Permutation Pattern Discovery in Biosequences

REVITAL ERES,2 GAD M. LANDAU,2,3 and LAXMI PARIDA4

ABSTRACT

Functionally related genes often appear in each other’s neighborhood on the genome; however, the order of the genes may not be the same. These groups or clusters of genes may have an ancient evolutionary origin or may signify some other critical phenomenon and may also aid in function prediction of genes. Such gene clusters also aid toward solving the problem of local alignment of genes. Similarly, clusters of protein domains, albeit appearing in different orders in the protein sequence, suggest common functionality in spite of being nonhomologous. In the paper, we address the problem of automatically discovering clusters of entities, be they genes or domains: we formalize the abstract problem as a discovery problem called the \( \pi \) pattern problem and give an algorithm that automatically discovers the clusters of patterns in multiple data sequences. We take a model-less approach and introduce a notation for maximal patterns that drastically reduces the number of valid cluster patterns, without any loss of information. We demonstrate the automatic pattern discovery tool on motifs on \( E. \ Coli \) protein sequences.

Key words: design and analysis of algorithms, combinatorial algorithms on words, discovery, data mining, clusters, patterns, motifs.

1. INTRODUCTION

Genes that appear together consistently across genomes are believed to be functionally related: these genes in each other’s neighborhood often code for proteins that interact with one another suggesting a common functional association. However, the order of the genes in the chromosomes may not be the same. In other words, a group of genes appears in different permutations in the genomes (Marcott et al., 1999; Overbeek et al., 1999; Snel et al., 2000). For example, in plants, the majority of snoRNA genes are organized in polycistrons and transcribed as polycistronic precursor snoRNAs (Brown et al., 2001). Also, the olfactory receptor(OR)-gene superfamily is the largest in the mammalian genome. Several of the human OR genes appear in cluster with 10 or more members located on almost all human chromosomes, and some chromosomes contain more than one cluster (Giglio et al., 2001).

---

1This paper is the extended journal version of (Parida et al., 2003).
2Department of Computer Science, University of Haifa, Haifa 31905, Israel.
3Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840.
4Computational Biology Center, IBM TJ Watson Research Center, Yorktown Heights, New York 10598.
As the available number of complete genome sequences of organisms grows, it becomes a fertile ground for investigation along the direction of detecting gene clusters by comparative analysis of the genomes. A gene $G$ is compared with its orthologs $G'$ in the different organism genomes. Even phylogenetically close species are not immune from gene shuffling, such as in Haemophilus influenzae and Escherichia Coli (Watanbe et al., 1997; Siefert et al., 1997). Also, a multicistronic gene cluster sometimes results from horizontal transfer between species (Lawrence and Roth, 1996), and multiple genes in a bacterial operon fuse into a single gene encoding multi-domain protein in eukaryotic genomes (Marcott et al., 1999).

If the functions of genes, say, $G_1G_2$ is known, the function of its corresponding ortholog clusters $G'_2G'_1$ may be predicted. Such positional correlation of genes as clusters and their corresponding orthologs have been used to predict functions of ABC transporters (Tomii and Kanehisa, 1998) and other membrane proteins (Kihara and Kanehisa, 2000).

The local alignment of nucleic or amino acid sequences, called the multiple sequence alignment problem, is based on similar subsequences; however, the local alignment of genomes (Ogata et al., 2000) is based on detecting locally conserved gene clusters. A measure of gene similarity is used to identify the gene orthologs. For example, genes $G_1G_2G_3$ may be aligned with $G'_3G'_1G'_2$: such an alignment is never detected in subsequence alignments.

Domains are portions of the coding gene (or the translated amino acid sequences) that correspond to a functional subunit of the protein. Often, these are detectable by conserved nucleic acid sequences or amino acid sequences. The conservation helps in a relatively easy detection by automatic motif discovery tools. However, the domains may appear in a different order in the distinct genes giving rise to distinct proteins. But they are functionally related due to the common domains. Thus, these represent functionally coupled genes such as operon structures for coexpression (Tamames et al., 1997; Dandekar et al., 1998).

In the paper, we address the problem of automatically discovering clusters of genes or domains. A similar problem is addressed by Nakaya et al. (2001) that integrates data from different sources, such as gene expression data and metabolic pathways, and works on a single genome at a time. Yet another variation has been addressed as the problem of finding common intervals in multiple permutations (Heber and Stoye, 2001). In this paper, we formalize the abstract problem as a discovery problem called the $\pi$ pattern problem and give an algorithm that automatically discovers the clusters of patterns (that appear in various permuted forms in the instances) in multiple data sequences. As there is not enough knowledge about forming an appropriate model to filter the meaningful from the apparently meaningless clusters, we take a model-less approach and introduce a notation for maximal patterns that drastically reduces the number of valid cluster patterns, without any loss of information, making it easier to study the results from an application viewpoint.

We demonstrate the automatic pattern discovery tool on motifs on E. Coli protein sequences. It is interesting to observe that permutations involving as many as eight motifs are discovered. Although its biological significance is yet to be established, nevertheless, it appears to be an interesting phenomenon.

Roadmap. In the next section, we formalize the problem. In the following section, we introduce our notion of maximality and its associated notation so that there is no loss of information. We next describe the algorithm and then give some experimental results, open problems, and conclusions.

2. THE $\pi$ PATTERN PROBLEM

We begin by giving some definitions.

Let $S = s_1s_2 \ldots s_n$ be a string of length $n$, and $P = p_1p_2 \ldots p_m$ a pattern, both over alphabet $\{1, \ldots, |\Sigma|\}$. 

**Definition 1 ($\Pi(s)$, $\Pi'(s)$).** Given a string $s$ on alphabet $\Sigma$, $\Pi(s) = \{\alpha \in \Sigma \mid \alpha = s[i], \text {for some } 1 \leq i \leq |s|\}$ and $\Pi'(s) = \{\alpha(t) \mid \alpha \in \Pi(s), t \text {is the number of times that } \alpha \text {appears in } s\}$.

For example, if $s = abcd$, $\Pi(s) = \{a, b, c, d\}$. If $s = abbccdac$, $\Pi'(s) = \{a(2), b(2), c(3), d\}$. Note that $d$ appears only once and we ignore the annotation altogether.
Definition 2 (p-occurs). A pattern $P$ p-occurs (permuted occurrence) in a string $S$ at location $i$ if:
$$\Pi'(P) = \Pi'_{i}(s_i \ldots s_{i+m-1}).$$

Definition 3 ($\pi$-pattern). Given an integer $K$, a Pattern $P$ is a $\pi$-pattern on $S$ if:

- $|P| > 1$, we rule out the trivial single character patterns.
- $P$ p-occurs at some $k' \geq K$ distinct locations on $S$; $L_{p} = \{i_1, i_2, \ldots, i_{k'}\}$ is the location list of $p$.

For example, consider $K = 2$, $\Pi'(P) = \{a(2), b(3), c\}$, and the string $S = aacbbbxxabcdabab$. Clearly, $P$ p-occurs at positions 1 and 9.

The problem of permutation pattern ($\pi$-pattern) discovery. Given a string $S$ and $K < n$, find all $\pi$-patterns of $S$ together with their location lists.

For example, if $S = abcdbacdaab$, then $P = \{a, b, c\}$ is a 4-$\pi$-pattern with location list $L_{p} = \{1, 5, 10, 11\}$.

The total number of $\pi$-patterns is $O(n^2)$, but is this number actually attained? Consider the following example.

Example 1. Let $S = abcdefghijjabcede fghi j$ and $k = 2$. The $\pi$-patterns below show that their number could be quadratic in the size of the input.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Location List</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1 = {a, b}$</td>
<td>$L_{p_1} = {1, 11}$</td>
</tr>
<tr>
<td>$P_2 = {a, b, c, d}$</td>
<td>$L_{p_2} = {1, 11}$</td>
</tr>
<tr>
<td>$P_3 = {a, b, c, d, e}$</td>
<td>$L_{p_3} = {1, 11}$</td>
</tr>
<tr>
<td>$P_4 = {a, b, c, d, e, f}$</td>
<td>$L_{p_4} = {1, 11}$</td>
</tr>
<tr>
<td>$P_5 = {a, b, c, d, e, f, g, h}$</td>
<td>$L_{p_5} = {1, 11}$</td>
</tr>
<tr>
<td>$P_6 = {a, b, c, d, e, f, g, h, i}$</td>
<td>$L_{p_6} = {1, 11}$</td>
</tr>
<tr>
<td>$P_7 = {a, b, c, d, e, f, g, h, i, j}$</td>
<td>$L_{p_7} = {1, 11}$</td>
</tr>
<tr>
<td>$P_8 = {b, c, d}$</td>
<td>$L_{p_8} = {2, 12}$</td>
</tr>
<tr>
<td>$P_9 = {b, c, d, e, f}$</td>
<td>$L_{p_9} = {2, 12}$</td>
</tr>
<tr>
<td>$P_{10} = {b, c, d, e, f, g, h}$</td>
<td>$L_{p_{10}} = {2, 12}$</td>
</tr>
<tr>
<td>$P_{11} = {b, c, d, e, f, g, h, i}$</td>
<td>$L_{p_{11}} = {2, 12}$</td>
</tr>
<tr>
<td>$P_{12} = {c, d}$</td>
<td>$L_{p_{12}} = {3, 13}$</td>
</tr>
<tr>
<td>$P_{13} = {c, d, e}$</td>
<td>$L_{p_{13}} = {3, 13}$</td>
</tr>
<tr>
<td>$P_{14} = {c, d, e, f}$</td>
<td>$L_{p_{14}} = {3, 13}$</td>
</tr>
<tr>
<td>$P_{15} = {c, d, e, f, g, h}$</td>
<td>$L_{p_{15}} = {3, 13}$</td>
</tr>
<tr>
<td>$P_{16} = {c, d, e, f, g, h, i}$</td>
<td>$L_{p_{16}} = {3, 13}$</td>
</tr>
<tr>
<td>$P_{17} = {c, d, e, f, g, h, i, j}$</td>
<td>$L_{p_{17}} = {3, 13}$</td>
</tr>
<tr>
<td>$P_{18} = {e, f}$</td>
<td>$L_{p_{18}} = {5, 15}$</td>
</tr>
<tr>
<td>$P_{19} = {e, f, g, h}$</td>
<td>$L_{p_{19}} = {5, 15}$</td>
</tr>
<tr>
<td>$P_{20} = {e, f, g, h, i, j}$</td>
<td>$L_{p_{20}} = {5, 15}$</td>
</tr>
<tr>
<td>$P_{21} = {f, g, h}$</td>
<td>$L_{p_{21}} = {6, 16}$</td>
</tr>
<tr>
<td>$P_{22} = {f, g, h, i, j}$</td>
<td>$L_{p_{22}} = {6, 16}$</td>
</tr>
<tr>
<td>$P_{23} = {g, h}$</td>
<td>$L_{p_{23}} = {7, 17}$</td>
</tr>
<tr>
<td>$P_{24} = {g, h, i, j}$</td>
<td>$L_{p_{24}} = {7, 17}$</td>
</tr>
<tr>
<td>$P_{25} = {i, j}$</td>
<td>$L_{p_{25}} = {9, 19}$</td>
</tr>
</tbody>
</table>

3. MAXIMAL PATTERNS

We give a general definition of maximality which holds even for different kinds of substring patterns such as rigid, flexible, and with or without wild cards (Parida, 2000). In the following, assume that $P$ is the set of all $\pi$-patterns on a given input string $S$. 
Definition 4. \( P_a \in \mathcal{P} \) is nonmaximal if there exists \( P_b \in \mathcal{P} \) such that (1) each \( p \)-occurrence of \( P_a \) on \( S \) is covered by a \( p \)-occurrence of \( P_b \) on \( S \), (each occurrence of \( P_a \) is a substring in an occurrence of \( P_b \)) and (2) each \( p \)-occurrence of \( P_b \) on \( S \) covers \( l \geq 1 \), \( p \)-occurrence(s) of \( P_a \) on \( S \). A pattern \( P_b \) that is not nonmaximal is maximal.

Clearly, \( \Pi'(P_a) \subset \Pi'(P_b) \). Although it seems counterintuitive, it is possible that \(|L_{pa}| < |L_{pb}| \). Consider the input \( S = abcdebca \ldots abcde \).

The equality \( Pa = \{d,e\} \) \( p \)-occurs only two times, but \( Pb = \{a,b,c,d,e\} \) \( p \)-occurs three times and by the definition \( Pa \) is nonmaximal with respect to \( Pb \).

To illustrate the case of \( l > 1 \) in the definition, consider \( S = abcdbac \ldots abcabcd \ldots abcdabc \).

\( Pa = \{a,b,c\} \) \( p \)-occurs two times in the first and third, and four times in the second \( p \)-occurrence of \( Pb = \{(a)2,(b)2,(c)2,d\} \). Also, by the definition, \( Pa \) is nonmaximal with respect to \( Pb \).

We further claim that such a nonmaximal pattern \( Pa \) can be “deduced” from \( Pb \) and the \( p \)-occurrences of \( Pa \) on \( S \) can be estimated to be within the \( p \)-occurrences of \( Pb \). This will be shown to be a consequence of Theorem 2 in the next section.

Theorem 1. Let \( M = \{P_j \in \mathcal{P} \mid P_j \text{ is maximal}\} \). \( M \) is unique.

This is straightforward to see. This result holds even when the patterns are substring patterns. In Example 1, pattern \( P_7 \) is the only maximal \( \pi \) pattern in \( S \).

3.1. Maximal notation

Recall that in case of substring patterns, the maximal pattern very obviously indicates the nonmaximal patterns as well. For example, a maximal pattern of the form \( abcd \) implicates \( ab, bc, cd, abc \), and \( bcd \) as possible nonmaximal patterns, unless they have occurrences not covered by \( abcd \). Do maximal \( \pi \) patterns have such an obvious form? In this section, we introduce a special notation based on observations discussed below. We next demonstrate how this notation makes it possible to represent maximal \( \pi \) patterns.

Theorem 2. Let \( Q \in \mathcal{P} \) and \( Q = \{Q' \mid Q' \text{ is nonmaximal w.r.t } Q \} \). Then there exists a permutation, \( \overline{Q} \), of \( \Pi'(Q) \) such that for each element \( Q' \in Q \), a permutation of \( \Pi'(Q') \) is a substring of \( \overline{Q} \).

Proof. Without loss of generality, let the ordering of the elements be as the one in the leftmost occurrence of \( Q \) on \( S \) as \( \overline{Q} \). Clearly, there is a permutation of \( \Pi'(Q') \) that is a substring of \( \overline{Q} \), else \( Q' \) is not a nonmaximal pattern by the definition.

Corollary 1. The ordering is not necessarily complete. Some elements may have no order with respect to some others.

Consider \( S = abcd ef \ldots cadbf e \ldots abcd ef \). Then \( P_1 = \{a,b,c,d\} \), \( P_2 = \{e,f\} \), and \( P_3 = \{a,b,c,d,e,f\} \) are the \( \pi \) patterns with three occurrences each on \( S \). Then the intervals denoted by brackets can be represented as

\[
(3(1a,b,c,d),1,(2e,f)2)3
\]

where the elements within the brackets can be in any order. A pair of brackets \((i \ldots) \) corresponds to the \( \pi \) pattern \( P_i \). An element is either a character from the alphabet or bracketed elements.

Corollary 2. A representation that captures the order of the elements of \( Q \) along with the intervals that correspond to each \( Q' \) encodes the entire set \( Q \).

This representation will appropriately annotate the ordering. The representation using brackets works except that there may intersecting intervals that could lead to clutter. When the intervals intersect, the
brackets need to be annotated. For example, \((a(b, d)c)\) can have at least two distinct interpretations: (1) \((a(2a, d)3)c\), or (2) \((a(2a, d)_1)c\).

Consider the input string \(S = abcd \ldots dcba \ldots abcd\). The \(\pi\) patterns are \(P_1 = ab\), \(P_2 = bc\), \(P_3 = cd\), \(P_4 = abc\), \(P_5 = bcd\), and \(P_6 = abcd\), each occurring three times. Using only annotated brackets will yield a cluttered representation as follows:

\[
(6(1(a(2(3b)1(3c)2)4d)3)s)_6.
\]

The annotation of the brackets is required to keep the pairing of the brackets unambiguous. It is clear that if two intervals intersect, then the intersection elements are immediate neighbors of the remaining elements. For example, if \((a(2a, c)_1)d\), then \((b, c)\) must be immediate neighbors of \((a)\) as well as \((d)\).

We introduce a symbol “-” to denote immediate neighbors, then the intervals never intersect. Further, they do not need to be annotated if they do not intersect. Thus, the previous example can be simply given as \(a-(b, c)-d\). The earlier cluttered representation of Equation 1 can be cleanly put as

\[
a-b-c-d.
\]

Next, consider Example 1. Using the notation, there is only one maximal \(\pi\) pattern given by \(M = a-b-(c, d)-e-f-g-h-i-j\) at locations 1 and 11 on \(S\). Notice that \(\Pi(P_f) = \Pi(M)\) and every other \(\pi\) pattern can be deduced from \(M\).

4. THE ALGORITHM

The input of the algorithm is a set of strings of total length \(n\). In order to simplify the explanation, we consider one string \(S\) of length \(n\) over an alphabet \(\Sigma\).

The algorithm computes the maximal \(\pi\) patterns in \(S\). It has two stages: (1) find all the \(\pi\) patterns in \(S\), and (2) find the maximal \(\pi\) patterns in \(S\). In our implementation, in stage 2, we use a straightforward computation using location lists of all the \(\pi\) patterns in \(S\) obtained at stage 1. The location lists of each pair of \(\pi\) patterns are checked to find if one \(\pi\) pattern is covered by the other one. Assume that stage 1 outputs \(p\) \(\pi\) patterns and the maximum length of a location list is \(\ell\); stage 2 runs in \(O(p^2\ell)\) time. From now on, only stage 1 will be discussed.

We assume that the size of the longest pattern is \(L\). Step \(\ell\) \((2 \leq \ell \leq L)\) of Stage 1 finds \(\pi\) patterns of length \(\ell\). The computation is based on an algorithm given by Amir et al. (2003). Different approaches are given by Didier (2003) and Schmidt and Stoye (YEAR?).

The algorithm moves a window of size \(\ell\) along string \(S\), adding and deleting a letter in each iteration. This is similar to the algorithm for computing the sum of every consecutive \(\ell\) elements of an array.

The algorithm maintains an array \(\text{NAME}[1 \ldots |\Sigma|]\) where \(\text{NAME}[q]\) keeps count of the number of appearances of letter \(q\) in the current window. Hence, the sum of the values of the elements of \(\text{NAME}\) is \(\ell\). In each iteration, the window shifts one letter to the right, and at most two variables of \(\text{NAME}\) are changed; one is increased by one (adding the rightmost letter), and one is decreased by one (deleting the leftmost letter of the previous window). Note that for a given window \(s_o s_{o+1} \ldots s_{o+\ell-1}\) \(\text{NAME}\) represents \(\Pi'(s_o s_{o+1} \ldots s_{o+\ell-1})\). There is one difference between \(\text{NAME}\) and \(\Pi'\); in \(\Pi'\), only the letters of \(\Pi\) are considered, and in \(\text{NAME}\) all letters of \(\Sigma\) are considered, but the values of letters that are not in \(\Pi\) are zero. At iteration \(j\), we define \(\text{NAME}\) to represent the substring \(s_j \ldots s_{j+\ell-1}\).

Algorithm’s implementation

In order to implement the sliding window technique described above, we maintain the following data structures:

- Two pointers \(i_{left}\) and \(i_{right}\). At every iteration \((i_{left}, i_{right})\) is the window under consideration.
- Array \(\text{NAME}[1 \ldots |\Sigma|]\).
The main part of the algorithm consists of a move of \( i_{\text{right}} \) and \( i_{\text{left}} \) to the right and an update of two entries of \( \text{NAME} \), as described in the following code:

**Main Part of Algorithm**

Repeat until \( i_{\text{right}} = n \)

\[
\begin{align*}
& \text{[ } i_{\text{right}} \text{ Move ]} \\
& i_{\text{right}} \leftarrow i_{\text{right}} + 1 \\
& \text{NAME}[S_{i_{\text{right}}}] \leftarrow \text{NAME}[S_{i_{\text{right}}}] + 1 \\
\end{align*}
\]

\[
\begin{align*}
& \{ \text{Skip } i_{\text{left}} \text{ Move if window size is less than } l \} \\
& \{ i_{\text{left}} \text{ Move } \} \\
& \text{NAME}[S_{i_{\text{left}}}] \leftarrow \text{NAME}[S_{i_{\text{left}}} - 1] \\
& i_{\text{left}} \leftarrow i_{\text{left}} + 1 \\
\end{align*}
\]

Compute the name of \( \text{NAME} \)

**end Main Part of Algorithm**

**Observation.** Substrings of \( S \), of length \( \ell \), that are permutations of the same string are represented by the same \( \text{NAME} \).

We have explained how the \( \text{NAMEs} \) of all substrings of length \( \ell \) of \( S \) are computed. However, we still have to find the \( \text{NAMEs} \) that appear more than \( K \) times.

Each distinct \( \text{NAME} \) is given a unique name—an integer in the range 0 \ldots n. The names are given by using the naming technique (Apostolico et al., 1988; Kedem et al., 1996), which is a modified version of the algorithm of Karp, Miller, and Rosenberg (1972).

**4.1. The naming technique**

Assume, for the sake of simplicity, that \(|\Sigma|\) is a power of 2. (If \(|\Sigma|\) is not a power of 2, \( \text{NAME} \) can be extended to an appropriate size by concatenating to its end repeated -1. The size of the resulting array is no more than twice the size of the original array.)

A name is given to each subarray of size \( 2^i \) that starts on a position \( j2^i + 1 \) in the array, where \( 0 \leq i \leq \log |\Sigma| \) and \( 0 \leq j < |\Sigma|/2^i \). Names are given first to subarrays of size 1, then 2, 4, \ldots, \(|\Sigma|\); at the end, a name is given to the entire array.

A subarray of size \( 2^i \) is a concatenation of two subarrays of size \( 2^{i-1} \). The names of these two subarrays are used as the input for the computation of the name of the subarray of size \( 2^i \). The process may be viewed as constructing a complete binary tree, which we will refer to as a naming tree. The leaves of the tree (level 0) are the elements of the initial array. Node \( x \) in level \( i \) is the parent of nodes \( 2x - 1 \) and \( 2x \) in level \( i - 1 \).

Our naming strategy is as follows. A name is a pair of previous names. At level \( j \) of the naming, we compute the name of subarray \( \text{NAME}_1 \text{NAME}_2 \) of size \( 2^j \), where \( \text{NAME}_1 \) and \( \text{NAME}_2 \) are consecutive subarrays of size \( 2^{j-1} \) each. We give as names the natural numbers in increasing order. Notice that every level uses only the names of the level below it; thus, the names we use at every level are numbers from the set \( \{1, \ldots, n\} \).

To give an array a name, we need to know only whether the pair of names of the composing subarrays has appeared previously. If it did, then the array gets the name of this pair. Otherwise, it gets a new name. It is necessary, therefore, to show a quick way to dynamically access pairs of numbers from a bounded range universe. This is discussed in Section 4.2

**Example 2.** Let \( \Sigma = \{a, b, c, d, e, f, g, h, i, j, k, \ell, m, n, o, p\} \), \(|\Sigma| = 16\). Assume a substring \( \text{cbo\elljkikgikl} \) of \( S \); the array \( \text{NAME} \) that represents this substring is as follows.

\[
\begin{array}{cccccccccccc}
0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 2 & 2 & 2 & 0 & 0 & 1 & 0
\end{array}
\]
Below is the result of naming the above NAME.

<table>
<thead>
<tr>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Suppose the window move adds the letter n. In the diagram below, we indicate in boldface the names that changed as a result of the change to NAME.

<table>
<thead>
<tr>
<th>14</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

From Example 2, one can see that a single change in NAME causes at most \( \log |\Sigma| \) names to change, since there is at most one name change in every level.

**Time.** We conclude that at every iteration, only \( O(\log |\Sigma|) \) names need to be handled, since only two elements of array NAME are changed.

We have seen that the name of the NAME array can be maintained at a cost of \( O(\log |\Sigma|) \) per iteration. What has to be found is whether the updated NAME array gets a new name, or a name that appeared previously. Before we show an efficient implementation of this task, let us bound the maximum number of different names our algorithm needs to generate for a fixed window size \( \ell \).

**Lemma 3 (Amir et al., 2003).** The maximum number of different names generated by our algorithm’s naming of size \( \ell \) window on a text of length \( n \) is \( O(n \log |\Sigma|) \). The maximum number of names generated at a fixed level \( j \) in the naming tree is \( O(n) \).

### 4.2. The pair recognition problem

We have seen earlier that it is necessary to show a quick way to dynamically access pairs of numbers from a bounded range universe. Formally, we would like a solution to the following problem:

**Definition 5.** The dynamic pair recognition problem is the following:

*INPUT:* A sequence of queries \( \{(a_i, b_i)\}_{i=1}^{\infty} \), where \( a_i, b_i \in \{1, \ldots, j\} \).

*OUTPUT:* Dynamically decide, for every query \( (a_i, b_i) \), whether there exist \( c, c < j \) such that \( (a_j, b_j) = (a_c, b_c) \).

At any point \( j \), the pairs we are considering all have their first element no greater than \( j \). Thus, accessing the first element can be done in constant time by direct access. This suggests “gathering” all pairs in trees rooted at their first element. However, if we make sure these trees are ordered by the second element and balanced, we can find elements by binary search in time that is logarithmic in the tree size.

**Algorithm’s implementation**

The algorithm maintains the following data structure:

- \( \text{BAL}[a] \) is a balanced binary tree of all pairs \( (a, b) \) that have been named so far, sorted by \( b \). Since \( a, b \) are increasing natural numbers, starting from 1, \( \text{BAL}[a] \) is directly accessed by \( a \).
• When $a$ is the name appearing as the root of the naming tree, two data structures are attached to its $\text{BAL}[a]$:
  — $\text{counter}_a$: counts the number of substrings named $a$.
  — $\text{location}_a$: holds the starting locations of those substrings in $S$.

The algorithm is now straightforward. We are given pair $(a, b)$ at time $j$ and need to recognize if it has appeared so far.

**Pair Recognition Algorithm**

```plaintext
if $(a, b) \in \text{BAL}[a]$ then name is name$(a,b)$. 
else:
  $j \leftarrow j + 1$
  add $(a, b)$ to $\text{BAL}[a]$
  name$(a, b) \leftarrow j$
  initialize empty $\text{BAL}[j]$

if name$(a, b)$ is the name appearing in the root of the naming tree then
  add $i_{\text{left}}$ to $\text{location}_a$(name$(a,b)$)
  $\text{counter}_a(a,b) \leftarrow \text{counter}_a(a,b) + 1$

end Algorithm
```

**Time.** The above solution for the pair recognition algorithm requires, for solving each query $(a_j, b_j)$, a search on a balanced search tree with all previous queries whose first pair element is $a_j$. In our case, since in every level there are at most $O(n)$ different numbers, the time for searching such a balanced tree is $O(\log |\text{BAL}[a]|) = O(\log(n))$.

4.3. **Time complexity**

Stage 1 of our algorithm runs $L$ times. In a step $\ell$, we first initialize $\text{name}$ and the naming tree in $O(\ell + |\Sigma|)$ time and then compute $n - \ell$ iterations. Each iteration includes at most two changes in $\text{name}$ and the computation of $O(\log |\Sigma|)$ names. Computing a name takes $O(\log n)$ time. Hence, the total running time of our algorithm is $O(Ln \log |\Sigma| \log n)$.

5. **EXPERIMENTAL RESULTS**

We show some preliminary results on *E. Coli* protein sequences. The input to our system is substring patterns detected on pruned a set of *E. Coli* sequences: in this pruned set, no pair of sequences is 90% or more similar in the sequences using standard sequence similarity measures. There are 8,394 protein sequences with a total of about 1,391,900 amino acids in the dataset. The following parameters were used to obtain the substring patterns: (1) quorum: the patterns appear at least five times; and (2) wild card density: the patterns have no more than two wild cards in a window of twelve bases. The number of such substring patterns is 207. The input sequences are now viewed as sequences of motifs/domains with a possibility of multiple occurrences at a location. Thus, the alphabet size for this problem is 207. The $\pi$ pattern discovery tool is run on this input file to yield the result. The input files are available from following site: `www.cs.nyu.edu/~parida/res/public/data/` as “ecobase.dat.gz” and “ecobase.mtfs.gz.”

Table 1 shows the result of discovering $\pi$ patterns on this data where the alphabet is the motif/domain with parameters as described above. Figure 1 shows an example of a permuted $\pi$ pattern of size 6.
Table 1. Patterns on Motifs/Domains of the E. Coli Protein Sequences

<table>
<thead>
<tr>
<th>Size of patterns</th>
<th>Total number of patterns</th>
<th>Number of maximal patterns</th>
<th>Percentage of maximal patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>161</td>
<td>98</td>
<td>61%</td>
</tr>
<tr>
<td>3</td>
<td>129</td>
<td>53</td>
<td>41%</td>
</tr>
<tr>
<td>4</td>
<td>95</td>
<td>55</td>
<td>58%</td>
</tr>
<tr>
<td>5</td>
<td>43</td>
<td>17</td>
<td>40%</td>
</tr>
<tr>
<td>6</td>
<td>27</td>
<td>19</td>
<td>70%</td>
</tr>
<tr>
<td>7</td>
<td>15</td>
<td>11</td>
<td>67%</td>
</tr>
<tr>
<td>8</td>
<td>7</td>
<td>7</td>
<td>100%</td>
</tr>
</tbody>
</table>

Example of a permuted maximal pattern of size 6 in E Coli sequences:

(325) 35 {53 36} 81 136 {72 8} 35 {159 21} 36 109 {140 82 57}

(498) 35 {159 21} {53 36} 81 136 {72 8} 35 187 {159 21} 36 109 {166 145 140 82 79 74 71 57}

6 4 [35 36 72 81 136 159] (325 1) (325 2) (498 0) (498 1)

35 GETL.YGESGSGKST
36 VGESGSGKST
72 IADEPTTLDV
81 PHQLSGG..QRV
136 LSGC.RQRVIA
150 IV.GSGSKST

line 325
VLAVENLNAIFMQDQKIAVRNLFSFLQREGLANVESGSQKSYTGLALMRRLLEQAGGLV
QCDMILLQRRSSHVIELSEQNAAAQMRHVGRADMAMIFQEPMTSLNPVFTVGEQIAESIRLHQ
NASSREEAMVEAKRMDFQVRPEAPQTSRLYPHQLSQGMRQRAMIALSCRPVAVLIADE
PTTALDVQIAQQLQLIQLKEGMGQFTHIDMGVAVAEQDRVVLMYQGEAVETGTEQ
IFHAPQHPTYRALLAAPQVLGAKGKLDYPRFRPLSLHAPKAPPSEQKTVDGEPVLRVR
NLTVRFPLRSLGNLVTRVEHVEKYVSDLWPGGETLSVYGEGSCKSTGRALLRLYQVEGG
EIHFGQGQRDTLPSGQAKLRRDIQFQFDYASPQDLDPRQTIGDIIEPLVRHLLQPKGDAARAVW
LLERGVLLPEHAWRYPHESPQRQGRICPAARLALPNKVIADAEAVSLDVRQIINNLQDLQR
DFGIAYLFISHDMAVERISHRVARNYLGQVEICPRAVFENPQHYTRKLLAAVPVAPESPQQR
PQRVLLSDDLPNSSNHLRGEVEAASLQCVQPGHYVAQPSQUEYAMRR

line 498
MTQTLAIENLVSFGRHQQTVRTVNDVSLQIEAGEGLAYGESGSQKSYTALTSLRLLLPSPP
VEYLSGDRIFQGESSLHIASDQTQLRGRVGKNKIEMFQEPVMVSPLNLHTLEKQLYAVSLHRGMRR
EAARGEILNCLDRIGRQAARKRLTYQPHQLSGGERQRTMIALATLSPPELIADDEPTTL
DVSQVAQILQLLRELQGELMGMLFTHHNSLRVKLAHRAVVMQNGRCVEQNYATTLFASPTH
PYTQKLLNQEPGDPVLPLEPASTQDRLDVEQQLAYFPQFIRPKGLRIVDHNVVVKNISETLRAGETL
GLYGEGSCKSTGLALRLINQSGSHDFGQPLQNLNRQLLPVSRRHIQFVQDFNPSNLPRLN
VLQIEEGLRVRQPTLSAARERQQQTVIAVMHEVLDPETHRYYPAEFSGGQQRQRIAIARALILKPSL
IIADDEPTSSLDKTVQAIQILLLKSLQQHQLAYLFISHDLHVRLACHIQIVIIQGEVEQGPCAR
VFATPQKEYTRQLLAL

Figure 1. An example to show how a pair of domains (motifs) are numbered. The top two lines numbered 325 and 498 represent the input line numbers in the data. Each number in the row represents a domain (motif). Numbers in braces represents multiple occurrence of the domains: for example, domains 53 and 36 occur at the same location. The next line shows that it is a pattern of size 6 (i.e., six domains in the permutation) occurring in four locations, twice in sequence 325 and twice in sequence 498. The next six lines show the mapping of the domain numbers to the actual domains. The permuted domains are displayed in the original sequences at the bottom. Using the maximality notation, the pattern is represented as (36-81-136-72), 35, 159.
6. CONCLUSIONS

Related genes often appear in each other’s neighborhood on the genome; however, the order of the genes may not be the same. Such gene clusters also aid toward solving the problem of local alignment of genes. Similarly, clusters of protein domains, albeit appearing in different orders in the protein sequence, suggest common functionality in spite of being nonhomologous. In the paper, we have addressed the problem of automatically discovering clusters as a discovery problem called the $\pi$ pattern problem and gave an algorithm that automatically discovers the clusters of patterns in multiple data sequences. We have taken a model-less approach and introduced a notation for maximal patterns that drastically reduces the number of valid cluster patterns. We conclude with two open problems and some preliminary results of the automatic pattern discovery tool on motifs on $E. Coli$ protein sequences.

ACKNOWLEDGMENTS

We have greatly benefited from earlier discussions with Jens Stoye. RE and GML supported by the Israel Science Foundation grant 282/01 and by the First Foundation of the Israel Academy of Science and Humanities. GML also supported by NSF grant CCR-0104307 and the IBM Faculty Partnership Award.

REFERENCES


Address correspondence to:
Revital Eres
Department of Computer Science
Haifa University
Mount Carmel
Haifa 31905, Israel

E-mail: revitale@cslx.haifa.ac.il