for pattern discovery and pattern matching in music. In Crawford, T. and Veltkamp, R. C., editors, *Content-Based Retrieval*, volume 6171 of *Dagstuhl Seminar Proceedings (DagSemProc)*, pages 1–23, Dagstuhl, Germany. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

Meredith, D. (2013). COSIATEC and SIATECCompress: Pattern discovery by geometric compression. *Proceedings of the 14th International Society for Music Information Retrieval Conference (ISMIR 2013)*, pages 263–268.

Meredith, D. (2016). Analysing music with point-set compression algorithms. In Meredith, D., editor, *Computational Music Analysis*, pages 335–366. Springer, Cham, Switzerland.

Meredith, D. (2019). RECURSIA-RRT: Recursive translatable point-set pattern discovery with removal of redundant translators. In Cellier, P. and Driessens, K., editors, *Machine Learning and Knowledge Discovery in Databases: International Workshops of ECML PKDD 2019*, pages 485–493, Würzburg, Germany. Springer.

Pellisier, P. (2026). M*LIB: A library of generic and type safe containers / data structures in pure C language. https://github.com/P-p-H-d/mlib. Accessed: 2026-08-14.

Schmuckler, M. A. and Moranis, R. (2023). Rhythm contour drives musical memory. *Attention, Perception, & Psychophysics*, 85(7):2502–2514.

Snyder, B. (2000). *Music and Memory: An Introduction*. MIT Press, Cambridge, Mass.

Ukkonen, E. (1995). On-line construction of suffix trees. *Algorithmica*, 14(3):249–260.

Wang, J.-Y., Kuo, Y.-C., and Su, L. (2025). Improving motif discovery of symbolic polyphonic music with motif note identification. *Transactions of the International Society for Music Information Retrieval*, 8(1):353–371.

Ziv, J. and Lempel, A. (1977). A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343.