Discovery of Order Preserving Subspace Clusters in Large Data Sets

Tao Tao, Hui Fang, ChengXiang Zhai, and Jiong Yang
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801
Email: {taotao,hfang,czhai}@uiuc.edu, jioyang@cs.uiuc.edu

Abstract

In many experimental situations, notably gene microarray experiments, we observe matrix data where every object achieves an expression value under every experimental condition. Subspace clustering on such data is to discover a set of objects, which display similar patterns of expression values under a set of conditions. The quality of clustering results highly depends on the definition of clusters. In this paper, we propose a novel definition of subspace clusters based on a combination of two existing constraints – order constraint and fluctuation constraint. The order constraint guarantees that the relative ranks of the expression values under all the conditions in a cluster will be the same for any object, while the fluctuation constraint restricts the fluctuation of changes of expression values from one condition to another. Combining them together leads to clusters with more consistent and coherent patterns of the expression values. It is computationally challenging to discover such clusters in large data sets since subspace clustering is NP-hard in general. In order to discover all the qualified clusters efficiently, we propose a novel deterministic algorithm that iteratively refines the cluster space through data projection and cluster space splitting. The experiments on both real data and synthetic data demonstrate the effectiveness and efficiency of our algorithm.

1 Introduction

Data clustering is a traditional unsupervised learning topic. It has been extensively studied for many years [5, 9] and has many applications. Traditional clustering algorithms rely on a distance measure between different objects to minimize the intra-cluster distances and maximize the inter-cluster distances at the same time. A cluster is a collection of objects that share similar properties.

In many experimental situations, notably gene microarray experiments, we observe matrix data where every *object* achieves an *expression value* under every experimental *condition*. Unlike traditional clustering problems, clustering such data is to discover the clusters embedded in the subspace of a high dimensional data set. The following information retrieval example illustrates the difference. Table 1 shows different people's interests in different topics. There are four people and eight topics. Everybody gives a score for every topic. For example, Jane gives 4 to movie. In order to cluster such data, traditional methods use two different approaches. One is to treat different topics as different attributes. In this example, each object has 8 attributes, so they are 8-dimensional data. A distance is defined in this 8-dimensional space to measure the similarity between different people. Unfortunately, it is hard to find several people that share similar interests in **all** 8 topics. Instead, several people may share similar interests in sports, but not in others. In this example, Jane, Tommy and John all like basketball more than tennis, and tennis more than football, but their interests in other topics are not similar. Another common clustering approach is to consider the triplet: (name, attribute, score), and convert this table into $4 \times 8 = 32$ triplets. However, this approach can generate "undesirable shape" cluster in the table. For example, it might group only (Jane, movie, 4) and (John, music,4) into a cluster, which is not the "rectangle shape" that we want. A cluster should include several objects , several attributes and all the scores at the crossing points. In the example above, if a cluster includes (Jane, movie) and (John, music), it should include (Jane, music) and (John, movie) as well.

The example above shows how the subspace clustering problem is different from traditional clustering problems. A major challenge here is to define appropriate criteria to find coherent subspace clusters, rather than define an appropriate similarity function as in a traditional clustering method. A major research question is what is an effective measure to judge whether a subspace is a qualified cluster. In general, we consider *pattern similarity* for this judgement. Again, we use the examples in Table 1. We plot these values in Figure 1. There is no obvious similarity between objects.

	movie	book	video	basketball	food	football	tennis	music
Jane	4	3	4	6	5	8	7	3
Tommy	3	4	2	5	6	7	6	1
John	6	7	5	8	3	10	9	4
Ali	8	5	1	2	7	9	10	7

Table 1. An information retrieval example.

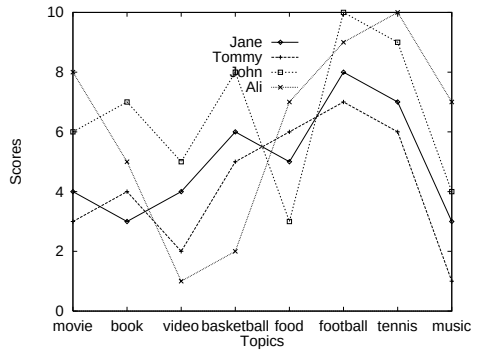

Figure 1. Matrix data.

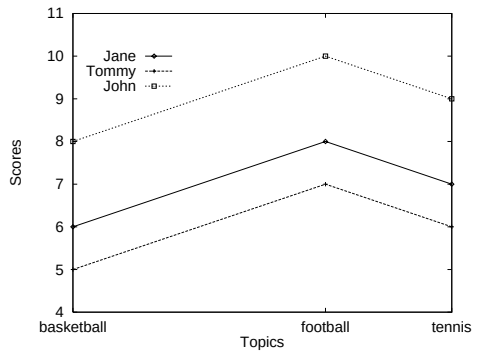

Figure 2. Similar patterns.

However, if we focus on only three attributes: (basketball, football, and tennis) and three objects: (Jane, Tommy, and John), and plot them in Figure 2, they look very similar now with only difference being the shift values. In reality, such shift values can be caused by personality or other unimportant experimental factors. Our purpose is to extract these similar patterns without being distracted by their shift values.

However, this "similarity" is still a vague concept. In reality, due to noise and other factors, it is almost impossible that different objects would generate exactly identical patterns with just difference in shift values. Thus our criteria should allow some small fluctuation in values. Of course, if the fluctuation is too dramatic, it should not be considered as noise.

Intuitively, a qualified cluster should satisfy two constraints:

1. Similar patterns should share the same relative ranks in terms of conditions/attributes. For example, if Jane, Tommy and John are in the same cluster, they should share the same relative ranks of interests under the conditions in this cluster. In the example above, they all like basketball more than tennis, and tennis more than football.

2. The fluctuation should be under control. Jane's interests in basketball and football differ by 2, while the difference for Ali is 7. They should probably not be in the same cluster even though they both like football more than basketball.

Each of these two constraints is studied in previous works, but none of them has considered both constraints together. The need for applying both constraints in clusters can be seen from Figure 3. Figure 3 is a real example extracted from some yeast microarray data [14]. Using only the fluctuation constraint will partition the three genes into a single cluster, or separate gene3 from gene1 and gene2. However, gene1 is different from gene2 and gene3 in terms of their pattern shapes. A constraint on their relative order of expression values can avoid this situation effectively. Also, using only one constraint, existing algorithms generate a large number of loose clusters making the further analysis difficult. It is reasonable to use more strict constraints to reduce the result space. Driven by these observations, we combine the two constraints together, and develop an efficient algorithm to find clusters based on the two constraints.

The contributions of this paper are 3-fold:

1. We propose a new subspace clustering model by combing two constraints, which are more appropriate for many subspace clustering problems. The two constraints restrict the pattern trend as well as pattern magnitude. As we can see later in this paper, the objects in such a cluster show consistent patterns.

2. We propose a novel algorithm with two properties:

   (a) It is a top-down depth-first searching algorithm. It has much smaller memory cost for large data

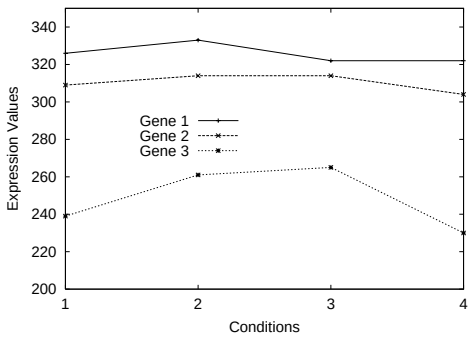

Figure 3. Problem without constraint on orders.

sets than a bottom-up algorithm, such as the p-clustering algorithm [15].

(b) It is a deterministic algorithm, thus avoids the completeness problem in a stochastic algorithm [7].

3. We use Gene Ontology [1] to evaluate our clustering results on real yeast microarray data.

The rest of this paper is structured as follows. In section 2, we review the related works, which motivate our clustering definition and algorithm. We define our clustering criteria in section 3, and describe our algorithm in section 4. Redundancy elimination is addressed in section 5. Experiment results are discussed in section 6. Conclusions and future work are in section 7.

2 Previous Work

The subspace clustering problem has been studied recently [2, 4, 6, 7, 13]. Cheng and Church [7] proposed bi-clustering algorithm. They designed the *mean squared residue score* as a measure for a qualified cluster. Suppose $\mathcal{C}$ is an experimental condition set and $\mathcal{O}$ is an experimental object set. $C \subseteq \mathcal{C}$ and $O \subseteq \mathcal{O}$ are subsets. $Y_{o,c}$ is an expression value that an object o achieves under a condition c. The mean square residue of (O, C) is defined as :

$$H(O,C) = \frac{1}{|O||C|} \sum_{o \in O, c \in C} (Y_{o,c} - Y_{o,C} - Y_{O,c} + Y_{O,C}).$$

where $Y_{o,C} = \frac{1}{|C|} \sum_{c \in C} Y_{o,c}$, $Y_{O,c} = \frac{1}{|O|} \sum_{o \in O} Y_{o,c}$, $Y_{O,C} = \frac{1}{|O||C|} \sum_{c \in C, o \in O} Y_{o,c}$ are the row and column means. Bi-clustering can discover more than one qualified clusters. However, as a stochastic algorithm, it cannot guarantee completeness—not every qualified cluster can be discovered. After discovering a cluster, this algorithm replaces the values in that cluster by random data. Thus, when the clusters overlap with each other, the random data ruin the overlapping part. Unfortunately, such overlapping also often happens in reality. The p-clustering algorithm [15] is designed to solve this problem. Unlike bi-clustering, it is deterministic algorithm. It defines a *pScore* of a 2×2 matrix: $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ as:

$$pScore = |(a - b) - (c - d)|$$

A sub-matrix is a p-cluster if and only if the pScore of every 2×2 matrix in the cluster is less than a predefined threshold δ. P-clustering algorithm can discover every cluster in a data set, no matter whether the clusters are overlapping or not. However, it only applies the fluctuation constraint; order constraint is not addressed at all. Thus it cannot avoid the problem in Figure 3.

Ben-Dor et al.'s paper [4] uses only the order constraint and claims that the conditions permutation for each object is the same in the clusters. The importance of order constraint for microarray data is discussed heavily there. They also prove that the problem is NP-hard. In that paper, they discuss two methods to discover clusters: a complete model and a partial model. Its complete model is simply to enumerate every combination, which is unacceptable in reality. Its partial model is a stochastic model. Like original bi-clustering algorithm, it cannot discover every qualified cluster, either.

In this paper, we extend the previous works by combining the order constraint and fluctuation constraint. The problem formulation will be discussed in following section.

3 Order Preserving Subspace Clusters

As discussed in the previous section, the order constraint plays an important role in the subspace clustering problem. In this section, we formally define this problem of clustering based on both order constraint and fluctuation constraint.

Let us assume that there are a set of *objects*, $\mathcal{O}$, whose size is $|\mathcal{O}|$, and a set of *conditions* $\mathcal{C}$, whose size is $|\mathcal{C}|$. Each object has an *expression value* under each condition. The data set, therefore, takes the form of a large matrix Y_{o_i,c_j} (We write Y_{o_i,c_j} as $Y_{i,j}$ when it does not cause confusion.), $i = 1, ..., |\mathcal{O}|$, $j = 1, ..., |\mathcal{C}|$, where $o_i \in \mathcal{O}$ indexes $|\mathcal{O}|$ different objects and $c_j \in \mathcal{C}$ indexes $|\mathcal{C}|$ different conditions. $Y_{i,j}$ is the expression value that object o_i achieves under condition c_j. Table 2 is such a data set. It has conditions from c_1 to c_5 and objects from o_1 to o_3. The numbers in

the table are the expression values. For example, $Y_{2,1} = 2$ means that object o_2 has expression value 2 under condition c_1.

	c_1	c_2	c_3	c_4	c_5
o_1	1	3	5	2	4
o_2	2	3	1	4	7
o_3	3	6	7	4	8

Table 2. Matrix data.

We define a *cluster* on the matrix data as $S = (O, C)$, where $O \subseteq \mathcal{O}$ is a subset of objects and $C \subseteq \mathcal{C}$ is a subset of conditions. As mentioned in section 1, the objects in a cluster should share the same relative magnitudes under different conditions, and their fluctuation should also be under control. Formally, different conditions and objects in a cluster should satisfy the following two conditions.

Definition 1 *(Order constraint) For any pair of objects o_1, o_2 ($o_1, o_2 \in O$) and any pair of conditions c_1, c_2 ($c_1, c_2 \in C$) in a cluster $S = (O, C)$, $Y_{o_1,c_1} \geq Y_{o_1,c_2} <=> Y_{o_2,c_1} \geq Y_{o_2,c_2}$*

Example 1 *In table 2, $(\{o_1, o_2\}, \{c_1, c_2\})$ satisfies the order constraint. However, $(\{o_1, o_2\}, \{c_1, c_3\})$ does not because $Y_{1,1} < Y_{1,3}$ but $Y_{2,1} > Y_{2,3}$.*

Definition 2 *(Fluctuation constraint) For any pair of objects o_1, o_2 and any pair of conditions c_1, c_2 in a cluster S: $|(Y_{o_1,c_1} - Y_{o_1,c_2}) - (Y_{o_2,c_1} - Y_{o_2,c_2})| < \delta$, where δ is a predefined positive threshold. Intuitively, this constraint means that the change of values on the two conditions between two objects is confined by δ. This constraint is precisely the pScore defined in [15].*

Example 2 *In table 2, suppose $\delta = 1$. $(\{o_1, o_2\}, \{c_1, c_2\})$ is qualified. However, $(\{o_1, o_2\}, \{c_1, c_5\})$ is not qualified because $(Y_{1,1} - Y_{1,5}) - (Y_{2,1} - Y_{2,5}) = 2 > \delta$.*

Note that both order constraint and fluctuation constraint have anti-monotonic property [11], *i.e.* if a set satisfies a constraint, all its subsets satisfy that constraints as well. We will see that this property helps us avoid re-checking constraints after a big table is split into small tables.

We call an $S = (O, C)$ **order preserving cluster**, or just **order cluster** for simplification, if and only if it satisfies both constraints above.

In general, order clusters can be discovered by enumerating all possible combinations and checking the constraints on them one by one. However, it is too expensive to be practical. In the next section, we will develop a novel algorithm to prune the searching space in order to improve the efficiency.

4 Order Clustering Algorithm

Similar to the p-clustering algorithm, our approach is deterministic. We can find all clusters satisfying both order constraint and fluctuation constraint.

In our algorithm, we use an operation called *projection* [8, 12] to enforce the order constraint. We enumerate every condition to project the table into a new suffix table. In each row of the new table, there are only the conditions whose expression values are larger than or equal to that of the current projected condition. This projection can be done recursively. During each projection, some objects are removed because the current condition is not in their rows (They are deleted during previous projections). We will show that the clusters, including the conditions on the projection path and the objects left in the suffix table, satisfy the order constraint. Projection is used to guarantee the order constraint. The details will be addressed in section 4.1. Projection is an efficient operation to shrink the table size and reduce its search space because it removes some objects from the table each time. After each projection, we can check its fluctuation constraint. Usually, not all entries in the table can satisfy it. A large table therefore splits into several smaller tables. Each table is a subset(sub cluster) satisfying the fluctuation constraint. Since they are subsets, they satisfy the order constraint as well due to the anti-monotonic property of our constraints. The splitting operation reduces the searching space more, and improves the algorithm's efficiency. This part of algorithm is called *splitting* and can be found in section 4.2. The main algorithm outline and a complete example are shown in section 4.3.

4.1 Projection

As mentioned in section 3, order clusters need to satisfy two constraints. In this section, we discuss how to enforce the first one—order constraint.

In the previous section, like fluctuation constraint, order constraint is defined on a pair of conditions, *i.e.* it is defined on every 2×2 matrix. Thus, the order here looks like a partial order, which only requires the order on a part of conditions. However, with careful observation, we find this is, in fact, a complete order. The order constraint is to enforce a global order on all conditions. Formally, we have the following lemma.

Lemma 1 *Given a cluster $S = (O, C)$, where $O = \{o_1, ..., o_{|O|}\}$ is an object set and $C = \{c_1, ..., c_{|C|}\}$ is a condition set, it satisfies the order constraint if and only if for every object o_i, the expression values under all conditions $c_j \in C$ have the same order of ranks. In other words, if an object o_1's expression values have the ascending or-*

der $c_{i_1}, c_{i_2}, ..., c_{i_{|C|}}$, all other objects in O should have the same ascending orders.

The correctness of this lemma is easy to see. An example would be sufficient to show why the lemma is correct.

Example 3 *In table 2, let us see $(\{o_1, o_2, o_3\}, \{c_1, c_2, c_5\})$. We extract this 3×3 matrix from our data set:*

$1, 3, 4$
$2, 3, 7$
$3, 6, 8$

.

Clearly, every 2×2 matrix satisfies the order constraint. It is also easy to verify that for all objects (o_1, o_2, and o_3), the ascending order of ranks of expression values is (c_1, c_2, c_5).

This lemma inspires a new point of view to enforce the order constraint. To find a cluster, satisfying the order constraint, is equivalent to finding a set of objects and a set of conditions, in which each object has the same order of ranks under all the conditions. In order to discuss it easily, we convert our original data format to the transactional database format. Each object is a tuple including $|C|$ data pairs. A data pair includes the condition name and its expression value. For example, we convert Table 2 into Table 3. In the original table, $Y_{1,1} = 1$, the pair $(c_1, 1)$ is put into the row of o_1. We now call a row following an object a *tuple*.

o_1	$(c_1, 1)$ $(c_2, 3)$ $(c_3, 5)$ $(c_4, 2)$ $(c_5, 4)$
o_2	$(c_1, 2)$ $(c_2, 3)$ $(c_3, 1)$ $(c_4, 4)$ $(c_5, 7)$
o_3	$(c_1, 3)$ $(c_2, 6)$ $(c_3, 7)$ $(c_4, 4)$ $(c_5, 8)$

Table 3. Sorted data.

Based on the new data format, we define the following concepts.

Definition 3 *(c_i-suffix) Given a tuple $S = \{(c_1, Y_{c_1}), (c_2, Y_{c_2}), ..., (c_n, Y_{c_n})\}$, a c_i-suffix is a subset S_{c_i} ($S_{c_i} \subseteq S$), which includes (c_j, Y_{c_j}) ($(c_j, Y_{c_j}) \in S$) if and only if $Y_{c_j} \geq Y_{c_i}$. The reason that we call it "suffix" is that the c_i-suffix includes all the pairs behind (c_i, Y_{c_i}) if we sort them in an ascending order in terms of expression values. If c_i has the largest expression value within S, the c_i-suffix is ϕ. If c_i is not in S, its suffix does not exist.*

Note that ϕ is different from "not existing"; ϕ means that the suffix does not include any pair. However, c_i itself is in the original tuple. Thus, during our projection algorithm, we keep object o in the new table if its c_i-suffix is ϕ, but remove it if its c_i-suffix does not exist. Since our clusters include all conditions on the projection path, even if a c_i-suffix is ϕ, its projection path includes c_i already.

Example 4 *In Table 3, the c_2-suffix of object o_1 is $\{(c_3, 5), (c_5, 4)\}$.*

Definition 4 *(c_i-suffix table) Given a data set (a table), for each object in this table, if its c_i-suffix set exists, we keep its c_i-suffix in the new table. We call the new table c_i-suffix table. We call the operation to obtain the c_i-suffix table **projection**.*

Example 5 *The c_2-suffix table of Table 3 is*

o_1	$(c_3, 5), (c_5, 4)$
o_2	$(c_4, 4), (c_5, 7)$
o_3	$(c_3, 7), (c_5, 8)$

.

In each line, it has the object name and its c_2-suffix. Because c_2 appears in all tuples in original table, every object has its c_2-suffix. Therefore, the c_2-suffix table includes all objects.

Definition 5 *(Path suffix table) Given a table and a sequence of conditions $SC = c_{i_1}, c_{i_2}, ..., c_{i_j}$ in order, a path suffix table, or just suffix table for simplification, is obtained by projecting $c_{i_k} (1 \leq k \leq j)$ to the table iteratively. We call SC a projection condition path. For instance, if the original table is T, we project c_{i_1} to get c_{i_1}-suffix table T_1, then project c_{i_2} on T_1 to get c_{i_1}, c_{i_2}-suffix table T_2. Iteratively, every c_{i_k} is projected in order, we can then get the SC-suffix table. A c_i-suffix table is a special case of suffix tables because its condition path includes only one single condition c_i.*

Example 6 *In order to obtain the (c_2, c_3) suffix table of Table 3, we project c_2 on the original table to get the c_2-suffix table, and then project c_3 on the c_2-suffix table to obtain the (c_2, c_3) suffix table. The (c_2, c_3) suffix table is*

o_1	$(c_5, 4)$
o_3	$(c_5, 8)$

.

Note that o_2 does no longer exist in this table because c_3 is not in the tuple of o_2 in c_2-suffix table.

Based on Lamma 1, to find a cluster that includes a set of conditions SC in order, is equivalent to finding the SC suffix table, which includes all the conditions in SC and all objects appearing in the SC suffix table.

In our implementation, in order to find the suffix table easily, we first sort the expression values of all objects. We can then use binary search to locate c_i, and finally generate its suffix set by copying all the items behind c_i and the items before c_i but having the same expression value as c_i. The copying operation could be virtual and not necessarily implemented really. Instead, we can use a pointer to record current projection position.

Example 7 *We sort the data in each tuple of Table 3 based on its expression values, and obtain*

o_1	$(c_1,1),(c_4,2),(c_2,3),(c_5,4),(c_3,5)$
o_2	$(c_3,1),(c_1,2),(c_2,3),(c_4,4),(c_5,7)$
o_3	$(c_1,3),(c_4,4),(c_2,6),(c_3,7),(c_5,8)$

.

If we want to obtain c_2-suffix of o_1, we check the expression value of c_2 (It is 3.), do binary search to locate the position $(c_2,3)$, and then add $(c_5,4)$ and $(c_3,5)$ into the new table. Because $2<3$, we do not include $(c_4,2)$. By the same method, we can get the c_2-suffix tuple for every condition and get the c_2-suffix table correspondingly.

It is worthwhile to mention that the projection operation enumerates all possible projection paths in fact. For example, one path could be c_1, c_2, while another path could be c_2, c_1. In general, different paths result in different suffix tables, which include different objects, but in some special cases, they do cause redundancy. This problem will be addressed in section 5.

4.2 Splitting

The projection operation can enforce the order constraint, but cannot guarantee fluctuation constraint by itself. In this section, we develop an operation called *splitting* to prune the tentative search space more and incorporate fluctuation constraint checking.

As we discussed before, each suffix table is a set that satisfies order constraint. In order to enforce the fluctuation constraint, we check it after each projection. This will cause one table to split into several small tables, which all satisfy both constraints. We call such a table a *condensed suffix table*.

Condensed suffix tables can be obtained from suffix tables in the following way. Given a projection path $SC = c_{i_1}, c_{i_2}, ..., c_{i_j}$ and one of its condensed SC-suffix tables T, we append a new condition $c_{i_{j+1}}$ to SC. Denote $\hat{SC} = c_{i_1}, c_{i_2}, ..., c_{i_j}, c_{i_{j+1}}$. After the projection operation on T, we get a $\hat{SC}$-suffix table. Since this table comes from a condensed SC-suffix table, the anti-monotonic property of our constraints ensures that its fluctuation constraint on all conditions in SC is automatically satisfied. We only need to check the fluctuation constraints between $c_{i_{j+1}}$ and $c_{i_k}(1 \le k \le j)$. In reality, we iteratively check the constraint from ($c_{i_{j+1}}$ and c_{i_1}) to ($c_{i_{j+1}}$ and c_{i_j}). After each checking, the table splits into several small tables. The following procedure of checking will be iteratively applied to each small table. In our implementation, we use sorting to make this checking easier. Assume we want to check the fluctuation constraint between $c_{i_{j+1}}$ and c_{i_1}. We would compute all the differences between column $c_{i_{j+1}}$ and c_{i_1}. After sorting them, we move a sliding window of size δ from the smallest value to the largest value. All objects falling into the same window would satisfy the fluctuation constraint. We use the following example to illustrate the operation.

Example 8 *The original table is*

o_1	$(c_1,1),(c_2,2),(c_3,3),(c_4,5)$
o_2	$(c_1,2),(c_2,2),(c_3,4),(c_4,7)$
o_3	$(c_1,3),(c_2,4),(c_3,8),(c_4,9)$
o_4	$(c_1,3),(c_2,3),(c_3,6),(c_4,8)$

.

Assume $\delta = 1$. The condensed (c_1, c_2)-suffix table is

o_1	$(c_3,3),(c_4,5)$
o_2	$(c_3,4),(c_4,7)$
o_3	$(c_3,8),(c_4,9)$

.

After extending the path to (c_1, c_2, c_3), its suffix table becomes

o_1	$(c_4,5)$
o_2	$(c_4,7)$
o_3	$(c_4,9)$
o_4	$(c_4,8)$

.

We then compute the difference between c_3 and c_1 ($c_3 - c_1$). They are (o_1:2, o_2:2, o_3:5, o_4:3). Sorting them, we get (o_1:2, o_2:2, o_4:3, o_3:5). We move a size 1 ($\delta = 1$) sliding window, and generate (o_1:2, o_2:2, o_4:3) and (o_3:5). Thus, we have two tables:

o_1	$(c_4,5)$
o_2	$(c_4,7)$
o_4	$(c_4,8)$

and

o_3	$(c_4,9)$

.

We use the same operation based on the differences between c_3 and c_2 ($c_3 - c_2$) on the two tables respectively. It is easy to check we will get three table finally: $\{o_1, o_2\}$, $\{o_2, o_4\}$, and $\{o_3\}$. If we require that the size of desired clusters should be of at least 2 objects, table $\{o_3\}$ can be dropped.

4.3 The Main Algorithm

Figure 4 outlines the main routine of the algorithm. It receives Y as input matrix data, and δ is the threshold for the fluctuation constraint. nc and nr are the minimal columns

Require: Data Y, FluctuationThreshold δ, ColumThreshold nc, RowThreshold nr
1: **/*Sort every column and get new array*/**
2: SORTCOLUMN(Y)
3: **for** i = 1 to $|C|$ **do**
4: $conditionPath$ = (i)
5: T = i-suffix table of Y
6: **for** j = 1 to $|C|$ **do**
7: **if** $j \neq i$ **then**
8: **/*Iteratively Project the data for every condition*/**
9: Projection(T, j, $conditionPath$ δ, nc, nr)
10: Remove j from $conditionPath$
11: **end if**
12: **end for**
13: **end for**

Figure 4. Algorithm PatternClustering.

and rows for a final cluster. A qualified cluster needs to satisfy both constraints and its size should be over nc and nr as well. The algorithm sorts the table first. It then projects every condition iteratively. $conditionPath$ stores the conditions in the projection path. Since the first level projection will not lead to splitting operation, we separate it from others.

Projection is a recursive routine. It is outlined in Figure 5. Its parameters include current suffix table Y, current projection condition cc, condition path pp and three other parameters: δ, nc and nr. It follows three steps. First, it receives the current condition cc and data set, and projects the data set to obtain the cc-suffix table of Y. Second, it obtains the differences of the expression values between current condition and previous conditions, and invokes the splitting routine to partition the suffix table into several small tables. Since it is easy to understand, we do not show the pseudo code for the splitting operation. The splitting routine returns a list of condensed tables. Finally, for each condensed table, if it has not enough objects or conditions to satisfy the nc or rc parameters, we drop the table. Otherwise, it invokes the *projection* routine recursively.

Figure 6 is a complete example. Assume that $\delta = 1$, $nc = 2$, $nr = 2$. The original data is on the top of the figure. It is projected iteratively from c_1 to c_5. Only the c_1 projection path is shown. Since the c_1-suffix table has only c_1 in the projection path, it is not necessary to check the fluctuation constraint, which requires two conditions as least. The table is projected iteratively again. Since every condition appears only once in a tuple, we do not need to project c_1 again. Enumeration is from c_2 now. The (c_1, c_2) suffix table still includes all objects. The fluctuation constraint on c_1, c_2 needs to be checked now. We list the difference $c_2 - c_1$ following every tuple. Note $c_2 - c_1$ is always nonnegative. Actually, the new projected condition always has expression values larger than or equal to those of the conditions projected before. This nonnegative property exists across the whole algorithm. The difference is $(2, 1, 3, 2)$.

Require: SuffixTable Y, CurrentCondition cc, ConditionPath pp FluctuationThreshold δ, ColumThreshold nc, RowThreshold nr
1: append cc at the end of pp
2: **/* T is new suffix table */**
3: $T = NULL$
4: **/*Projection to get cc-suffix table of Y */**
5: **for** i = 1 to $|O|$ **do**
6: **if** cc in tuple of o_i **then**
7: Obtain $cc-$suffix: S of o_i
8: **if** left items in S + length of $pp > nc$ **then**
9: **/*Otherwise, it cannot satisfy nc threshold in the future */**
10: add S to new suffix table T
11: **end if**
12: **end if**
13: **end for**
14: obtain difference table D between cc and each of previous conditions.
15: Ts = Split(T,D,δ,nr) **/*Ts is splitting table list*/**
16: **for** Each table T in Ts and each condition cn($cn \notin pp$) **do**
17: Projection(T,cn,pp,δ,nc,nr)
18: **end for**
19: Remove cc from pp

Figure 5. Algorithm Projection.

We sort the order of objects based on these difference values. Thus, from top to bottom, they are (o_2, o_1, o_4, o_3). A $size = 1$ sliding window cuts the table into two smaller tables, shown on the bottom. Both tables have 2 conditions in the path and 3 objects in the table. They are both qualified order clusters. We continue processing these tables recursively.

Algorithm Complexity: Given a current table T of m columns and n rows, a projection operation needs to locate the current condition, which takes $O(\log m)$ time. The location operation is applied to every object. So projection operation has complexity $O(n \log m)$. A splitting operation needs to sort the rows, which takes $O(n \log n)$ time. A sliding window operation costs us $O(n)$ time. So, we have $O(n \log n)$ time complexity for each splitting operation. As the algorithm continues, both m and n decrease quickly. The algorithm, therefore, achieves good efficiency. However, due to the essential NP-hard property, the worse-case time complexity for the whole algorithm is exponential.

5 Redundancy Elimination

In general, the projection and splitting operations lead to different condense suffix tables on different paths. However, in some special situations, the tables in different branches are the same. In this section, we discuss the issue of redundancy.

There are two situations that redundancy can be generated. One is due to the identical expression values under different conditions for the same object. In the projection operation to obtain the c_i-suffix, we find the current projected condition first and then add the pairs *before* current pair if they share identical expression values. Redundancy

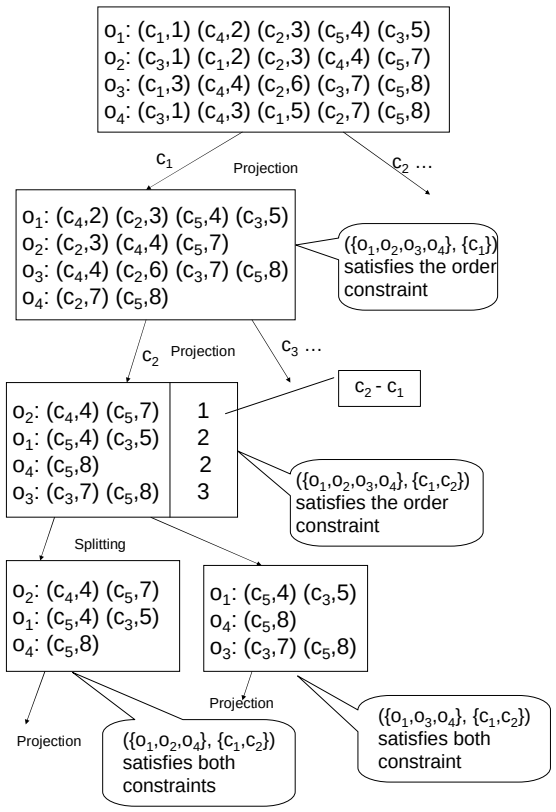

Figure 6. A complete example.

could be generated here.

Example 9

o_1	$(c_1, 2), (c_2, 2), (c_3, 3)$
o_2	$(c_1, 3), (c_2, 3), (c_3, 5)$

In this table, the (c_1, c_2) projection path and (c_2, c_1) projection path both lead to the same cluster $(\{o_1, o_2\}, \{c_1, c_2\})$. Their suffix table is also identical. It is

o_1	$(c_3, 3)$
o_2	$(c_3, 5)$

.

There is another kind of redundancy. It is generated by splitting operation. When the fluctuation constraint is enforced, a single big table splits into several small tables. These tables could be overlapping, but not identical. However, when the algorithm continues, these tables could split again. When the overlapping parts are separated as a condensed table, redundancy could be generated. For example,

Example 10 *Assume the original table is:*

o_1	$(c_1, 1), (c_2, 2), (c_3, 3)$
o_2	$(c_1, 1), (c_2, 2), (c_3, 3)$
o_3	$(c_1, 1), (c_2, 3), (c_3, 5)$
o_4	$(c_1, 1), (c_2, 3), (c_3, 5)$
o_5	$(c_1, 1), (c_2, 4), (c_3, 7)$
o_6	$(c_1, 1), (c_2, 4), (c_3, 7)$

and $\delta = 1$. After checking the fluctuation constraint on c_1 and c_2, the table splits into two small tables: (o_1, o_2, o_3, o_4) and (o_3, o_4, o_5, o_6). The algorithm then continues projecting c_3. It is easy to check that the first table splits into two tables: (o_1, o_2) and (o_3, o_4). The second table also splits into two tables: (o_3, o_4) and (o_5, o_6). Clearly, (o_3, o_4) is redundant.

A general redundancy removal method is to do post-processing for the final results. After generating all qualified clusters, a hash table is built. The hash key is the conditions and the objects. Whenever there is a hash confliction, a redundant result is detected and can be dropped.

However, it is desirable to detect and avoid the redundancy in the precess of clustering, since this would further improve the efficiency. We now show that this is possible for the first kind of redundancy.

The first kind of redundancy can be eliminated in the middle of processing. Based on the analysis above, identical expression values are the primary reason why such kind of redundancy is generated. Intuitively, if all objects in the suffix table have identical expression values under two conditions on the path, it means that the two conditions are totally exchangeable and some possible redundancy could be generated. After we change the order of the two conditions to make up another path, all objects in the original table will appear in the new table. Formally, we have the following lemma.

Lemma 2 *Assume a projection path is SC and its suffix table is T. If there exist two conditions in SC: c_{i_1} and c_{i_2}, such that $Y_{o,c_{i_1}} = Y_{o,c_{i_2}}$ for every object o in T. We claim there should exist another projection path $\hat{SC}$ and its suffix table $\hat{T}$ such that $\hat{SC}$ includes exactly the same conditions as SC (different order) and $\hat{T}$ includes every object of T as well as its suffix tuples.*

Proof *Assume $SC = \alpha c_{i_1} \beta c_{i_2}$, where α and β are sequences of conditions. SC-suffix table is T. We set $\hat{SC} = \alpha c_{i_2} \beta c_{i_1}$. Its suffix table is $\hat{T}$. For any object o in table T, we know $Y_{o,c_{i_1}} = Y_{o,c_{i_2}}$. We assume the suffix tuple of object o is S. Because S exists in table T, it follows that for every condition $c_i \in \beta$. $Y_{o,c_i} = Y_{o,c_{i_1}} = Y_{o,c_{i_2}}$. Therefore, no matter what the order of c_i, c_{i_1}, c_{i_2} is, its suffix tuple should be the same as S because we always put*

identical values into the new suffix table. The suffix tuple of object o under $\hat{SC}$ is S. Thus it should exist in table $\hat{T}$.

Lemma 2 gives us a condition to detect redundancy. If the situation in Lemma 2 occurs, we know there is another path including c_{i_2} and c_{i_1}; we only need keep one of them. In our implementation, if i_1 is smaller than i_2, we keep it. Otherwise, if i_1 is larger than i_2, we drop it. For example, in example 9, we keep the path (c_1, c_2) and its suffix table and drop the path (c_2, c_1).

It is unclear to cope with the second type of redundancy efficiently. This would be an interesting topic for further research.

6 Experiments

We tested our algorithm with both synthetic and real data sets. In section 6.1, we evaluate the quality of our clustering algorithm results. In section 6.2, we test the scalability and parameter sensitivity of our algorithm. Throughout this section, we use N and M to represent, respectively, the number of rows and number of columns in a data set. nr is the minimal number of rows for a qualified order cluster, while nc is the minimal number of columns for a qualified order cluster. δ is the threshold for the fluctuation constraint.

6.1 Quality of Clusters

We evaluate the quality of our clustering results with a real data set—a gene expression data. This yeast microarray data set contains the expression levels of 2884 genes(objects) under 17 conditions [7, 14]. The data is organized into a matrix format. Each row corresponds to a gene and each column corresponds to a condition. Each entry represents a relative abundance of mRNA of a gene under a specific condition. The data range is from 0 to 600.

We use Gene Ontology(GO) to evaluate the quality of discovered clusters. Gene Ontology is a project to provide controlled vocabulary(GO terms) for describing the molecular function and cellular location of gene products and the biological process in which they are involved [1]. In general, Gene Ontology has a tree structure. There are three GO trees: molecular function, cellular component, and biological process. They are used by multiple databases to annotate gene products, so that this common vocabulary can be used to compare gene products across species. The tree structure represents the relationship between different terms. A parent represents a general function while its children have more specific ones. Since genes can be mapped to GO terms, it provides a valid measure for the goodness of our clusters.

To see if the combination of the fluctuation constraint and order constraint helps us to discover more coherent clusters, we compare our algorithm with the p-clustering algorithm, which uses only the fluctuation constraint. We set $\delta = 10$, $nc = 4$ and $nr = 45$. We get 8 clusters for both p-clustering algorithm and our order clustering algorithm. 6 out of the 8 clusters are identical for both p-clustering and order clustering, so we focus on comparing the quality of the two pairs of different clusters. We map all genes in a cluster to GO terms whenever it is possible and ignore those to which there are no corresponding GO terms. After mapping, we find a nearest common ancestor in the GO tree for all of them. The depth of the nearest common ancestor from the tree root is an indicator for the goodness of a cluster; the deeper the nearest common ancestor is, the better the cluster is.

Of the two pairs of clusters left, one pair is identical after mapping to GO terms. So we focus on the other. For this pair, after mapping to GO tree, the p-cluster includes 4 conditions and 35 genes while the order cluster includes 4 conditions and 31 genes—4 genes fewer than the p-cluster. We use all three GO trees (molecular function, cellular component, and biological process) to compare their quality. The depth of the nearest common ancestors are listed in Table 4.

	function	component	process
order cluster	11	1	11
p cluster	11	1	1

Table 4. The depth of nearest common ancestor of all genes in a cluster.

The component tree is a small tree under development. It is reasonable to see the nearest common ancestor is the root. In the function tree, both of them have depth 11. However, in the process tree, there is large difference between the ordering clustering result and the p-clustering result. We further figure out the exact positions of these genes, and show them in Figure 7. We can see the 4 genes are separated from others in the second level. It is clear that the order clustering algorithm indeed remove some unrelated genes from the p-clustering result.

6.2 Algorithm Performance

In order to evaluate the performance of our clustering algorithm, we use several synthetic data sets. We first randomly generate values ranging from 0 to 600 with uniform sampling probability, and then embed a fixed number of clusters in the raw data to ensure that there exist enough qualified clusters in the synthetic data sets. The algorithm is implemented using C++ on a Linux machine with a 2.4GHz CPU and 1G memory.

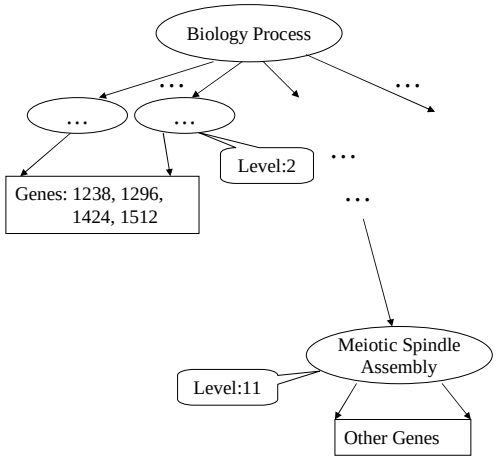

Figure 7. Evaluation of a cluster using a GO tree.

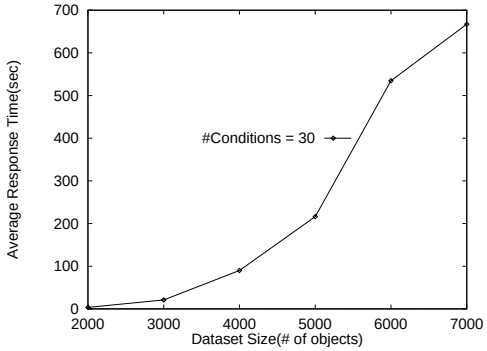

Figure 8. Response time varying # of rows in data sets.

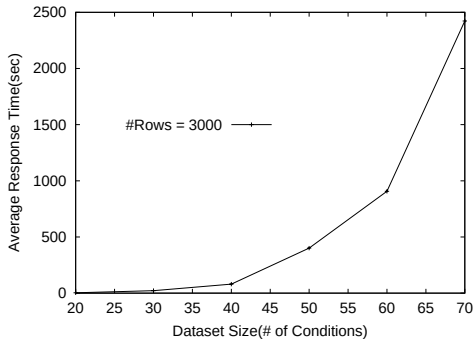

Figure 9. Response time varying # of columns in data sets.

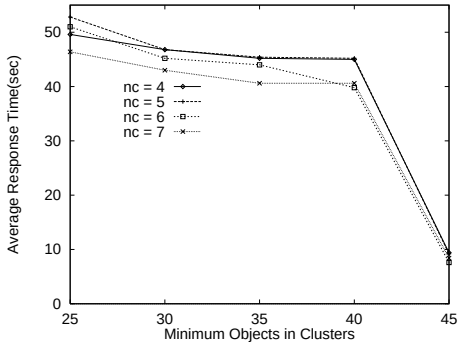

Figure 10. Response time varying different nr and nc.

We first evaluate the scalability on both columns and rows. The running time is the average time obtained from 10 synthetic data sets. The results are shown in Figure 8 and Figure 9.

In Figure 8, we fix the number of columns of synthetic data to 30, and change the number of rows from 2000 to 7000. We embed 30 clusters in each data set. The minimal number of columns of embedded clusters is 6, while the minimal number of rows is $0.01N$, where N is the total number of rows in the data set. We set $\delta = 6$. The plot shows the superlinear property.

Figure 9 is designed to test the scalability of the number of columns. We fix the number of rows to 3000, and change the number of columns from 20 to 70. Similarly, we embed 30 clusters in each data set. The minimal number of rows of embedded clusters is 30, and minimal number of columns is $0.2M$, where M is the number of columns. We again set $\delta = 6$.

Obviously, the row scalability is better than column scalability. Fortunately, in reality, the number of rows is usually much larger than the number of columns. Taking gene microarray for example, a single focus experiment has very few columns. The yeast microarray data set in our experiment has only 17 columns. Even for an experiment with a large heterogeneous condition set, the number of columns could only be up to several hundreds [13]. On the other hand, the number of rows can be up to 10 thousand or even more. Our algorithm favors data sets with large number of rows.

Next, we consider the influence of parameter setting. We simulate 5 data sets. Each of them has 3000 rows and 30 columns. We embedded 30 clusters in each data set. These clusters have a minimum of 40 rows and 10 columns. In Figure 10, there are four curves for $nc = 4, 5, 6, 7$ respectively. In general, the bigger nc is, the faster the program runs. At the same time, when nr is increasing, the running

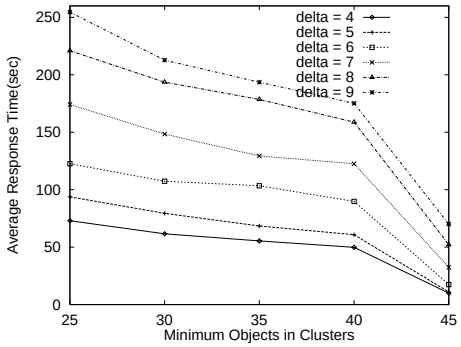

Figure 11. Response time varying different nr and δ.

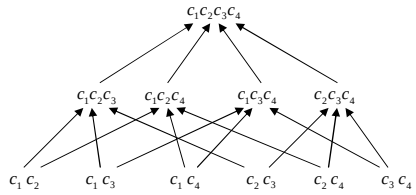

Figure 12. Bottom-up Merging.

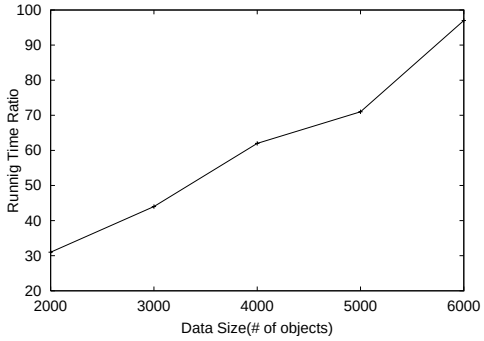

Figure 13. Running time ration of baseline algorithm and order clustering algorithm.

time is decreasing quickly. There is a dramatic drop after nr is larger than 40, because the clusters, which we embedded manually, have the size 40. There are 6 curves in Figure 11, corresponding to that δ is changing from 4 to 9. Obviously, δ has a significant impact on the running time. A bigger δ can reduce the running time substantially.

Finally, we compare the performance of our algorithm with another baseline algorithm, which is a revised p-clustering algorithm. Unlike our algorithm, the p-clustering algorithm has no order constraint and uses a bottom-up approach. We revise it to incorporate the order constraint, and keep its bottom-up search property at the same time. We take this revision as our baseline.

The baseline algorithm starts with clusters containing only two conditions. Sorting allows it easily find out all objects, which can be in the same cluster under the fluctuation constraint. An order constraint filtering is applied to drop any cluster that does not satisfy the order constraint. After this step, the system generates many small clusters, which include two conditions and several objects and satisfy both constraints. Thereafter, the system iteratively combines small clusters to form bigger clusters. It is easy to see that qualified clusters satisfy Apriori property [3]. A bigger cluster is qualified only if all its subsets are qualified. In this way, a lot of small clusters are pruned. Figure 12 shows this merging process. A cluster, including $\{c_1, c_2, c_3, c_4\}$, depends on four three-condition sets: $\{c_1, c_2, c_3\}$, $\{c_1, c_2, c_4\}$, $\{c_1, c_3, c_4\}$, $\{c_2, c_3, c_4\}$. Each of them depends on three two-condition sets. Only if all sets in the lower level satisfy the constraints, would its upper level satisfy it.

Figure 13 is a comparison between the baseline algorithm and our order clustering algorithm. The data sets have 30 columns. The numbers of rows are from 2000 to 6000. We embedded 30 clusters, with at least $0.01N$ rows and 6 columns. We set $\delta = 6$. This figure plots the ratio of the running time of baseline algorithm to our algorithm. It is easy to see that the order clustering is much faster than baseline algorithm. With the size of data increasing, the ratio increase as well. Therefore, the order clustering algorithm has much better scalability.

7 Conclusions and future work

As a main contribution, this paper proposes a novel clustering model. Unlike existing works, this model imposes two constraints — order constraint and fluctuation constraint — upon clusters. A combination of both constraints restricts both the relative order of conditions for objects and the fluctuation of expression values. It, therefore, reduces the result space and obtains more consistent clusters. Driven by the model, a novel top-down depth-first algorithm is proposed to exhaustively discover all the clusters. Experiments on both synthetic data and real life data show its effectiveness and efficiency.

There are several open issues that warrant further investigation.

First, a strict order constraint is too strong in some applications. Sometimes, a more "soft" order constraint may be more effective. For example, instead of requiring all conditions satisfy the order constraint, we can partition them into several groups. The order constraint can be applied within each group. Thus, it generates a partial order constraint. How to efficiently formulate this problem and make the group partition flexible is a challenging problem.

Second, redundancy is generated during the algorithm. This issue is addressed in section 5. Post-processing is an

easy, general method. However, a simple on-the-fly redundancy removal algorithm can improve much the speed. We have explored it to some extend in this paper. It is still unclear how to efficiently detect and remove both types of redundancy elimination.

Third, true clusters can overlap with each other, and the values in the overlapping part are likely different from non-overlapping values. If the difference is small, both p-clustering algorithm and order clustering algorithm can recognize them. However, if the difference is substantial, this problem becomes very difficult to cope with. Plaid model [10] seems to be the only existing work that has handled this situation. Further investigation in this direction is worthwhile.

References

[1] Gene ontology consortium (2000) gene ontology: tool for the unification of biology. *Nature Genet*, pages 25–29.

[2] R. Agrawal, J. Gehrke, D. Gunopulos, and P. Raghavan. Automatic subspace clustering of high dimensional data for data mining applications. *SIGMOD*, pages 94–105, 1998.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. *Proc. 20th Int. Conf. Very Large Data Bases, VLDB*, pages 487–499, 1994.

[4] A. Ben-Dor, B. Chor, R. Karp, and Z. Yakhini. Discovering local structure in gene expression data: The order-preserving submatrix problem. *Proc. RECOMB*, pages 49–57, 2002.

[5] P. Berkhin. Survey of clustering data mining techniques. 2002.

[6] C. H. Cheng, A. W.-C. Fu, and Y. Zhang. Entropy-based subspace clustering for mining numerical data. *Knowledge Discovery and Data Mining*, pages 84–93, 1999.

[7] Y. Cheng and G. M. Church. Biclustering of expression data. *Proc. ISMB*, pages 93–103, 2000.

[8] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: Frequent pattern-projected sequential pattern mining. *KDD*, 2000.

[9] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[10] L. Lazzeroni and A. Owen. Plaid models for gene expression data. *Statistica Sinica*, 12:61–86, January 2000.

[11] J. Pei, J. Han, and L. V. S. Lakshmanan. Mining frequent item sets with convertible constraints. *ICDE*, pages 433–442, 2001.

[12] J. Pei, J. Han, B. Mortazavi-Asl, and H. Pinto. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. *ICDE*, 2001.

[13] A. Tanay, R. Sharan, and R. Shamir. Discovering statistically significant biclusters in gene expression data. *Bioinformatics*, 18(suppl.1):S136–S144, 2002.

[14] S. Tavazoie, J. D. Hughes, M. J. Campbell, J. C. Raymond, and G. M. Church. Systematic determinationof genetic network architecture. *Nature Genetics*, 22:281–285, 1999.

[15] H. Wang, W. Wang, J. Yang, and P. S. Yu. Clustering by pattern similarity in large data sets. *SIGMOD*, 2002.