Available Online at www.ijcsmc.com

International Journal of Computer Science and Mobile Computing

A Monthly Journal of Computer Science and Information Technology

ISSN 2320–088X

IJCSMC, Vol. 3, Issue. 4, April 2014, pg.1019 – 1034

RESEARCH ARTICLE

Finding Correlated CCC-Biclusters from Gene Expression Data

Geethamani.S[1], Vijayalakshmi.S[2]

[1]Assistant Professor, Sri Ramakrishna College of Arts & Science for Women, Coimbatore, India
[2]Assistant Professor, Sri Ramakrishna College of Arts & Science for Women, Coimbatore, India
[1] geethura@yahoo.co.in
[2] dvijayashruthi@yahoo.com

ABSTRACT: Several non-supervised machine learning methods have been used in the analysis of gene expression data obtained from microarray experiments. Recently, biclustering, a non-supervised approach that performs simultaneous clustering on the row and column dimensions of the data matrix, has been shown to be remarkably effective in a variety of applications. The goal of biclustering is to find subgroups of genes and subgroups of experimental conditions, where the genes exhibit highly correlated behaviors. These correlated behaviors correspond to coherent expression patterns and can be used to identify potential regulatory modules possibly involved in regulatory mechanisms. Many specific versions of the biclustering problem have been shown to be (Non-deterministic polynomial) NP-complete. However, identifying biclusters in time series expression data, it can restrict the problem by finding only maximal biclusters with contiguous columns. This restriction leads to a tractable problem. The motivation of the biological processes start and finish in an identifiable contiguous period of time, leading to increased (or decreased) activity of sets of genes forming biclusters with contiguous columns. In this context, an algorithm that find and reports all maximal contiguous column coherent biclusters. (CCC-Biclusters), in time linear in the size of the expression matrix. Each relevant CCC-Bicluster identified corresponds to the discovery of a coherent expression pattern shared by a group of genes in a contiguous subset of time-points and identifies a potentially relevant regulatory module. The linear time complexity of CCC-Biclustering is obtained by manipulating a discretized version of the gene expression matrix and using efficient string processing techniques based on suffix trees. The results of the proposed algorithm in synthetic and real data that show the effectiveness of the approach and the relevance of CCC-Biclustering in the discovery of regulatory modules. These results were obtained by applying the algorithm to the transcriptomic expression patterns occurring in Saccharomyces cerevisiae in response to heat stress. The results show not only the ability of the proposed methodology to extract relevant information compatible with documented biological knowledge, but also the utility of using this algorithm in the study of other environmental stresses, and of regulatory modules, in general.

1. INTRODUCTION

1.1 DATA MINING

Data Mining is the process of sorting through large amount of data and picking out relevant information. Data mining extract novel and useful knowledge from large repositories of data and has become an effective analysis and decision means in corporation. Data mining is an important tool to transform data into information and it also used to find the hidden information in a database. Data mining techniques have been widely used in various applications such as marketing, business, scientific discovery, pattern mining, medical diagnosis and spatial data mining. Data mining consists of two main tasks predictive and descriptive.

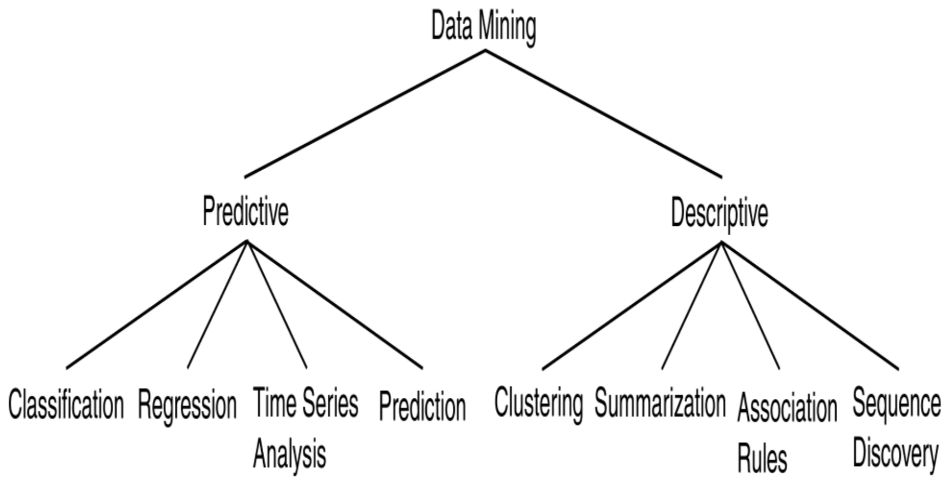


FIGURE 1.1 DATA MINING TYPES

1.2 DNA MICROARRAY

Compared with the traditional approach to genomic research, which has focused on the local examination and collection of data on single genes, microarray technologies have now made it possible to monitor the expression levels for tens of thousands of genes in parallel. The two major types of microarray experiments are the cDNA microarray and oligonucleotide arrays (abbreviated oligochip). Despite differences in the details of their experiment protocols, both types of experiments involve three common basic procedures.

A microarray is a small chip (made of chemically coated glass, nylon membrane or silicon), onto which tens of thousands of DNA molecules (probes) are attached in fixed grids. Each grid cell relates to a DNA sequence.

Typically, two mRNA samples (a test sample and a control sample) are reverse-transcribed into DNA (targets), labeled using either fluorescent dyes or radioactive is topics, and then hybridized with the probes on the surface of the chip.

Chips are scanned to read the signal intensity that is emitted from the labeled and hybridized targets. Clustering is the most popular approach of analyzing gene expression data and has proven successful in many applications, such as discovering gene pathway, gene classification, and function prediction. There is a very large body of literature on clustering in general and on applying clustering techniques to gene expression data in particular. Several representative algorithmic techniques have been developed and experimented in clustering gene expression data, which include but are not limited to hierarchical clustering, self-organizing maps, and graphic theoretic approaches (e.g., CLICK).

1.3 PRE-PROCESSING OF GENE EXPRESSION DATA

A microarray experiment typically assesses a large number of DNA sequences (genes, DNA clones, or expressed sequence tags [ESTs]) under multiple conditions. These conditions may be a time series during a biological process (e.g., the yeast cell cycle) or a collection of different tissue samples (e.g., normal versus cancerous tissues). In this research the cluster analysis of gene expression data without making a distinction among DNA sequences, which will uniformly be called "genes". Similarly, uniformly refer to all kinds of experimental conditions as "samples" if no confusion will be caused.

The original gene expression matrix obtained from a scanning process contains noise, missing values, and systematic variations arising from the experimental procedure. Data pre-processing is indispensable before any cluster analysis can be performed. Some problems of data pre-processing have themselves become interesting research topics. Those questions are beyond the scope of this survey; an examination of the problem of missing value estimation, and the problem of data normalization.

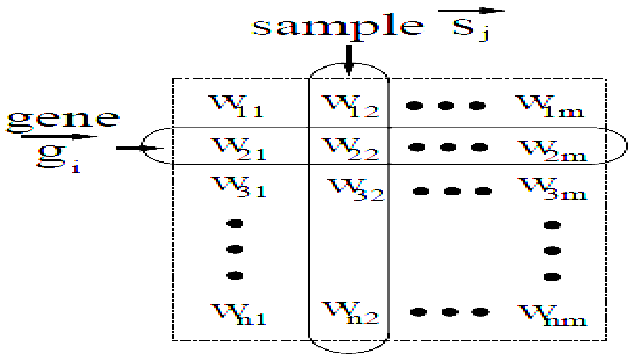

sample $\overrightarrow{s_j}$
gene $\overrightarrow{g_i}$
w_{11} w_{12} ••• w_{1m}
w_{21} w_{22} ••• w_{2m}
w_{31} w_{32} ••• w_{3m}
w_{n1} w_{n2} ••• w_{nm}

Figure 1.2: A gene expression matrix

One or more of the following pre-processing procedures: filtering out genes with expression levels which do not change significantly across samples; performing a logarithmic transformation of each expression level; or standardizing each row of the gene expression matrix with a mean of zero and a variance of one. In the following discussion of clustering algorithms, the details of pre-processing procedures and assume that the input data set has already been properly pre-processed.

1.4APPLICATIONS OF CLUSTERING GENE EXPRESSION DATA

Clustering techniques have proven to be helpful to understand gene function, gene regulation, cellular processes, and subtypes of cells. Genes with similar expression patterns (co-expressed genes) can be clustered together with similar cellular functions. This approach may further understanding of the functions of many genes. Furthermore, co-expressed genes in the same cluster are likely to be involved in the same cellular processes, and a strong correlation of expression patterns between those genes indicates co-regulation. Searching for common DNA sequences at the promoter regions of genes within the same cluster allows regulatory motifs specific to each gene cluster to be identified and co-regulatory elements. The inference of regulation through the clustering of gene expression data also gives rise to hypotheses regarding the mechanism of the transcriptional regulatory network. Finally, clustering different samples on the basis of corresponding expression profiles may reveal sub-cell types which are hard to identify by traditional morphology-based approaches.

1.4.1 Introduction to Clustering Techniques

First introduce the concepts of clusters and clustering. We will then divide the clustering tasks for gene expression data into three categories according to different clustering purposes. Finally, the issue of proximity measure has been explained in detail.

1.4.1.1 Clusters and Clustering

Clustering is the process of grouping data objects into a set of disjoint classes, called clusters, so that objects within a class have high similarity to each other, while objects in separate classes are more dissimilar. Clustering is an example of unsupervised classification. “Classification” refers to a procedure that assigns data objects to a set of classes. “Unsupervised” means that clustering does not rely on predefined classes and training examples while classifying the data objects. Thus, clustering is distinguished from pattern recognition or the areas of statistics known as discriminate analysis and decision analysis, which seek to find rules for classifying objects from a given set of pre-classified objects.

1.4.1.2 Categories of gene expression data clustering

Currently, a typical microarray experiment contains 10^3 to 10^4 genes, and this number is expected to reach to the order of 10^6. However, the number of samples involved in a microarray experiment is generally less than 100. One of the characteristics of gene expression data is that it is meaningful to cluster both genes and samples. On one hand, co-expressed genes can be grouped in clusters based on their expression patterns. In gene-based clustering, the genes are treated as the objects, while the samples are the features. On the other hand, the samples can be partitioned into homogeneous groups. Each group may correspond to some particular macroscopic phenotype, such as clinical syndromes or cancer types. Such sample-based clustering regards the samples as the objects and the genes as the features. The distinction of gene-based clustering and sample based clustering is based on different characteristics of clustering tasks for gene expression data. Clustering algorithms, such as K-means and hierarchical approaches, can be used both to group genes and to partition samples.

1.4.1.3 Proximity measurement for gene expression data

Proximity measurement measures the similarity (or distance) between two data objects. Gene expression data objects, no matter genes or samples, can be formalized as numerical. The proximity between two objects and is measured by a proximity function of corresponding vectors. However, for gene expression data, the overall shapes of gene expression patterns (or profiles) are of greater interest than the individual magnitudes of each feature. Euclidean distance does not score well for shifting or scaled patterns (or profiles). To address this problem, each object vector is standardized with zero mean and variance one before calculating the distance.

Pearson’s correlation coefficient is widely used and has proven effective as a similarity measure for gene expression data. However, empirical study has shown that it is not robust with respect to outliers, thus potentially yielding false positives which assign a high similarity score to a pair of dissimilar patterns. If two patterns have a common peak or valley at a single feature, the correlation will be dominated by this feature, although the patterns at the remaining features may be completely dissimilar.

1.5 TYPE OF BICLUSTER

Gene expression matrix can be analyzed in two ways. For gene based clustering, genes are treated as data objects, while samples are considered as features. Conversely, for sample-based clustering, samples serve as data objects to be clustered, while genes play the role of features. The third category of cluster analysis applied to gene expression data, which is subspace clustering, treats genes and samples symmetrically such that either genes or samples can be regarded as objects or features. Gene-based, sample-based and subspace clustering face very different challenges, and different computational strategies are adopted for each situation. In this section, we will introduce the gene-based clustering, sample-based clustering, and subspace clustering techniques, respectively.

1.5.1 Gene-based Clustering

The problem of clustering genes based on their expression patterns. The purpose of gene-based clustering is to group together co-expressed genes which indicate function and co-regulation. The challenges of gene-based clustering and then review a series of clustering algorithms which have been applied to group genes.

CCC BICLUSTER

CCC-Biclustering uses a generalized suffix tree to identify, in time linear on the size of the expression matrix, all maximal biclusters with contiguous columns that exhibit coherent expression evolutions over time. In a CCC-Bicluster, all genes have exactly the same discretized expression pattern. CCC-Biclustering depends strongly on the suffix tree construction and on the data structures used, we will revise here the Ukkonen´s algorithm for suffix tree construction at a higher level and give the intuition behind its linear time construction.

Let A0 be an NG rows by NT columns gene expression matrix defined by its set of rows (genes), G, and its set of columns (time-points), T. In this context, A0 ij represents the expression level of gene i in time-point j. We address the case where gene expression levels in matrix A0 can be discretized to a set of symbols that represent distinctive activation levels. After the discretization process, matrix A0 is transformed into matrix A, where Aij 2 represents the discretized value of the expression level of gene i in time-point j. In Figure 2 a three symbol alphabet = fD;N, Ug was used, where ‘D’ (down), ‘N’ (no-change), and ‘U’ (up) mean that the expression levels decreased, didn’t change or increased between consecutive time points Tj and Tj+1. When Tj , Tj+1 or both are missing, the symbol ‘-’ is used.

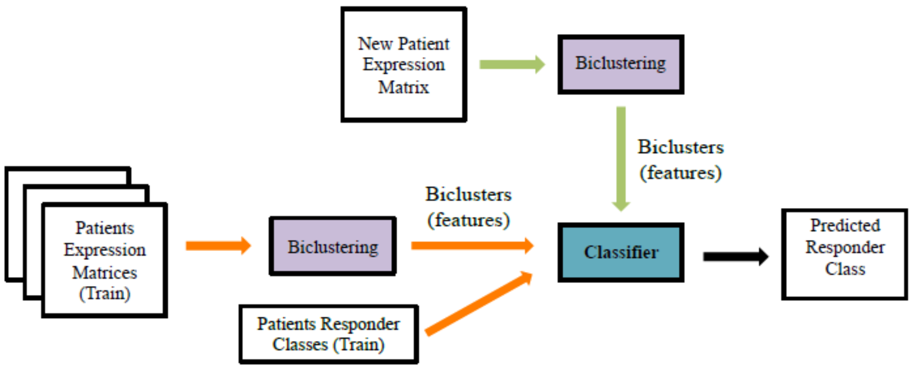

New Patient Expression Matrix
Biclustering
Biclusters (features)
Patients Expression Matrices (Train)
Biclustering
Biclusters (features)
Classifier
Predicted Responder Class
Patients Responder Classes (Train)

Figure 3.1: Basic workflow of a biclustering-based classifier.

Consider now the matrix obtained by preprocessing matrix A using a simple alphabet transformation, that appends the column number to each symbol in the matrix and the generalized suffix tree built for the set of strings corresponding to each row in A. CCC-Biclustering is a linear time biclustering algorithm that finds and reports all maximal CCC-Biclusters based on their relationship with the nodes in the generalized suffix tree.

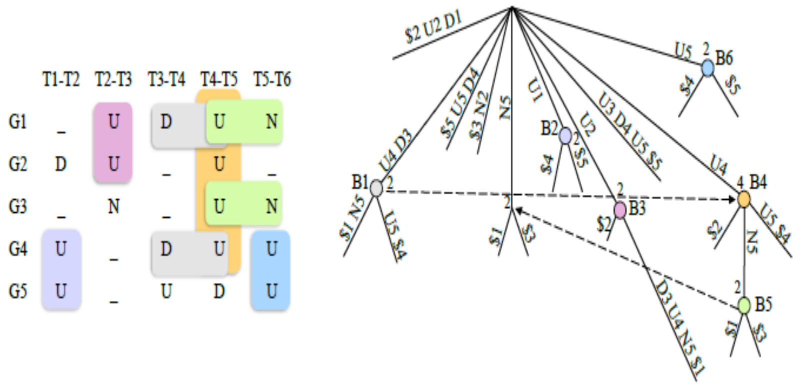


Figure 3.2: Maximal CCC-Biclusters in the discretized matrix and related nodes in the suffix tree.

1.5.2 Ukkonen's Algorithm

A contiguous column coherent bicluster (ccc-bicluster), AIJ =(I, J), is a subset of rows I = {i1, . . . , ik} and a contiguous subset of columns J = {r, r+1, . . . , s−1, s} from matrix A such that Aij = Alj , ∀i, l ∈ I and j ∈ J.

In this setting, each ccc-bicluster defines a string *S* that is common to every row in the ccc-bicluster, between columns *r* and *s* of matrix *A*. Figure 2(b) illustrates two ccc-biclusters that appear in the expression matrix. These two ccc-biclusters are maximal, in the sense that they are not properly contained in any other ccc-biclusters. Ukkonen's algorithm to construct suffix trees uses the concepts of implicit suffix tree and suffix link to achieve a linear time construction.

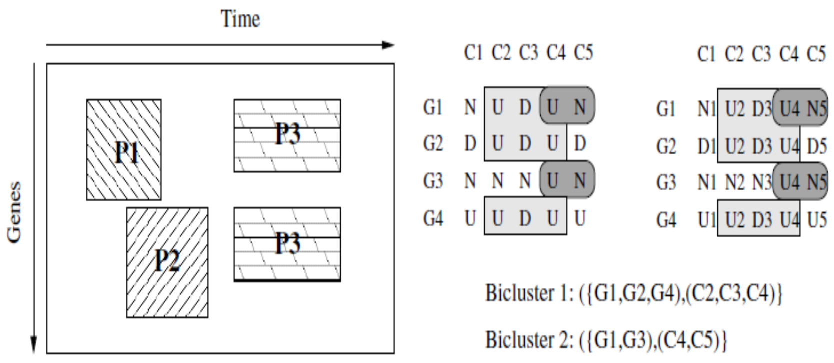

Time
Genes
P1
P2
P3
P3
C1 C2 C3 C4 C5
G1
G2
G3
G4
Bicluster 1: ({G1,G2,G4),(C2,C3,C4)}
Bicluster 2: ({G1,G3),(C4,C5)}

Figure. 3.3. Biclusters in time series gene expression data

An implicit suffix tree for a string *S* is a tree obtained from the suffix tree *T* constructed for the string *S*$ by removing every copy of the symbol $ from de edge labels of the tree, then removing any node *v* that does not have at least two children. In particular, an implicit suffix tree for a prefix *S*[1..*i*] of string *S* is similarly defined by taking the suffix tree for *S*[1...*i*]$ and deleting $ symbols, edges and nodes as above. The implicit suffix tree of the string *S*[1..*i*] is denoted by *Ti*, $1 \geq i \geq |S|$. Let $x\alpha$ denote an arbitrary string, where *x* denotes a single character and α denotes a (possibly empty) substring. For any internal node *v* with string-label $x\alpha$, if there is another node *s*(*v*) with string-label ®, then a pointer from *v* to *s*(*v*) is called a suffix link. The pair (*v; s*(*v*)) will denote the suffix link from *v* to *s*(*v*). As a special case, if α is empty, $x\alpha$ has a suffix link leading to the root, (*v; root*).

Ukkonen's algorithm starts by constructing an implicit suffix tree *Ti* for each prefix of *S*[1...*i*] of a string *S*, starting from *T*1 and incrementing *i* by one until *TjSj* is built, where *jSj* is the number of characters in *S*. The true suffix tree from *S* is then constructed from *TjSj*

The algorithm follows *three suffix extension rules* in order to make sure that the suffix *S*[*j*...*i*+ 1] is in the tree after the extension of *S*[*j*..*i*] with character *S*[*i* + 1]:

Rule 1: If path *S*[*j*...*i*] ends at a leaf, concatenate *S*[*i* + 1] to the end of its edge label;

Rule 2: If path *S*[*j*...*i*] ends before a leaf, and doesn't continue by *S*[*i* + 1], connect the end of the path to a new leaf *j* by an edge labeled by character *S*[*i* + 1]. If the path ended at the middle of an edge, split the edge and insert a new internal node as a parent of leaf *j*;

Rule 3: If the path can be continued by *S*[*i*+1] do nothing (the suffix *S*[*j*...*i*+1] is already in the tree).

Given these suffix extension rules, once the end of a suffix *S*[*j*...*i*] of *S*[1...*i*] is in the current tree, the execution of the extension rules in order to ensure that suffix *S*[1...*i* + 1] is in the tree can be performed in constant time. The key issue in implementing Ukkonen's algorithm is then how to locate the end of all the *i* + 1 suffixes of *S*[1....*i*]. This is achieved by using suffix links and three implementation tricks.

Let's start by an intuitive motivation about how suffix links can be used to speed up path traversals. Extension *j* (of phase *i*+1) finds the end of the path *S*[*j*...*i*] in the tree (and extends it with character *S*[*i* + 1]. Similarly, extension *j* + 1 finds the end of the path *S*[*j* + 1...*i*]. Assuming that *v* is an internal node with string-label *S*[*j*] on the path *S*[*j*...*i*], then we can avoid traversing the path when locating the end of path *S*[*j* + 1...*i*], by starting the traverse from the suffix link of *v*, *s*(*v*). This can be done because these suffix links always exist and are in fact easy to set. The certainty about the existence of the suffix link (*v; s*(*v*)) comes from the observation that if an internal node *v* is created during extension *j* (of phase *i* + 1), then extension *j* + 1 will find out the node *s*(*v*). Why is this true? Let *v* be a node with string-label *x*. This node can only be created by extension Rule 2, that is, *v* can only be inserted at the end of path *S*[*j*...*i*], which continued by some character *c 6*= *S*[*i* + 1]. In this context, the paths *x c* and *c* have been inserted in the tree before phase *i*+1. This means that in extension *j*+1, node *s*(*v*) is either found since it is already in the tree or created at the end of path = *S*[*j* + 1...*i*].

Consider now the extensions of phase *i* + 1. Extension 1 extends path *S*[1...*i*] with character *S*[*i*+1]. This can be done easily since the path *S*[1...*i*] always ends at leaf 1, and is thus extended by Rule 1. As such, extension 1 can be performed in constant time, if we maintain a pointer to the edge at the end of *S*[1...*i*]. The subsequent extensions *j* +1, *j* = 1... *i*, Since extension *j* has located the end of the path *S*[*j*...*i*] can start from there, and walk up at most one node either to the root, or to a node *v* that has a suffix link *s*(*v*) from it.

In case to traverse the path *S*[*j* +1...*i*] explicitly downwards starting from the root. In case the suffix link to node *s*(*v*) let *x* be the edge-label of *v*. This means *S*[*j*...*i*] = *x* for some 2 § . In this scenario, to follow the suffix link of *v*, and continue by matching *downwards* from node *s*(*v*) (which is now labeled. Having found the end of path = *S*[*j* +1...*i*], we apply the extension rules to ensure that it is extended with the character *S*[*i* + 1]. Finally, if a new internal node *u* was created in extension *j*, we set its suffix link to point to the end node of path *S*[*j* + 1...*i*]. However, speed up these explicit traversals by using the implementation trick 1 *(*skip/count in Gusfield [1]): each path *S*[*j*....*i*], which is followed in extension *j*, is known to exist in the tree, as such, the path can be followed by choosing the correct edges, instead of examining each character. Let *S*[*k*] be the next character to be matched on path *S*[*j*....*i*]. Now an edge labeled by *S*[*p*....*q*] can be traversed simply by checking that *S*[*p*] = *S*[*k*], and skipping the next *q ¡p* character of S[j...i]. This improves the time to traverse a path from time proportional to its string-depth to time proportional to its node-depth.

The suffix links and the first trick the total time of a phase is now *O*(*jSj*). However, there are *jSj* phases and the total time bound is still *O*(*jSj*2). In order to achieve the desired linear time bound we just need a simple implementation detail and two more implementation tricks. The implementation detail concerns edge-label compression. In fact, with the current implementation, the edge-labels in the suffix tree might contain *O*(*jSj*) characters which makes the space required for the suffix tree to be *O*(*jSj*2). As the time of the algorithm is at least as large as the size of its input, that many characters makes an *O*(*jSj*) time bound impossible [1]. However there is a simple alternative scheme for edge-labeling: instead of explicitly write a substring *S*[*p…q*] as edge-label, we can write only a pair of indices on the edge, (*p; q*), specifying the start and end positions of that substring in *S*. Since the algorithm has a copy of *S*, it can locate any particular character in *S* in constant time given its position in the string.

For example, when matching along an edge, the algorithm uses the index pair written on the edge to retrieve the needed characters from *S* and then performs the comparisons on those characters. The extension rules are also easily implemented in this labeling scheme: when the extension rule 2 applies in a phase *i* + 1, we just have to label the newly created edge with the index pair (*i* + 1*; i* + 1), and when rule 1 applies (on a leaf edge), we only need to change the index pair on that leaf edge from (*p; q*) to (*p; q* + 1). Since the number of edges is at most 2*jSj* ¡ 1, the suffix tree uses only *O*(*jSj*) symbols and *O*(*jSj*) space.

The implementation trick 2 ("show stopper" in Gusfield [1]) is based on the observation that some extensions can be found unnecessary to compute explicitly. In fact, Rule 3 is a "show stopper" since if the path *S*[*j…i* + 1] is already in the tree, so are the paths *S*[*j* + 1*…i* + 1]…. *S*[*i*+1]. As such, phase *i*+1 can be finished at the first extension *j* that applies Rule 3. The implementation trick *3 ("*once a leaf, always a leaf" in Gusfield [1]) is based on the observation that a node created as a leaf remains a leaf thereafter because no extension rule adds children to a leaf. In fact, if extension *j* created a leaf (numbered *j*), the extension *j* of any later phase *i* + 1 applies Rule 1 (concatenating the next character *S*[*i* + 1] to end of the edge-label of *j*). As such, explicit applications of Rule 1 can be eliminated as follows: using edge-label compression described above and representing the end position of each terminal edge by a global value *e* standing for "the current end position" means that in phase *i*+1, when a leaf edge is first created ad would normally be labeled with *S*[*p….i*+1] instead of writing the pair of indexes (*p; i* + 1), as explained above, we write (*p; e*). The symbol *e* is a global index and is set to *i* + 1 once in each phase. In phase *i* + 1, since we know that rule 1 will apply in extensions 1 through *ji* at least, we do not need explicit work to implement those *ji* extensions. Instead, we only need constant time to increment the value of *e* and then do explicit work for (some) extensions starting with *ji* + 1.

Using suffix links, edge-compression and tricks 1, 2 and 3, Ukkonen'a algorithm builds the implicit suffix trees *T*1 through *TjSj* in *O*(*jSj*) total time. Moreover, in order to create the true suffix tree *T* we just have to use the final implicit suffix tree *SjSj*. We first add a string terminal symbol $ to the end of *S* and let Ukkonen's algorithm continue with this symbol. Next, we just have to replace every index *e* on every leaf node with the value *jSj*. This is achieved by a simple traversal of the tree, visiting each leaf each, which takes *O*(*jSj*) [1]. In order to construct the generalized suffix tree for a set of string *fS*1*… SjRjg* all with the same length *jCj* and sharing the same alphabet § *0* , we first build the implicit suffix trees from *TS*1 1 through *S*1*TS jCj* 1 using Ukkonen's algorithm and then insert the strings *Si* on the tree starting with the last implicit suffix tree of *Si*¡1: *TS jCj i*¡1 , 2 · *i* · *jRj*. Finally we transform the last implicit suffix tree *TS jCj jRj* in a true suffix tree by adding a string terminal symbol $*i* to the end of *Si* and let Ukkonen's algorithm continue with these symbols one at the time. Assuming that the nodes in the implicit suffix tree organize their children by lexicographic order of the first character of their edge-labels, this is done in a way such that $1 ….. $*jRj* and every string terminator $*i* is lexicographically smaller then any character in §*0*. This enables the insertion of leaves corresponding to terminators in the root node (and other nodes) always in the first position of the data structures storing the children of each node. This is done in *O*(*jRjjCj*), since there are *jRj* strings of length *jCj* and inserting each string in the suffix tree using Ukkonen's algorithm takes *O*(*jRjjCj*) as we have seen above.

1.5.3 Suffix Trees

A string *S* is an ordered list of characters written contiguously from left to right. For any string *S*, *S*[*i..j*] is the (contiguous) sub string of *S* that starts at position *i* and ends at position *j*. In particular, *S*[1*..i*] is the prefix of *S* that ends at position *i* and *S*[*i..*|*S*|] is the suffix of *S* that starts at position *i*, where |*S*| is the number of characters in *S*. A suffix tree is a data structure built over all the suffixes of a string *S* that exposes its internal structure. This data structure has been extensively used to solve a large number of string processing problems.

Three types of nodes in the construction of the generalized suffix tree *T*: the root, internal nodes and leaf nodes. The root stores an array called children with *jCjj*§*j* + *jRj* positions where each position is a pointer to the first of its "potential" children, which can either be internal nodes, leaf nodes or a null pointer. The array is sorted in lexicographic order of the first character of the edge-label in the nodes. The first *R* positions store the *jRj* string terminators, the next positions store the nodes whose first character in the edge-label starts with §*0*[1].... *0*[*jCj*]*...0*[*jCj*§]]. In this setting, children [*j* + *jRj*] is null if there is not a suffix of any of the strings *Si* starting with the character in §*0*[*j*], that is, if the potential node whose edge-label starts with the character §*0*[*j*] does not exists.

Each internal node *v* stores a pointer to its first child, a pointer to its right sibling, a pointer to the node we get by following its suffix link (if it exists), the index pair representing its edgelabel (start and end position os the substring), its string-length, *P*(*v*), the number of leafs in its subtree, *L*(*v*) and a flag indicating if it corresponds to a maximal CCC-Bicluster or not. The first child of the node is the first element of a linked-list of nodes (either internal nodes or leaves nodes) corresponding to its children sorted in lexicographic order of the first character of their edge-labels. The right sibling of each internal node *v* is also the first element of a linked-list storing all its siblings (nodes whose parent is the parent of *v*). Leaf nodes store the same information as internal nodes except the pointer to the first child.

In order to enable the construction of a suffix tree obeying this definition when one suffix of *S* matches a prefix of another suffix of *S*, a character terminator, that does not appear nowhere else in the string, is added to its end. For example, the suffix tree for the string *S*=TACTAG is presented in Fig. 3. The suffix tree construction for a set of strings, called a generalized suffix tree, can be easily obtained by consecutively building the suffix tree for each string of the set. The leaf number of the single string suffix tree is now converted to two numbers: one identifying the string and other the starting position (suffix) in that string.

Suffix trees can be built in time that is linear on the size of the string, using several algorithms. Generalized suffix trees can be built in time linear on the sum of the sizes of the strings. Ukkonen's algorithm, used in this work, uses suffix links to achieve a linear time construction.

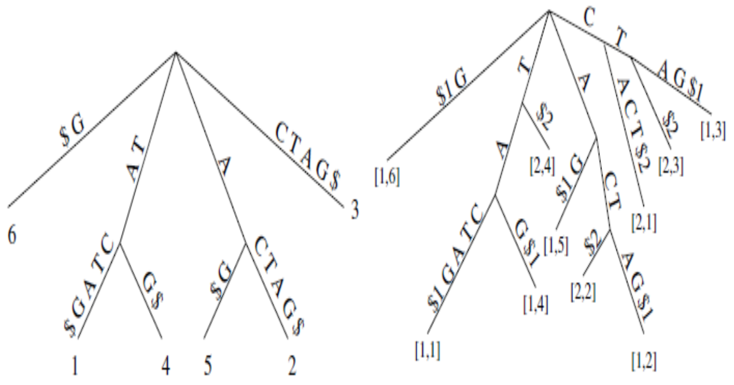


Figure. 6. Example of a suffix tree for the string *S*=TACTAG and a generalized suffix tree for the strings *S*1=TACTAG and *S*2=CACT

Ccc-bicluster right-maximal if it cannot be extended to the right by adding one more column at the end, and left-maximal if it cannot be extended to the left by adding one more column at the beginning. Stated more plainly, a ccc-bicluster is maximal if no more rows nor contiguous columns (either at the right or at the left) can be added to it while maintaining.

A new alphabet $\Sigma = \Sigma\times\{1 \ldots m\}$, where each element $\Sigma\_$ is obtained by concatenating one symbol in Σ and one number in the range $\{1 \ldots m\}$. In order to do this alphabet transformation we use a function $f : \Sigma \times \{1 \ldots m\}$ defined by $f(a, k) = a|k$ where $a|k$ represents the character in $\Sigma\_$ obtained by concatenating the symbol a with the number k. For example, if $\Sigma = \{U,D,N\}$ and $m = 3$, then $\Sigma\_ = \{U1, U2, U3, D1, D2, D3, N1, N2, N3\}$. For this case, $f(U, 2) = U2$ and $f(D, 1) = D1$.

Consider now the set of strings *S = {S*1…….. *Sn}* obtained by mapping each row *AiC* in matrix *A* to string *Si* such that *Si*(*j*) = *f*(*Aij, j*). Each of these strings has *m* characters and corresponds to the symbols in a row of matrix *A* after the above alphabet transformation. After this transformation, the matrix corresponding to the discredited matrix. Let *T* be the generalized suffix tree obtained from the set of strings *S*. Let *v* be a node of *T* and let *P*(*v*) be the path-length of *v*, that is, the number of characters in the string that labels the path from the root to node *v*. Additionally, let *B*(*v*) be the branch-length of *v* and let *L*(*v*) denote the number of leaves in the sub-tree rooted at *v*, in case *v* is an internal node. It is easy to verify that every internal node of the generalized suffix tree *T* corresponds to one ccc- bicluster of the matrix *A*. This is so because an internal node *v* in *T* corresponds to a given substring that is common to every row that has a leaf rooted in *v*. Therefore, node *v* defines a ccc-bicluster that has *P*(*v*) columns and a number of rows equal to *L*(*v*). It is also true that all the leaves except the ones whose path label is simply a terminator also identify ccc-biclusters.

The figure also shows that there are six internal nodes, other than the root. Each one of these nodes corresponds to one ccc-bicluster. Furthermore, each leaf node with branch-length greater than one is also a ccc-bicluster. However, some of these ccc-biclusters are trivial, since they represent biclusters with only one column (nodes with branch-labels *N*1, *U*4 and *N*5, since they have branch-labels with only one character), or are represented by leaves (trivial ccc-biclusters with only one row even when they are maximal). Others are non-maximal (nodes with branch-labels *D*3*U*4 and *N*5), since they have an incoming suffix link from a node with the same number of leaves. As such, only the internal nodes with branch labels *U*2*D*3*U*4 and *U*4*N*5 identify maximal, non-trivial ccc-biclusters. These

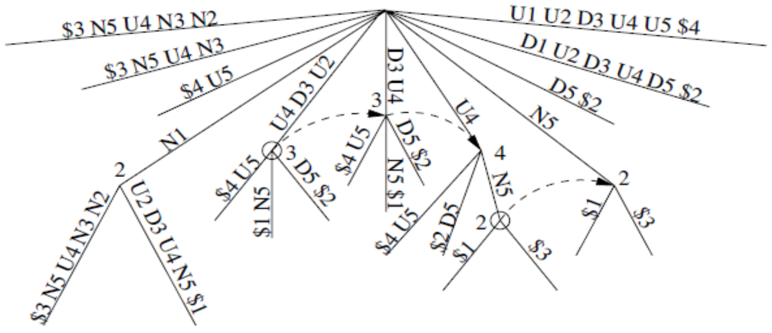


Figure. 3.4. Generalized suffix tree for the matrix

Note that the rows in each ccc-bicluster are obtained from the terminators in the leaves in the subtree of each node *v*, while the columns in each ccc-bicluster are obtained from the value of *P*(*v*) and the information on the branch-label that connects *v* to its parent. In fact, the value of *P*(*v*) and the first character of the path label of *v* is needed to identify the set of columns that belong to the bicluster.

1.5.4 Score Matrix based on Profiles Similarities

Another strategy to compute the score matrix between the test patients and the training set relies on the fact that each CCC-Bicluster is represented by a pattern of symbols, a profile, and representative of the coherent evolution in the expression of the genes in the bicluster along the bicluster time-points. A profile is said to be shared between patients if it identifies a similar expression pattern and represents biclusters which have the required minimum number of genes and/or time-points in common, a parameter determined empirically.

The score matrix between patients is computed, such that an entry (i; j) represents the number of profiles shared between train patient i and test patient j. Instead of the sum of shared profiles between patients, the entry (i; j) of the score matrix can also be computed with a polynomial kernel (a quadratic kernel in general). The idea is to penalize the patients with a larger number of biclusters, since a higher number of profile matches could be due to random events. Filtering Non-Discriminative Bi clusters based on Profiles A given profile is kept in the filtered set, if and only if it contributes more to the discrimination than to the confusion between classes, that is, a profile in a train patient's set of profiles is maintained if and only if it is shared by more patients of the same class than of the other class. A minimum number of shared genes and/or time-points can also be used and fine-tuned.

1.5.5 Score Matrix based on Symbol Pairing with Time-Lags

As one might expect, even when the same genes are involved in a given mechanism in different patients, the expression evolution pattern for one patient might be delayed when compared to others'. As such, the possibility of time-lags in gene expression should be taken into account, as it is a consequence of the patient-specific response rate, and shown to be of particular importance in previous time series expression studies. In this approach, all biclusters (or filtered ones) of the test patient are analyzed and a parameter for a maximum time-lag (number of time-points to consider in the delay) is defined. Then, for each test bicluster, a comparison is made between the discretized symbols, computing the number of perfect matches, considering translations in the time-points, from 0 (the original position) to the maximum time-lag, and its symmetric, thus allowing translations in both directions along the time axis (Figure 4).

The time-lag returning the highest score is chosen, and the binary submatrix resulting from that specific comparison is written in a final matrix. The sum of this matrix represents the score between the two patients, the entry (i; j) of the score matrix for the whole set of patients.

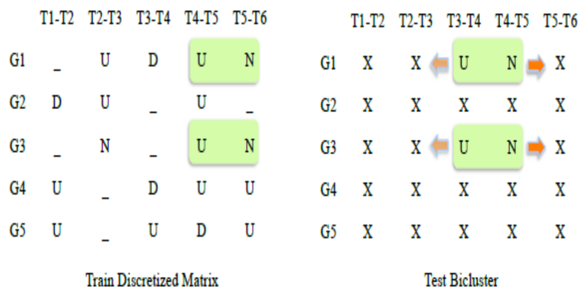


Figure 3.5: Symbolic comparison between a test bicluster

2. RESULTS AND DISCUSSIONS

2.1 EXPERIMENTAL RESULTS

Select the largest number of clusters for which the between-cluster p-values correspond to rejection. To test whether the set of genes discovered by our method shows significant enrichment with respect to a specific GO annotation provided by Gene Ontology Consortium],calculate the probability of obtaining the observed enrichment by chance (p-value).

$$p = 1 - \sum_{i=0}^{z-1} \frac{\binom{K}{i}\binom{M-K}{N-i}}{\genfrac{}{}{0pt}{}{M}{N}}$$

In specific, the probability p that a GO term is significantly enriched in a bi-cluster is calculated as where N is the number of genes in the bi-cluster, M is the total number of genes in the genome, z is the number of genes in the bi-cluster with a given GO term, and K is the total number of genes with a given GO term.

Table I shows the top CCC-Biclusters discovered sorted in ascending order of *p*-value. It is clear from the presented results that the CCC-algorithm coupled with the statistical significance test described in the project is able to identify the CCC-Biclusters planted together with a number of highly overlapping CCC-Biclusters.

TABLE I : TOP CCC-BICLUSTERS RECOVERED WITHOUT FILTERING HIGHLY OVERLAPPING EXPRESSION PATTERNS (AFTER SORTING THE DISCOVERED CCC-BICLUSTERS USING THE STATISTICAL SIGNIFICANCE *p*-VALUE).

ID	Expression Pattern	#Time -Points	#Genes	p-Value	CCC-Bicluster
24475	DDNUDDNDDNDD	12(35-46)	18	4.13E-56	MATCH 7
5790	UDNNDUNUDNDD	12(26-37)	19	1.37E-55	MATCH 8
17868	NUNUUNDDNDNU	12(21-32)	19	3.72E-55	MATCH 1
25020	DNDNDDNNNNDD	12(33-44)	18	6.10E-54	MATCH 10
2438	UDDUNUDDU	9(37-45)	23	8.43E-40	MATCH 6 LOST 1 GENE1
5937	UUDNNDUNUDNDD	13(25-37)	12	2.76E-37	OVERLAP 8
15158	NUNDNNDDUNN	11(30-40)	16	4.08E-37	MATCH 3 LOST 3 GENES1
34531	UNUDUDNDUU	10(2-11)	16	7.17E-34	MATCH 5
16797	NDUNDNUUU	9(25-33)	20	8.20E-33	MATCH 9
23592	DDUNUDDU	8(28-45)	24	1.38E-31	OVERLAP 6

Table II: Comparison of Number of clusters in existing Algorithm

Bi-clustering Algorithms	Number of Clusters
CCC	185
LIST	160
OPSM	17
ISA	36
SAMBA	109
CC	100

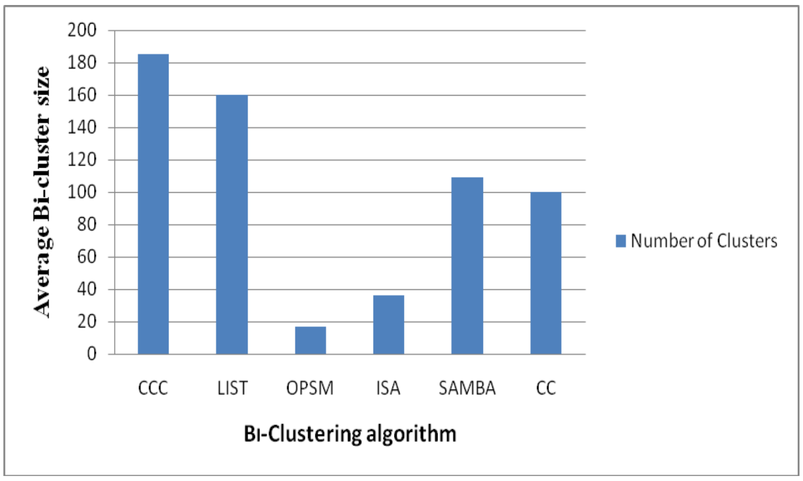


Table III: Comparison of Gene average in existing Algorithm

Bi-clustering Algorithms	Number of clusters	Average Number of Genes
CCC	185	300
LIST	160	209.73
OPSM	17	373.1
ISA	36	82.5
SAMBA	109	65.1
CC	100	78.7

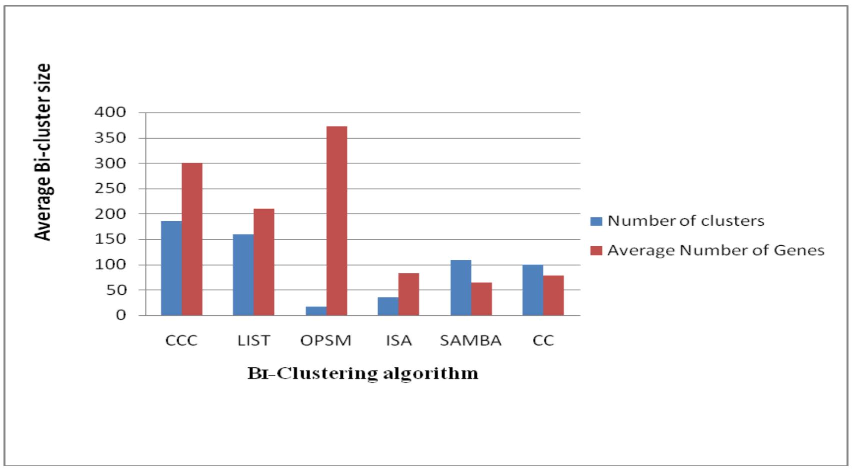


Table IV: Comparison of number conditions in genes

Bi-clustering Algorithms	Number of clusters	Average Number of Genes	Average Number of Conditions
CCC	185	300	7.8
LIST	160	209.73	9
OPSM	17	373.1	9.8
ISA	36	82.5	8.83
SAMBA	109	65.1	17.9
CC	100	78.7	13.3

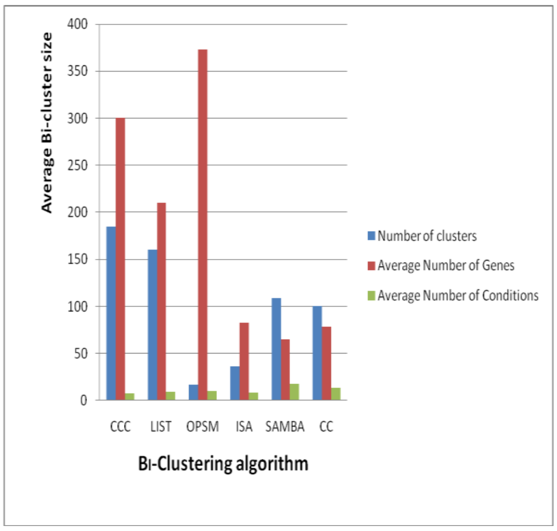
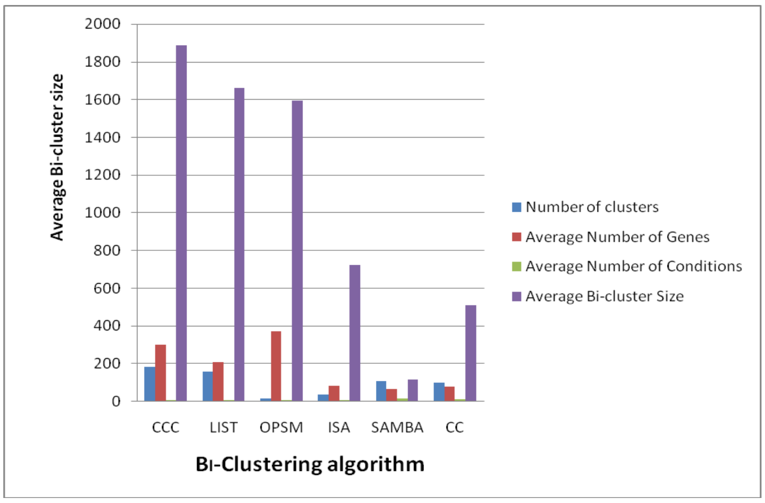


Table V: Comparison of average Bi-cluster Size

Bi-clustering Algorithms	**Number of clusters**	**Average Number of Genes**	**Average Number of Conditions**	**Average Bi-cluster Size**
CCC	185	300	7.8	1885.12
LIST	160	209.73	9	1661.69
OPSM	17	373.1	9.8	1595.4
ISA	36	82.5	8.83	722
SAMBA	109	65.1	17.9	118.63
CC	100	78.7	13.3	509.59



3. CONCLUSION AND FUTURE ENHANCEMENT

Analyzing gene expression time series using biclustering. It was designed to comply with the broad specifications of a software tool, essentially focused on user friendliness, platform independence, modularity, reusability and efficiency. Finds and reports all maximal contiguous column coherent biclusters (CCC-Bi clusters), in time linear in the size of the expression matrix. Each relevant CCC-Bicluster identified corresponds to the discovery of a coherent expression pattern shared by a group of genes in a contiguous subset of time-points and identifies a potentially relevant regulatory module. The linear time complexity of CCC-Biclustering is obtained by manipulating a discretized version of the gene expression matrix and using efficient string processing techniques based on suffix trees.

The effectiveness of the approach and the relevance of CCC-Biclustering in the discovery of regulatory modules. These results were obtained by applying the algorithm to the transcriptomic expression patterns occurring in Saccharomyces cerevisiae in response to heat stress.

Future works will focus on some improvements for the proposed algorithm with regard to the overlapping among genes and to the fitness function.

REFERENCES

[1] R.L.F. Cordeiro, A.J.M. Traina, C. Faloutsos, and C. Traina Jr., “Finding Clusters in Subspaces of Very Large, Multi-Dimensional Data Sets,” Proc. IEEE 26th Int’l Conf. Data Eng. (ICDE), pp. 625- 636, 2010.

[2] C. Traina Jr., A.J.M. Traina, C. Faloutsos, and B. Seeger, “Fast Indexing and Visualization of Metric Data Sets Using Slim-Trees,” IEEE Trans. Knowledge Data Eng., vol. 14, no. 2, pp. 244-260, Mar./ Apr. 2002.

[3] C. Traina Jr., A.J.M. Traina, L. Wu, and C. Faloutsos, “Fast Feature Selection Using Fractal Dimension,” Proc. 15th Brazilian Symp. Databases (SBBD), pp. 158-171, 2000.

[4] H.-P. Kriegel, P. Kro¨ger, and A. Zimek, “Clustering High- Dimensional Data: A Survey on Subspace Clustering, Pattern-Based Clustering, and Correlation Clustering,” ACM Trans. Knowledge Discovery from Data, vol. 3, no. 1, pp. 1-58, 2009.

[5] C. Domeniconi, D. Gunopulos, S. Ma, B. Yan, M. Al-Razgan, and D. Papadopoulos, “Locally Adaptive Metrics for Clustering High Dimensional Data,” Data Mining and Knowledge Discovery, vol. 14, no. 1, pp. 63-97, 2007.

[6] A.K.H. Tung, X. Xu, and B.C. Ooi, “Curler: Finding and Visualizing Nonlinear Correlation Clusters,” Proc. ACM SIGMOD Int’l Conf. Management of Data, pp. 467-478, 2005.

[7] C. Aggarwal and P. Yu, "Redefining Clustering for High- Dimensional Applications," IEEE Trans. Knowledge and Data Eng., vol. 14, no. 2, pp. 210-225, Mar./Apr. 2002.

[8] E.K.K. Ng, A.W. chee Fu, and R.C.-W. Wong, "Projective Clustering by Histograms," IEEE Trans. Knowledge and Data Eng., vol. 17, no. 3, pp. 369-383, Mar. 2005.

[9] G. Moise, J. Sander, and M. Ester, "Robust Projected Clustering," Knowledge Information Systems, vol. 14, no. 3, pp. 273-298, 2008.

[10] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan, "Automatic Subspace Clustering of High Dimensional Data for Data Mining Applications," SIGMOD Record, vol. 27, no. 2, pp. 94- 105, 1998.

[11] C.C. Aggarwal, J.L. Wolf, P.S. Yu, C. Procopiuc, and J.S. Park, "Fast Algorithms for Projected Clustering," SIGMOD Record, vol. 28, no. 2, pp. 61-72, 1999.

[12] M.L. Yiu and N. Mamoulis, "Iterative Projected Clustering by Subspace Mining," IEEE Trans. Knowledge and Data Eng., vol. 17, no. 2, pp. 176-189, Feb. 2005.