Mining Periodic Patterns with Gap Requirement from Sequences [*]

Minghua Zhang Ben Kao David W. Cheung Kevin Y. Yip

Department of Computer Science,
The University of Hong Kong, Hong Kong.
{*mhzhang, kao, dcheung, ylyip*}*@cs.hku.hk*

ABSTRACT

We study a problem of mining frequently occurring periodic patterns with a gap requirement from sequences. Given a character sequence S of length L and a pattern P of length l, we consider P a frequently occurring pattern in S if the probability of *observing* P given a randomly picked length-l subsequence of S exceeds a certain threshold. In many applications, particularly those related to bioinformatics, interesting patterns are *periodic* with a *gap requirement*. That is to say, the characters in P should match subsequences of S in such a way that the matching characters in S are separated by gaps of more or less the same size. We show the complexity of the mining problem and discuss why traditional mining algorithms are computationally infeasible. We propose practical algorithms for solving the problem, and study their characteristics. We also present a case study in which we apply our algorithms on some DNA sequences. We discuss some interesting patterns obtained from the case study.

1. INTRODUCTION

The completion of whole-genome sequencing of various organisms facilitates the detection of many kinds of interesting patterns in DNA and protein sequences. It is now well known that the genomes of most plants and animals contain large quantity of repetitive DNA fragments. For instance, it is estimated that one third of the human genome is composed of families of reiterated sequences [9]. The genomes are thus far from pieces of random strings, and it is widely believed that a substantial amount of currently unknown information can be extracted from the sequences.

A large number of studies on genome sequence mining are related to the identification of periodic patterns. This is largely due to the abundance and variety of periodic patterns existing in the genomes. From the short three base pair (bp) periodicity in protein coding DNA [5] and the medium-length repetitive motifs found in some proteins [4] to the mosiac of very long DNA segments in the genome of warm-blooded vertebrates [3], periodic patterns of different lengths and types are found at both genomic and proteomic levels. Some of the patterns have been identified as having significant biological and medical values. For example, some repeats have been shown to affect bacterial virulence to human [19], while the excessive expansions of some Variable Number of Tandem Repeats (VNTRs) are the suspected cause of some nervous system diseases [14]. Efficient algorithms for searching periodic patterns from long sequences are therefore of growing importance.

Computationally, a DNA or protein sequence is treated as a long string of characters with a finite alphabet. The alphabet used in modeling a DNA sequence is usually the four-character set $\{A, C, G, T\}$ representing the four nitrogenous bases Adenine, Cytosine, Guanine and Thymine. For protein sequences, the commonly used alphabet is the set of twenty amino acids.

Two types of periodic patterns have received much attentions: tandem repeats and base pair oscillations. Given a (DNA or protein) sequence $s = s_1 s_2 s_3 \cdots s_L$ of length L and an integer p (the period), a tandem repeat is a subsequence $s_i s_{i+1} s_{i+2} \cdots s_{i+2p-1}$ where $s_{i+j} = s_{i+p+j}$, for $0 \leq j < p$. The basic computational problem is to find all tandem repeats in a given sequence. There are many variations of the problem, considering issues like the number of periods (tandem repeats vs. tandem arrays), the maximality of patterns, whether errors (insertions, deletions and substitutions) are allowed and the corresponding cost functions, palindromic reverses, and efficient approximate solutions. A recent survey on the works can be found in [9]. We are particularly interested in tandem repeats that are related to the three-dimensional structure of the sequence. For example, the protein sequence of the molecule porcine ribonuclease inhibitor (SwissProt entry RINI_PIG [2]) consists of an alternating pattern of two kinds of repeats with lengths 29 and 28 residues [4]. The two can be combined to form a repeating unit of 57 residues, and there are 7.5 such units in the molecule. As a result, the protein has a horseshoe shape with the interior face formed by a parallel β sheet of 17 β strands and the exterior face formed by 16 α helices[1].

It should be noted that the repeats are not error-free. For instance, a phase shift is found in one of the repeats, which

[*]This research is supported by Hong Kong Research Grants Council grant HKU 7040/02E.

[1]A figure of the protein can be found in Figure 1 of [4] at http://www.nslij-genetics.org/dnacorr/.

may be due to the insertion or deletion of a short sequence.

The second kind of important periodic pattern is base pair oscillations, which correspond to unexpected correlations between bases of distance p. For example, the probability of having a 'T' located p bp after an 'A' can be calculated as $\frac{n_{AT}(p)}{L-p}$, where $n_{AT}(p)$ is the number of such occurrences in the sequence and $L-p$ is the number of base pairs located p bp apart. If base pairs of distance p are independent, then the expected probability will be $pr(A)pr(T)$, which is the product of the probabilities of occurrence of the two individual bases in the sequence. The difference $\frac{n_{AT}(p)}{L-p} - pr(A)pr(T)$ can be used to reflect the correlation between the two bases at a distance of p apart [7]. It has been shown in [20, 7] that some base pairs exhibit an abnormally high correlation at a period of 10-11 base pairs and its multiples in many kinds of organisms. It is believed that a partial reason for the phenomenon is related to the helical structure of the DNA, which has a period of about 10-11 base pairs in some organisms [7]. In other words, for some base pairs, if the first one is found in a certain position, there is an abnormally high probability of finding the second one after about one helical turn. Some interesting periodic patterns may thus be found in successive bases with similar 3D orientations.

Our study is based on the above observation. We would like to search for frequent periodic patterns that consist of bases physically located one helical turn after another. Symbolically, a pattern is defined as a subsequence

$$s_i s_{i+g^1} s_{i+g^1+g^2} \cdots s_{i+g^1+g^2+\cdots+g^{l-1}},$$

where l is the length (number of bases) in the subsequence and $g^j, 1 \leq j < l$ is the length of period j. Unlike previous studies, we define g^j as a range of integers instead of a fixed integer. The reason for this setting is two-fold: 1) the actual period of a helical turn is usually not an integer and 2) the actual period may vary in organisms. The introduction of a variable period thus provides a flexible way to capture any interesting patterns hidden in a sequence.

While the primary focus of our study is on the periodic patterns in DNA sequences due to its 3D structure, the techniques being developed can also be applied to mine other kinds of sequences, in which case the variable period can be used to model the maximum allowed insertions/deletions within a single period.

The rest of the paper is organized as follows. Section 2 mentions some related works. Section 3 formally defines our computational problem. In Section 4 we prove a couple of important theorems that lead to the derivation of efficient algorithms for our mining problem. Section 5 presents the algorithms. In Section 6 we analyze the algorithms' performance. Section 7 presents a case study in which we document an interesting finding obtained by applying our algorithm to mining DNA sequences. Finally, Section 8 concludes the paper.

2. RELATED WORKS

Besides the studies on tandem repeats and base-pair oscillation, there are other related works that include studies on mining patterns from biological sequences with certain support requirement. For example, the `TEIRESIAS` algorithm [15] is designed for discovering patterns that are composed of characters (such as $\{A, C, T, G\}$) and wild-cards (which match any characters) from biological sequences. Although wild-cards provide some flexibility in specifying a pattern, too many unrestricted wild-cards in a pattern would render the pattern uninteresting. Therefore, the authors restrict the number of wild-cards that can be present in the extracted patterns. In another study [8], the `Pratt` algorithm is proposed for mining restricted patterns from a sequence database. The restrictions include the maximum number of characters and wild-cards in a pattern.

BLAST [1] is one of the famous algorithms in the area of bioinformatics. Given a query sequence, it searches for matched sequences from a database. In essence, BLAST is a search algorithm with a query as input, while our model is focused on mining unknown knowledge.

From the area of data mining, one related problem is to find frequent sequences from transactional databases. Many efficient algorithms have been proposed for the problem [16, 22, 24, 12]. Their goal is to find patterns that appear in at least a certain number of sequences in the database. All the algorithms are based on the well-known Apriori property. Unfortunately, as we will see later, this property does not hold for our problem. Also, the sequence mining algorithms find patterns across sequences. On the other hand, our model is to discover patterns within a sequence. Moreover, the characteristics of the biological sequences (e.g., very long sequence with very few different characters) makes a direct application of those sequence mining algorithms inefficient.

There are also some algorithms on mining frequent patterns from a single sequence [10, 6]. In [10], the input sequence is divided into some overlapped windows of fixed width w, and every two neighboring windows share a common segment of length $(w-1)$. In [6], a sequence is divided into non-overlapping windows. In both papers, a pattern is frequent if it appears in at least a certain number of windows. With this definition, it is shown that the Apriori property applies. By segmenting a sequence into windows and counting the number of windows in which a pattern occurs greatly simplifies the design of the mining algorithm. The drawback is that patterns that span multiple windows cannot be discovered, and that in some cases, a suitable window width is difficult to determine. Our model does not have those restrictions.

Yang et al. studied asynchronous periodic patterns in time series data [21]. In their model, shifts in the occurrence of patterns are permitted to filter out random noises. They also consider a range of periods instead of the prespecified ones as used in [6]. However, although a range of period length is considered, they still has a restriction on the maximal length of the period. Also, the Apriori property is applicable on patterns of the same period.

3. PROBLEM DEFINITION

In this section we give a formal definition of the periodic pattern mining problem. To simplify our discussion, let us first define a number of notations and terms.

A sequence from which we extract frequent patterns is called a **subject sequence**. Let $\sum$ be the **alphabet** of all possible characters that occur in a subject sequence. For example, $\sum = \{A, C, G, T\}$ for DNA sequences; for protein sequences, $\sum$ is the set of 20 amino acids.

A **wild-card** (denoted by a single dot, '.') is a special symbol that matches any character in $\sum$. A **gap** is a sequence of wild-cards. The **size** of a gap refers to the number of wild-cards in it. For example, the size of '.....' is 5. We use $g(N)$ to represent a gap of size N; we use $g(N, M)$ to represent a gap whose size is within the range $[N, M]$. The range $[N, M]$ is called a **gap requirement**.

A **pattern** is a sequence of characters and gaps that begins and ends with characters. We define the **length** of a pattern P, denoted by $|P|$, as the number of characters in P. For example, if $P = A..T.C$, then $|P| = 3$. Note that the wild-card symbols are not counted towards the pattern's length.

Given a pattern P, a substring Q of P is called a **sub-pattern** of P if Q itself is also a pattern (i.e., Q also starts and ends with characters). If $|P| \geq 2$, its sub-pattern containing the first $|P| - 1$ characters is called the **prefix** of P. Similarly, the sub-pattern of P that contains the last $|P|-1$ characters is called the **suffix** of P. We use $\mathit{prefix}(P)$ and $\mathit{suffix}(P)$ to represent the prefix and suffix of P, respectively. For example, $\mathit{prefix}(A..T.C) = A..T$ and $\mathit{suffix}(A..T.C) = T.C$.

Given a subject sequence S (a pattern P), we use $S[i]$ ($P[i]$) to represent the i-th character of S (P). For example, if $S = ACGTA$, then $S[1] = A$, $S[2] = C$, etc. If $P = A..T.C$, then $P[1] = A$, $P[2] = T$.

For our problem of mining periodic patterns from a sequence, we are interested in patterns of the following form:

$$a_1 g(N, M) a_2 g(N, M) \dots a_{l-1} g(N, M) a_l \qquad (1)$$

where $a_i \in \sum$ for $1 \leq i \leq l$, and N, M are two user supplied numbers that specify the minimum and maximum gap sizes between two successive characters in a pattern, respectively. If the gap size requirement is understood, as a shorthand, we express a pattern P by simply specifying the characters it contains (i.e., $a_1 a_2 \dots a_l$). For example, if $N = 8$ and $M = 10$, the pattern written as ATC refers to the pattern $Ag(8, 10)Tg(8, 10)C$. Since the mining problem is defined with specified values of N and M, in the following discussion, we use the shorthand notation for patterns, unless otherwise stated.

Given a sequence S of length L, an **offset sequence** of length l is a sequence of integers $[c_1, \dots, c_l]$, such that $1 \leq c_j \leq L$ for all j, and $c_{j+1} - c_j - 1 \in [N, M]$ for all $1 \leq j \leq l - 1$. Essentially, an offset sequence is simply a sequence of positions of S that satisfies the gap requirement.

Our goal is to determine frequently occurring patterns given a subject sequence S. Hence, we need to define the term *frequency* and how often a pattern P occurs before we consider it *frequent* in S. We define **frequency** of a pattern P by the probability of observing P if we randomly pick $|P|$ positions of S (i.e., a random offset sequence) that satisfy the gap requirement. Also, a pattern P is considered **frequent**, if its frequency exceeds certain user-specified threshold value, ρ_s.

Given a sequence S, a pattern P, and an offset sequence $[c_1, \dots, c_{|P|}]$, we say that P matches S w.r.t. the offset sequence if $S[c_j] = P[j]$ for all $1 \leq j \leq |P|$. We define the **support** of P w.r.t. S (denoted by $\mathit{sup}(P)$) as the number of distinct offset sequences with respect to which P matches S. For example, if $S = AAGCC$, $P = AC$, and gap requirement is $[2, 3]$, then P matches S w.r.t. the offset sequence $[1, 4]$ since $S[1] = P[1]$ and $S[4] = P[2]$. Similarly, P matches S w.r.t. the offset sequences $[1, 5]$ and $[2, 5]$. So $\mathit{sup}(P) = 3$. A straightforward way to compute P's support is to enumerate all possible offset sequences, check the

Symbol	Definition
S	A subject sequence
P	A pattern
N	The minimum gap between 2 successive characters in a pattern
M	The maximum gap between 2 successive characters in a pattern
L	Length of S; $L = \|S\|$
l	Length of P; $l = \|P\|$
W	Flexibility of a gap; $W = M - N + 1$
$\mathit{minspan}(l)$	The minimum span of a length-l pattern $\mathit{minspan}(l) = (l-1)N + l$
$\mathit{maxspan}(l)$	The maximum span of a length-l pattern $\mathit{maxspan}(l) = (l-1)M + l$
l_1	The length of a longest pattern whose maximum span is $\leq \|S\|$ $l_1 = \lfloor \frac{L+M}{M+1} \rfloor$
l_2	The length of a longest pattern whose minimum span is $\leq \|S\|$ $l_2 = \lfloor \frac{L+N}{N+1} \rfloor$

Table 1: Notations

contents of S with respect to all those offset sequences, and determine the fraction of the offset sequences with respect to which P matches S. If the fraction exceeds the required threshold ρ_s, P is frequent; otherwise P is infrequent.

To determine whether a pattern P of length l is frequent with respect to a sequence S, we need two numbers: (1) N_l, the number of offset sequences of length l in S and (2) $\mathit{sup}(P)$. If the **support ratio**, $\mathit{sup}(P)/N_l$, is larger than ρ_s, P is a frequent pattern.

In the following section, we derive a formula for computing N_l. In Section 5, we derive algorithms for computing all patterns P that satisfy the frequency requirement.

4. MATHEMATICAL ANALYSIS

In this section we derive a recurrence equation for determining the value of N_l. We also prove several important theorems that allow us to formulate efficient algorithms for solving the periodic pattern mining problem. For reference, Table 1 shows the various symbols and their definitions we use in this section.

We use the variable W to denote the *flexibility* of the gap requirement. For example, if the gap requirement is $[4, 6]$, then the flexibility is $6 - 4 + 1 = 3$. That is to say, if the first character of a pattern P matches the sequence S at a certain position, say j (i.e., $P[1] = S[j]$), then there are three possible positions of S for $P[2]$ to match against, namely, $S[j+5]$, $S[j+6]$ and $S[j+7]$. Also, the larger is the flexibility, the larger is the number of offset sequences that satisfy the gap requirement, and so, the value of N_l will be larger.

We use $\mathit{minspan}(l)$ to denote the minimum span of a length-l pattern P. As an example, with a gap requirement of $[3, 4]$, a length-3 pattern spans at least 9 positions of the subject sequence. This is obtained by taking the smallest gap of 3 positions between the first and the second characters of P, and 3 positions between the second and the third. (Figure 1 illustrates the concept.) Since a length-l pattern has l characters and $l-1$ gaps and the minimum gap size is N, the minimum span is thus equal to $(l-1)N+l$. Similarly, we can determine the maximum span of a length-l pattern (denoted by $\mathit{maxspan}(l)$), which is equal to $(l-1)M+l$.

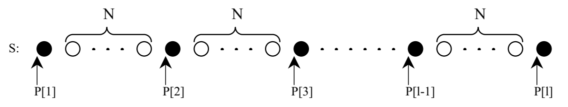


Figure 1: Illustration of *minspan*

Given a length-L sequence S, we use the symbol l_1 to denote the length of the longest patterns whose maximum span does not exceed L. The number l_1 can be obtained by solving $maxspan(l_1) = (l_1 - 1)M + l_1 \leq L$, which gives $l_1 = \lfloor \frac{L+M}{M+1} \rfloor$. Similarly, l_2 denotes the length of the longest patterns whose minimum span does not exceed L. We have $l_2 = \lfloor \frac{L+N}{N+1} \rfloor$. Since $M \geq N$, we have $l_2 \geq l_1$.

4.1 Determining N_l

Given a pattern length l, a subject sequence length L, and a gap requirement $[N, M]$, we would like to calculate N_l, the number of distinct length-l offset sequences. Here, we first summarize the result. We consider three cases:

1. For $l > l_2$, $N_l = 0$.
2. For $l \leq l_1$, $N_l = \left[L - (l-1)(\frac{M+N}{2} + 1)\right] W^{l-1}$.
3. For $l_1 < l \leq l_2$, N_l can be determined by a recursive formula.

Case 1 ($l > l_2$): The minimum span of a length-l pattern exceeds the subject sequence's length. Hence, there are no length-l offset sequences.

Case 2 ($l \leq l_1$): The maximum span of a length-l pattern is less than or equal to the subject sequence's length. In this case, we find that N_l grows exponentially with respect to l. Also, the larger is the flexibility of the gap requirement (W), the larger is N_l. Let us consider an example to illustrate how big N_l is. In one of the experiments we performed, we used a DNA sequence fragment that consists of 1,000 characters (i.e., $L = 1,000$), a gap requirement of $N = 9$ and $M = 12$, and so $W = 4$. The number of length-10 offset sequences N_{10} is about 235 million.[2]

Case 3 ($l_1 < l \leq l_2$): The boundary cases in which the span of a length-l pattern may or may not exceed the subject sequence's length. In this case, instead of a closed-form formula, we provide a computable recursive formula for N_l.

The analysis for deriving N_l for the cases is rather lengthy. Interested readers are referred to [23] for details.

4.2 Determining Frequent Patterns

Like many other data mining problems, our objective is to discover frequent patterns from data under a definition of "frequent". A common difficulty shared by most mining problems is that the number of patterns is huge. So a straightforward method of enumerating all possible patterns and counting their supports is not feasible. Traditional mining algorithms achieve efficiency by various pruning techniques that aim at drastically reducing the number of patterns that need to be checked. One very important property that enables effective pruning is the *Apriori* property,

[2]A typical helix turn of some organism is about 10 to 11 characters. We use a slightly larger gap requirement so that most patterns of interest are considered.

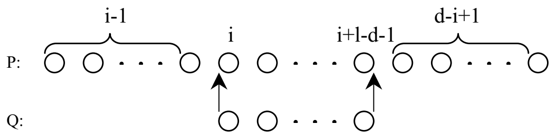


Figure 2: Patterns P and Q

which states that "the support of a pattern cannot exceed the support of any of its sub-patterns." The *Apriori* property is shown to hold under many data mining problems and models. The well-known `Apriori` algorithm [13] is a classic example that uses the *Apriori* property. In `Apriori`, an itemset X cannot be frequent if any proper subset of X is not frequent, and in which case, X is pruned.

For our mining problem, the *Apriori* property, however, does not hold. As a simple example, consider the sequence $S = ACTTT$, the pattern $P_1 = AT$ and its sub-pattern $P_2 = A$. If the gap requirement is $[1, 3]$, we see that $sup(P_1) = 3$ (corresponding to the offset sequences $\{[1, 3], [1, 4], [1, 5]\}$) while $sup(P_2) = 1$ (corresponding to the offset sequence $\{[1]\}$). Hence, the support of a pattern can exceed the support of its sub-pattern.

To achieve pruning, we derive an *apriori-like* property. Theorems 1 and 2 summarize the property.

THEOREM 1. *Given a length-l pattern P and a length-$(l-d)$ sub-pattern $Q = P[i]P[i+1] \ldots P[i+l-d-1]$ of P, where $1 \leq i \leq d+1$, we have $sup(Q) \geq sup(P)/W^d$.*

Proof: Let U be the set of all length-l offset sequences with respect to which P matches S. We have $sup(P) = |U|$. We partition U into R subsets $U_1, \ldots, U_R$ such that two offset sequences $A = [c_{a_1}, \ldots, c_{a_l}]$ and $B = [c_{b_1}, \ldots, c_{b_l}]$ are in the same subset U_j if and only if $c_{a_k} = c_{b_k} \forall i \leq k \leq i+l-d-1$. We see that each U_j corresponds to a unique offset sequence with respect to which Q matches S. Therefore, $sup(Q) \geq R$. Since the offset sequences in a given U_j only differ in the first $i-1$ offsets and the last $d-i+1$ offsets (see Figure 2), the cardinality of each U_j cannot exceed $W^{(i-1)+(d-i+1)}$ or W^d. Hence, R, the number of subsets U_j's must be at least equal to $|U|/W^d$. Therefore,

$$sup(Q) \geq R \geq |U|/W^d = sup(P)/W^d.$$

□

Theorem 1 is an important one in that it allows us to prune a large number of candidate patterns from consideration. In particular, if a length-l pattern P is frequent, then by definition, we have $sup(P)/N_l \geq \rho_s$. Now, consider a length-$(l-d)$ sub-pattern Q of P. Theorem 1 requires that

$$\frac{sup(Q)}{N_{l-d}} \geq \frac{sup(P)}{N_{l-d}W^d} \geq \frac{N_l}{N_{l-d}W^d}\rho_s = \lambda_{l,d} \cdot \rho_s, \qquad (2)$$

where $\lambda_{l,d} = \frac{N_l}{N_{l-d}W^d}$. That is, the support ratio of Q also has to attain a certain value.

One can also verify the following transitivity property of λ:

$$\lambda_{l,d_1+d_2} = \lambda_{l,d_1} \cdot \lambda_{l-d_1,d_2} \quad \forall 0 \leq d_1 \leq l \text{ and } \forall 0 \leq d_1+d_2 \leq l. \qquad (3)$$

K_r	K_1	K_2	K_3	K_4	K_5	K_6	K_7	K_8
Value	2	1	2	1	0	0	0	0

Table 2: K_r of sequence $ACGTCCGT$

As an example, if $l \leq l_1$, then by Equation 2 and the value of N_l stated in Section 4.1, one can easily show that,

$$\frac{sup(Q)}{N_{l-d}} \geq \frac{N_l}{N_{l-d}W^d}\rho_s = \frac{L-(l-1)(\frac{M+N}{2}+1)}{L-(l-d-1)(\frac{M+N}{2}+1)}\rho_s. \quad (4)$$

Here $\lambda_{l,d} = \frac{N_l}{N_{l-d}W^d} = \frac{L-(l-1)(\frac{M+N}{2}+1)}{L-(l-d-1)(\frac{M+N}{2}+1)}$.

For a long subject sequence (i.e., large L), a small pattern length (i.e., small l), and a small d, the fraction $\lambda_{l,d}$ is very close to 1. Therefore, if a length-l pattern P is frequent (i.e., its support ratio exceeds ρ_s), Theorem 1 implies that any length-$(l-d)$ sub-pattern Q of P has to have its support ratio exceed $\lambda_{l,d} \cdot \rho_s$, or *almost* ρ_s as well. Hence, we obtain a property that is very close to the *apriori* property. One can derive an efficient pruning algorithm based on that observation.

In the proof of Theorem 1, we bound the cardinality of the set U_j by W^d. The bound is obtained by considering the extreme case that given an offset sequence $A = [c_{a_1}, \ldots, c_{a_i}, \ldots c_{a_{i+l-d-1}}, \ldots c_{a_l}] \in U_j$, any perturbation of the first $i-1$ offsets and the last $d-i+1$ offsets (as long as the gap requirement is still satisfied) results in another offset sequence in U_j. In other words, any such perturbation gives us an offset sequence w.r.t. which P matches S. That is to say, no matter how we change the first $i-1$ offsets $[c_{a_1}, \ldots, c_{a_{i-1}}]$, we observe the same sequence of characters $S[c_{a_1}] = P[1], \ldots, S[c_{a_{i-1}}] = P[i-1]$, and the same can be said for the last $d-i+1$ offsets. The bound is obviously too loose.

We now consider a method of tightening the bound. Given a small value m, we consider all length-$(m+1)$ offset sequences of the form $[(r), (r+g^1), \ldots, (r+g^1+\ldots+g^m)]$, where each $g^j \in [N+1, M+1]$. Let us inspect S according to those offset sequences and use K_r to denote the frequency count of the most frequently occurring patterns observed. We repeat the exercise for each value of $1 \leq r \leq L$. Finally, we take $e_m = \max_{r=1}^{L} K_r$. We illustrate the idea with a simple example. Suppose $S = ACGTCCGT$, the gap requirement is $[1,2]$, and $m = 2$. We first calculate K_1. There are 4 possible length-$(m+1)$ (or length-3) offset sequences whose first element is equal to 1: $[1,3,5]$, $[1,3,6]$, $[1,4,6]$ and $[1,4,7]$, and they correspond to patterns AGC, AGC, ATC, and ATG, respectively. We see that AGC is the most frequently occurring pattern and its count is 2, so $K_1 = 2$. For K_2, the relevant offset sequences are $[2,4,6]$, $[2,4,7]$, $[2,5,7]$ and $[2,5,8]$. Since these 4 offset sequences give 4 different patterns CTC, CTG, CCG and CCT, by definition $K_2 = 1$. Other K_r values are calculated similarly. The results are shown in Table 2. Finally we get $e_m = \max_{r=1}^{8} K_r = 2$.

Semantically, for any offset r, the value e_m tells us how many times at most we will see the same character sequence in S under the offset sequence $[r, r+g^1, \ldots, r+g^1+\ldots+g^m]$ however we perturb the last m offsets in the sequence. Essentially, we use e_m to replace W^m as a better bound since $\frac{W^m}{e_m} \geq 1$. In the above example, $\frac{W^m}{e_m} = \frac{2^2}{2} = 2$. In typical DNA sequences, we find that the ratio $\frac{W^m}{e_m}$ becomes larger as m increases.

To illustrate how the value e_m is used, let us re-visit Theorem 1 again and consider the following example. Suppose the sub-pattern Q is taken from the first $l-8$ characters of P (i.e., $Q = P[1]P[2]\ldots P[l-8]$). If we follow the proof of Theorem 1 again, we see that all offset sequences $A = [c_{a_1}, \ldots, c_{a_{l-8}}, c_{a_{l-7}}, \ldots, c_{a_l}]$ in U_j only differ in the last 8 offsets. Now, suppose we have determined the value of e_m for the case $m = 3$. We know that, however we perturb the offsets $c_{a_{l-7}}$, $c_{a_{l-6}}$, $c_{a_{l-5}}$, the maximum number of times that we see the same character sequence (namely, $P[l-7]$, $P[l-6]$ and $P[l-5]$) over those three offsets is e_m. The same is true for the offsets $c_{a_{l-4}}$, $c_{a_{l-3}}$, $c_{a_{l-2}}$. And finally, there are at most W^2 ways for us to perturb the offsets $c_{a_{l-1}}$ and c_{a_l}. Hence, $|U_j| \leq e_m^2 W^2$. This bound could be much smaller than W^8 Theorem 1 uses. With this discussion, the following theorem can be easily proved.

THEOREM 2. *Given a length-l pattern P and a length-$(l-d)$ sub-pattern $Q = P[1]\ldots P[l-d]$ of P such that $s = \lfloor d/m \rfloor$ and $t = d - sm$, we have $sup(Q) \geq \frac{sup(P)}{e_m^s W^t}$.*

From Theorem 2, we know that if a length-l pattern P is frequent, then the length-$(l-d)$ sub-pattern Q of P such that $Q = P[1]\ldots P[l-d]$ must have its support ratio lower-bounded by:

$$\begin{aligned} sup(Q)/N_{l-d} &\geq \frac{sup(P)}{e_m^s W^t N_{l-d}} \\ &\geq \frac{N_l}{N_{l-d}e_m^s W^t}\rho_s \\ &= \frac{W^{d-t}}{e_m^s} \cdot \lambda_{l,d} \cdot \rho_s \\ &= (\frac{W^m}{e_m})^s \cdot \lambda_{l,d} \cdot \rho_s \\ &= \lambda'_{l,d} \cdot \rho_s, \qquad (5) \end{aligned}$$

where $s = \lfloor d/m \rfloor$, $t = d - sm$, and $\lambda'_{l,d} = (\frac{W^m}{e_m})^s \cdot \lambda_{l,d}$.

5. ALGORITHMS

In the previous section we discuss why pruning is an important issue in typical mining problems. In this section we propose efficient algorithms that apply pruning techniques.

5.1 MPP

Consider Equation 4 in Section 4 again. We note that if a length-l pattern P is frequent w.r.t. the support threshold ρ_s, then any length-$(l-d)$ sub-pattern Q of P must have a support ratio not less than $\lambda_{l,d} \cdot \rho_s$. This leads to the following apriori-like mining algorithm. We call our algorithm `MPP`.

First, let us assume that the user has a rough idea about the length of the longest frequent patterns in the subject sequence S. Let n represents such a length. `MPP` guarantees that all frequent patterns of length less than or equal to n are returned. For the longer frequent patterns, `MPP` will take a *best-effort approach*, i.e., it will return as many of those frequent patterns as it could.

To obtain all frequent patterns of length less than or equal to n, Equation 4 suggests that we obtain all length-1 patterns whose support ratios are not less than $\lambda_{n,n-1} \cdot \rho_s$.

(Other length-1 patterns would not be the constituents of any longer frequent patterns of interest.) From those patterns, we *join* them to obtain a set of length-2 *candidate patterns*. We examine the subject sequence and collect all those candidate patterns whose support ratios are not less than $\lambda_{n,n-2} \cdot \rho_s$. We then join those patterns collected to form a set of length-3 candidate patterns and so on. In general, during the i-th iteration, the algorithm determines a set (denoted by $\hat{L}_i$) of length-i patterns whose support ratios are not less than $\lambda_{n,n-i} \cdot \rho_s$. In the $(i+1)$-st iteration, patterns in $\hat{L}_i$ are joined to form the set of candidate length-$(i+1)$ patterns (denote by C_{i+1}). Patterns in C_{i+1} whose support ratios are not less than $\lambda_{n,n-(i+1)} \cdot \rho_s$ are collected in $\hat{L}_{i+1}$. The process repeats until either (1) `MPP` generates an empty candidate set, or (2) when $i = n+1$.

For the second case, `MPP` would have returned all frequent patterns of lengths less than or equal to n. To find other longer frequent patterns, `MPP` reverts to a basic Apriori-like method. That is, during each iteration $i > n$, `MPP` generates candidate set C_i based on L_{i-1}. It then checks the patterns in C_i and collects those whose support ratios are not less than ρ_s in L_i. The process repeats until `MPP` generates an empty candidate set. Note that in this candidate pattern generation process, a length-$(n+k)$ pattern P (where $k > 0$) is generated (and potentially is returned by the algorithm as a frequent pattern) only if there is a length-n sub-pattern Q of P whose support ratio is not less than ρ_s. From Equation 4, however, we see that a length-$(n+k)$ pattern can be frequent if all of its length-n sub-patterns have their support ratios reach $\lambda_{n+k,k} \cdot \rho_s$, which is less than ρ_s. In other words, there could be length-$(n+k)$ frequent patterns that are not generated and are thus missed. As a result, `MPP` can only guarantee that all frequent patterns of lengths less than or equal to n are discovered.

There are a few issues concerning `MPP` as outlined above:

- First, if the user does not have a good idea about how long frequent patterns are, he may choose an arbitrarily large n. In that case, pruning is not effective. For example, consider the case when `MPP` is determining L_4. The pruning condition requires that every length-4 candidate sequence with a support ratio not less than $\lambda_{n,n-4} \cdot \rho_s$ be included in L_4. If n is very large, $\lambda_{n,n-4}$ is very small, and few candidates can be removed.

- Second, for DNA sequences, the size of the alphabet, (e.g., $|\{A, C, T, G\}|$), is small. The number of combinations of short patterns is thus very small. Hence, short patterns are likely frequent. For example, in our experiment, we find that patterns of lengths one or two are always frequent. These patterns are thus uninteresting.

- Third, given a length-i candidate pattern P, checking its support might require us to examine the subject sequence S with respect to every length-i offset sequences. As we have discussed in Section 4, the number of length-i offset sequences equals N_i, a very large number even for a moderate value of i.

For the first issue, if $n > l_1$, `MPP` restricts n to l_1. That is to say, `MPP` will only guarantee the extraction of all frequent patterns whose lengths are less than or equal to l_1. We remark that even without a theoretical guarantee that *all* patterns longer than l_1 are found, the drawback, in practice, may not be detrimental. Incidentally, in all of the experiments we performed on DNA sequences, very long frequent patterns do not occur.

For the second issue, `MPP` starts with length-3 patterns, assuming that shorter ones are uninteresting. `MPP` does not count their supports and saves a bit of computation.

For the third issue, `MPP` uses an index structure called *partial index list* (*PIL*) to avoid examining all offset sequences when counting a pattern's support count.

Given a subject sequence S and a length-l pattern P, $PIL(P)$ is a list of (x, y) pairs where all x's are of distinct values. If the pair (x, y) is in $PIL(P)$, then there are exactly y offset sequences of the form $[x, c_2, \ldots, c_l]$ with respect to which P matches S (i.e., $P[1] = S[x], \ldots, P[l] = S[c_l]$). For example, if $S = AACCGTT$, $P = ACT$, $[N, M] = [1, 2]$, then $PIL(P) = \{(1, 3), (2, 2)\}$. This is because P matches S with respect to three offset sequences with the first offset equals 1 (namely, $\{[1, 3, 6], [1, 4, 6], [1, 4, 7]\}$) and two offset sequences with the first offset equals 2 (namely, $\{[2, 4, 6], [2, 4, 7]\}$).

There are two properties of $PIL(P)$:

1. Given $PIL(P)$, one can easily compute $sup(P)$, which is just the sum of all y's in the list. Using our previous example, since $PIL(P) = \{(1, 3), (2, 2)\}$, we have $sup(P) = 3 + 2 = 5$.

2. For a pattern P, let $prefix(P) = Q_1$, $suffix(P) = Q_2$. $PIL(P)$ can be computed from $PIL(Q_1)$ and $PIL(Q_2)$ using the following simple procedure.

```
1  $\forall (x, y) \in PIL(Q_1)$
2    $t = 0$
3    $\forall (x', y') \in PIL(Q_2)$ s.t. $x' - x - 1 \in [N, M]$
4      $t = t + y'$
5    if ($t > 0$), insert $(x, t)$ in $PIL(P)$
```

Figure 3 shows the algorithm `MPP`. The algorithm basically follows our previous discussion. For generating length-$(i+1)$ candidates, `MPP` considers every pair of length-i patterns P_1 and P_2 in $\hat{L}_i$. If $suffix(P_1) = prefix(P_2)$, then the candidate pattern $P_1[1]P_2$ is put into C_{i+1}. For example, $P_1 = ACG$ and $P_2 = CGT$ generate $ACGT$. `MPP` also calculates the *PIL* list of the candidate using $PIL(P_1)$ and $PIL(P_2)$. The *PIL* list of the candidate pattern allows us to determine its support count and therefore whether the candidate should be added to the set $\hat{L}_{i+1}$ or not. Finally, all patterns in all $\hat{L}_i$'s with support ratios not less than ρ_s are returned to the user.

5.2 MPPm

The efficiency of `MPP` relies on how effective pruning is. In this subsection we discuss how `MPP` can be refined to improve its pruning effectiveness and thus to achieve better efficiency.

Recall that given a value of n (the length of the longest frequent patterns the user is interested in obtaining), a candidate length-i pattern Q in the set C_i is pruned if its support ratio is less than $\lambda_{n,n-i} \cdot \rho_s$. So, the larger the value $\lambda_{n,n-i}$ is, the more effective is the pruning. By the definition of λ, that implies a small $n - i$, or equivalently, a small n and a large i. While we don't have many choices for i (the algorithm always starts with mining length-3 patterns, so i starts at 3), the above argument indicates that a reasonably

```
Algorithm MPP($S$, $\rho_s$, $N$, $M$, $n$)
  calculate $W, l_1, l_2$
  if $n > l_1$, $n = l_1$
  for i=3 to $n$
    calculate $N_i, \lambda_{n,n-i}$
  for $i = n+1$ to $l_2$
    calculate $N_i$, and set $\lambda_{n,n-i} = 1$
  $C_3$ = the set of all length-3 patterns
  scan $S$ to compute the $PIL$s of all patterns in $C_3$
  for each pattern $P$ in $C_3$
    get $sup(P)$ from $PIL(P)$
    if $sup(P) \geq \rho_s N_3$, put $P$ into $L_3$
    if $sup(P) \geq \lambda_{n,n-3}\rho_s N_3$, put $P$ into $\hat{L}_3$
  $i := 3$
  while ($\hat{L}_i \neq \emptyset$)
    $C_{i+1} := Gen(\hat{L}_i)$
    $\forall P \in C_{i+1}$
      compute $PIL(P)$ to get $sup(P)$
      if $sup(P) \geq \rho_s N_{i+1}$, put $P$ into $L_{i+1}$
      if $sup(P) \geq \lambda_{n,n-(i+1)}\rho_s N_{i+1}$, put $P$ into $\hat{L}_{i+1}$
    $i := i+1$
  Return $L_3 \cup L_4 \cup \ldots \cup L_{i-1}$
```

Figure 3: Algorithm MPP

small value of n can potentially speed up the algorithm. As an example, in our experiment, we use a DNA fragment of 1,000 characters, a gap requirement of [9,12], and a support threshold $\rho_s = 0.003\%$, the longest pattern mined has a length of 13. If the user has a good idea of how long frequent patterns are, and picks $n = 13$ as the algorithm's input, our experiment shows that `MPP` could achieve good pruning and is efficient. The question is "what if the user does not know what n to pick?" We will come back to this question shortly.

In the derivation of Theorem 2, we discussed how to derive a tighter bound that leads to a more effective pruning strategy. Without repeating the details, the approach is to pick a small number m and analyze the subject sequence to obtain a number e_m. From Theorem 2 and Equation 5, we know that if there is a length-k frequent pattern P, then the sub-pattern Q of P that consists of the first $k-d$ characters (i.e., $Q = P[1]\ldots P[k-d]$) must have its support ratio not less than $\lambda'_{k,d} \cdot \rho_s$, where $\lambda'_{k,d} = (\frac{W^m}{e_m})^s \cdot \lambda_{k,d}$, $s = \lfloor d/m \rfloor$.

Now, let us consider the "pick-the-n" problem again. If the user does not have a good idea of n, our approach is to find a reasonable value *automatically.* The idea is to count the supports of all length-3 patterns. Then, for every value of $3 < k \leq l_1$, we check and see if there is any length-3 pattern Q whose support is not less than $\lambda'_{k,k-3} \cdot N_3 \cdot \rho_s$. If no such Q exists, then by Theorem 2, we know that there are no length-k frequent patterns. Finally, the value of n is taken as the largest k such that length-k frequent patterns may exist. We thus modify `MPP` with the above procedure of automatically determining n applied. We call the modified algorithm `MPPm`.

6. EXPERIMENT RESULTS AND ANALYSIS

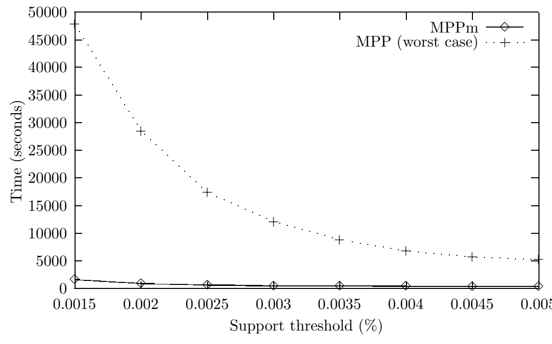
To analyze the performance of the mining algorithms, we perform an extensive experimental study. This section shows some representative results and discusses some interesting properties of the algorithms.

The data used in the experiments is the Homo Sapiens (human) DNA sequence AX829174 downloaded from the National Center for Biotechnology Information website [11]. The sequence consists of 10,011 characters. In the experiments, we randomly pick a length-L segment from AX829174 as the subject sequence for various values of L.

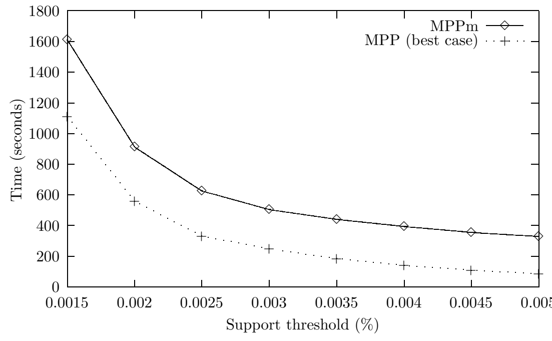
As we have discussed, the difference between `MPP` and `MPPm` is that `MPP` relies on a user input, n, which specifies an estimate of the length of the longest frequent patterns in the subject sequence, while `MPPm` tries to determine the estimate automatically. It is thus interesting to see how the estimation accuracy affects the performance of the algorithms. In our first experiment, we run `MPP` and `MPPm` for various values of support threshold, ρ_s. We note that different values of ρ_s yield different sets of frequent patterns. Let us use $n_o(\rho_s)$ to denote the length of the longest frequent patterns under a certain value of ρ_s. For each ρ_s, we execute `MPP` twice, one with the user input $n = n_o(\rho_s)$ (i.e., the best case scenario where the user has a perfect estimate of n), and the other with $n = l_1$ (i.e., the worst case scenario where the user has no idea about n and uses the largest value).

Figure 4 shows the performance of the algorithms. In this experiment, the subject sequence length $L = 1,000$, the gap requirement is $[9, 12]$ and `MPPm` uses $m = 10$.

L=1,000, $[N, M] = [9, 12]$, $m = 10$



(a)



(b)

Figure 4: `MPPm` vs. `MPP` (worst case) and `MPP` (best case)

First, we observe from the figure that as the support threshold increases, the execution times of the algorithms decrease. This is because a larger ρ_s gives fewer frequent patterns to extract. Also, from Figure 4(a), we see that without a reasonable estimate of n, the performance of MPP (worst case) is very bad. MPPm, on the other hand, is much more efficient due to its ability to determine a much smaller n. As an example, when $\rho_s = 0.003\%$[3], the experiment result shows that the longest frequent pattern has a length of $n_o(0.003\%) = 13$. While MPP uses $n = l_1 = 77$, MPPm estimates a value of $n = 22$. As we have explained in the previous section, a small value of n enables a much better pruning condition when the algorithms are determining which candidate patterns in C_i should be collected in the set $\hat{L}_i$. It explains why MPPm is much more efficient than MPP under the worst case.

Figure 4(b) compares MPPm against MPP when the user has a perfect estimate of n. From the figure, we see that MPPm is less efficient than MPP (best case). There are two reasons why MPPm takes longer time to execute. First, determining the value e_m (so that MPPm can apply Theorem 2 to estimate a value of n) requires MPPm to check quite a number of length-m patterns in the subject sequence S (see the discussion preceding Theorem 2). This overhead is not required for MPP. Second, MPP (best case) uses a smaller (and accurate) n value than MPPm does. Pruning is thus more effective. The performance difference, however, is not as big as that between MPPm and MPP (worst case). For example, in the above experiments, MPPm is 1.5 to 3.7 times slower than MPP (best case), and it is 16 to 30 times faster than MPP (worst case).

Our experiment result also shows that both MPP and MPPm are much more efficient than the straight forward way of enumerating all candidates. Since the *Apriori* property does not hold, the enumeration algorithm has to count all possible candidates. In our experiment settings, the number of candidates counted by the enumeration method is 4^i for C_i . On the other hand, both MPP and MPPm are able to prune a large number of candidates. Table 3 shows the number of candidates processed by the enumeration algorithm, MPP (worst case), MPPm and MPP (best case), respectively. The enumeration algorithm is impractical due to the large number of candidates it needs to process. The number of candidates MPP (worst case) has to deal with is also large, however, it becomes computable. For MPPm, it counts much fewer candidates than MPP (worst case), which explains why MPPm is much faster than MPP (worst case). MPP (best case) processes even fewer candidates than MPPm. Therefore it has the shortest execution time. The large difference between MPP (worst case) and MPP (best case) indicates that the efficiency of the MPP algorithm is dominated by the user input n, the estimated length of the longest frequent patterns.

To further illustrate the effect of the user input n, we execute MPP over different values of n. In this experiment $n_o(\rho_s) = 13$. Figure 5 shows the result. As expected, the worse is the estimate (a larger n), the slower is MPP. What is interesting about this figure is the execution time of MPP when $n = 10$, a value that is *smaller* than $n_o(\rho_s)$, the length of the longest frequent patterns in S. That is to say, when the user *under-estimate* $n_o(\rho_s)$. From the figure, we see that the execution time of MPP is smaller than the case when n equals the *real* maximum length $n_o(\rho_s)$.

[3]Recall that a pattern P is frequent if $sup(P) \geq N_l\rho_s$. Since N_l is exponentially large w.r.t. l, even a small ρ_s implies a fairly large support count of P.

	Enumeration Algorithm	MPP (worst case)	MPPm	MPP (best case)
C_3	64	64	64	64
C_4	256	256	256	256
C_5	1024	1024	1024	1024
C_6	4096	4096	4096	4096
C_7	16384	16384	16384	16384
C_8	65536	65528	54588	50609
C_9	262144	231161	17464	12198
C_{10}	1048576	177140	2926	2262
C_{11}	4194304	37543	1057	783
C_{12}	16777216	16114	346	222
C_{13}	4^{13}	7552	42	26
C_{14}	4^{14}	2919	6	3
C_{15}	4^{15}	1009	-	-
C_{16}	4^{16}	356	-	-
C_{17}	4^{17}	43	-	-
C_{18}	4^{18}	8	-	-
C_{19}	4^{19}	-	-	-
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_{77}	4^{77}	-	-	-

Table 3: Number of candidates counted by different algorithms

Recall that for a given user input n, MPP will find all frequent patterns of lengths less than or equal to n. For longer patterns, MPP takes a best-effort approach and tries to return those frequent patterns as many as possible. Therefore, if $n < n_o(\rho_s)$, not all frequent patterns are guaranteed to be found. The small execution time of MPP when $n = 10$ as shown in Figure 5, however, hints at an adaptive approach to determine a suitable n value. Specifically, if a user has no idea of a good n value, we could run MPP using a small n, let's say 10. After MPP finishes execution, it will return all frequent patterns of length less than or equal to n plus a number of longer frequent patterns. We could note the longest pattern discovered, use its length to refine n and re-execute MPP. This process could continue until we cannot refine n further. Although we do not explore this ap-

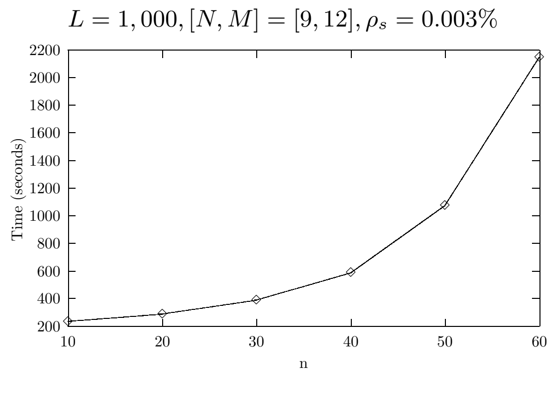


Figure 5: Performance of MPP under different user input n

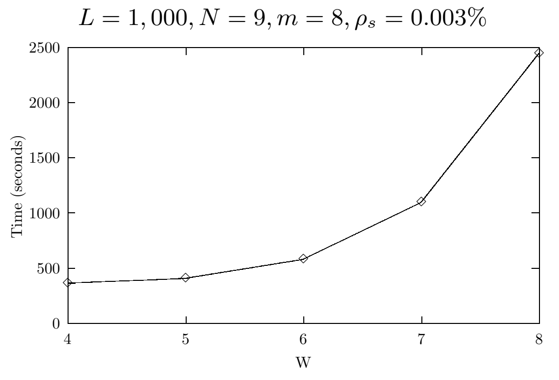


Figure 6: Performance of MPPm under different values of *W*

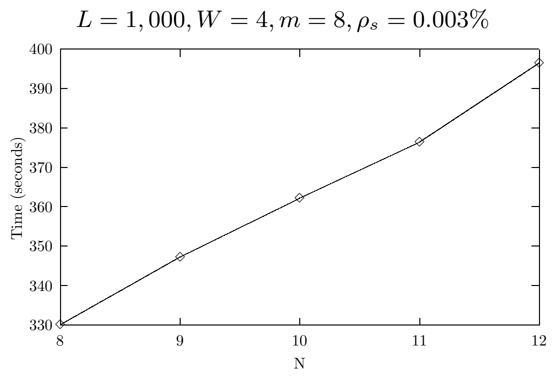


Figure 7: Performance of MPPm under different values of *N*

proach further in this paper, we remark that since the cost of running MPP with a small n is low, running it a few times adaptively as the way we just described could still be a very efficient method.

In another experiment, we study the effect of the flexibility of the gap W. We fix $N = 9$ and hence the gap requirement is $[9 \ldots W + 8]$. Figure 6 shows the performance of MPPm when W changes from 4 to 8. From the figure, we see that the larger is W, the larger is the execution time of the algorithm. This is because, for a given l, the number of length-l offset sequences, N_l, is proportional to W^{l-1} (see Section 4.1). That is, the larger the value of W, the larger is N_l. Hence, the *PIL* lists with which the algorithm uses to count patterns' supports are long. Therefore, more computational effort is needed. From the figure, we see that for MPPm (and MPP) to be practical, the gap flexibility, W, has to be reasonably small. Fortunately, the helical structure of DNA sequences does not imply a large flexibility. For example, in some organism, a helical turn consists of 10 to 11 base pairs, which implies a flexibility of 2.

In the next experiment, we fix the gap flexibility W to 4 and vary the value of N. The gap requirement is thus $[N, N + 3]$. Figure 7 shows the performance of MPPm as N varies from 8 to 12. From the figure, we see that the execution time of MPPm increases with N. Recall that after MPPm has estimated a value of n, it basically follows the logic of MPP. In particular, during the iteration in which MPPm determines the set $\hat{L}_i$, a candidate pattern in C_i is removed if its support ratio is less than $\lambda_{n,n-i} \cdot \rho_s$. According to Equation 4, $\lambda_{n,n-i} = \frac{L-(n-1)(\frac{M+N}{2}+1)}{L-(i-1)(\frac{M+N}{2}+1)}$. One can verify that $\lambda_{n,n-i}$ is a decreasing function of N. Hence, the smaller the value of N, the larger $\lambda_{n,n-i}$ is, and more candidate patterns can be pruned. This leads to a more efficient algorithm.

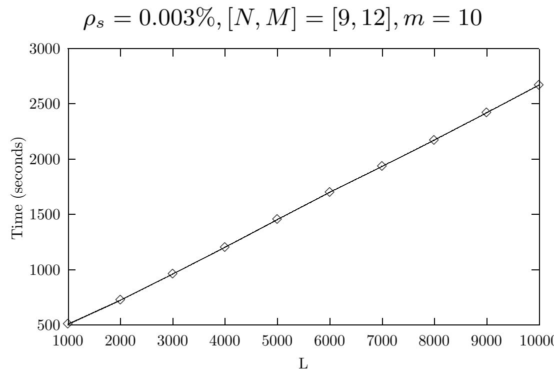


Figure 8: Performance of MPPm for various values of *L*

Our final experiment studies the scalability property of the algorithm. Figure 8 shows the execution time of MPPm as the length of the subject sequence (L) varies from 1,000 to 10,000 characters. The result shows that MPPm scales linearly with the sequence length.

7. A CASE STUDY

In this section we report a case study in which interesting patterns are mined using our algorithms. We applied MPPm to mine a number of DNA sequences, including the whole genomes of the bacteria H. influenzae, H. pylori, M. genitalium and M. pneumoniae. We segmented the genomes into short fragments of 100 kilo-bases (kb), and ran the algorithm on each fragment using a gap size of $[10, 12]$ and a support threshold of 0.006%. The length of the longest patterns discovered was 10 bases (characters). We observed a very interesting result: the bases 'A' and 'T' constitute much more to the periodic patterns than 'C' and 'G'. For instance, there are 256 length-8 patterns that consist of only 'A's and 'T's. We found that all such patterns were frequent in some fragments of all four genomes. Some of these patterns were even frequent in every fragment examined. As an example, if we consider fragments from bacteria genomes only, then on average, about 250 of the 256 length-8 patterns that consist of only 'A's and 'T's were frequent in a given fragment. On the other hand, length-8 patterns that consist of more than one 'C' or 'G' were unlikely to be frequent. For example, there are $4^8 = 65,536$ possible length-8 patterns, among which $2^8 = 256$ contain only 'A's and 'T's, and $8 \times 2 \times 2^7 = 2,048$ contain exactly one 'C' or 'G'. So, the number of possible patterns that have more than one 'C' or 'G' is $65,536 - 256 - 2,048 = 63,232$. We found that among these patterns, on average, only 3.9 of them were frequent in a DNA fragment of bacteria genomes. Also, none such frequent patterns is common in all genomes.

The results are consistent with the findings of a previous study [7], which shows the periodic occurrence of 'A' and 'T' in yeast and various bacteria and archaea with a period length of 10-11 base pairs. Our results complement its findings by showing that beyond the regularity that occurs between nucleotide pairs, the patterns actually last for quite a number of contiguous cycles. Also, some patterns are ubiquitous in the genomes, not restricting to any specific regions.

In a previous work that extensively studies ApA dinucleotide periodicity (the regular occurrence of base 'A' after another base 'A' separated by a fixed period) in various eubacteria, archaebacteria, eukaryotes and organelles, it has been suggested that the periodic patterns are more prominent in eubacteria than in eukaryotes [17]. For instance, the genome of H. sapiens (human) shows very weak periodicity, as compared to the eubacteria and some lower eukaryotes such as the baker yeast S. cerevisiae. We would like to verify whether the periodic patterns are really weakened in higher eukaryotes, or strong periodic patterns still exist, but they are composed of other bases or do not exhibit a rigorous periodicity with a fixed period length. We downloaded short pieces of the genomes of the eukaryotes H. sapiens, C. elegans and D. melanogaster, cut them into 100kb fragments, and repeated the above experiments. To our surprise, all of the 256 length-8 patterns that consists of 'A' and 'T' only are still frequent in some fragments of all three sequences. This result may imply that the flexible gap requirement is able to tolerate some variations in the sequences, such as the insertion or deletion of a nucleotide within a period that affects the period length.

Besides, some patterns not detected in the bacterial genomes are observed in the eukaryote sequences, many of which consist of more 'C's and 'G's. For instance, the length-8 pattern composing of 'G's only is frequent in some fragments of all three sequences. In one of the fragments of H. sapiens, the pattern composing of 16 G's only is also found to be frequent! All these suggest that the nucleotides involved in the periodic patterns in bacteria and eukaryotes are quite different.

Some former studies suggest two explanations for the dinucleotide oscillations [18, 17, 7]: (1) they are related to the helical shape of the DNA. In particular, the repetition of specific base-pair stacks with this periodicity would cause uni-directional deflection of the DNA curvature; (2) the alternation of hydrophobic and hydrophilic amino acids in α-helices leads to a periodicity of about 3.5 amino acids in protein sequences, which corresponds to 10-11 bases in DNA sequences. Both explanations are still possible given the new findings. The new results also further suggest that in eukaryotes, the maintenance of the DNA curvature may involve more 'C's and 'G's than in bacteria. Also, to verify the second explanation, it is useful to actually look for some proteins with a corresponding coding DNA sequence that exhibits the mined periodic patterns.

Finally, we have applied our algorithm on mining DNA sequences of many different species. We found that there are unique periodic patterns for each species. Some of these patterns are very interesting. For example, for C. elegans, we found periodic patterns that repeat themselves, such as `ATATATATATA`, `GTAGTAGTAGT`, etc. As another example, a unique periodic pattern for H. sapiens consists of 17 'G's. Biologists may find those patterns insightful.

8. CONCLUSION

This paper studied the problem of mining periodic patterns with a gap requirement from sequences. We formally defined the data-mining model and proved several important theorems that lead to the derivation of efficient algorithms. We proposed two algorithms, namely, `MPP` and `MPPm` for solving the problem. Extensive experiments had been done to illustrate the various performance characteristics of the algorithms. We found that for cases in which the user has a good estimate of the length of the longest frequent patterns, `MPP` is the most efficient algorithm. On the other hand, if the user does not provide the estimate, `MPPm` is able to determine a reasonably good one. We applied `MPPm` on a number of real DNA sequences. Much of our mining result is consistent with findings from previous studies.

9. REFERENCES

[1] Stephen F. Altschul, Warren Gish, Webb Miller, Eugene W. Myers, and David J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[2] A. Bairoch and B. Boeckmann. The swiss-prot protein sequence data bank. *Nucleic Acids Research*, 20(Suppl):2019–2022, 1992.

[3] Giorgio Bernardi, Birgitta Olofsson, Jan Filipski, Marino Zerial, Julio Salinas, Gerard Cuny, Michele Meunier-Rotival, and Francis Rodier. The mosaic genome of warm-blooded vertebrates. *Science*, 228(4702):953–958, 1985.

[4] Eivind Coward and Finn Drablos. Detecting periodic patterns in biological sequences. *Bioinformatics*, 14(6):498–507, 1998.

[5] J. W. Fickett and C. S. Tung. Assessment of protein coding measures. *Nucleuic Acids Research*, 20:6441–6450, 1992.

[6] Jiawei Han, Guozhu Dong, and YiWen Yin. Efficient mining of partial periodic patterns in time series database. In *Proc. of 15th International Conference on Data Engineering, ICDE99*, pages 106–115, 1999.

[7] H. Herzel, O. Weiss, and E. N. Trifonov. 10-11 bp periodicities in complete genomes reflect protein structure and DNA folding. *Bioinformatics*, 15(3):187–193, 1999.

[8] Inge Jonassen. Efficient discovery of conserved patterns using a pattern graph. Technical Report Report No. 118, University of Bergen, 1996.

[9] Stefan Kurtz, Enno Ohlebusch, Chris Schleiermacher, Jens Stoye, and Robert Giegerich. Computation and visualization of degenerate repeats in complete genomes. In *Proceedings of the 8th International Conference on Intelligent Systems for Molecular (ISMB-00)*, 2000.

[10] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, Nov 1997.

[11] http://www.ncbi.nlm.nih.gov.

[12] Jian Pei, Jiawei Han, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns by prefix-projected growth. In *Proc. 17th IEEE International Conference on Data Engineering (ICDE)*, Heidelberg, Germany, April 2001.

[13] T. Imielinski R. Agrawal and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, page 207, Washington, D.C., May 1993.

[14] P.S. Reddy and D.E. Housman. The complex pathology of trinucleotide repeats. *Current Opinion in Cell Biology*, 9(3):364–372, 1997.

[15] Isidore Rigoutsos and Aris Floratos. Combinatorial pattern discovery in biological sequences: the teiresias algorithm. *Bioinformatics*, 14(1), 1998.

[16] Ramakrishnan Srikant and Rakesh Agrawal. Mining sequential patterns: Generalizations and performance improvements. In *Proc. of the 5th Conference on Extending Database Technology (EDBT)*, Avignion, France, March 1996.

[17] Masaru Tomita, Masahiko Wada, and Yukihiro Kawashima. ApA dinucleotide periodicity in prokaryote, eukaryote, and organelle genomes. *Journal of Molecular Evolution*, 49:182–192, 1999.

[18] E. N. Trifonov. 3-, 10.5-, 200- and 400-base periodicities in genome sequences. *Physica A*, 249:511–516, 1998.

[19] A. van Belkum, S. Scherer amd W. van Leeuwen, D. Willemse, L. van Alphen, and H. Verbrugh. Variable number of tandem repeats in clinical strains of haemophilus influenzae. *Infection and Immunity*, 65(12):5017–5027, 1997.

[20] J. Widom. Short-range order in two eukaryotic genomes: Relation to chromosome structure. *Journal of Moleular Biology*, 259:579–588, 1996.

[21] Jiong Yang, Wei Wang, and Philip S. Yu. Mining asynchronous periodic patterns in time series data. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 275–279, Boston, MA USA, 2000.

[22] Mohammed J. Zaki. Efficient enumeration of frequent sequences. In *Proceedings of the 1998 ACM 7th International Conference on Information and Knowledge Management(CIKM'98)*, Washington, United States, November 1998.

[23] Minghua Zhang, Ben Kao, David W. Cheung, and Kevin Y. Yip. Mining periodic patterns with gap requirement from sequences. Technical Report CS Technical Report TR-2005-6, The University of Hong Kong, 2005.

[24] Minghua Zhang, Ben Kao, C.L. Yip, and David Cheung. A GSP-based efficient algorithm for mining frequent sequences. In *Proc. of IC-AI'2001*, Las Vegas, Nevada, USA, June 2001.