Chapter 4

Mining Frequent Approximate Sequential Patterns

Feida Zhu, Xifeng Yan, Jiawei Han, and Philip S. Yu

Contents

4.1 Introduction

Frequent sequential pattern mining remains one of the most important data-mining tasks since its introduction in Ref. [1]. With the ubiquity of sequential data, it has found broad applications in customer analysis, query log analysis, financial stream data analysis, and pattern discovery in genomic DNA sequences in bioinformatics. Extensive research on the topic has brought about general sequential pattern mining algorithms like Refs. [2–7] and constraint-based ones like Refs. [8,9]. Periodic pattern mining in temporal data sequences has also been studied [10,11].

However, all these mining algorithms follow the exact matching sequential pattern definition. It has been shown that the capacity to accommodate approximation in the mining process has become critical due to inherent noise and imprecision in data, e.g., gene mutations in genomic DNA sequence mining. The notion of approximate sequential pattern has been proposed in Ref. [12], in which an algorithm called ApproxMap is designed to mine consensus patterns. While mining consensus patterns provide one way to produce compact mining result under general distance measures, it remains a challenge how to efficiently mine the complete set of approximate sequential patterns under some distance measure that is stricter yet equally useful in many cases. The Hamming distance model, which counts only mismatches, is one of such.

We look at bioinformatics: for example (1) The identification of repeats serves as a critical step in many biological applications on a higher level such as a preprocessing step for genome alignment, whole genome assembly, and a postprocessing step for BLAST queries. For repeat families that are relatively new in the evolution, the set of repeats found under the Hamming distance model captures almost the complete set. (2) The limited knowledge that biologists currently have of these repeats makes it often hard for them to evaluate the relative significance among different repeats. It is therefore worth the effort to mine the complete set. Existing tools like RepeatMasker [13] only solve the problem of pattern matching, rather than pattern discovery without prior knowledge. (3) Many research works for the repeating patterns have been on an important subtype: the tandem repeats [14], where repeating copies occur together in the sequence. However, as shown by our experiments, these methods would miss those patterns whose supporting occurrences appear globally in the entire data sequence, which account for the majority of the complete set of frequent patterns.

REPuter [15] is the closest effort toward mining frequent approximate sequential patterns under the Hamming distance model. Unfortunately, REPuter achieves its efficiency by strictly relying on the suffix tree for constant-time longest common prefix computation in seed extension. Consequently, the type of approximate patterns that REPuter is able to mine is inevitably limited. In particular, it can only discover patterns with two occurrences and mismatches at identical positions across the support set. The mining problem targeted by REPuter is essentially a different one.

To uncover more interesting approximate patterns in DNA sequences, we establish a more general model for approximate sequential pattern mining problem. Our general philosophy is a "break-down-and-build-up" one based on the following observation. Although for an approximate pattern, the sequences in its support set may have different patterns of substitutions, they can in fact be classified into groups, which we call *strands*. Each strand is a set of sequences sharing a unified pattern representation together with its support. The idea is that by "breaking down" the support sets of the approximate patterns into strands, we are able to design efficient algorithms to compute them. Using a suffix-tree-based algorithm, we can in linear time mine out the initial strands, which are all the exact-matching repeats. These initial strands will then be iteratively assembled into longer strands in a local search fashion, until no longer strands can be found. In the second "build-up" stage, different strands are then grouped based on their constituting sequences to form a support set so that the frequent approximate patterns would be identified. By avoiding incremental growth and global search, we are able to achieve great efficiency without losing the completeness of the mining result. Instead of mining only the patterns repeating within a sliding window of some fixed size, our algorithm is able to mine all globally repeating approximate patterns.

4.2 Problem Formulation

In our problem setting, we focus on mining approximate sequential patterns under the Hamming distance model. Hamming distance, which is defined for two strings of equal length, is the number of substitutions required to change one into the other.

DEFINITION 4.1 Hamming Distance
For two strings $S = \langle s_1, s_2, \ldots, s_n \rangle$ and $P = \langle p_1, p_2, \ldots, p_n \rangle$ of a same length n, the Hamming distance between them is defined as

$$Dist(S,P) = |I|, \quad I = \{i | s_i \neq p_i, 1 \leq i \leq n\}$$

The Hamming distance between two strings S and P is denoted as $Dist(S,P)$. In our model, two sequential patterns are considered approximately the same if and only if they are of equal length and their distance is within a user-specified error tolerance. We therefore use string or substring to refer to all sequential patterns in the rest of the chapter. Given a string $S = \langle s_1, s_2, \ldots, s_n \rangle$ of length n, another string $Z = \langle z_1, \ldots, z_m \rangle$ is a substring of S if there exists an index i of S such that $z_j = s_{i+j}$ for all $1 \leq j \leq m$. In this case, S is a superstring of Z. We use $|S|$ to denote the length of a string S.

Given an input string S, we are interested in finding all frequent approximate substrings of S, i.e., for each such substring, the set of substrings that are considered approximately the same must be sufficiently large.

DEFINITION 4.2 Frequent Approximate Substring
Given a string S, a substring P of S is a frequent approximate substring if and only if there exists a set U of substrings of S and for each $W \in U$, $Dist(P,W) \leq |P|\delta$, and $|U| \geq \theta$, where θ is the minimum frequency threshold and δ is the error tolerance threshold. U is called the support set of P, denoted as P_{sup}.

Notice that U is represented as a set of indices of S as all substrings in U share the same length with P.

As in frequent itemset mining, the definition of frequent approximate substring also gives rise to redundancy in the mining result. Consider the three substrings in Figure 4.1.

Suppose S_1 is a frequent approximate substring, with its distances to S_2 and S_3 being both 2 in this case. If we delete the last character A from all three strings, the resulting substring S_1 is still a frequent approximate substring with its distances to S_2

$S_1 =$ $ATCCG$ T $ACTATG$ T $TCAGTTGCAGCCA$

$S_2 =$ $ATCCG$ G $ACTATG$ A $TCAGTTGCAGCCA$

$S_3 =$ $ATCCG$ A $ACTATG$ G $TCAGTTGCAGCCA$

FIGURE 4.1: Three substrings from input.

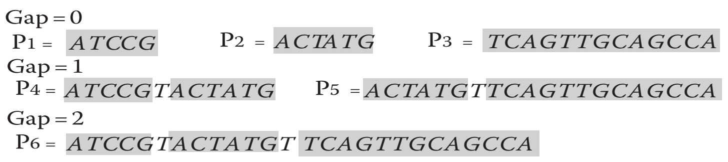

FIGURE 4.2: Ideal mining result.

and S_3 unchanged. This remains true as we delete more characters so long as the error tolerance requirement is satisfied. It would be considered redundancy in many cases if all such shorter substrings of S_1 are also reported. Ideally, we would like to report a substring only when it can not be extended without changing its distance to some substring in its support set. In the example of Figure 4.1, we would like to report six frequent approximate substrings as shown in Figure 4.2. We therefore define the *closeness* for a frequent approximate substring.

DEFINITION 4.3 Closed Frequent Approximate Substring
Given a string S, a frequent approximate substring P of S is closed if and only if there exists no frequent approximate substring Z of S such that (1) Z is a superstring of P, (2) there exists a bijection between Z_{sup} and P_{sup} such that for each $S_i \in P_{sup}$, there exists a $S'_i \in Z_{sup}$ such that S'_i is a superstring of S_i, and (3) $Dist(Z,S'_i) = Dist(P,S_i)$ for some $S_i \in P_{sup}$.

In this chapter, we study the problem of mining all closed frequent approximate substrings from a given data string. For brevity, all frequent approximate substrings mined by our algorithm are closed for the rest of the chapter. A frequent approximate substring will be abbreviated as a FAS. Formally, the frequent approximate substring mining problem (FASM) is defined as follows.

DEFINITION 4.4 FASM
Given a string S, a minimum frequency threshold θ and an error tolerance threshold δ, the FASM problem is to find all closed frequent approximate substrings P of S.

4.3 Algorithm Design

In general, for a FAS P, consider any two substrings S_1 and S_2 in P_{sup}. Aligning S_1 and S_2, we observe an alternating sequence of maximal-matching substrings and gaps of mismatches: $Pattern(S_1,S_2) = \langle M_1,g_1,M_2,g_2,\ldots,M_k\rangle$, where $M_i, 1 \leq i \leq k$ denote the maximal-matching substrings shared by S_1 and S_2. $g_i, 1 \leq i < k$ denote the

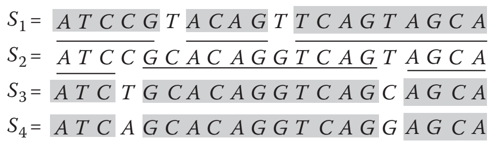
S_1= A T C C G T A C A G T T C A G T A G C A
S_2= A T C C G C A C A G G T C A G T A G C A
S_3= A T C T G C A C A G G T C A G C A G C A
S_4= A T C A G C A C A G G T C A G G A G C A

FIGURE 4.3: Alignment of S_1, S_3, and S_4 against S_2.

number of mismatches in the ith gap. Consider four substrings S_1,S_2,S_3, and S_4 as shown in Figure 4.3.

In this case, $Pattern(S_1,S_2) = \langle ATCCG,1,ACAG,1,TCAGTTGCA\rangle$. All four substrings are of length 20. If the error tolerance threshold $\delta = 0.1$ and minimum frequency threshold $\theta = 4$, then S_2 is a FAS since the other three substrings are within Hamming distance 2 from S_2. For each substring, the bounding boxes indicate the parts that match exactly with S_2. We can therefore define the notion of a strand, which is a set of substrings that share one same matching pattern.

DEFINITION 4.5

A set U of substrings $U = \{S_1,\ldots,S_k\}$ is a strand *if and only if for any two pairs of substrings $\{S_{i_1},S_{j_1}\}$ and $\{S_{i_2},S_{j_2}\}$ of U, $Pattern(S_{i_1},S_{j_1}) = Pattern(S_{i_2},S_{j_2})$.*

By definition, all the substrings in a strand U share the same alternating sequence of maximal-matching substrings and gaps of mismatches, which we call $Pat(U)$. We use $|Pat(U)|$ to denote the length of the substrings in U. We use $Gap(U)$ to denote the number of gaps in $Pat(U)$ and $Miss(U)$ to denote the number of total mismatches in $Pat(U)$. Given a strand U with its corresponding matching pattern $Pat(U) = \langle M_1,g_1,\ldots,M_k\rangle$, $Dist(S_i,S_j) = Miss(U) = \sum_{i=1}^{k-1} g_i$, for all $S_i,S_j \in U$ and $i \neq j$. All substrings in a strand share a same distance from one another. Define $Plist(U)$ to be the support set of a strand U, i.e., $Plist(U)$ is the set of indices where each substring in U occurs. A strand U is represented by the pair $\langle Pat(U),Plist(U)\rangle$.

We call a strand U valid if the distance between any two substrings of U satisfy the user-specified error tolerance threshold, i.e., $Miss(U) \leq |Pat(U)|\delta$. Similar to the notion of the closeness of a FAS, we have the definition for the closeness of a strand. A strand U is closed if and only if there exists no strand U' such that (1) there exists a bijection between the set of substrings of U and U' such that for each $P \in U$, there is a $P' \in U'$ and P' is a superstring of P, and (2) $Miss(U) = Miss(U')$.

A FAS could belong to multiple strands. For any given FAS, the observation is that its support set is exactly the union of all its closed valid strands.

LEMMA 4.1

For a FAS P, its support set $P_{sup} = \cup_{U \in X} U$, where X is the set of all P's closed valid strands.

PROOF We first prove $P_{sup} \subseteq \cup_{U \in X} U$. For any substring $W \in P_{sup}$, by the definition of P_{sup}, we have $Dist(P,W) \leq |P|\delta$. Let U be the strand uniquely defined by P and W. Then $Miss(U) = Dist(P,W) \leq |P|\delta$, which means U is valid. Since P is closed, extending P would change $Dist(P,W)$, and accordingly $Miss(U)$. As such U is also closed. Hence $W \in \cup_{U \in X} U$. We then show $\cup_{U \in X} U \subseteq P_{sup}$. For any substring $W \in \cup_{U \in X} U$, let U' be a valid and closed strand of P which contains W. Then $Dist(P,W) = Miss(U') \leq |P|\delta$ by the definition of a valid strand. Since P and W belong to a same strand, they are of equal length. The fact that U' is closed means that extending P would change $Dist(P,W)$. Hence $W \in P_{sup}$. ☐

We therefore have the following approach to decide if a given substring P is a FAS: Find all the closed valid strands of P and let the union of them be X. P is a FAS if and only if the cardinality of X is at least θ. Consider the example in Figure 4.3 in which the error tolerance is 0.1 and minimum frequency threshold is 4. Both strands $\{S_1,S_2\}$ and $\{S_2,S_3,S_4\}$ are valid. Suppose these two strands are also closed, then combining them we get a support set of size 4, satisfying the frequency requirement. As such, S_2 is a FAS.

Our algorithm solves the FASM in two steps.

1. *Growing strand*: Compute a set of closed valid strands initially. The set of initial strands is the set of all maximal exact repeats. More precisely, for each initial strand U, $Pat(U) = \langle M_1 \rangle$, $Miss(U) = 0$, and U is closed. These initial strands are computed by *InitStrand* using the suffix tree of the input sequence S. Similar approach has been used in REPuter [15] to mine exact repeats. By a linear-time suffix tree implementation as in Ref. [16], we are able to identify all initial strands in time linear to the input size. To mine out all closed valid strands, we iteratively call the following procedure to grow the current set of strands until no new strands are found: We scan the entire tape and, for each strand encountered, checks on both ends to see if the current strand can be grown by assembling neighboring strands. Let the result set be X.

2. *Grouping strand*: Once we have mined out all the closed valid strands in the first step, we compute the support set for each frequent approximate substring. The idea of grouping the strands is the following. Given the set X of all closed valid strands, we construct a substring relation graph G from X. The vertex set is all the substrings in the strands of X, each vertex representing a distinct substring. There is an edge between two substrings if and only if the Hamming distance between two substrings is within the error tolerance. Since all the substrings in one valid strand share the same distance among each other and the distance is within the error tolerance, all corresponding vertices in G form a clique. After scanning all the strands in X, we would construct a graph G which is a union of cliques. Then by our observation, a substring is a frequent approximate substring if and only if the degree of the corresponding vertex is greater than or equal to the minimum frequency threshold. Figure 4.4 illustrates the idea.

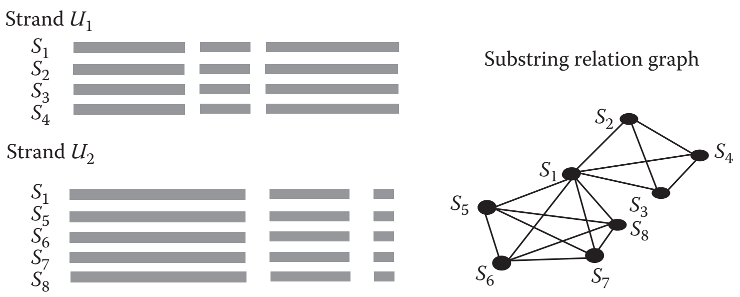

FIGURE 4.4: Two strands U_1 and U_2 and their substring relation graph.

4.3.1 Growing Strand

The main algorithm for growing strands is shown in Algorithm 4.1. The procedure *GrowthOnce*() is shown in Algorithm 4.2.

ALGORITHM 4.1 StrandGrowth

```
Input:  The original sequence $S$
        Error tolerance threshold $\delta$
Output: $Tape[1 \dots |S|]$

1: $Tape \leftarrow InitStrand(S)$;
2: while new valid strands are found
3:     $Tape \leftarrow GrowthOnce(S, Tape, \delta)$;
4: return $Tape$;
```

ALGORITHM 4.2 GrowthOnce

```
Input:  The original sequence $S$
        $Tape[1 \dots |S|]$
        Error tolerance threshold $\delta$
Output: $Tape$

1: for $i = 1$ to $|S|$
2:     for each substring $U_i$
3:         Check for a distance $d$ to the right of $U_i$
4:         for each $U'_j$ found at distance $d'$
5:             $Pat(U'') \leftarrow \langle Pat(U), d', Pat(U') \rangle$
6:             if $U''$ is valid
7:                 $Plist(U'') \leftarrow Plist(U) \otimes_{d'} Plist(U')$;
```

```
8:                  Insert $Plist(U'')$ into $Tape$;
9:          Check for a distance $d$ to the left of $U_i$
10:       **for each** $U'_j$ found at distance $d'$
11:             $Pat(U'') \leftarrow \langle Pat(U'), d', Pat(U) \rangle$
12:             **if** $U''$ is valid
13:                  $Plist(U'') \leftarrow Plist(U') \otimes_{d'} Plist(U)$;
14:                  Insert $Plist(U'')$ into $Tape$;
15: **return** $Tape$;
```

Initial Strand Computation

The set of initial strands is the set of all maximal exact repeats. More precisely, for each initial strand U, $Pat(U) = \langle M_1 \rangle$, $Miss(U) = 0$, and U is closed. These initial strands are computed by *InitStrand* using the suffix tree of the input sequence S. Similar approach has been used in REPuter [15] to mine exact repeats. A suffix tree is a data structure that compactly encodes the internal structure of a string. As such, it can be used to solve some complicated string problems in linear time. In particular, it enables us to mine out all frequent maximal exact-matching substrings of S with a running time linear in the length of S. A suffix tree $\mathscr{T}$ for a string S of n characters is a rooted directed tree with exactly n leaves numbered 1 to n. Each internal node of $\mathscr{T}$, except for the root has at least two children. Each edge is labeled with a nonempty substring of S. No two edges going out of a node have the labels on the edge beginning with the same character. A suffix tree encodes all suffixes of S by the following: for any leaf node i, the concatenation of edge-labels on the path from the root to leaf i corresponds exactly to the suffix of S beginning from index i, which is $S[i,\ldots,n]$. For instance, if $S = \langle ATTCGATTAC \rangle$, the suffix tree for S is as shown in Figure 4.5.

The first linear-time algorithm for constructing a suffix tree was given by Weiner [17] in 1973. We therefore have the following theorem:

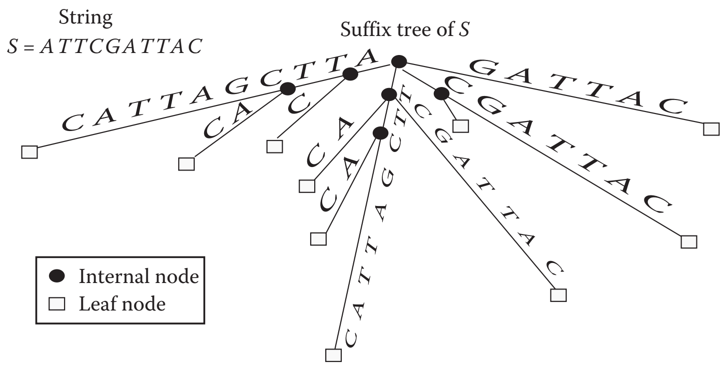

FIGURE 4.5: Suffix tree for a string S.

THEOREM 4.1 (Weiner 1973 [17])

Given a data sequence S, a suffix tree $\mathscr{T}$ can be constructed for S in $O(|S|)$ time. The size of $\mathscr{T}$ is $O(|S|)$.

InitStrand mines out all maximal frequent exact-matching substrings of length at least l_{min} and occurs at least θ times. The algorithm first constructs in linear time a suffix tree for S and then use the same idea as in Ref. [16] to identify the left diverse nodes in linear time to find all maximal frequent exact-matching substrings, as shown in Algorithm 4.3. For each node v of a suffix tree, define the *label of the path* for v as the concatenation of edge labels from the root to v, and is denoted as $\Xi(v)$. Denote as $Child_{num}(v)$ the number of leaves in the subtree rooted at v. Notice that $Child_{num}(v)$ for each node v can be computed along the tree traversal. Once we find a set of substrings matching with each other exactly at a node v, we can create a valid strand U for the set by setting the strand's pattern $Pat(U)$ as $\Xi(v)$ and $Plist(U)$ as the list of the numbers of all the leaves in v's subtree sorted in increasing order.

ALGORITHM 4.3 InitStrand

```
Input:  The original sequence S
          The minimum length threshold $l_{min}$
          The minimum frequency threshold $\theta$
Output: $Tape[1 \ldots |S|]$

1: Build the suffix tree $\mathscr{T}$ for S
2: Traverse $\mathscr{T}$
3:     for each internal node v
4:         if $|\Xi(v)| \geq l_{min}$ and $Child_{num}(v) \geq \theta$
5:             Generate a strand U for $\Xi(v)$
6:             Insert U into Tape
7: return Tape;
```

Notice that the suffix tree $\mathscr{T}$ can be built in linear time. Since Line 2 to Line 7 is a tree traversal and thus takes time proportional to the size of the tree, we conclude that *InitStrand* runs in time linear in the length of S. We have the following theorem from Ref. [16]. We conclude from Theorem 4.2 that *InitStrand* runs in time $O(|S|)$ for a given input string S.

THEOREM 4.2

All maximal frequent repeats in S can be found in $O(|S|)$ time.

4.3.2 Grouping Strand

Once we have mined out all the closed valid strands in the first step, we use *StrandGroup* to compute the support set for each frequent approximate substring. The idea of grouping the strands is the following. Given the set X of all closed valid strands, we construct a substring relation graph G from X. The vertex set is all the substrings in the strands of X, each vertex representing a distinct substring. There is an edge between two substrings if and only if the Hamming distance between two substrings is within the error tolerance. Since all the substrings in one valid strand share the same distance among each other and the distance is within the error tolerance, all corresponding vertices in G form a clique. After scanning all the strands in X, we would construct a graph G which is a union of cliques. Then by Lemma 4.1, a substring is a frequent approximate substring if and only if the degree of the corresponding vertex is greater than or equal to the minimum frequency threshold, as illustrated in Figure 4.4. Algorithm 4.4 shows the following algorithm.

ALGORITHM 4.4 StrandGroup

```
Input:  $Tape[1 \dots |S|]$
        The minimum frequency threshold $\theta$
Output: all frequent approximate substrings

1: for each strand $U$ found while scanning $Tape$
2:     for each substring $P \in Plist(U)$
3:         Add $|Plist(U)| - 1$ neighbors to $P$
4: for each substring $P$
5:     if $P$ has at least $\theta$ neighbors
6:         Output $P$
```

4.3.3 Completeness of Mining Result

We now prove that our algorithm would generate the complete set of FASs. The sketch of the proof is as follows. We first prove that *StrandGrowth* would generate all closed valid strands by induction on the number of gaps of the strands. Lemma 4.1 then tells us that we could compute the support sets of all FASs and identify them by grouping the strands by Algorithm 4.4.

LEMMA 4.2
InitStrand would generate all closed valid strands U such that $Gap(U) = 0$.

PROOF By Theorem 4.2, we know that *InitStrand* would generate all frequent maximal exact-matching substrings. By the definition of a strand, the strands formed by these exact-matching substrings would have their number of gaps to be zero since no mismatches exist in these substrings. Hence the validity of the strands. The fact that these frequent substrings are maximal means that the strands thus constructed are closed. ∎

StrandGrowth discovers a new strand by always attempting to assemble two current closed valid strands. Before we prove the theorem, we need one more lemma to show that any closed valid strand can be generated from two shorter closed valid strands.

LEMMA 4.3

Given a closed valid strand U with $Pat(U) = \langle M_1, g_1, \ldots, M_m \rangle$. U can always be divided into two shorter closed valid strands U_1 and U_2 such that $Pat(U_1) = \langle M_1, g_1, \ldots, M_i \rangle$ and $Pat(U_2) = \langle M_{i+1}, g_{i+1}, \ldots, M_m \rangle$ for some $1 \le i \le m-1$.

PROOF Given a closed valid strand U with $Pat(U) = \langle M_1, g_1, \ldots, M_m \rangle$, let U_1 and U_2 be such that $Pat(U_1) = \langle M_1, g_1, \ldots, M_{m-1} \rangle$ and $Pat(U_2) = \langle M_m \rangle$. By definition, U_2 is a closed valid strand. If it is also true for U_1, we are done. Otherwise, the only possibility would be that U_1 is not valid. Since the entire strand U is valid, it follows that, if we move the last exact-matching substring M_{m-1} from $Pat(U_1)$ to $Pat(U_2)$ and obtain two new strands such that $Pat(U_1') = \langle M_1, g_1, \ldots, M_{m-2} \rangle$ and $Pat(U_2') = \langle M_{m-1}, g_{m-1}, M_m \rangle$, we must conclude that U_2' is again a closed valid strand. We then check if U_1' is valid. If not, we again move the last exact-matching substring M_{m-2} from $Pat(U_1')$ to $Pat(U_2')$, and so on so forth. Since at the end, if we have $Pat(U_1') = \langle M_1 \rangle$ and $Pat(U_2') = \langle M_2, g_2, \ldots, M_m \rangle$, both strands must be valid, we conclude that there exists some $i, 1 \le i \le m-1$, such that U can be divided into U_1 and U_2 where $Pat(U_1) = \langle M_1, g_1, \ldots, M_i \rangle$ and $Pat(U_2) = \langle M_{i+1}, g_{i+1}, \ldots, M_m \rangle$. ∎

Now we are ready to show that we would be able to generate all the closed valid strands.

THEOREM 4.3

StrandGrowth would generate all closed valid strands.

PROOF We prove by induction on the number of gaps $Gap(U)$ for any closed valid strand U. When $Gap(U) = 0$, the claim is true by Lemma 4.2. Assume that the claim is true for $Gap(U) = k \ge 0$. When $Gap(U) = k+1$, by Lemma 4.3, U can be divided into two shorter closed valid strands U_1 and U_2 such that $Pat(U_1) = \langle M_1, g_1, \ldots, M_i \rangle$ and $Pat(U_2) = \langle M_{i+1}, g_{i+1}, \ldots, M_m \rangle$ for some $1 \le i \le m-1$. By induction, both U_1 and U_2 will be generated by *StrandGrowth*. As such, when d is large enough in *StrandGrowth*, U_1 and U_2 will be assembled into U. The claim is thus also true for $Gap(U) = k+1$. We therefore conclude that *StrandGrowth* would generate all valid canonical strands. ∎

Since it is easy to verify the correctness of Algorithm 4.4 and we have proved Lemma 4.1, we would generate the support sets of all frequent approximate substrings and thus identify them. We therefore claim the following theorem for the completeness of our mining result.

THEOREM 4.4

Given an input data string S, StrandGrowth and StrandGroup would mine the complete set of frequent approximate substrings from S.

4.3.4 Local Search

One salient feature of *StrandGrowth* is that only local search is performed when checking on both ends of a strand for strand growth. We therefore need to determine the value of d in *GrowthOnce*. If d is set to be too big, then in the worst case, we would scan the entire data string each time we check for a strand. The running time of *GrowthOnce* would then be $\Omega(|X|^2)$, where X is the set of all valid canonical strands. On the other hand, if d is set to be too small, we could fail to guarantee the completeness of the mining result. Consider the following example in Figure 4.6. Suppose we have two valid strands U_1 and U_2 such that $|Pat(U_1)| = 20, Miss(U_1) = 0$ and $|Pat(U_2)| = 40, Miss(U_2) = 0$. There is a gap of 7 mismatches between them. Suppose the error tolerance is $\delta = 0.1$. Notice that a valid strand U can accommodate further mismatches on either ends up to a distance of $|Pat(U)|\delta - Miss(U)$. Then U_1 can accommodate $d_1 = 2$ extra mismatches and U_2 can accommodate $d_2 = 4$ extra mismatches.

However, as Figure 4.6 shows, the tricky part is that if we only search forward d_1 from U_1 and backward d_2 from U_2, we would fail to identify the chance to assemble them due to the fact that the gap is larger than the sum of d_1 and d_2. Even if we search forward from U_1 for a distance that doubles d_1, we could still miss U_2. Fortunately, searching backward from U_2 for a distance of $2d_2$ would let us reach U_1. Then how to decide on the value of d such that we would guarantee the completeness of the mining result, and at the same time, scan as small a portion of the data string as possible? It turns out we have the following theorem to help determine the value for d.

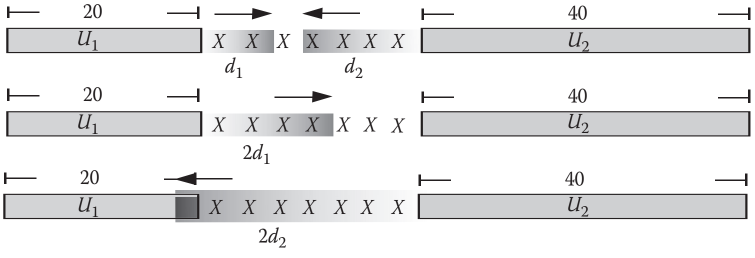

FIGURE 4.6: Assembling two strands U_1, U_2.

THEOREM 4.5

Given the error tolerance δ *and a strand* U*, searching for a distance* $d = 2(|Pat(U)|\delta - Miss(U))/(1-\delta)$ *would guarantee the completeness of the mining result.*

PROOF Suppose a closed valid strand U can be assembled by two shorter strands U_1 and U_2. Assuming U_1 occurs before U_2 in the data sequence. We only need to show that one of the following must happen: (1) searching forward a distance d from the end of U_1 would reach U_2 (2) searching backward a distance d from the beginning of U_2 would U_1. Suppose when assembling U_1 and U_2 into U, the gap between them is g. Since U is valid, we have $Miss(U) \leq |Pat(U)|\delta$, i.e., $Miss(U_1) + Miss(U_2) + g \leq (|Pat(U_1)| + |Pat(U_2)| + g)\delta$. Therefore, $g \leq (|Pat(U_1)|\delta - Miss(U_1))/(1-\delta) + (|Pat(U_2)|\delta - Miss(U_2))/(1-\delta)$. Without loss of generality, assume $|Pat(U_1)| \geq |Pat(U_2)|$. As such, $g \leq 2(|Pat(U_1)|\delta - Miss(U_1))/(1-\delta)$. This means U_2 would be encountered when searching forward a distance of $d = 2(|Pat(U_1)|\delta - Miss(U_1))/(1-\delta)$ from U_1. $\square$

Theorem 4.5 tells us that we do not have to search too far for us to guarantee the completeness. In fact, it is easy to observe that we at most search twice the distance of an optimal algorithm. Notice that any strand encountered within a distance of $\hat{d} = (|Pat(U)|\delta - Miss(U))/(1-\delta)$ can be assembled with the current strand to form a new valid strand, since the current strand itself can accommodate all the mismatches in a gap of length $\hat{d}$. As such to guarantee a complete mining result, any algorithm would have to check at least a distance of $\hat{d}$. We therefore check at most twice the distance of an optimal algorithm.

4.4 Performance Study

We used a real soybean genomic DNA sequence, CloughBAC, for our experiment. CloughBAC is 103,334 bp in length. There are altogether 182,046 closed approximate sequences of length at least 5. The longest closed approximate sequence is of length 995. The error tolerance δ is set as 0.1. The minimum frequency threshold θ is set as 3.

Figure 4.7 shows those of size up to 40 while Figure 4.8 shows the rest of the mining result, which are of size from 40 to 995. It can be observed that, in this particular soybean genomic DNA sequence, the approximate sequences are dense around the size of 10 and become sparse from size 15 to form a long tail.

We define the *spread* for an approximate sequence to be the distance between the index of its first occurrence and that of its last occurrence. A globally repeating approximate sequence has a large spread since its occurrences are not confined to a particular portion of the data sequence. As such, the larger the spread, the harder it is to discover the sequence by a sliding-window-based method. The spreads of all the

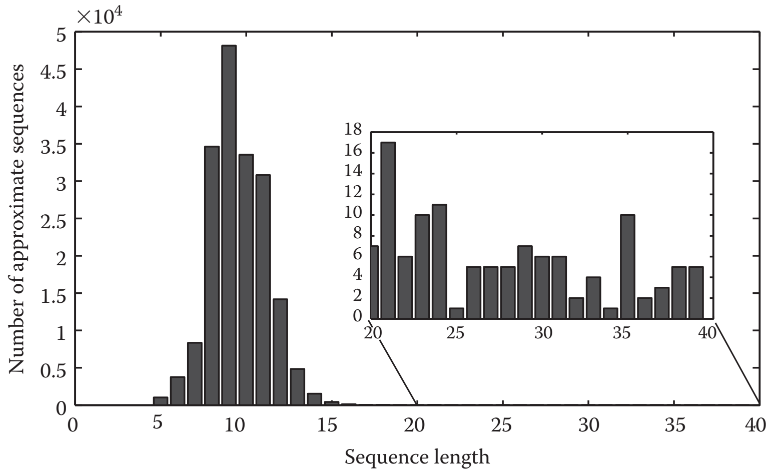

FIGURE 4.7: Sequences of size up to 40 bps.

approximate sequences in the mining result are plotted in Figure 4.9. It is evident that the majority of them actually have spreads comparable to the length of the orginal data sequence. Indeed as shown in Figure 4.10, the advantage of our mining approach compared against a sliding-window-based approach manifests itself in the fact that even a sliding window half the size of the original data sequence would discover all the occurrences of only 30% of the complete mining result. Furthermore, Figure 4.11 shows the average gaps between two successive occurrences of the approximate sequences in the complete mining result. Most of the sequences have an average gap

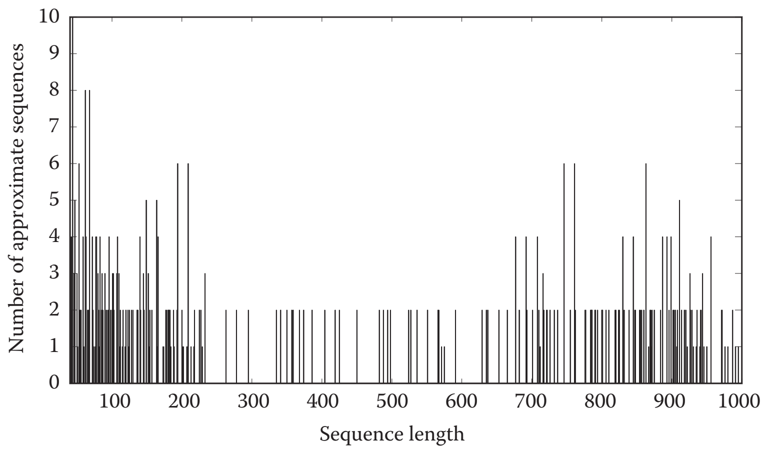

FIGURE 4.8: Sequences of size from 40 to 1000 bps.

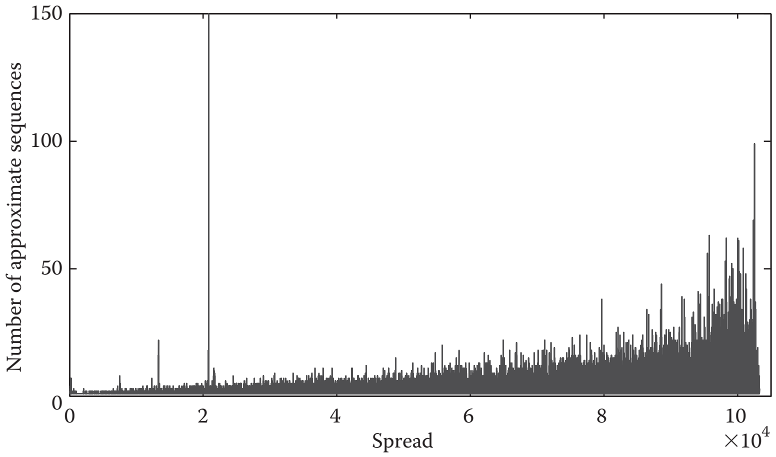

FIGURE 4.9: Spread of the mining result.

of size 1/10 of the original data sequence, which makes it hard for any locally aware approaches with a fixed intelligence radius to identify any repeating occurrences.

Figure 4.12 shows the running time of our algorithm as the number of output approximate sequence increases. It is compared against the one without the local search technique to demonstrate its importance in boosting the mining efficiency. The running time of our algorithm is observed to be linear in the output size. Figure 4.13 illustrates the run time performance with varied error tolerance δ. The bar chart, with its y-axis on the right-hand side of the figure, shows the corresponding numbers of output sequences as δ increases. More lenient error tolerance results in more output sequences and consequently a longer running time. Figure 4.14 illustrates the run

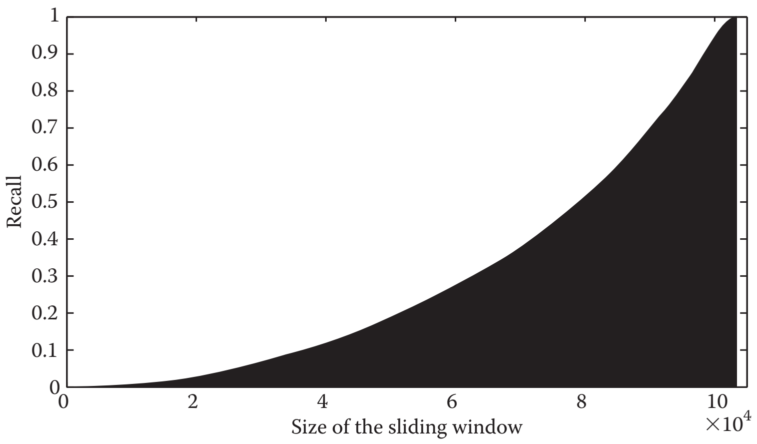

FIGURE 4.10: Recall of sliding window approach.

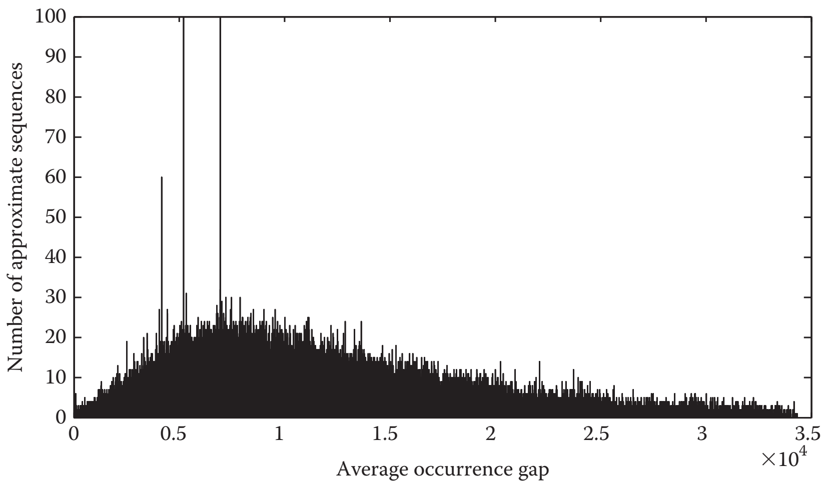

FIGURE 4.11: Average occurrence gap.

time performance with varied minimum frequency threshold θ. The bar chart, with its y-axis on the right-hand side of the figure, shows the corresponding numbers of output sequences as θ increases. Observe that as the minimum frequency threshold increases, the output size decreases sharply while the running time almost remains the same. This is because regardless of the minimum frequency threshold for the output, all sequences with at least two occurrences have to be computed during the strand growing stage, which is responsible for most of the mining cost. It is only in the strand grouping stage that a greater minimum frequency threshold helps to reduce the running time. The influence of θ on the mining cost is therefore less significant.

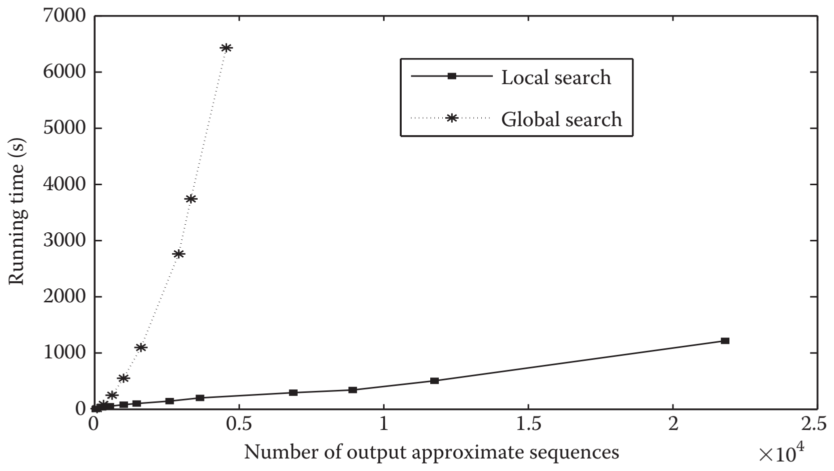

FIGURE 4.12: Run time.

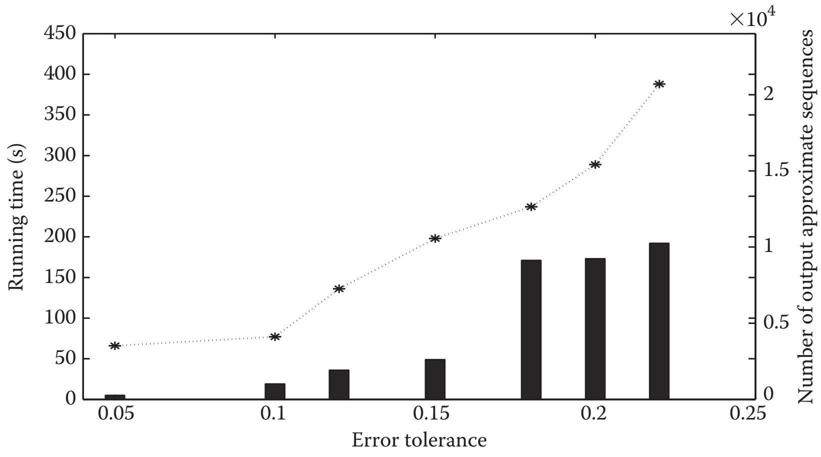

FIGURE 4.13: Run time with varied δ.

4.5 Related Work

Agrawal and Srikant first introduced the sequential pattern mining problem in Ref. [1], and later, based on the a priori property [18], continued to develop a generalized and improved algorithm [16]. A succession of sequential pattern mining algorithms have been proposed since then for performance improvements, including SPADE [5], PrefixSpan [6], and SPAM [7]. These algorithms either use a vertical id-list format (e.g., SPADE), a vertical bitmap data representation (e.g., SPAM), or a

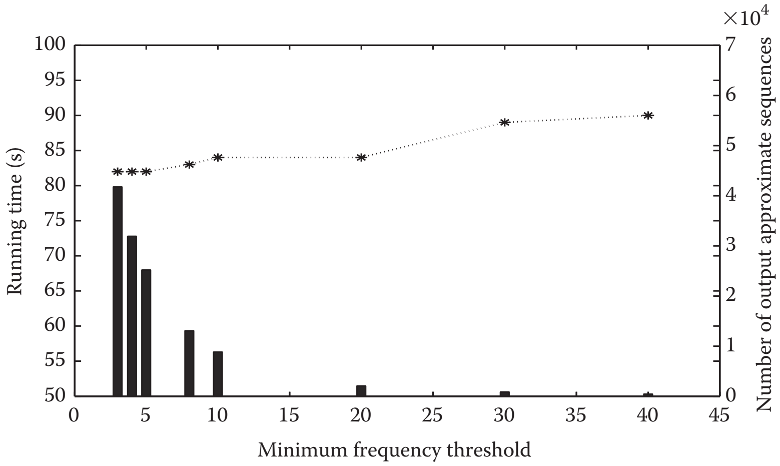

FIGURE 4.14: Run time with varied θ.

horizontal data set format (e.g., PrefixSpan) to enhance mining efficiency. There are also constraint-based ones like Refs. [8,9]. Periodic pattern mining in temporal data sequences have also been studied [10,11]. To make the mining result more friendly for user understanding, algorithms have been developed to mine frequent closed sequences. CloSpan [19] follows the candidate maintenance-and-test approach and uses techniques like CommonPrefix and Backward Sub-Pattern Pruning. BIDE [20] improves scalability by avoiding candidate maintenance and applying BI Directional Extension. When approximation is taken into consideration in the frequent sequential pattern definition, the size of the mining result could be prohibitively huge under a general distance measure. ApproxMap [12] approached this problem by mining instead the consensus patterns, which are a subset of long and representative patterns. Algorithms aimed at mining the complete answer set like Ref. [21], which have been studied in music information retrieval, suffer running time cost as high as $O(n^4)$ where n is the data sequence length. Algorithms in bioinformatics community have been focusing on approximate pattern matching and generate popular tools like RepeatMasker [13]. Most of these algorithms target at finding tandem repeats. REPuter [15] uses suffix tree to find maximal exact repeats and employs a suffix-tree-based constant time longest common prefix algorithm to extend them. However, REPuter cannot discover patterns with more than two occurrences and mismatches present at different positions across the support.

4.6 Future Work

The fast-growing data-intensive applications today present plenty of sequential pattern mining research problems for the new generation. Some future work directions include

1. *Multiple input sequences*: The current mining framework can be naturally extended to the multiple sequence scenario where the input is a set of long sequences. In many cases, the occurrences of a frequent substring in each input sequence are counted only once in the support computation. Suffix-tree with certain enhancement can handle repeats in multiple sequences. However, efficient discovery of frequent approximate substrings for multiple sequences requires further extension of the current algorithm.

2. *Other approximation definitions*: It is evident that there are many approximate sequential pattern definitions other than the Hamming distance model studied in this Chapter. In many bio-applications, the distance between two strings is defined in a much more complicated way than by gaps.

3. *Online sequential pattern mining*: The huge amount of data generated at a galloping speed in many real-life applications would eventually require efficient online mining algorithms for sequential patterns. It is interesting to study how to discover useful frequent patterns when only partial data can be stored and near-real-time response is desired.

4. *Anomaly mining*: In many trajectory data mining, e.g., commercial logistics applications, anomalous patterns (i.e., those outliers significantly deviate from the frequent patterns) are the mining targets. These patterns raise alert, drawing human attention for further examination.

4.7 Conclusions

Mining frequent approximate sequential pattern has been an important data-mining task. Its version in biological applications—finding repeats —has long been a topic of extensive research. Existing algorithms in bioinformatics communities solve pattern matching rather than mining. In particular, most algorithms are designed to find tandem repeats. In data-mining community, algorithms have been developed to mine a set representative patterns to avoid the combinatorial explosion due to the general distance definition. In this Chapter, we proposed the definition of *closed* frequent approximate sequential patterns. We aim to solve the problem of mining the complete set of frequent approximate sequential pattern mining under Hamming distance. Our algorithm is based on the notion of classifying a pattern's support set into *strands*, which makes possible both efficient computation and compact representation of it. By combining a suffix-tree-based initial strand mining and iterative strand growth, we adopt a local search optimization technique to reduce time complexity. We also proved that our local search strategy guarantees the completeness of the mining result. Our performance study shows that our algorithm is able to mine out globally repeating approximate patterns in biological genomic DNA data with great efficiency.

Acknowledgment

We are grateful to Professor Matt Hudson and Kranthi Varala for providing us with the genome sequence data and many helpful discussions and suggestions.

References

[1] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 1995 International Conference on Data Engineering (ICDE'95)*, pp. 3–14, Taipei, Taiwan, March 1995.

[2] F. Masseglia, F. Cathala, and P. Poncelet. The PSP approach for mining sequential patterns. In *Proceedings of the 1998 European Symposium on Principle*

of Data Mining and Knowledge Discovery (PKDD'98), pp. 176–184, Nantes, France, September 1998.

[3] R. Srikant and R. Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proceedings of the 5th International Conference on Extending Database Technology (EDBT'96),* pp. 3–17, Avignon, France, March 1996.

[4] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. FreeSpan: Frequent pattern-projected sequential pattern mining. In *Proceedings of the 2000 ACM SIGKDD International Conference on Knowledge Discovery in Databases (KDD'00),* pp. 355–359, Boston, MA, August 2000.

[5] M. Zaki. SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning,* 40: 31–60, 2001.

[6] J. Pei, J. Han, B. Mortazavi-Asl, H. Pinto, Q. Chen, U. Dayal, and M.-C. Hsu. PrefixSpan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 2001 International Conference on Data Engineering (ICDE'01),* pp. 215–224, Heidelberg, Germany, April 2001.

[7] J. Ayres, J. Flannick, J. E. Gehrke, and T. Yiu. Sequential pattern mining using bitmap representation. In *Proceedings of the 2002 ACM SIGKDD International Conference on Knowledge Discovery in Databases (KDD'02),* pp. 429–435, Edmonton, Canada, July 2002.

[8] M. Garofalakis, R. Rastogi, and K. Shim. SPIRIT: Sequential pattern mining with regular expression constraints. In *Proceedings of 1999 International Conference on Very Large Data Bases (VLDB'99),* pp. 223–234, Edinburgh, UK, September 1999.

[9] J. Pei, J. Han, and W. Wang. Constraint-based sequential pattern mining in large databases. In *Proceedings of the 2002 International Conference on Information and Knowledge Management (CIKM'02),* pp. 18–25, McLean, VA, November 2002.

[10] J. Han, G. Dong, and Y. Yin. Efficient mining of partial periodic patterns in time series database. In *Proceedings of 1999 International Conference on Data Engineering (ICDE'99),* pp. 106–115, Sydney, Australia, April 1999.

[11] C. Bettini, X. Sean Wang, and S. Jajodia. Mining temporal relationships with multiple granularities in time sequences. *Bulletin of the Technical Committee on Data Engineering,* 21:32–38, 1998.

[12] H.-C. Kum, J. Pei, W. Wang, and D. Duncan. ApproxMap: Approximate mining of consensus sequential patterns. In *Proceedings of the 2003 SIAM International Conference on Data Mining (SDM'03),* pp. 311–315, San Francisco, CA, May 2003.

[13] Institute for Systems Biology. Repeatmasker. In http://www.repeatmasker.org/webrepeatmaskerhelp.html, 2003.

[14] G. M. Landau and J. P. Schmidt. An algorithm for approximate tandem repeats. In *Proceedings of the 4th Annual Symposium on Combinatorial Pattern Matching*, number 684, pp. 120–133, Padova, Italy, 1993. Springer-Verlag, Berlin.

[15] S. Kurtz, J. V. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. Reputer: The manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 22: 4633–4642, 2001.

[16] D. Gusfield. *Algorithms on Strings, Trees and Sequences, Computer Science and Computation Biology*. Cambridge University Press, 1997.

[17] P. Weiner. Linear pattern matching algorithms. In *Proceedings of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pp. 1–11, 1973.

[18] R. Agrawal and R. Srikant. Fast algorithm for mining association rules in large databases. In Research Report RJ 9839, IBM Almaden Research Center, San Jose, CA, June 1994.

[19] X. Yan, J. Han, and R. Afshar. CloSpan: Mining closed sequential patterns in large datasets. In *Proceedings of the 2003 SIAM International Conference on Data Mining (SDM'03)*, pp. 166–177, San Fransisco, CA, May 2003.

[20] J. Wang and J. Han. BIDE: Efficient mining of frequent closed sequences. In *Proceedings of the 2004 International Conference on Data Engineering (ICDE'04)*, pp. 79–90, Boston, MA, March 2004.

[21] Jia-Lien Hsu, Arbee L. P. Chen, and Hung-Chen Chen. Finding approximate repeating patterns from sequence data. In *Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR'04)*, pp. 246–250, Barcelona, Spain, October 2004.