Mining Order-Preserving Submatrices from Data with Repeated Measurements

Chun Kit Chui, Ben Kao
Department of Computer Science
The University of Hong Kong
{ckchui,kao}@cs.hku.hk

Kevin Y. Yip
Department of Computer Science
Yale University
yuklap.yip@yale.edu

Sau Dan Lee
Department of Computer Science
The University of Hong Kong
sdlee@cs.hku.hk

Abstract

Order-preserving submatrices (OPSM's) have been shown useful in capturing concurrent patterns in data when the relative magnitudes of data items are more important than their absolute values. To cope with data noise, repeated experiments are often conducted to collect multiple measurements. We propose and study a more robust version of OPSM, where each data item is represented by a set of values obtained from replicated experiments. We call the new problem OPSM-RM (OPSM with repeated measurements). We define OPSM-RM based on a number of practical requirements. We discuss the computational challenges of OPSM-RM and propose a generic mining algorithm. We further propose a series of techniques to speed up two time-dominating components of the algorithm. We clearly show the effectiveness of our methods through a series of experiments conducted on real microarray data.

1 Introduction

Among all data mining problems, Order-Preserving Submatrix (OPSM) has important applications particularly in the area of bioinformatics. The general OPSM problem applies to a matrix of numerical data values. The objective is to discover a subset of attributes (columns) over which a subset of tuples (rows) exhibit a similar pattern of rises and falls in the tuples' values. For example, when analyzing gene expression data from microarray experiments, genes (rows) with concurrent changes of mRNA expression levels across different time points (columns) may share the same cell-cycle related properties [12]. Due to the high level of noise in typical microarray data, it is usually more meaningful to compare the *relative* expression levels of different genes at different time points rather than their absolute values. Genes that exhibit simultaneous rises and falls of their expression values across different time points or experiments reveal interesting patterns and knowledge. As an example, Figure 1 shows the expression levels (y-axis) of two different sets of genes under four experimental conditions (x-axis)[1] in the two graphs. The two sets of genes belong to different functional categories. From the figure we see that genes of the same group exhibit similar expression patterns even though their absolute expression values under the same experiment vary.

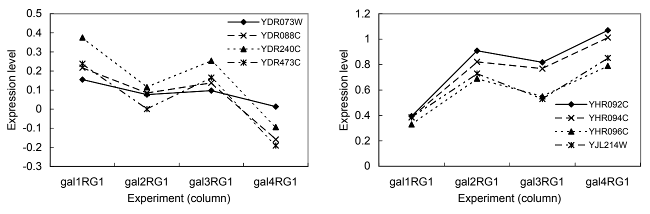

Figure 1. Concurrent expression patterns of two sets of genes from different functional categories

The original OPSM problem was first proposed by Ben-Dor et al. [2].

Definition 1 *Given an $n \times m$ matrix (dataset) D, an order-preserving submatrix (OPSM) is a pair (R, P), where R is a subset of the n rows (represented by a set of row ids) and P is a permutation of a subset of the m columns (represented by a sequence of column ids) such that for each row in R, the data values are monotonically increasing with respect to P, i.e., $D_{iP_j} < D_{iP_{j'}}, \forall i \in R, 1 \leq j < j' \leq |P|$.*[2]

For example, Table 1 shows a dataset with 4 rows and 4 columns. The values of rows 2, 3 and 4 rise from a to b, so $(\{2, 3, 4\}, \langle a, b \rangle)$ is an OPSM. For simplicity, in this study we assume that all values in a row are unique.

We say that a row *supports* a permutation if its values increase monotonically with respect to the permutation. In the above example, rows 2, 3 and 4 support the permutation $\langle a, b \rangle$, but row 1 does not. For a fixed dataset, the rows that support a permutation can be unambiguously identified. In

[1]See Section 7 for a description of the real dataset used.
[2]We use D_{pq} to denote the data item in row p and column q.

	a	b	c	d
row 1	49	38	115	82
row 2	67	96	124	48
row 3	65	67	132	95
row 4	81	115	133	62

Table 1. A dataset without repeated measurements

the following discussion, we will refer to an OPSM simply by its permutation, which will also be called a *pattern*.

An OPSM (and its corresponding pattern) is said to be frequent if the number of supporting rows is not less than a support threshold ρ [4]. Given a dataset, the OPSM mining problem is to identify all frequent OPSM's. In the gene expression context, these OPSM's correspond to groups of genes that have similar activity patterns, which may suggest shared regulatory mechanisms and protein functions.

A drawback of the basic OPSM mining problem is that it is very sensitive to noisy data. In microarray experiments, each value in the dataset is a physical measurement that is subject to different kinds of errors. To combat errors, experiments are often repeated and multiple measured values (called replicates) are recorded. The replicates allow a better estimate of the actual physical quantity. Indeed, as the cost of microarray experiments has been dropping, research groups have been obtaining replicates to strike for higher data quality. For example, in the microarray dataset we use in our study, each experiment is repeated 4 times to produce 4 measurements of each data point. Studies have clearly shown the importance of having multiple replicates in improving data quality [9].

Different replicates, however, may support different OPSM's. In our previous example, if the value of column a is slightly increased in row 3, say from 65 to 69, then row 3 will no longer support the pattern $\langle a, b\rangle$, but will support $\langle b, a\rangle$ instead. As another example, Table 2 shows a dataset with two more replicates added per experiment. From this new dataset, we see that it is no longer clear whether row 3 supports the $\langle a, b\rangle$ pattern. For instance, while the replicates a_1, b_1 support the pattern, the replicates a_2, b_3 do not.

	a_1	a_2	a_3	b_1	b_2	b_3	c_1	c_2	c_3	d_1	d_2	d_3
row 1	49	55	80	38	51	81	115	101	79	82	110	50
row 2	67	54	130	96	85	82	124	92	94	48	37	32
row 3	65	49	62	67	39	28	132	119	83	95	89	64
row 4	81	83	105	115	110	87	133	108	105	62	52	51

Table 2. A dataset with repeated measurements

Our examples illustrate that the original OPSM definition is not robust against noisy data. It also fails to take advantage of the additional information provided by data replicates. There is thus a strong motivation to revise the definition of OPSM to handle repeated measurements. Such a definition should satisfy the following requirements:

(1) If a pattern is supported by all combinations of the replicates of a row, the row should contribute a high support to the pattern. For example, for row 3, the values of column b are clearly smaller than those of column c. All $3 \times 3 = 9$ replicate combinations of b and c values (b_1, c_1), (b_1, c_2), ..., (b_3, c_3) support the $\langle b, c\rangle$ pattern. Row 3 should thus strongly support $\langle b, c\rangle$.

(2) If the value of a replicate largely deviates from other replicates, it is probably due to error. The replicate should not severely affect the support of a given pattern. For example, we see that row 2 generally supports the pattern $\langle a, c\rangle$ if we ignore a_3, which is abnormally large (130) when compared to a_1 (67) and a_2 (54). The support of $\langle a, c\rangle$ contributed by row 2 should only be mildly reduced due to the presence of a_3.

(3) If the replicates largely disagree on their support of a pattern, the overall support should reflect the uncertainty. For example, in row 4, the values of b and c are mingled. Thus, row 4 should not *strongly* support $\langle b, c\rangle$ or $\langle c, b\rangle$.

The first two requirements can be satisfied by summarizing the replicates by robust statistics such as medians, and mining the resulting dataset using the original definition of OPSM. However, the third requirement cannot be satisfied by any single summarizing statistic. This is because under the original definition, a row can only either fully support or fully not support a pattern. The information of uncertainty is thus lost. To tackle this problem, we define a new OPSM problem based on the concept of *fractional support*:

Definition 2 *The fractional support $s_i(P)$ of a pattern P contributed by a row i is the number of replicate combinations of row i that support the pattern, divided by the total number of replicate combinations of the columns in P.*

For example, for row 1, the pattern $\langle a, b, d\rangle$ is supported by 8 replicate combinations: $\langle a_1, b_2, d_1\rangle$, $\langle a_1, b_2, d_2\rangle$, $\langle a_1, b_3, d_1\rangle$, $\langle a_1, b_3, d_2\rangle$, $\langle a_2, b_3, d_1\rangle$, $\langle a_2, b_3, d_2\rangle$, $\langle a_3, b_3, d_1\rangle$, and $\langle a_3, b_3, d_2\rangle$ out of $3^3 = 27$ possible combinations. The fractional support $s_1(\langle a, b, d\rangle)$ is therefore 8/27. We use $sn_i(P)$ and $sd_i(P)$ to denote the numerator and the denominator of $S_i(P)$, respectively. In our example, $sn_1(\langle a, b, d\rangle) = 8$ and $sd_1(\langle a, b, d\rangle) = 27$.

The definition of fractional support satisfies all the three requirements we stated above. Firstly, if all replicate combinations of a row support a certain pattern, the fractional support contributed will be 1. Secondly, if a replicate of a column j deviates from the others, the replicate can at most change the fractional support by $\frac{1}{r(j)}$, where $r(j)$ is the number of replicates of column j. This has small effects when the number of replicates $r(j)$ is large. Finally, if only a fraction of the replicate combinations support a pattern, the resulting fractional support will be fuzzy (away from 0 and 1), which reflects the uncertainty.

The total fractional support of a pattern P (or simply the support of P), is defined as the sum of all the fractional supports of P contributed by all the rows: $s(P) = \sum_i s_i(P)$. A pattern P is frequent if its support is not less than a given support threshold ρ. Our new OPSM mining problem OPSM-RM (OPSM with repeated measurements) is to identify all frequent patterns in a data matrix with replicates.

From the definition of fractional support, we can observe the combinatorial nature of the OPSM-RM problem — the number of replicate combinations grows exponentially with respect to the pattern length. One of the objectives of this work is to derive efficient algorithms for mining OPSM-RM. By proving a number of interesting properties and theorems, we propose pruning techniques that can significantly reduce mining time.

2 Related work

The conventional order-preserving submatrices (OPSM) mining problem was motivated and introduced in [2] to analyze gene expression data without repeated measurements. In [2], it was proved that the problem is NP hard. A greedy heuristic mining algorithm was proposed, which does not guarantee the return of all OPSM's or the best OPSM's.

Since then, mining efficiency has been the main research issue. In [4], the *monotonic* and *transitive* properties of OPSM's were proved. Based on the properties, a candidate set generation-and-test framework was proposed to mine all OPSM's. It makes use of a new data structure, the head-tail trees, for efficient candidate generation. The study reported in [6] concerned the high computational cost of mining OPSM's from massive data. They defined the *twig clusters*, which are OPSM's with large numbers of columns and naturally low supports. They proposed a *KiWi* framework to efficiently mine the twig clusters. None of the above studies, however, handle data with repeated measurements.

The OP-clustering approach [10] generalizes the OPSM model by grouping attributes into equivalent classes. A depth-first search algorithm was proposed for mining all error-tolerated clusters. The model attempts to handle error in single expression values rather than exploiting extra information obtained from repeated measurements. In [3], the problem of mining OPSM's over multiple time points was considered. There are different experimental conditions in each time point, and a pattern is required to be consistent over the time points. An evolutionary algorithm was proposed to explore the search space.

3 Basic algorithm

In this section we discuss a straightforward algorithm for solving the OPSM-RM problem. We use an alternative representation of a matrix dataset that is more convenient for our discussion [6]. For each row, we sort all the values in ascending order, and record the resulting column names as a data sequence. For example, row 1 in Table 2 is represented by the data sequence $\langle b, a, d, b, a, c, a, b, d, c, d, c\rangle$. The advantage of such a representation is that given a row i and a pattern P, the count $sn_i(P)$ is equivalent to the number of subsequences in the data sequence that match P. For example, $sn_1(\langle a, b, d\rangle) = 8$ because there are 8 subsequences in $\langle b, a, d, b, a, c, a, b, d, c, d, c\rangle$ that match the pattern $\langle a, b, d\rangle$. In the following discussion, when we mention a row, we refer to the row's sorted data sequence.

Theorem 1 *Let P_1 and P_2 be two patterns such that P_1 is a subsequence of P_2. For any row i, $s_i(P_2) \leq s_i(P_1)$.*

Proof. It is sufficient to show that the theorem is true for patterns whose lengths differ by 1, i.e., $|P_2| = |P_1| + 1$. We can repeat the argument to prove the theorem for patterns of arbitrary lengths. Let j be the column that is in P_2 but not in P_1, and $r(j)$ be the number of replicates in column j. Each subsequence of row i that matches P_1 can potentially be extended to match P_2 by inserting a column j replicate. Since there are only $r(j)$ such replicates, at most $r(j)$ such extensions are possible. Hence, $sn_i(P_2) \leq r(j) \cdot sn_i(P_1)$. On the other hand, the total number of possible replicate combinations is multiplied by a factor of $r(j)$, i.e., $sd_i(P_2) = r(j) \cdot sd_i(P_1)$. Therefore $s_i(P_2) = \frac{sn_i(P_2)}{sd_i(P_2)} \leq \frac{r(j)\cdot sn_i(P_1)}{r(j)\cdot sd_i(P_1)} = s_i(P_1)$. □

The above monotonic property implies the following Apriori property:

Corollary 1 *Let P_1 and P_2 be two patterns such that P_1 is a subsequence of P_2. P_2 is frequent only if P_1 is frequent.*

Proof. If P_1 is infrequent, $s(P_1) < \rho$. By Theorem 1, $s_i(P_2) \leq s_i(P_1)$ for all row i. So, $s(P_2) = \sum_i s_i(P_2) \leq \sum_i s_i(P_1) = s(P_1) < \rho$. Pattern P_2 is therefore infrequent. □

The Apriori property ensures that an OPSM can be frequent only if all its subsequences (i.e., sub-patterns) are frequent. This suggests an iterative mining algorithm as shown in Figure 2.

As in frequent itemset mining [1], the algorithm iteratively generates the set $Cand_k$ of length-k candidate patterns, and verifies their supports. Those patterns that pass the support threshold are recorded in the set $Freq_k$, which are then used to generate the candidates of the next iteration.

We remark that in the original OPSM problem (without data replicates), all candidates are by definition frequent and thus support verification is not needed. This is due to the transitivity property: if a row supports both patterns $\langle a, b, c\rangle$ and $\langle b, c, d\rangle$, the value at column a must be smaller than

```
Algorithm OPSM-RM
INPUT: raw data matrix $D$, support threshold $\rho$
OUTPUT: the set of all frequent patterns
1:  Transform $D$ into sequence dataset $D'$
2:  $Cand_2$ := {all possible patterns of length 2}
3:  $k = 2$
4:  $Freq_k$ := verify($Cand_k$, $D'$, $\rho$)
5:  while $Freq_k \neq \emptyset$ do
6:      $k := k + 1$
7:      $Cand_k$ := generate($Freq_{k-1}$)
8:      $Freq_k$ := verify($Cand_k$, $D'$, $\rho$)
9:  end while
10: return $Freq_2 \cup ... \cup Freq_k$
```

Figure 2. An Apriori algorithm for OPSM-RM

that at column d, and so it must also support $\langle a, b, c, d\rangle$. However, when there are replicates, the fractional support of a pattern can be smaller than those of all its sub-patterns. For example, the sequence $\langle b, a, d, b, a, c, a, b, d, c, d, c\rangle$ has a fractional support of $4/9$ for $\langle a, b\rangle$, $8/9$ for $\langle b, c\rangle$ and $8/9$ for $\langle a, c\rangle$, but the support for $\langle a, b, c\rangle$ is only $9/27 = 3/9$. Support verification is thus necessary for OPSM-RM.

The efficiency of the algorithm depends on the two core functions **generate** and **verify**. For example, significant speed-up can be achieved if effective pruning techniques are applied so that **generate** produces a smaller set of candidate patterns. In the following we briefly describe the basic algorithms for implementing the **generate** and **verify** functions.

Generate. A convenient way to generate length-k candidates from length-(k-1) frequent patterns is to use the head-tail trees. We briefly describe the data structure here. Readers are referred to [4] for details. Each length-(k-1) frequent pattern P derives two length-(k-2) sub-patterns, called a head pattern P_1 and a tail pattern P_2. P_1 is obtained from P by removing the last symbol of P while P_2 is obtained by removing the first symbol. For example, if $P = \langle a, b, c\rangle$ then $P_1 = \langle a, b\rangle$ and $P_2 = \langle b, c\rangle$. All the head patterns derived from all the length-(k-1) frequent patterns are collected and are stored as strings in a compressed data structure. For each head pattern P_1, a reference to all the frequent patterns from which P_1 is derived is also stored. In our implementation, we use a prefix tree [8] to store the head patterns. We call it the head tree. Similarly, tail patterns are collected and are stored in a tail tree.

To generate length-k candidates, the two trees are traversed in parallel to identify frequent patterns with common sub-strings. For example, if both $P_1 = \langle a, b, c\rangle$ and $P_2 = \langle b, c, d\rangle$ are frequent patterns, then the common substring $\langle b, c\rangle$ will appear in both the head tree (due to P_2) and the tail tree (due to P_1). References to P_1 and P_2 are retrieved. The two patterns are then joined to derive the candidate $\langle a, b, c, d\rangle$.

Verify. Candidate patterns obtained from **generate** are stored as strings in another compressed data structure. Again, we use a prefix tree implementation. To count the candidates' supports, we scan the dataset. For each row i, we traverse the candidate tree and locate all candidate patterns that are subsequences of the data sequence of row i. For each such candidate pattern P, we increment its support $s(P)$ by $s_i(P)$.

Support counting can be made more efficient by compressing data sequences using run-length encoding. Given a data sequence of a row, consecutive occurrences of the same column symbol are replaced by one single symbol and an occurrence count. For example, the data sequence $\langle d, d, d, a, a, b, b, c, c, b, c, a\rangle$ of row 2 in Table 2 is compressed to $\langle d(3), a(2), b(2), c(2), b(1), c(1), a(1)\rangle$. The advantage of compressing data sequences is that the compressed sequences are shorter (in our example, 7 symbols) than the originals (in our example, 12 symbols). The shorter data sequences allow more efficient subsequence matching in support counting. For example, the pattern $\langle d, c, a\rangle$ matches the above compressed data sequences two times (instead of 9 times against the uncompressed sequence): $\langle d(3), ., ., c(2), ., ., a(1)\rangle$ and $\langle d(3), ., ., ., ., c(1), a(1)\rangle$. To determine $sn_i(P)$, we multiply the occurrences for each match and sum the results. In the above example, we have $sn_2(\langle d, c, a\rangle) = 3 \cdot 2 \cdot 1 + 3 \cdot 1 \cdot 1 = 9$.

4 MinBound

From Theorem 1 we know that the support of a pattern contributed by a row cannot exceed the corresponding supports of its sub-patterns. We can make use of this observation to help deduce an upper bound to the support of a candidate pattern. If this upper bound is less than the support threshold ρ, the candidate pattern can be pruned. Fewer candidates result in a faster verification and support counting process, and thus a more efficient mining algorithm.

In this section we discuss a simple bounding technique called MinBound. Recall that in candidate generation, a candidate pattern P is generated by joining two sub-patterns, say, P_1 and P_2. For example, the candidate $\langle a, b, c, d\rangle$ is obtained by joining $\langle a, b, c\rangle$ and $\langle b, c, d\rangle$. Note that both P_1 and P_2 must be frequent and therefore their fractional supports given by each row of the dataset should have already been previously computed. If such supports are cached, we can determine an upper bound of $s(P)$ by

$$s(P) = \sum_{i=1}^{n} s_i(P) \leq \sum_{i=1}^{n} \min\{s_i(P_1), s_i(P_2)\}.$$

For example, for row 1 in Table 2, $s_1(\langle a, b, c\rangle) = 9/27$ and $s_1(\langle b, c, d\rangle) = 7/27$. Therefore an upper bound of

$s_1(\langle a,b,c,d\rangle)$ is $\min\{9/27, 7/27\} = 7/27$. Note that the exact value of $s_1(\langle a,b,c,d\rangle)$ is 6/81 = 2/27.

5 Computing supports by head-tail arrays

Generally, the upper bounds derived by MinBound are not very tight. In this section we introduce head-tail arrays, a data structure that allows the calculation of the exact support of candidate patterns. Although powerful, head-tail arrays are very memory demanding and are thus impractical. Fortunately, the arrays can also be used to derive fairly tight bounds for the support, in which case the memory requirements can be substantially reduced by maintaining only certain statistics. The details will be given in Section 6.

Recall that a length-k candidate pattern P is generated by two length-(k-1) frequent sub-patterns P_1 and P_2, which correspond to the head (i.e., P_1 = P[1..k-1]) and tail (i.e., P_2 = P[2..k]) of P. Given a row i, our goal is to compute the fractional support $s_i(P)$ based on certain support count information we have previously obtained about P_1 and P_2 with respect to row i. To illustrate, let us use row 1 in Table 2 and $P = \langle a,b,c,d\rangle$ as a running example. Let

$$S_1 = \langle \overset{1}{b}, \overset{2}{a}, \overset{3}{d}, \overset{4}{b}, \overset{5}{a}, \overset{6}{c}, \overset{7}{a}, \overset{8}{b}, \overset{9}{d}, \overset{10}{c}, \overset{11}{d}, \overset{12}{c} \rangle$$

be the data sequence of row 1. (Symbol indices are shown for ease of reference.) Also, we have $P_1 = \langle a,b,c\rangle$ and $P_2 = \langle b,c,d\rangle$. The fractional support $s_i(P)$ can be computed by constructing the following two auxiliary arrays.

The *head array* H concerns the head sub-pattern P_1. It contains $r(P[1])$ entries (recall that $r(P[1])$ is the number of replicates of column $P[1]$). The l-th entry of the head array records the number of times P[2..k-1] appear after the l-th occurrence of $P[1]$ in S_i. In our example, $P[1] = a$ and there are $r(P[1]) = r(a) = 3$ replicates, so the head array has 3 entries. Also, P[2..k-1] = $\langle b,c\rangle$. The 3 entries of the head array thus record the number of times $\langle b,c\rangle$ occurs in S_1 after each of the 3 a's. The head array is thus:

H:

5	2	2

The first entry is 5 because after the first a (position 2), there are 5 occurrences of $\langle b,c\rangle$ in S_1, at positions $(4,6),(4,10),(4,12),(8,10)$ and $(8,12)$. Similarly, the second entry is 2 because after the second a (position 5), there are 2 occurrences of $\langle b,c\rangle$, at $(8,10)$ and $(8,12)$.

The *tail array* T concerns the tail sub-pattern P_2. It consists of sn_i(P[2..k-1]) entries. The l-th entry of the array records the number of times $P[k]$ appears after the l-th occurrence of P[2..k-1] in S_i, where the occurrences are in lexicographic order according to the positions of the occurrences. In our example, P[2..k-1] = $\langle b,c\rangle$ and there are $sn_1(\langle b,c\rangle) = 8$ occurrences of $\langle b,c\rangle$ in S_1. In lexicographic order, the positions of these occurrences are: $(1,6)$, $(1,10)$, $(1,12)$, $(4,6)$, $(4,10)$, $(4,12)$, $(8,10)$ and $(8,12)$. The tail array thus has 8 entries, one for each occurrence of $\langle b,c\rangle$. For our example, $P[k] = d$. Each entry in the tail array thus records the number of d's that appear after the corresponding $\langle b,c\rangle$ in S_1. Our tail array is:

T:

2	1	0	2	1	0	1	0

Since the first occurrence of $\langle b,c\rangle$ is (1,6) and there are 2 d's after that (at positions 9 and 11), the first entry of the tail array is 2. The other entries can be determined similarly.

By arranging the occurrences of $\langle b,c\rangle$ in lexicographic order, we ensure that all occurrences of $\langle b,c\rangle$ that appear after a certain position in S_1 are all associated with the rightmost entries of the tail array. This helps us in determining the number of occurrences of a pattern. For example, let us determine the number of $\langle a,b,c,d\rangle$ in S_1 that start with the first a (position 2). From the head array, we know that there are 5 $\langle b,c\rangle$'s after the first a. Because of the lexicographic order, these 5 $\langle b,c\rangle$'s must be associated with the 5 rightmost entries of the tail array. According to the tail array, there are 2, 1, 0, 1, and 0 d's after those 5 $\langle b,c\rangle$'s respectively. Therefore, there are 2 + 1 + 0 + 1 + 0 = 4 $\langle b,c,d\rangle$'s after the first a. Similarly, there is 1 $\langle b,c,d\rangle$ after the second a and 1 after the third a. So in total there are $4+1+1 = 6$ occurrences of $\langle a,b,c,d\rangle$ in S_1.

We can generalize the above computation for any head array H and tail array T. We call the resulting sum the "HT-sum", which can be expressed by the following formula:

$$\textit{HT-sum}(H,T) = \sum_{p=1}^{|H|} \sum_{q=1}^{H[p]} T[|T| - q + 1]. \qquad (1)$$

Since $sd_i(P)$, the total number of replicate combinations, is given by $\prod_{j=1}^{|P|} r(P[j])$, the fractional support $s_i(P) = sn_i(P)/sd_i(P)$ can be readily determined.

6 HTBound

In Section 5 we show that given a length-k candidate pattern P and its generating sub-patterns P_1 and P_2, if we have constructed the head array H and the tail array T, then $sn_i(P)$ (and thus the fractional support $s_i(P)$) can be computed by HT-sum. However, the tail array contains sn_i(P[2..k-1]) entries, which, in the worst case, is exponential to the pattern's length. It is thus impractical to construct or store all the tail arrays. In this section we show that it is possible to compute an *upper bound* of the HT-sum by storing only 3 numbers without ever constructing the head and tail arrays. We call this bound the HTBound. Similar to the idea of MinBound, the HTBound allows us to prune candidate patterns for a more efficient mining algorithm. We will show at the end of this section that HTBound is tighter than

MinBound. To better illustrate the concepts, we continue with our running example considering the data sequence S_1, the length-k candidate pattern $P = \langle a, b, c, d \rangle$, its head sub-pattern $P_1 = \langle a, b, c \rangle$ and tail sub-pattern $P_2 = \langle b, c, d \rangle$.

To determine the HTBound of P, we need the following three values, all obtainable in previous iterations of the mining algorithm. (We show the corresponding values of our running example in parentheses.)

- $sn_i(P_1)$ ($sn_1(\langle a, b, c \rangle) = 9$). This value is obtained in the (k-1)-st iteration. Note that it is also equal to the sum of the entries in the head array.
- $sn_i(P_2)$ ($sn_1(\langle b, c, d \rangle) = 7$). This value is obtained in the (k-1)-st iteration. Note that it is equal to the sum of the entries in the tail array.
- $sn_i(P[2..k\text{-}1])$ ($sn_1(\langle b, c \rangle) = 8$). This value is obtained in the (k-2)-nd iteration. Note that this value is equal to the number of entries in the tail array. Also, no entry in the head array can exceed this value.

We assume that the number of replicates for each column is stored as metadata, i.e., we know $r(j)$ for all column j. In particular, we know $r(P[1])$ and $r(P[k])$. Note that the former equals the number of entries in the head array, while no entry in the tail array can exceed the latter. In our example, $r(P[1]) = r(a) = 3$, so H has 3 entries, and $r(P[k]) = r(d) = 3$, so no entry in T exceeds 3.

The above values thus constrain the sizes, sums and maxima of H and T. For convenience, we call them the "constraint counts". The following property, easily verifiable by the definition of head array, states another constraint on H:

Property 1 *The entries in the head array H are non-increasing (from left to right).*

Our idea of upper bounding HT-sum(H,T) is to show that there exists a pair of arrays H^* and T^* that can be obtained from H and T through a series of transformations. We will prove that (1) each transformation will not reduce the HT-sum and hence HT-sum$(H, T) \leq$ HT-sum(H^*, T^*); (2) H^* and T^* can be constructed using solely the constraint counts. Because of (2), H and T need not be materialized and stored. We will show a closed-form formula for HT-sum(H^*,T^*), which serves as an upper bound of HT-sum(H,T), in terms of the constraint counts. The transformations are based on the following "push" operations:

Definition 3 *A push-right operation on an array A from entry l to entry l' reduces $A[l]$ by a positive value v and increases $A[l']$ by v, where $l < l'$.*

Definition 4 *A push-left operation of an array A from entry l to entry l' reduces $A[l]$ by one and increases $A[l']$ by one, where $l' < l$.*

Essentially, the push operations push values towards the right and left of an array respectively. Here are two useful properties of the push operations:

Lemma 1 *With a fixed head array, each push-right operation on the tail array T cannot reduce the HT-sum.*

Proof. A formal proof is given in [5]. In summary, in the procedure of computing the HT-sum (Section 5), for each entry in the head array, a number of rightmost entries of the tail array are summed. Since each push-right operation on T transfers a positive value from an entry to another entry on its right, the sum cannot be reduced. □

Lemma 2 *If the tail array is non-increasing, each push-left operation on the head array cannot reduce the HT-sum.*

Proof. A formal proof is given in [5]. Here, we illustrate the proof by an example. Consider our example head array $H = [5, 2, 2]$. If we push-left on H from entry $H[3]$ to $H[2]$ by a value of 1, we get $\hat{H} = [5, 3, 1]$. In calculating the HT-sum, the entries $H[2] = 2$ and $H[3] = 2$ each requests the sum of the two rightmost entries in T, i.e., $T[t\text{-}1]$ and $T[t]$ where $t = |T|$. On the other hand, the entries $\hat{H}[2] = 3$ and $\hat{H}[3] = 1$ request the sum of the three rightmost entries in T (i.e., $T[t\text{-}2..t]$) and the value of the rightmost entry in T (i.e., $T[t]$), respectively. So the net difference HT-sum$(\hat{H}, T) -$ HT-sum(H, T) = $T[t\text{-}2]$ - $T[t\text{-}1]$. *If T is non-increasing*, $T[t\text{-}2] \geq T[t\text{-}1]$ and thus HT-sum$(H, T) \leq$ HT-sum$(\hat{H}, T)$. □

Note that Lemma 2 is applicable only if the tail array is non-increasing. In our running example, however, T does not satisfy the non-increasing requirement. Fortunately, we can show that by applying a number of push-right operations, we can transform T to a T' that is non-increasing. With T', Lemma 2 applies, and we can perform a number of push-left operations to transform H to an H^*. Finally, we apply push-right operations to transform T' to a T^*. In this transformation process, by Lemmas 1 and 2, we have HT-sum$(H, T) \leq$ HT-sum$(H, T') \leq$ HT-sum$(H^*, T') \leq$ HT-sum(H^*, T^*). To complete the discussion, we need to define the contents of T', H^* and T^*, and to show that (1) T' so defined is non-increasing and that it can be obtained by transforming T via a number of push-right operations; (2) H^* can be obtained from H via a number of push-left operations, each of which preserves the non-increasing property of the entries, and the content of H^* so defined can be derived from the constraint counts; and (3) T^* can be obtained from T' via a number of push-right operations and its content so defined can be derived from the constraint counts. To accomplish the above, we need to prove a few properties of T first.

Recall that T contains $sn_i(P[2..k\text{-}1])$ entries and that the l-th entry of T records the number of $P[k]$ that appears after the l-th occurrence of P[2..k-1] in the data sequence S_i. In our example, P[2..k-1] = $\langle b, c\rangle$ and there are 8 occurrences of it in S_1 at positions $(1, 6)$, $(1, 10)$, $(1, 12)$, $(4, 6)$, $(4, 10)$, $(4, 12)$, $(8, 10)$ and $(8, 12)$. Let us group the entries together if they correspond to the same occurrence of P[2]. In our example, P[2] = b. The three occurrences of b are positions (1), (4) and (8). So we group the first 3 entries (which correspond to $\langle b, c\rangle$ at $(1, 6)$, $(1, 10)$, $(1, 12)$) together. Similarly, the remaining entries in T are divided into two more groups. We note that each group forms a segment in the T array, called an *interval*. In our example, the intervals are:

T:	2	1	0	2	1	0	1	0

Given an interval I in T, we define the *interval average* of I as the average of the entries in I. For example, the interval averages of the 3 intervals in our example T are 1, 1, and 0.5, respectively. Here is an important property of the interval averages:

Lemma 3 *The interval averages are non-increasing.*

Proof. A formal proof is given in [5]. In summary, consider any interval I and its immediate right neighbor interval I'. We can show that I must contain I' as its rightmost entries (e.g., the second interval ([2,1,0]) contains the third interval ([1,0]) at its right end). We can also show that if I contains additional entries (other than those of I'), the average of these additional entries must be at least as large as the interval average of I' (e.g., the additional entry [2] in the second interval is larger than the third interval's average, which is 0.5). Therefore, the interval average of I must not be smaller than the interval average of I'. Hence, the interval averages are non-increasing. □

With the concept of intervals, we are ready to define T':

Definition 5 *Array T' is the same as T in terms of its size, the number of intervals, and the length of each interval. For each interval I in T', the value of each entry in I is equal to the average value of the corresponding interval in T.*

With our running example, we have,

T:	2	1	0	2	1	0	1	0
T':	1	1	1	1	1	1	0.5	0.5

The following lemma states the desired properties of T'.

Lemma 4 *T' is (a) non-increasing, and (b) obtainable from T via a number of push-right operations.*

Proof. (a): Within each interval, entries in T' share the same value, so they are non-increasing. Also, the entries in T' contain the interval averages of T. By Lemma 3, these averages are non-increasing. So, the entries in T' are non-increasing across intervals.

(b): A formal proof is given in [5]. In summary, for each interval of T, we use push-right operations to obtain the corresponding interval of T'. Here we use our example to illustrate. The entries of the first interval of T are non-increasing, therefore we can repeatedly move the excessive values from the leftmost entry to the next one by push-right operations, forming $(1, 2, 0)$ and then $(1, 1, 1)$. □

Next, we define H^*. Recall that the l-th entry of H records the number of times the pattern P[2..k-1] occurs after the l-th P[1] in S_i. So, no entry in H can exceed sn_i(P[2..k-1]). H^* is obtained from H by pushing as much value to the left as possible, subject to the constraint that no entry in H^* exceeds sn_i(P[2..k-1]). H and H^* thus have the same size and sum. Let x be the number of entries in H, y = sn_i(P[2..k-1]), and z be the sum of all entries in H. H^* is given by

$$H^*[m] = \begin{cases} y & 1 \leq m \leq \lfloor \frac{z}{y} \rfloor \\ z \bmod y & m = \lfloor \frac{z}{y} \rfloor + 1 \\ 0 & \lfloor \frac{z}{y} \rfloor + 2 \leq m \leq x \end{cases} \quad (2)$$

In our example, x = 3, y = 8, and z = 9. H^* is thus:

H:	5	2	2
H^*:	8	1	0

Note that x, y, and z can be obtained from the constraint counts, so H^* can be constructed directly from the constraint counts based on Equation 2 without materializing H.

Lemma 5 *H^* is obtainable from H by a number of push-left operations that preserve the non-increasing property.*

Proof. There are three types of entries in H^*: (1) 0-valued entries, all rightmost; (2) max-valued entries, all leftmost; (3) zero or one remainder entry. For any entry j of H, we call it a donor, a receiver or a remainder entry if the j-th entry of H^* is of type-1, type-2 or type-3, respectively. We repeatedly perform the following: take the rightmost donor that is non-zero, and use a push-left operation to move one from it to the leftmost receiver that is not equal to the maximum value y yet, or to the remainder entry if all receivers are already equal to y. After all the donors are made 0 by the above procedure, if there is a receiver that is still smaller than y by an amount w, we push w from the remainder entry to the receiver to obtain H^*. It is easy to see that each operation preserves the non-increasing property of the array. □

For our example, there is a donor $H[3]$, a receiver $H[1]$, and a remainder entry $H[2]$. We first use two push-left operations to move 2 from $H[3]$ to $H[1]$ to form $(7, 2, 0)$. Then since the receiver still has not reached the maximum value $y = 8$, we use a push-left to move 1 from $H[2]$ to $H[1]$ to form $(8, 1, 0)$, which is equal to H^*.

Finally, we define T^* and show how it can be obtained from T'. Recall that T has $sn_i(P[2..k\text{-}1])$ entries with a sum of $sn_i(P_2)$. Let $x = sn_i(P[2..k\text{-}1])$ and $z = sn_i(P_2)$. T^* is constructed by distributing an integral amount of z evening among the x entries, with the reminder distributed to the rightmost entries of T^*. That is,

$$T^*[m] = \begin{cases} \lfloor \frac{z}{x} \rfloor & 1 \leq m \leq x - (z \bmod x) \\ \lceil \frac{z}{x} \rceil & x - (z \bmod x) + 1 \leq m \leq x \end{cases}$$

In our example, x = 8 and z = 7. T^* is thus:

T':	1	1	1	1	1	1	0.5	0.5
T^*:	0	1	1	1	1	1	1	1

It is obvious that T^* can be constructed from the constraint counts alone.

Lemma 6 *T^* can be obtained from T' by a number of push-right operations.*

Proof. Since the entries in T' are non-increasing and those in T^* are non-decreasing, if $T'[1] = T^*[1]$, then all corresponding entries in the two arrays are equal and no push-right operations are needed. Otherwise, $T'[1] > T^*[1]$, and we can move the difference $T'[1]-T^*[1]$ to $T'[2]$ by a push-right operation. If we now ignore the first entry of each array, the same argument then applies to the second entry. We can repeat the process to equalize every pair of corresponding entries of the two arrays. □

One can verify the following closed-form formula of HT-sum(H^*,T^*). For clarity, let us define a few values:

$$\begin{aligned} h_1 &= \left\lfloor \frac{sn_i(P[1..k-1])}{sn_i(P[2..k-1])} \right\rfloor, \\ h_2 &= sn_i(P[1..k-1]) \bmod sn_i(P[2..k-1]), \\ t_1 &= \left\lfloor \frac{sn_i(P[2..k])}{sn_i(P[2..k-1])} \right\rfloor, \\ t_2 &= (sn_i(P[2..k]) \bmod sn_i(P[2..k-1])), \end{aligned}$$

Finally,

$$\begin{aligned} & \text{HT-sum}(H^*, T^*) \\ = \; & h_1 \cdot sn_i(P[2..k]) + \\ & \begin{cases} h_2(t_1+1) & \text{if } h_2 \leq t_2 \\ t_2(t_1+1) + (h_2 - t_2)t_1 & \text{otherwise} \end{cases} \end{aligned} \quad (3)$$

Note that the above computation only requires the constraint counts. Therefore HT-sum(H^*,T^*) can be calculated without materializing any of H, H^*, T, T' or T^*. For our running example, h_1 = 1, h_2 = 1, t_1 = 0, t_2 = 7, and HT-sum$(H^*, T^*) = 1 \times 7 + 1 \times (0+1) = 8$. Our HTBound thus equals HT-sum$(H^*, T^*)/sd_1(P) = 8/81$. Note that the exact support is 6/81 and the MinBound is 7/27 = 21/81 (see Section 4). HTBound is thus much tighter than MinBound in this example. This is not mere coincidence. We can show that the HTBound is indeed theoretically guaranteed to be better than the MinBound.

Lemma 7 *HTBound is always at least as tight as MinBound.*

Due to space limitation, readers are referred to [5] for a proof of Lemma 7.

7 Experiments

In this section we evaluate our methods using a real microarray dataset that was also used in some previous studies on mining data with repeated measurements [11, 13]. It is a subset of a dataset obtained from a study of gene response to the knockout of various genes in the galactose utilization (GAL) pathway of the yeast Saccharomyces cerevisiae [7][3]. In our dataset, the columns correspond to the knockout experiments of 9 GAL genes and the wild type, growing in media with or without galactose, yielding a total of $2(9 + 1) = 20$ experiments (columns). Each experiment has 4 replicates. There are 205 rows corresponding to genes that exhibit responses to the knockouts. The genes belong to four different classes according to their functional categories. Figure 1 in Section 1 shows some example values of our dataset (only 1 replicate per column is shown).

We compare the performance of three methods: (1) **Basic**, which applies the basic Apriori algorithm (see Figure 2) with data compression, (2) **MinBound**, which is the Basic method plus candidate pruning using MinBound, and (3) **HTBound**, which is the Basic method plus candidate pruning using HTBound. To test the scalability of the methods, we insert synthetic replicates, columns and rows to form larger datasets. To synthesize additional replicates, for each gene and each experiment, we follow standard practice to model the values by a Gaussian distribution with the mean and variance equal to the sample mean and variance of the 4 replicates. The expression values of new replicates are then sampled from the Gaussian. New columns are synthesized by randomly drawing an existing column, discarding the existing expression values, but keeping the fitted Gaussians and sampling new values from them. This way of construction mimics the addition of knockout experiments of genes in the same sub-paths of the original ones. Finally, new rows are synthesized as in the synthesis of new columns, but with an existing row as template instead of a column. This way of construction mimics the addition of genes that have similar responses as some existing ones, due

[3]The dataset can be downloaded at `http://genomebiology.com/content/supplementary/gb-2003-4-5-r34-s8.txt`.

to co-occurrence in the same downstream pathways. In the experiments, the default support threshold is 20%.

We first compare the patterns mined under three different settings: (1) Apply the original OPSM method on only one set of replicates at a time. Since there are 4 replicates per column, the basic OPSM method is applied 4 times. We call these OPSM-i ($i = 1..4$). (2) Replace replicates by their averages and apply the basic OPSM method (OPSM-avg). (3) Consider all replicates and apply our approach (OPSM-RM). We use Fisher's exact test [6] to compute the p-value of each pattern. Intuitively, a small p-value indicates that the genes that support the pattern are highly likely to belong to the same biological class. A pattern is said to be *significant* if its p-value is less than 0.01. In microarray data analysis, the classes are usually unknown and the mined patterns help identify genes that are biologically related. Biologists need to perform costly small-scale experiments to verify the results. A successful mining algorithm should therefore return as few insignificant patterns as possible, in order to minimize the cost.

From the mined patterns, we observe that OPSM-RM always returns fewer insignificant long patterns at various support thresholds. As an illustration, Table 3 shows the number of insignificant patterns of length 4 or more at 20% support threshold.

OPSM-1	OPSM-2	OPSM-3	OPSM-4	OPSM-avg	OPSM-RM
11	6	33	53	14	0

Table 3. Number of insignificant long patterns

Our result shows that if we consider only one set of replicates (OPSM-i), or only the averages of the replicates (OPSM-avg), many insignificant long patterns are returned. Some genes that may not be functionally related could support the same long patterns due to noise in data. Since these insignificant patterns are not returned by OPSM-RM, our result shows that OPSM-RM is robust against data noise.

We now focus on the OPSM-RM model. First, we compare the efficiency of the three methods (Basic, MinBound, HTBound) by applying them to a dataset with 5,000 rows. We report the running time (Figure 3 left) and the number of unpruned candidates that need verification (Figure 3 right) in different iterations of the algorithms, i.e., for different pattern lengths k. For the graph on the right, we also show the actual number of frequent patterns mined for reference.

The figure shows that the two bounding techniques are very effective in speeding up the mining time by pruning infrequent candidates. The pruning effectiveness is most pronounced in iteration 4, in which the number of candidates is the highest. In addition, the pruning power of the HTBound is always stronger than MinBound, which is consistent with Lemma 7. From the figure, we also see that the number of unpruned candidates under HTBound is very

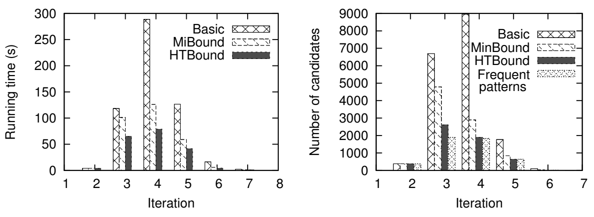

Figure 3. Speed performance in different iterations

close to the actual number of frequent patterns. In particular, there are in total 17,900 candidates generated by the Basic method, among which 4,751 are frequent. There are therefore 13,149 infrequent candidates. Under HTBound, there are only 5,538 unpruned candidates, among which 5,538-4,751 = 787 are infrequent. So, HTBound has pruned (13149-787)/13149 = 94% of all infrequent candidates.

To study the effect of the support threshold, we repeat the comparisons at different thresholds. A dataset with 205 rows is used in this experiment. The results are shown in Figure 4. The left panel shows the running times, and the right panel shows the running times as percentages of the Basic method.

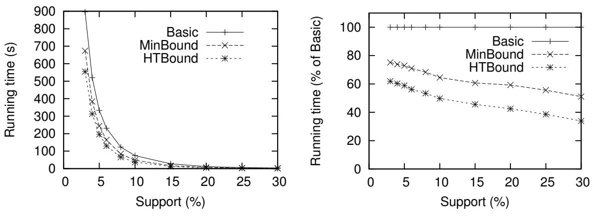

Figure 4. Running time at various support thresholds

In general, the bounding methods significantly improve the efficiency of the mining algorithm, and HTBound is more effective than MinBound in all cases. We observe that at higher support thresholds, the two bounds are capable of pruning more candidates. Yet even at low thresholds, the bounds could still provide substantial performance gains.

Next, we study the scalability of the methods by varying the number of rows, columns, and replicates per column. The results are shown in Figures 5, 6 and 7, respectively.

The relative pruning power of the two methods as compared to the basic algorithm remains largely stable in all three sets of experiments. Also, the running time remains reasonable when there are many replicates, columns or rows, which demonstrates the practicality of our new definition of OPSM in analyzing large datasets.

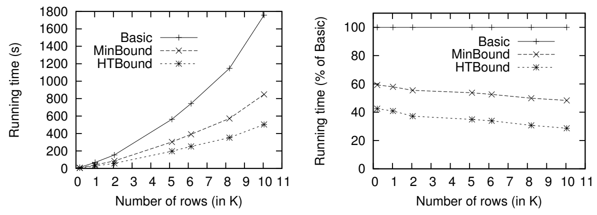

Figure 5. Scalability w.r.t. number of rows

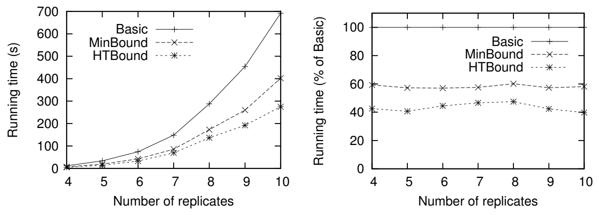

Figure 7. Scalability w.r.t. number of replicates

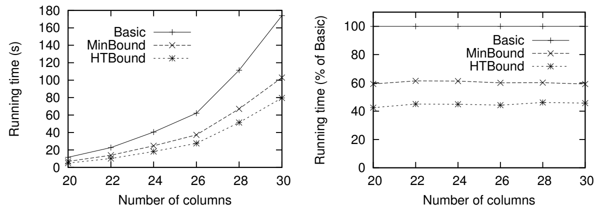

Figure 6. Scalability w.r.t. number of columns

8 Concluding remarks

In this paper we have described the problem of high noise level to the mining of OPSM's, and discussed how it can be alleviated by exploiting repeated measurements. We have listed some practical requirements for OPSM-RM, and proposed a concrete definition that fulfills the requirements. We have described a basic Apriori mining algorithm that utilizes a monotonic property of the definition. Its performance depends on the component functions **generate** and **verify**. We have suggested a sequence compression method for reducing the running time of **verify**. For **generate**, we have proposed two pruning methods based on the MinBound and the HTBound. The latter makes use of the head and tail arrays, which are useful both in constructing and proving the bound. We have performed experiments on real microarray data to demonstrate the effectiveness of the pruning methods, and the scalability of the algorithm.

We will continue our study on the head and tail arrays to see if it is possible to further improve the HTBound. We will also apply our technique to the analysis of other microarray datasets, to look for new findings due to our new definition of OPSM.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499, 1994.

[2] A. Ben-Dor, B. Chor, R. M. Karp, and Z. Yakhini. Discovering local structure in gene expression data: the order-preserving submatrix problem. *Journal of Computational Biology*, 10(3-4):373–384, 2003.

[3] S. Bleuler and E. Zitzler. Order preserving clustering over multiple time course experiments. In *EvoWorkshops 2005*, volume 3449 of *LNCS*, pages 33–43, 2005.

[4] L. Cheung, K. Y. Yip, D. W. Cheung, B. Kao, and M. K. Ng. On mining micro-array data by order-preserving submatrix. *International Journal of Bioinformatics Research and Applications*, 3(1):42–64, 2007.

[5] C. K. Chui, B. Kao, K. Y. Yip, and S. D. Lee. Mining order-preserving submatrices from data with repeated measurements. Technical Report TR-2008-16, HKU CS, September 2008. `http://www.cs.hku.hk/research/techreps/document/TR-2008-16.pdf`.

[6] B. J. Gao, O. L. Griffith, M. Ester, and S. J. M. Jones. Discovering significant opsm subspace clusters in massive gene expression data. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 922–928, 2006.

[7] T. Ideker, V. Thorsson, J. A. Ranish, R. Christmas, J. Buhler, J. K. Eng, R. Bumgarner, D. R. Goodlett, R. Aebersold, and L. Hood. Integrated genomic and proteomic analyses of a systematically perturbed metabolic network. *Science*, 292(5518):929–934, 2001.

[8] D. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching, Third Edition*. Addison-Wesley, 1997.

[9] M.-L. T. Lee, F. C. Kuo, G. A. Whitmore, and J. Sklar. Importance of replication in microarray gene expression studies: Statistical methods and evidence from repetitive cDNA hybridizations. *Proceedings of the National Academy of Sciences of the United States of America*, 97(18):9834–9839, 2000.

[10] J. Liu and W. Wang. OP-Cluster: Clustering by tendency in high dimensional space. In *Proceedings of the Third IEEE International Conference on Data Mining*, pages 187–194, 2003.

[11] M. Medvedovic, K. Y. Yeung, and R. E. Bumgarner. Bayesian mixture model based clustering of replicated microarray data. *Bioinformatics*, 20(8):1222–1232, 2004.

[12] P. T. Spellman, G. Sherlock, M. Q. Zhang, V. R. Iyer, K. Anders, M. B. Eisen, P. O. Brown, D. Botstein, and B. Futcher. Comprehensive identification of cell cycle-regulated genes of the yeast saccharomyces cerevisiae by microarray hybridization. *Molecular Biology of the Cell*, 9(12):3273–3297, 1998.

[13] K. Y. Yeung, M. Medvedovic, and R. E. Bumgarner. Clustering gene-expression data with repeated measurements. *Genome Biology*, 4(R34), 2004.