1. THE SIATEC-C ALGORITHM

The SIATEC-C algorithm computes TECs of patterns in a point-set D, such that, the IOI between adjacent points in a pattern is at most a given threshold δ. The patterns are produced by computing all MTPs in the input point-set and then cutting them at gaps that exceed the IOI threshold δ. SIATEC-C also avoids producing small patterns when the points covered by the pattern are already covered by a larger discovered pattern. The main goal of SIATEC-C is to produce candidate patterns for further processing in a computationally efficient way. By using a point-set representation, SIATEC-C can discover patterns in polyphonic music and also polyphonic patterns. The algorithm is loosely based on the SIATEC algorithm, the full name of the algorithm is thus SIATEC with pattern *cutting*. By restricting the pattern discovery to patterns without large gaps, SIATEC-C can avoid producing patterns with isolated members while reducing the running time and memory footprint. This approach aims at similar results as compactness trawling by [1]. Cutting patterns at large IOI gaps has also been suggested by [2].

The algorithm begins by sorting the input point-set in ascending lexicographical order to produce the point-set D_s. The variables T and W are used for tracking a sliding window for each point in D_s in the onset dimension. The sliding windows are used in computing MTPs in order to restrict the number of MTPs that need to be kept in memory simultaneously. The array T keeps track of the index from where to continue computation on each iteration, and the array W keeps track of the upper bounds of the windows. The indexing in the pseudocode starts at 1 and array access is denoted by brackets. For T the value at index i stores the index in D_s for where the next sliding window starts. The value at index i of W stores the upper bound of the sliding window for ith point of D_s. The indices in T are initialized to the range from 1 to n (line 4) and on line 5 the upper bounds are initialized for each point $p \in D_s$ to be $p.x + \delta$. The values in the array C keep track of the size of the largest pattern occurrence that covers the corresponding point in D_s. The initial values for the cover array C are set to 0.

The *difference index* structure I is computed by the COMPUTEDIFFINDEX function described in algorithm 2. Difference vectors between all pairs of points, p_i and p_j, for which the IOI between the points does not exceed the theshold δ, are computed in the main loop on lines 3–8. The differences along with the index-pairs $\langle i, j \rangle$ are stored in the intermediate array I'. The array is sorted in ascending lexicograhpical order by the difference vectors and indices. On line 10 the sorted array is partitioned by the difference vectors, so that an array of entries of the form $\langle v, [\langle s_1, t_1 \rangle, \ldots, \langle s_i, t_i \rangle] \rangle$ is created. Each entry contains a difference vector v and the corresponding *source* and *target* indices. The source indices s_i are the indices of points in D_s that can be translated by v within D_s, and the corresponding target indices are of the points that are produced by translating the point at the source index by v. The array I thus has a single entry for each difference vector v between points in D_s such that their IOI does not exceed δ. Partitioning can be implemented by simply iterating through the sorted array and joining adjacent elements with the same difference vector. The index structure I is sorted in ascending order of difference vectors and all source and target indices for an entry are also in ascending order.

Algorithm 1 SIATEC-C Algorithm

```
1:  function SIATEC-C(D, δ)
2:      D_s ← SORT_Lex(D)
3:      n ← |D_s|
4:      T ← [1, 2, ..., n]
5:      W ← INITWINDOWBOUNDS(D_s, δ)
6:      C ← [0, 0, ... 0] of n zeros
7:      I ← COMPUTEDIFFINDEX(D_s, δ)
8:      while T[1] ≤ n do
9:          M ← COMPUTEMTPSINWINDOW(D_s, T, W)
10:         M' ← CUTANDSORT(M, δ)
11:         for P ∈ M' do
12:             if IMPROVESCOVER(P, C) then
13:                 τ ← FINDTRANSLATORS(P, I, D_s, C)
14:                 OUTPUTTEC(P, τ)
```

Algorithm 2 SIATEC-C: Compute the difference vector index

```
1:  function COMPUTEDIFFINDEX(D_s, δ)
2:      I' ← []
3:      for i ← 1 to |D_s| − 1 do
4:          for j ← i + 1 to |D_s| do
5:              p_i ← D_s[i]; p_j ← D_s[j]
6:              if IOI(p_i, p_j) > δ then
7:                  break
8:              I' append ⟨p_j − p_i, ⟨i, j⟩⟩
9:      I ← SORT_Lex(I')
10:     I ← PARTITIONBYDIFF(I)
11:     return I
```

In the main loop of the SIATEC-C algorithm (lines 8–14 of 1), MTPs are computed for translation vectors within the sliding windows by the COMPUTEMTPSINWINDOW function. The MTPs are computed by first computing all translations between pairs of points where the target point is within the sliding window of the source point. The indices of the source points are stored in pairs with the tranlations. The array thus produced is sorted in ascending lexicographical order and partitioned by the translation vectors. The function is otherwise equal to the SIA algorithm [3], except that the difference vectors are limited by the sliding windows defined by the arrays T and W, and the indices of the MTP and its translated occurrence are also stored. The sliding windows are used to avoid keeping all $O(n^2)$ differences in memory at the same time. On each iteration the indices in T are updated to the point just outside the current window and then the sliding window upper bounds in W are incremented by δ.

The produced MTPs can have gaps in them that exceed

Algorithm 3 SIATEC-C: Find translators and update cover

```
1:  function FindTranslators(P, I, D_s, C)
2:      V ← VEC(P)
3:      v ← V[1]
4:      A ← { t | ⟨s, t⟩ ∈ FindIndices(v, I) }
5:      for i ∈ [2, ..., |V|] do
6:          v ← V[i]
7:          A' ← FindIndices(v, I)
8:          A ← { t | ⟨s, t⟩ ∈ A' ∧ s ∈ A }
9:      l ← P[|P|]
10:     τ ← { D_s[i] − l | i ∈ A }
11:     C ← UpdateCover(P, A, C, I)
12:     return τ
```

the threshold δ. Thus the MTPs are cut on line 10 to produce the set of patterns M', where the IOI between no adjacent patterns points exceeds δ. The patterns are also sorted in descending order of size to ensure that larger patterns are handled first. The function ImprovesCover checks if the pattern, or its translated version, is larger than any of the patterns that cover the same points. This is achieved by using the indices stored with the patterns and the array C. A pattern is considered to improve the cover only if it improves the cover value of at least one point. This step reduces the number of small and duplicate patterns that would be otherwise output by the algorithm. Small patterns may be output by the algorithm even if a larger pattern covering the same points is discovered. This occurs in the case that the small pattern is found on an earlier iteration of the main loop (lines 8–14).

Instead of using sliding windows to keep the space complexity subquadratic, the MTPs could be computed using the method by [4]. The threshold value δ defines the width of the sliding windows, however, a separate window width parameter could be used. This choice is made mainly for simplifying the analysis of the algorithm.

The FindTranslators function described in algorithm 3 is used for finding all translators of a pattern P by using the difference index I and the vectorized representation of P. The function also updates the cover array C.

The case of a single point pattern, $|P| = 1$, needs to be handled separately. The description of handling a single point is omitted from the FindTranslators function, as it is a simple procedure: the translators are computed by computing all difference vectors between the only point in the pattern and all other points in D_s. The cover is updated by iterating through C and updating any zero values to 1. Patterns with a single point can also be discarded altogether as they are unlikely to be musically important patterns.

The algorithm for finding translators works by finding translationally equivalent prefixes of P, and extending the prefixes until they are full occurrences of P. The prefixes are not kept in memory, instead the array A keeps track of the indices of the last points of translationally equivalent prefixes in D_s. The algorithm begins by computing V, the vectorized representation of P, and using its first diffence v to initialize A. The FindIndices function returns the array of source-target index pairs from I for a given vector. As the entries in I are in ascending order of difference vectors, FindIndices can be simply implemented using binary search on the difference vectors of the entries. On line 4 A is initialized to contain the target indices of v in D_s. At this point A thus contains the indices of the last points of all translationally equivalent occurrences of a two-point prefix of P.

After initialization, the prefixes are extended to cover the entire length of P in the main loop on lines 5–8. The vector v is updated to the next difference difference in V, that is, the next difference vector by which the prefixes can be extended. The index pairs corresponding to v are stored in a temporary variable A' on line 7. In order to find the indices of points that can be used to extend the prefixes, while keeping the condition of translational equivalence, the indices in A are matched with the source indices in A' and the corresponding target indices in A' are returned. Finding the matching indices requires computing the intersection of A with the source indices of A'. As the indices stored in the entries of I are in ascending order, the intersection can be computed in linear time by iterating through both arrays once while collecting the corresponding target indices to an array. The array containing the intersection is thus also in ascending order. At the end of each iteration A contains the indices of the last points of translationally equivalent prefixes of P of length $i + 1$. After the loop has finished, A contains the indices of the last points of all patterns that are translationally equivalent to P. The translators of the pattern P are computed on line 10 by computing the differences between the last point of P and the points at the indices stored in A.

The cover C is updated on line 11 by using the indices stored in A. As A contains the indices of the last points of all patterns that are translationally equivalent to P, a similar traversal using $VEC(P)$ and I can be used to find all indices of points covered by $TECP, D_s$. By reversing the order of traversal, and using the values in A as the initial values, the traversal will only find the indices of points covered by $TEC(P, D_s)$. During the traversal each value in C is updated to be $|P|$ if the current value at the index is not already at least $|P|$.

The SIATEC-C algorithm does not output TECs of MTPs, however, its goal is to produce patterns similar to MTPs but without isolated members. Therefore the correctness of the MTP computation in SIATEC-C warrants a brief consideration. The correctness of the SIA algorithm [3] is based on finding all points translatable by the same vector in the input point-set. In SIA this is achieved by computing all difference vectors between points, storing the vectors along with the indices of points, and then partitioning by the vectors to find all points translatable by the same vector. With windowing in SIATEC-C, it is required that all difference vectors within the point-set that are equal are computed on the same iteration. The windowing in SIATEC-C uses the onset dimension, thus all differ-

ences with the same onset dimension are computed on the same iteration. The indices stored in T are used for convenience, unlike in SIAR [1] which limits the windows by indices. Other dimensions are not considered in the sliding windows, thus all difference vectors that are equal are found on the same iteration and SIATEC-C produces all MTPs in the input point-set.

Finding all translators of a pattern P in FINDTRANSLATORS is shown to be correct in the following theorem.

Theorem 1.1. *Let P, $|P| > 1$ be a pattern that occurs in point-set D_s such that no IOI between adjacent points in P exceeds δ. Let I be the difference index computed for D_s and δ. Then* FINDTRANSLATORS *returns all translators of P in D_s.*

Proof. All references to lines of pseudcode in this proof refer to algorithm 3. Recall that $P \equiv_\mathrm{T} Q \iff VEC(P) = VEC(Q)$ [3]. Thus finding all patterns translationally equivalent to P can be achieved by finding all patterns whose vectorized representation is equal to $VEC(P)$.

The invariant of the FINDTRANSLATORS algorithm is that A contains the indices of the last points of all patterns $Q_i \subset D_s$ such $VEC(Q_i) = VEC(P)[1, i]$, where $VEC(P)[1, i]$ denotes a prefix of length i and $i = 1$ initially. When $i = |VEC(P)|$ then $VEC(Q_i) = VEC(P)$.

The array A is initialized with the target indices of points that can be translated by $VEC(P)[1]$ in D_s. Thus after initialization on line 4, A contains the last indices of points belonging to patterns such that $VEC(Q_1) = VEC(P)[1, 1]$.

On each iteration of the loop on lines 5–8 the source and target index pairs of all points translatable by $VEC(P)[i]$ are retrieved from I. Of these index pairs, the ones whose source index matches a target index stored in A are used to produce the updated array A. Thus at the end of each iteration, A contains the last indices of patterns Q, for which $VEC(Q_i) = VEC(P)[1, i]$. All required index pairs can be found in I, as there are no IOI gaps larger than δ in P and I contains the source and target index pairs for all difference vectors whose onset dimension is at most δ. Therefore, after the last iteration A contains the indices of the last points of patterns for which $VEC(Q_i) = VEC(P)$, that is, $Q_i \equiv_\mathrm{T} P$.

For two translationally equivalent patterns P and Q, the translator $t : P = Q + t$ is simply the difference between any two points at the same index in both patterns. Therefore the translators can be obtained by computing the difference between the last point of the pattern P and the points at the indices stored in A. Therefore, FINDTRANSLATORS correctly computes all translators of P in D_s. □

Next we will analyze the time and space complexity of the SIATEC-C algorithm.

Theorem 1.2. *Let D be a 2-dimensional point-set with n points. Let m be the largest number of points in any span of length δ in the onset dimension and let h be the number of points in the largest MTP in D. Then the worst case time complexity of* SIATEC-C *is $O(hn^2 \log nm)$.*

Proof. Computing the difference index I requires computing $O(nm)$ difference vectors, sorting them, and partitioning. Thus COMPUTEDIFFINDEX runs in $O(nm \log nm)$ time. Initializing the arrays T and W can be performed in linear time.

The number of iterations the main loop on lines 8–14 of algorithm 1 executes is approximately $\frac{n}{\delta} = O(n)$. Computing the MTPs requires also computing $O(nm)$ difference vectors, sorting and partitioning them into MTPs, and then sorting the MTPs by size, thus running in $O(nm \log nm)$ time. However, the total amount of computation required to compute MTPs in the loop performs the same number of difference vector computations and comparisons as computing all MTPs and sorting them by size, thus the total amount of work needed for MTP computation during the execution of the algorithm is $O(n^2 \log n)$ just as in SIA [3].

Cutting and filtering the MTPs by IMPROVESCOVER iterating through $O(n^2)$ points and performing constant time operations on each point.

For a pattern P, the size of its vectorized represenation is $|P| - 1$. The FINDTRANSLATORS function's main loop (algorithm 3 lines 5–8) is thus run on $O(n^2)$ difference vectors in total. For each difference vector, the loop finds the index pairs from I in $O(\log nm)$ time using binary search and computes the intersection of A with the source indices in A'. The number of index pairs that can be found for a difference vector v in I is equal to the size of the largest MTP in D, denoted by h. Thus computing the intersection of sorted arrays is linear in h, resulting in time complexity of $O(h \log nm)$ for a single difference vector in $\vec{P}$. Overall, finding all translators for all produced patterns has a worst case time complexity of $O(hn^2 \log nm)$.

The overall worst case time complexity of the algorithm is thus dominated by computing the translators, resulting in a worst case time complexity of $O(hn^2 \log nm)$.

□

The largest MTP in a point-set can contain $n - 1$ points in the case of the artificial D_{min} point-set. Thus a less detailed worst case time complexity of SIATEC-C could also be denoted as $O(n^3 \log n)$.

Theorem 1.3. *Let D be a 2-dimensional point-set with n points, and let m be the largest number of points in any span of length δ in the onset dimension. Then the worst case space complexity of* SIATEC-C *is $O(nm)$.*

Proof. Computing the difference index I requires storing $O(nm)$ difference vectors and corresponding index pairs, and after partitioning the number of difference vectors and index pairs does not increase. Therefore I takes $O(nm)$ space.

The arrays T and W used for keeping track of the sliding windows are both linear in the size of the input point-set. Therefore they do not dominate the space complexity of SIATEC-C.

On each iteration of the main loop on lines 8–14 of algorithm 1 the MTP computation requires keeping $O(nm)$ difference vectors in memory.

In the FINDTRANSLATORS function the number of index pairs contained in A or A' is at most the equal to the size of the largest MTP in D. Therefore the space complexity of FINDTRANSLATORS is $O(n)$.

The space complexity of SIATEC-C is dominated by I and the MTP computation, therefore the worst case space complexity is $O(nm)$. □

The structure I could be replaced with a *hash-map* structure allowing constant expected time retrieval of index pairs by FINDINDICES. This would lead to an expected time complexity of $O(hn^2)$ for SIATEC-C. In the experiments conducted for this paper the use of hashing was not found to produce significant practical running time improvements over using binary search. Alternatively the algorithm by [4] could be used for computing the MTPs instead of using sliding windows. However, this would not change the time complexity of the algorithm as it is dominated by finding the translators of the patterns.

2. REFERENCES

[1] T. Collins, *Improved methods for pattern discovery in music, with applications in automated stylistic composition*. PhD thesis, The Open University, 2011.

[2] G. A. Wiggins, "Models of musical similarity," in *Musicae Scientiae, Discussion Forum 4A*, 2007, pp. 315–338.

[3] D. Meredith, K. Lemström, and G. A. Wiggins, "Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music," *Journal of New Music Research*, vol. 31, no. 4, pp. 321–345, 2002.

[4] A. Laaksonen and K. Lemström, "On the memory usage of the SIA algorithm family for symbolic music pattern discovery," in *Eighth International Conference on Mathematics and Computation in Music*, 2022, in press.