Permutation Pattern Discovery in Biosequences *

Revital Eres[†] University of Haifa

Gad M. Landau[‡] University of Haifa & Polytechnic University

Laxmi Parida [§] IBM T J Watson Research Center

Abstract

Functionally related genes often appear in each others 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 it genes or domains: we formalize the abstract problem as a discovery problem called the πpattern 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 others 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 appear in different permutations in the

*This paper is the extended journal version of [ELP03].

†Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 828-8376, FAX: (972-4) 824-9331; email: `revitale@cslx.haifa.ac.il`; partially supported by by the Israel Science Foundation grant 282/01 and by the FIRST Foundation of the Israel Academy of Science and Humanities.

‡Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: `landau@poly.edu`; partially supported by NSF grant CCR-0104307, by the Israel Science Foundation grant 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, and by IBM Faculty Partnership Award.

§Computational Biology Center, IBM TJ Watson Research Center, Yorktown Heights, New York 10598 email`parida@us.ibm.com`

genomes [PNR+99, OFD+99, SLBH00]. For example in plants, the majority of snoRNA genes are organized in polycistrons and transcribed as polycistronic precursor snoRNAs [BCL+01]. Also, the olfactory receptor(OR)-gene superfamily is the largest in the mammalian genome. Several of the human OR genes appear in cluster with ten or more members located on almost all human chromosomes and some chromosomes contain more than one cluster [GBM+01].

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* [WMIG97, SMA+97]. Also, a multicistronic gene cluster sometimes results from horizontal transfer between species [LR96] and multiple genes in a bacterial operon fuse into a single gene encoding multi-domain protein in eukaryotic genomes [PNR+99].

If the functions of genes say G_1G_2 is known, the function of its corresponding ortholog clusters $G_2'G_1'$ may be predicted. Such positional correlation of genes as clusters and their corresponding orthologs have been used to predict functions of ABC transporters [TK98] and other membrane proteins [KK00].

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* [OFG00] 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 $G_3'G_1'G_2'$: 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 sub-unit of the protein. Often, these are detectable by conserved nucleic acid sequences or amino acid sequences. The conservation helps in a relative 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 forming operon structures for co-expression [TCOV97, DSHB98].

In the paper we address the problem of automatically discovering clusters of genes or domains. A similar problem is addressed in [NGK01] 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 [HS01]. In this paper, we formalize the abstract problem as a discovery problem called the πpattern 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 πPattern Problem

We begin by giving some definitions.

Let $S = s_1 s_2 \ldots s_n$ be a string of length n, and $P = p_1 p_2 \ldots p_m$ a pattern, both over alphabet $\{1, ..., |\Sigma|\}$.

Definition 1 *($\Pi(s)$, $\Pi'(s)$) Given a string s on alphabet Σ,*

$$\Pi(s) = \{\alpha \in \Sigma \mid \alpha = s[i], \textit{ for some } 1 \leq i \leq |s|\} \textit{ and}$$

$$\Pi'(s) = \{\alpha(t) \mid \alpha \in \Pi(s), t \textit{ is the number of times that } \alpha \textit{ appears in } s\}$$

For example if $s = abcda$, $\Pi(s) = \{a, b, c, d\}$. 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: $\Pi'(P) = \Pi'(s_i \ldots s_{i+m-1})$.*

Definition 3 *(πpattern) Given an integer K, a Pattern P is a πpattern 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. $\mathcal{L}_p = \{i_1, i_2, \ldots, i_{k'}\}$ is the location list of p.*

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

The Problem of Permutation Pattern (πPattern) Discovery. Given a string S and $K < n$, find all πpatterns of S together with their location lists.

For example, if $S = abcdbacdabacb$, then $P = \{a, b, c\}$ is a 4-πpattern with location list $\mathcal{L}_p = \{1, 5, 10, 11\}$.

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

Example 1 *Let* $S = abcdefghijabdcefhgij$ *and* $k = 2$. *The* π*patterns below show that their number could be quadratic in the size of the input.*

$$
\begin{array}{ll}
P_1 = \{a, b\}, & \mathcal{L}_{p_1} = \{1, 11\} \\
P_2 = \{a, b, c, d\}, & \mathcal{L}_{p_2} = \{1, 11\} \\
P_3 = \{a, b, c, d, e\}, & \mathcal{L}_{p_3} = \{1, 11\} \\
P_4 = \{a, b, c, d, e, f\}, & \mathcal{L}_{p_4} = \{1, 11\} \\
P_5 = \{a, b, c, d, e, f, g, h\}, & \mathcal{L}_{p_5} = \{1, 11\} \\
P_6 = \{a, b, c, d, e, f, g, h, i\}, & \mathcal{L}_{p_6} = \{1, 11\} \\
P_7 = \{a, b, c, d, e, f, g, h, i, j\}, & \mathcal{L}_{p_7} = \{1, 11\} \\
P_8 = \{b, c, d\}, & \mathcal{L}_{p_8} = \{2, 12\} \\
P_9 = \{b, c, d, e, f\}, & \mathcal{L}_{p_9} = \{2, 12\} \\
P_{10} = \{b, c, d, e, f, g, h\}, & \mathcal{L}_{p_{10}} = \{2, 12\} \\
P_{11} = \{b, c, d, e, f, g, h, i, j\}, & \mathcal{L}_{p_{11}} = \{2, 12\} \\
P_{12} = \{c, d\}, & \mathcal{L}_{p_{12}} = \{3, 13\} \\
P_{13} = \{c, d, e\}, & \mathcal{L}_{p_{13}} = \{3, 13\} \\
P_{14} = \{c, d, e, f\}, & \mathcal{L}_{p_{14}} = \{3, 13\} \\
P_{15} = \{c, d, e, f, g, h\}, & \mathcal{L}_{p_{15}} = \{3, 13\} \\
P_{16} = \{c, d, e, f, g, h, i\}, & \mathcal{L}_{p_{16}} = \{3, 13\} \\
P_{17} = \{c, d, e, f, g, h, i, j\}, & \mathcal{L}_{p_{17}} = \{3, 13\} \\
P_{18} = \{e, f\}, & \mathcal{L}_{p_{18}} = \{5, 15\} \\
P_{19} = \{e, f, g, h\}, & \mathcal{L}_{p_{19}} = \{5, 15\} \\
P_{20} = \{e, f, g, h, i, j\}, & \mathcal{L}_{p_{20}} = \{5, 15\} \\
P_{21} = \{f, g, h\}, & \mathcal{L}_{p_{21}} = \{6, 16\} \\
P_{22} = \{f, g, h, i, j\}, & \mathcal{L}_{p_{22}} = \{6, 16\} \\
P_{23} = \{g, h\}, & \mathcal{L}_{p_{23}} = \{7, 17\} \\
P_{24} = \{g, h, i, j\}, & \mathcal{L}_{p_{24}} = \{7, 17\} \\
P_{25} = \{i, j\}, & \mathcal{L}_{p_{25}} = \{9, 19\}
\end{array}
$$

3 Maximal Patterns

We give a general definition of maximality which holds even for different kinds of substring patterns such as rigid, flexible, with or without wild cards [Par00].

In the following, assume that $\mathcal{P}$ is the set of all πpatterns on a given input string S.

Definition 4 $P_a \in \mathcal{P}$ *is* **non-maximal** *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 non-maximal is* **maximal**.

Clearly, $\Pi'(P_a) \subset \Pi'(P_b)$. Although it seems counter-intuitive, but it is possible that $|\mathcal{L}_{p_a}| < |\mathcal{L}_{p_b}|$. Consider the input $S = abcdebca \ldots\ldots abcde$. $P_a = \{d, e\}$ p-occurs only two times but $P_b = \{a, b, c, d, e\}$ p-occurs three times and by the definition P_a is non-maximal with respect to P_b.

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

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

We further claim that such a non-maximal pattern P_a can be "deduced" from P_b and the p-occurrences of P_a on S can be estimated to be within the p-occurrences of P_b. This will be shown to be a consequence of Theorem 2 in the next section.

Theorem 1 *Let $\mathcal{M} = \{P_j \in \mathcal{P} |\ P_j$ is maximal$\}$. $\mathcal{M}$ 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 πpattern in S.

3.1 Maximality Notation

Recall that in case of substring patterns, the maximal pattern very obviously indicates the non-maximal patterns as well. For example a maximal pattern of the form *abcd* implicates *ab*, *bc*, *cd*, *abc*, *bcd* as possible non-maximal patterns, unless they have occurrences not covered by *abcd*. Do maximal πpatterns 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 πpatterns.

Theorem 2 *Let $Q \in \mathcal{P}$ and $\mathcal{Q} = \{Q' |\ Q'$ is non-maximal w.r.t Q $\}$. Then there exists a permutation, $\overline{Q}$, of $\Pi'(Q)$ such that for each element $Q' \in \mathcal{Q}$, 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 non-maximal pattern by the definition. ∎

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

Consider $S = abcdef \ldots\ldots cadbfe \ldots\ldots abcdef$. Then $P_1 = \{a, b, c, d\}$, $P_2 = \{e, f\}$ and $P_3 = \{a, b, c, d, e, f\}$ are the πpatterns 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 $({}_i \ldots)_i$ corresponds to the πpattern 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 $\mathcal{Q}$.*

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) $({}_1a({}_2b,d)_2c)_1$, or, (2) $({}_1a({}_2b,d)_1c)_2$.

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

$$({}_6({}_1({}_4a({}_2({}_5b)_1({}_3c)_2)_4d)_3)_5)_6 \tag{1}$$

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 $({}_1a({}_2b,c)_1d)_2$, 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

$$a\text{-}b\text{-}c\text{-}d$$

Next, consider Example 1. Using the notation, there is only one maximal πpattern given by $M = a$-b-(c,d)-e-f-(g,h)-i-j at locations 1 and 11 on S. Notice that $\Pi(P_7) = \Pi(M)$ and every other πpattern 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 Σ.

The algorithm computes the maximal πpatterns in S. It has two stages: (1) Find all the πpatterns in S, and (2) Find the maximal πpatterns in S. In our implementation, in Stage 2 we use a straightforward computation using location lists of all the πpatterns in S obtained at Stage 1. The location lists of each pair of πpatterns are checked to find if one πpattern is covered by the other one. Assume that stage 1 outputs p πpatterns, and the maximum length of a location list is ℓ, 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 ℓ $(2 \leq \ell \leq L)$ of Stage 1, finds πpatterns of length ℓ. The computation is based on an algorithm given by Amir et. al. [AALS03]. Different approaches are given in [D03, SS03]

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

The algorithm maintains an array $NAME[1\ldots|\Sigma|]$ where $NAME[q]$ keeps count of the number of appearances of letter q in the current window. Hence, the sum of the values of the elements of $NAME$ is ℓ. In each iteration the window shifts one letter to the right, and at most 2 variables of $NAME$ 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 $s_a s_{a+1} \ldots s_{a+\ell-1}$ $NAME$ represents $\Pi'(s_a s_{a+1} \ldots s_{a+\ell-1})$. There is one difference between $NAME$ and Π', in Π' only the letters of Π are considered and in $NAME$ all letters of Σ are considered, but the values of letters that are not in Π are zero. At iteration j we define $NAME$ to represents 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 $NAME[1..|\Sigma|]$.

The main part of the algorithm consists of a move of i_{right} and i_{left} to the right and an update of two entries of $NAME$, as described in the following code:

Main Part of Algorithm

Repeat until $i_{right} = n$

{ i_{right} Move }
$i_{right} \leftarrow i_{right} + 1$
$NAME[S_{i_{right}}] \leftarrow NAME[S_{i_{right}}] + 1$

{ i_{left} Move }
$NAME[S_{i_{left}}] \leftarrow NAME[S_{i_{left}}] - 1$
$i_{left} \leftarrow i_{left} + 1$

Compute the name of $NAME$

end Main Part of Algorithm

Observation: Substrings of S, of length ℓ, that are permutations of the same string are represented by the same $NAME$.

We have explained how the $NAME$s of all substrings of length ℓ of S are computed. However, we still have to find the $NAME$s that appear more than K times.

Each distinct $NAME$ is given a unique name - an integer in the range $0 \ldots n$. The names are given by using the *naming* technique [AIL$^+$88, KLP96], which is a modified version of the algorithm of Karp, Miller and Rosenberg [KMR72].

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, $NAME$ 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 2 subarrays of size 2^{i-1}. The names of these 2 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 $NAME_1 NAME_2$ of size 2^j, where $NAME_1$ and $NAME_2$ are consecutive subarrays of size 2^{j-1} each. We give as names the natural numbers in increasing order. Notice that every level only uses the names of the level below it, thus the names we use at every level are numbers from the set $\{1, ..., n\}$.

To give an array a name, we need only to know if 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* $cbo\ell jikgik\ell j$ *of* S, *the array* $NAME$ *that represents this substring is:*

0	1	1	0	0	0	1	0	2	2	2	2	0	0	1	0

Below is the result of naming the above $NAME$.

<table>
<tr><td colspan="16">11</td></tr>
<tr><td colspan="8">9</td><td colspan="8">10</td></tr>
<tr><td colspan="4">6</td><td colspan="4">7</td><td colspan="4">8</td><td colspan="4">7</td></tr>
<tr><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">3</td><td colspan="2">5</td><td colspan="2">5</td><td colspan="2">4</td><td colspan="2">3</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>2</td><td>2</td><td>2</td><td>2</td><td>0</td><td>0</td><td>1</td><td>0</td></tr>
</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>
<tr><td colspan="16">14</td></tr>
<tr><td colspan="8">9</td><td colspan="8">13</td></tr>
<tr><td colspan="4">6</td><td colspan="4">7</td><td colspan="4">8</td><td colspan="4">6</td></tr>
<tr><td colspan="2">2</td><td colspan="2">3</td><td colspan="2">4</td><td colspan="2">3</td><td colspan="2">5</td><td colspan="2">5</td><td colspan="2">2</td><td colspan="2">3</td></tr>
<tr><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>2</td><td>2</td><td>2</td><td>2</td><td>0</td><td>1</td><td>1</td><td>0</td></tr>
</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 ℓ.

Lemma 3 *[AALS03] The maximum number of different names generated by our algorithm's naming of size ℓ 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 $\{(a_j, b_j)\}_{j=1}^{\infty}$, where $a_j, b_j \in \{1, ..., j\}$.
OUTPUT: Dynamically decide, for every query (a_j, b_j), 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 suggest "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:

- $BAL[a]$ 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, $BAL[a]$ 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 $BAL[a]$:
  - $counter_a$ - counts the number of substrings named a.
  - $location\_list_a$ - 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

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

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

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 $O(\log |BAL[a]|) = O(\log(n))$.

4.3 Time Complexity

Stage 1 of our algorithm runs L times. In a step ℓ we first initialize $NAME$ and the naming tree in $O(\ell + |\Sigma|)$ time and then compute $n - \ell$ iterations. Each iteration includes at most two changes in $NAME$, 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 set of *E Coli* sequences: in this pruned set, no pair of sequences is ninety percent 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 data set. The following parameters were used to obtain the substring patterns. (1) quorum: the patterns appear at least five times, (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 πpattern 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 πpatterns on this data where the alphabet is the motif/domain with parameters as described above. Figure 1 shows an example of a permuted πpattern of size 6.

Size of πPatterns	**Total number of πPatterns**	**Number of Maximal πPatterns**	**Percentage of Maximal πPatterns**
2	161	98	61%
3	129	53	41%
4	95	55	58%
5	43	17	40%
6	27	19	70%
7	15	11	67%
8	7	7	100%

Table 1: πpatterns on motifs/domains of the *E Coli* protein sequences.

6 Conclusions

Related genes often appear in each others 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 πpattern problem and give 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 are greatly benefited from earlier discussions with Jens Stoye.

References

[AALS03] A. Amir, A. Apostolico, G. M. Landau, and G. Satta. Efficient text fingerprinting via Parikh mapping. *Journal of Discrete Algorithms*, 2003. to appear.

[AIL+88] A. Apostolico, C. Iliopoulos, G. M. Landau, B. Schieber, and U. Vishkin. Parallel construction of a suffix tree with applications. *Algorithmica*, 3:347–365, 1988.

[BCL+01] Brown, Clark, Leader, Simpson, and Lowe. *RNA*, 7:1817–1832, 2001.

[DSHB98] T Dandekar, B Snel, M Huynen, and P Bork. *Trends Biochem. Sci.*, 23:324–328, 1998.

[D03] G. Didier. Common intervals of two sequences. In *Proc. Third International Workshop on Algorithms in Bioinformatics (WABI)*, Lecture Notes in Computer Science **2812**, Springer-Verlag, 17-24, 2003.

[ELP03] R. Eres, G.M. Landau, and L. Parida. A Combinatorial Approach to Automatic Discovery of Cluster-Patterns. In *Proc. Third International Workshop on Algorithms in Bioinformatics (WABI)*, Lecture Notes in Bioinformatics **2812**, Springer-Verlag, 139-150, 2003.

[GBM+01] S Giglio, K W Broman, N Matsumoto, V Calvari, G Gimelli, T Neuman, H Obashi, L Voullaire, D Larizza, R Giorda, J L Weber, D H Ledbetter, and O Zuffardi. Olfactory receptor-gene clusters, genomic-inversion polymorphisms, and common chromosome rearrangements. *Am. J. Hum. Genet.*, 68(4):874–883, 2001.

[HS01] Steffen Heber and Jens Stoye. Finding all common intervals of k permutations. In *Proc. of the Twelfth Symp. on Comp. Pattern Matching*, volume 2089 of *Lecture Notes in Computer Science*, pages 207–218. Springer-Verlag, 2001.

[KK00] D. Kihara and M. Kanehisa. *Genome Res*, 10:731–743, 2000.

[KLP96] Z. M. Kedem, G. M. Landau, and K. V. Palem. Parallel suffix-prefix matching algorithm and application. *SIAM Journal of Computing*, 25(5):998–1023, 1996.

[KMR72] R. Karp, R. Miller, and A. Rosenberg. Rapid identification of repeated patterns in strings, arrays and trees. In *Symposium on Theory of Computing*, volume 4, pages 125–136, 1972.

[LR96] J. G. Lawrence and J. R. Roth. *Genetics*, 143:1843–1860, 1996.

[NGK01] Akhiro Nakaya, Susumo Goto, and Minoru Kanehisa. Extraction of correlated gene clusters by multiple graph comparison. *Genome Informatics*, No 12:44–53, 2001.

[OFD+99] R Overbeek, M Fonstein, M Dsouza, G D Pusch, and N Maltsev. The use of gene clusters to infer functional coupling. *Proc. Natl. Acad. Sci. USA*, 96(6):2896–2901, 1999.

[OFG00] H. Ogata, W. Fujibuchi, and S. Goto. *Nucleic Acids Res*, 28:4021–4028, 2000.

[Par00] Laxmi Parida. Some results on flexible-pattern matching. In *Proc. of the Eleventh Symp. on Comp. Pattern Matching*, volume 1848 of *Lecture Notes in Computer Science*, pages 33–45. Springer-Verlag, 2000.

[PNR+99] E M Marcott M Pellegrini, H L Ng, D W Rice, T O Yeates, and D Eisenberg. Detecting protein function and protein-protein interactions. *Science*, 285:751–753, 1999.

[SS03] T. Schmidt and J. Stoye. Quadratic time algorithms for finding common intervals in two and more sequences.

[SLBH00] B. Snel, G Lehmann, P Bork, and M A Huynen. A web-server to retrieve and display repeatedly occurring neighborhood of a gene. *Nucleic Acids Research*, 28(18):3443–3444, 2000.

[SMA+97] J L Siefert, K A Martin, F Abdi, W R Widger, and G E Fox. *J. Mol. Evol.*, 45:467–472, 1997.

[TCOV97] J Tamames, G Casari, C Ouzounis, and A Valencia. *J. Mol. Evol.*, 44:66–73, 1997.

[TK98] K. Tomii and M. Kanehisa. *Genome Res*, 8:1048–1059, 1998.

[WMIG97] H Watanbe, H Mori, T Itoh, and T Gojobori. *J. Mol. Evol.*, 44:S57–S64, 1997.

Example of a permuted maximal πpattern 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..VGESGSGKS.T*
36 *VGESGSGKS.T*
72 **IADEPTT.LDV**
81 **PHQLSGG..QRV**
136 *LSGG.RQRV.IA*
159 *LVG.SGSGKS.T*

line 325

VLAVENLNIAFMQDQQKIAAVRNLSFSLQR*GETLAIVGESGSGKSVT*ALALMRLLEQAGGLV
QCDKMLLQRRSREVIELSEQNAAQMRHVRGADMAMIFQEPMTSLNPVFTVGEQIAESIRLHQ
NASREEAMVEAKRMLDQVRIPEAQTILSRY**PHQLSGGMRQRV***MIA*MALSCRPAVL**IADE**
PTTALDVTIQAQILQLIKVLQKEMSMGVIFITHDMGVVAEIADRVLVMYQGEAVETGTVEQ
IFHAPQHPYTRALLAAVPQLGAMKGLDYPRRFPLISLEHPAKQAPPIEQKTVVDGEPVLRVR
NLVTRFPLRSGLLNRVTREVHAVEKVSFDLWP*GETLSLVGESGSGKSTT*GRALLRLVESQGG
EIIFNGQRIDTLSPGKLQALRRDIQFIFQDPYASLDPRQTIGDSIIEPLRVHGLLPGKDAAARVAW
LLERVGLLPEHAWRYPHEFSGGQRQRICIARALALNPKVIIADEAVSALDVSIRGQIINLLLDLQR
DFGIAYLFISHDMAVVERISHRVAVMYLGQIVEIGPRRAVFENPQHPYTRKLLAAVPVAEPSRQR
PQRVLLSDDLPSNIHLRGEEVAAVSLQCVGPGHYVAQPQSEYAFMRR

line 498

MTQTLLAIENLSVGFRHQQTVRTVVNDVSLQIEA*GETLALVGESGSGKSVT*ALSILRLLPSPP
VEYLSGDIRFHGESLLHASDQTLRGVRGNKIAMIFQEPMVSLNPLHTLEKQLYEVLSLHRGMRR
EAARGEILNCLDRVGIRQAAKRLTDY**PHQLSGGERQRV***MIA*MALLTRPELL**IADEPTTAL**
DVSVQAQILQLLRELQGELNMGMLFITHNLSIVRKLAHRVAVMQNGRCVEQNYAATLFASPTH
PYTQKLLNSEPSGDPVPLPEPASTLLDVEQLQVAFPIRKGILKRIVDHNVVVKNISFTLRA*GETL*
*GLVGESGSGKSTT*GLALLRLINSQGSIIFDGQPLQNLNRRQLLPIRHRIQVVFQDPNSSLNPRLN
VLQIIEEGLRVHQPTLSAAQREQQVIAVMHEVGLDPETRHRYPAEFSGGQRQRIAIARALILKPSL
IILDEPTSSLDKTVQAQILTLLKSLQQKHQLAYLFISHDLHVVRALCHQVIILRQGEVVEQGPCAR
VFATPQQEYTRQLLALS

Figure 1: An example to show how a pair of domains (motifs) 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 (ie. 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 on the original sequences at the bottom. Using the maximality notation the pattern is represented as: (36-81-136-72), 35, 159.