Mining Order-Preserving Submatrices from Data with Repeated Measurements

Kevin Y. Yip, Ben Kao, Xinjie Zhu, Chun Kit Chui, Sau Dan Lee and David W. Cheung

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 exact values. For instance, in analyzing gene expression profiles obtained from microarray experiments, the relative magnitudes are important both because they represent the change of gene activities across the experiments, and because there is typically a high level of noise in data that makes the exact values untrustable. 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 show the effectiveness and efficiency of our methods through a series of experiments conducted on real microarray data.

Index Terms—H.2.8.d Data mining, H.2.8.a Bioinformatics, H.2.8.i Mining methods and algorithms

1 Introduction

Order-Preserving Submatrix (OPSM) is a data pattern particularly useful for discovering trends in noisy data. The 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 similar 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 [2]. 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) 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.

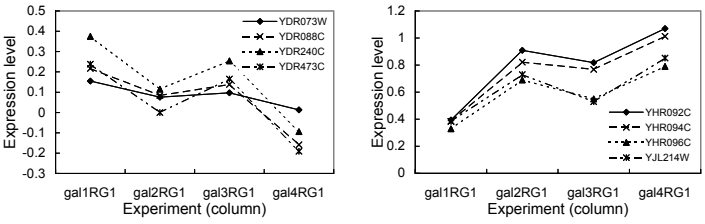


Fig. 1. Concurrent expression patterns of two sets of genes from different functional categories

The original OPSM problem was first proposed by Ben-Dor et al. [3]:

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., $D_{iP_j} < D_{iP_{j'}}, \forall i \in R, 1 \leq j < j' \leq |P|$, where D_{rc} denotes the value at row r and column c of D.

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

- *K. Y. Yip is with the Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong and the Department of Molecular Biophysics and Biochemistry, Yale University, New Haven, Connecticut, USA.*
  E-mail: kevinyip@cse.cuhk.edu.hk
- *B. Kao, X. Zhu, C. K. Chui, S. D. Lee and D. W. Cheung are with the Department of Computer Science, University of Hong Kong, Hong Kong.*
  E-mail: {kao, xjzhu, ckchui, sdlee, dcheung}@cs.hku.hk

A preliminary version of this paper was published in the Eighth IEEE International Conference on Data Mining (ICDM'08) [1]. The main additions in this extended version are: (1) An efficient data structure for candidate verification (Section 3.2.2); (2) A tighter version of HTBound (Section 6); (3) Completely new sets of experiments that involve the new data structure, which scale much better to larger datasets (Section 7.2); and (4) New experiments that show the mined OPSM's are potentially more biologically significant when repeated measurements are considered in the OPSM model (Section 7.1).

TABLE 1
A dataset without repeated measurements

	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

dataset, the rows that support a permutation can be unambiguously identified. In the following discussion, we will refer to an OPSM simply by its permutation, which will also be called a *pattern*.

An OPSM is said to be frequent if the number of supporting rows is not less than a support threshold, ρ [4]. Given a dataset, the basic 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/or protein functions.

In microarray experiments, each value in the dataset is a physical measurement subject to different kinds of errors. A drawback of the basic OPSM mining problem is that it is sensitive to noisy data. 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.

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 some of the microarray datasets we use in our study, each experiment is repeated 3 times to produce 3 measurements of each data point. Studies have clearly shown the importance of having multiple replicates in improving data quality [5].

Different replicates, however, may support different OPSM's. For example, Table 2 shows a dataset with two more replicates added per experiment. From this 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_1, b_2 do not.

TABLE 2
A dataset with repeated measurements

	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

Our example illustrates that the original OPSM definition is not robust against noisy data. It also fails to take advantage of the additional information provided by replicates. There is thus a need 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), and is thus likely an error. 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 neither strongly support $\langle b, c\rangle$ nor $\langle c, b\rangle$.

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, and thus the information of uncertainty is lost. To tackle this problem, we propose a new definition of OPSM and the corresponding mining 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 $sd_1(\langle a, b, d\rangle) = 27$.

If we use fractional support to indicate how much a row supports an OPSM, all the three requirements we stated above are satisfied. Firstly, if all replicate combinations of a row support a certain pattern, the fractional support contributed will be one, the maximum fractional support. Secondly, if one 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 supports a pattern, the resulting fractional support will be fuzzy

(away from 0 and 1), which reflects the uncertainty.

Based on the definition of fractional support, the support of a pattern P is defined as the sum of 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:

Definition 3: Given a dataset, the OPSM-RM problem asks for the set of all OPSMs each of which having a total fractional support from all rows not less than a given support threshold.

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. The objective 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 submatrix (OPSM) mining problem was motivated and introduced by Ben-Dor et al. [3] to analyze gene expression data without repeated measurements. They 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. Cheung et al. [4] proved the *monotonic* and *transitive* properties of OPSM's. 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 by Gao et al. [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. In the study by Bleuler and Zitzler [7], 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. None of the above studies, however, handle data with repeated measurements.

The OP-clustering approach by Liu and Wang [8] 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.

More generally, OPSM is related to the problems of pattern-based subspace clustering [9], biclustering [10], [11] and sequence mining [12], [13], all of which look for patterns in specific subspaces/subsequences. Comparisons of the different methods have been reported by other groups previously [14], [15].

3 Basic algorithm

In this section we discuss a straightforward algorithm for solving the OPSM-RM problem. We use an alternative representation of datasets that is more convenient for our discussion [6]. For each row of a dataset, 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 [16], the algorithm iteratively generates the set $Cand_k$ of length-k candidate patterns, and verifies their supports. Patterns that pass the support threshold are added to the set $Freq_k$, which are then used to generate 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

Algorithm OPSM-RM
INPUT: raw data matrix D, support threshold ρ
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', ρ)
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', ρ)
9: **end while**
10: **return** $Freq_2 \cup ... \cup Freq_k$

Fig. 2. An Apriori algorithm for OPSM-RM

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 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 $9/27 = 3/9$, which is smaller than its supports of all three length-2 sub-patterns. 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 describe the basic algorithms for implementing the **generate** and **verify** functions.

3.1 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. For each length-(k-1) frequent pattern P, two length-(k-2) sub-patterns are derived, 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 [17] to store the head patterns. We call it the head tree. Similarly, tail patterns are collected and are stored in another prefix tree called the 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 sub-string $\langle b, c\rangle$ 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 $\langle a, b, c, d\rangle$.

Notice that since the pattern $\langle a, b, c, d\rangle$ is frequent only if all four length-3 sub-patterns of it are frequent, one may also check if $\langle a, b, d\rangle$ and $\langle a, c, d\rangle$ are frequent before adding $\langle a, b, c, d\rangle$ to the candidate set. This can be done by following the corresponding paths of their heads in the head tree and check if they are linked from the leaf nodes. Although this checking can potentially further reduce the number of candidates, the number of patterns to check is enormous when k is large, and the saving may not be worth the cost. This additional checking is thus not performed in our implementation.

3.2 Verify

To verify whether a candidate pattern is frequent, we need to compute its total fractional support. Directly computing the support from the database D would require a lot of database scans and thus take a long time. The computational overhead can be broken down into two parts: the time to locate and access the relevant rows for each pattern, and the time to compute the fractional support of each row.

In Section 3.2.1, we briefly describe a straight-forward verification procedure that uses a prefix tree. While the use of the tree helps avoid excessive access of irrelevant patterns, it is not very efficient in computing fractional support from individual rows. In Section 3.2.2, we propose a data structure called counting array that can greatly speed up the computation of fractional support. Finally, in Section 3.2.3, we describe a data compression scheme that can further improve the efficiency of the verification process.

3.2.1 Prefix-tree

Candidate patterns obtained from **generate** are stored as strings in a prefix tree. 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 $s_i(P)$. Since the process traverses only the portion of the tree that contains subsequences of row i, it is more efficient than a brute-force database scan. However, since each column label could appear multiple times in row i due to the replicates, support counting requires a lot of back-tracking during tree traversal, which is not efficient. To avoid back-tracking completely, we propose a data structure called counting array for computing the exact support efficiently.

3.2.2 Counting array

Our goal is to compute the fractional support of a pattern P by row i, $s(P) = \frac{sn_i(P)}{sd_i(P)}$. Since the denominator

$sd_i(P) = \prod_{j=1}^{|P|} r(P[j])$ is simply the product of the number of replicates of the involved columns, the only difficulty is the computation of the numerator $sn_i(P)$.

For each suffix P' of P, we construct a *counting array* to store (1) the positions of the first label in P' in the sequence of row i, and (2) the number of occurrences of P' starting from that position. We illustrate it by an example. Suppose the sequence for row i is

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

(label positions are shown for ease of reference), $P = \langle a, b, c, d\rangle$, and we want to calculate $sn_i(P)$. We first consider the first (i.e., shortest) suffix of P, d. The counting array is:

$$d : 7(3), 9(2), 12(1)$$

The array contains three entries. The first one indicates that the first d occurs at position 7, and there are in total 3 occurrences of d in i from position 7 onwards. Similarly, the second and third d's occur at positions 9 and 12, respectively, and there are in total 2 and 1 occurrences from the two positions onwards.

Next, we consider the second suffix of P, cd. The counting array is:

$$cd : 2(6), 8(3), 11(1)$$

Again, the three entries indicate that there are three c's in the sequence, at positions 2, 8 and 11, respectively. Also, from these three positions onwards, there are 6, 3 and 1 occurrences of cd.

The counting arrays for the remaining two suffixes, bcd and $abcd(= P)$ are:

$$bcd : 1(10), 6(4), 10(1)$$

$$abcd : 3(12), 4(8), 5(4)$$

With the counting array for $abcd$, $sn_i(P)$ is simply the value in parentheses of its first entry, i.e., 12. This is because the value states the number of times P occurs in the sequence from the position at which the first a occurs onwards — precisely the definition of $sn_i(P)$.

The reason that we need the counting arrays for all suffixes is that each array can be constructed easily from the previous one. For example, to construct the array for $abcd$, we start from the last entry and work backwards. Suppose we already know that the last occurrence of a in the sequence is at position 5 (which can be easily pre-determined and cached before starting the mining process by a single scan of the sequence), we want to know how many $abcd$'s are there from this position onwards. This is equivalent to asking how many bcd's are there after the position. From the second entry of the array for bcd, we know the answer is 4. Next, we work on the second entry of the $abcd$ array. The question is how many occurrences of $abcd$ are there from position 4 onwards. This is equivalent to asking the number of occurrences of bcd after position 4 (which will be prefixed by this a to become the whole $abcd$ pattern), plus the number of occurrences of $abcd$ after position 4. In other words, it is the sum of the second entry of the bcd array, and the third entry of the $abcd$ array, which is equal to 4 + 4 = 8. Finally, we work on the first entry of the $abcd$ array. Using the previous logic, the entry is equal to the second entry of the bcd array, plus the second entry of the $abcd$ array, which is equal to 4 + 8 = 12.

In general, the value of each entry of an array can be computed by summing a proper entry in the previous array and the next entry of the current array. The proper entry in the previous array is the first one with a position after the current position. Algorithmically, a pointer can be used to keep track of this proper entry. Let $CA(P)$ be the counting array for pattern P, where for each entry k, $CA(P)[k].pos$ is the occurrence position of the k-th $P[1]$ in the sequence and $CA(P)[k].count$ is the number of occurrences of P from that position onwards, then $CA(P)$ is constructed using the following algorithm:

Function Verify
INPUT:
$CA(P)$ with all $CA(P)[i].pos$ filled but not $CA(P)[i].count$
Completely filled $CA(P')$ where P' is the tail of P
OUTPUT:
Completely filled $CA(P)$

```
1:  ptr := |CA(P')|
2:  for k = |CA(P)| downto 1
3:      while ptr > 0 and CA(P)[k].pos > CA(P')[ptr].pos
4:          ptr := ptr − 1
5:      end while
6:      CA(P)[k].count := CA(P')[ptr + 1].count
7:      if k ≠ |CA(P)| // No next entry for the last entry
8:          CA(P)[k].count := CA(P)[k].count+
            CA(P)[k + 1].count
9:      end if
10: end for
11: return CA(P)
```

Fig. 3. Computing fractional support of row i for pattern P using the counting arrays

Assuming that all label positions are cached, the algorithm has a time complexity of $O(r(P[1]))$, and does not require scanning the dataset. Although in our illustration the counting arrays of all suffixes are involved, only the one for the tail P' (which was computed in the previous iteration when P' was the candidate pattern) is needed in action when computing the fractional support of P. Therefore the space requirement is $O(r(P'[1]))$ (the length of the array) for each candidate pattern P.

3.2.3 Data compression

Support counting can be made even more efficient by pre-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 (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 before support verification. Fewer candidates result in a smaller workload in the verification step, and thus a more efficient mining algorithm.

In this section we first discuss a simple bounding technique called MinBound. In the coming sections we develop a tighter bound by using a more advanced bounding technique.

Recall that in candidate generation, a candidate pattern P is generated by joining two sub-patterns, the head P_1 and the tail 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. We can then 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

The upper bounds derived by MinBound are not very tight in general. 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. Suppose the data sequence of row 1 is as follows:

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

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] appears 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:

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 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:

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 it (at positions 9 and 11), the first entry of the tail array is 2. The other entries are 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 associated with the rightmost entries of the tail array. This helps us determine 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. 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 has the following formula:

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

As discussed before, 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.

In order to avoid redundant summations of the last entries of the tail array, we can construct a cumulative tail array with the same length as the tail array and the l-th entry being the sum of the last entries of the tail array starting the l-th one. Since it is easier to explain various properties using the original tail array, we will keep using it in our coming discussion, but remark that the cumulative version is more efficient if we are to compute the exact support using HT-sum.

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 $sn_i(P[2..k\text{-}1])$ 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 has been 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 has been 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 has been 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 five 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 4: 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 5: 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 our technical report [18]. 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, push-left operations on the head array cannot reduce the HT-sum.

Proof: A formal proof is given in our technical report [18]. 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-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-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-2] - T[t-1]. *If T is non-increasing*, T[t-2] ≥ T[t-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 this 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^*). We can thus use HT-sum(H^*, T^*) as an upper bound of $sn_i(P)$.

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-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 our technical report [18]. 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 can now define T':

Definition 6: 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 our technical report [18]. 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 above the interval average 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 these

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-1]) entries with a sum of $sn_i(P_2)$. Let x = sn_i(P[2..k-1]) and z = $sn_i(P_2)$. T^* is constructed by distributing an integral amount of z evenly among the x entries, with the reminder distributed to the rightmost entries of T^*. That is,

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

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,

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

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.

Readers are referred to our technical report [18] for a formal proof.

6.1 An improved HTBound

Using the ideas developed above, we have also identified a simpler and slightly tighter HTBound. We keep the description of how the old HTBound was derived as it contains a lot of interesting ideas, and its proofs make it easy to derive the following new bound.

The main ingredient of the improved HTBound is a new T^*, which is defined as follows:

Definition 7: Array T^* is the same as T in terms of its size, and each entry is equal to the average of T.

With our running example, we have,

T:	2	1	0	2	1	0	1	0
T^*:	7/8	7/8	7/8	7/8	7/8	7/8	7/8	7/8

Again, it is obvious that T^* can be constructed from the constraint counts alone. Since it is also non-decreasing, the proof of Lemma 6 is also valid for showing that this T^* can be produced from T' by push-right operations.

The corresponding HT-sum based on this T^* is simple:

$$\begin{aligned} \textit{HT-sum}(H^*, T^*) &= \sum_{p=1}^{|H|} \sum_{q=1}^{H[p]} T[|T| - q + 1] \\ &= \sum_{p=1}^{|H|} \sum_{q=1}^{H[p]} \frac{sn_i(P[2..k])}{sn_i(P[2..k-1])} \\ &= \frac{sn_i(P[1..k-1])sn_i(P[2..k])}{sn_i(P[2..k-1])} \end{aligned} \quad (4)$$

For our example, the HT-sum is equal to $(9 \times 7)/8 = 63/8 = 7.875$, which gives the new HTBound of $7.875/81$, slightly better than $8/81$ given by the old HTBound.

In fact, the new HTBound is guaranteed to be at least as tight as the old one. It is easy to see that the old T^* can be produced from the current one using push-right operations. Since each push-right operation on the tail array cannot reduce the HTBound according to Lemma 1, the HT-sum derived from the new T^* is not larger than the old one, and thus after normalizing by the same denominator $sd_i(P)$, the new HTBound is at least as tight as the old one.

7 EXPERIMENTS

In this section we describe experiments that we perform to evaluate the validity of the new OPSM-RM model, and the efficiency of our mining algorithm.

7.1 Validity of the new OPSM-RM model

7.1.1 Setup

To justify the proposal of the more complex OPSM-RM model as compared to the basic OPSM model, we need to show that the mined frequent patterns of OPSM-RM are potentially more biologically significant. We use real microarray gene expression datasets to perform this test. We randomly download seven microarray datasets of the baker's yeast Saccharomyces cerevisiae with replicates from the Gene Expression Omnibus (GEO) database [19], as shown in Table 3.

TABLE 3
Datasets used to evaluate the OPSM-RM model

ID	Probes (rows)	Samples (columns)	Replicates
GDS1611(wt)	9,335	16	3
GDS1611(mut)	9,335	16	3
GDS2002	5,617	10	3
GDS2003	5,617	10	3
GDS2712	9,335	7	3
GDS2713	9,335	7	3
GDS2715	9,335	7	3

We use various ways to evaluate the biological significance of mined patterns. First, we evaluate them using protein-protein interactions (PPIs) [14], [20], [21]. It has been shown, in various model organisms, that proteins encoded by genes with correlated expression are more likely to physically interact [22], [23], [24], [25], [26]. One concrete way to evaluate the mined frequent patterns is to check what fraction of genes having same patterns interact physically. For OPSM-RM, by definition each gene has a fractional support for a pattern. In this part of analysis we define the set of genes associated with a pattern as those with a fractional support no less than an inclusion threshold t. If the patterns based on the OPSM-RM model have higher fractions of PPIs than the traditional OPSM model, the former is potentially capable of revealing more functional relationships between proteins in terms of their interactions.

To ensure the generality of our conclusion, we use three different sets of PPIs with different levels of precision and coverage: BioGRID-10, BioGRID-200, and DIP-MIPS-iPfam [27]. For each set, we compute the background probability of finding a protein interaction between two random proteins, by dividing the number of known interactions by the total number of protein pairs. Then for each mined frequent pattern, we compute the within-pattern probability of protein interaction by dividing the number of known interactions between the proteins supported by the pattern by the number of protein pairs. Finally, we compute the odd-ratio as the within-pattern probability divided by the background probability. A large odd-ratio would indicate a pattern with significantly more protein interactions than a random protein set. We also use the numbers of genes in 1) the whole dataset, 2) the set of genes that support the pattern, 3) the set of genes whose proteins interact and 4) the intersection of 2 and 3 to form a 2x2 contingency table, and compute the probability of getting an intersection at least as large as the observed value using Fisher's exact test. To eliminate the effect of gene set size on these p-values, we sample equal number of genes from each gene set before computing the PPI p-values. We then summarize the p-values of all the gene sets involved by determining the fraction of gene sets being statistically significant, with a p-value less than 0.01.

Besides using PPIs, we also check the enrichment of functional annotations within the gene sets using DAVID [28], a popular tool for performing functional analysis. For each pattern, from the associated set of genes the number of genes that belong to various functional categories are counted, and statistical enrichment is given by a Benjamini-Hochberg corrected p-value [29]. Again, for OPSM-RM the gene sets are determined based on the inclusion threshold t. We summarize these p-values by counting the fraction of statistically significant patterns. We use the whole set of genes in a dataset as the background, and adopt the default parameter values on the DAVID web site.

We use the results from the PPI and DAVID analyses to compare the frequent patterns mined from four different procedures: 1) traditional OPSM, using only one set of replicates (abbreviated as OPSM-x, where x is the replicate number), 2) traditional OPSM, using the average of all replicates (abbreviated as OPSM-avg), 3) traditional OPSM, using the median of all replicates (abbreviated as OPSM-med), and 4) OPSM-RM.

We repeat the tests with multiple values for the cutoff t and the support threshold ρ for frequent patterns.

7.1.2 Results

Table 4 shows the average odd ratios for the patterns mined from GDS2003 at support threshold $\rho = 10\%n$ (n is the total number of rows in the dataset), evaluated by the protein interactions in BioGRID-10. Each row corresponds to the patterns of a different iteration (i.e., pattern length), and each column corresponds to one OPSM model. We include results for two values of the inclusion threshold t.

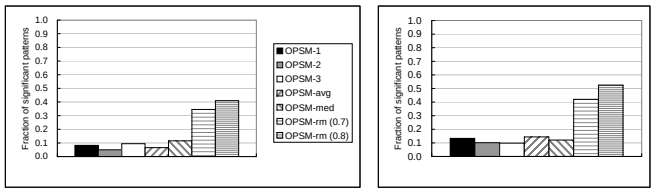
TABLE 4
Odd ratios of patterns mined from GDS2003 at $\rho = 10\%n$, evaluated by BioGRID-10

Iter.	OPSM-1	OPSM-2	OPSM-3	OPSM-avg	OPSM-med	OPSM-RM (t=0.7)	OPSM-RM (t=0.8)
2	1.1	1.1	1.1	1.1	1.1	1.2	1.2
3	1.3	1.2	1.3	1.3	1.3	1.8	2.0
4	1.5	1.5	1.5	1.6	1.6	2.9	3.3

From the results, we see that all methods have all average odd ratios above 1, which suggests that proteins supporting same patterns generally tend to interact more often than random. However, the odd ratios differ between iterations, and between different models. First, the odd ratios correspond to later iterations are higher, which is expected as a longer pattern guarantees that the supporting proteins have similar gene expression trends across more samples. Second, the odd ratios of the OPSM-RM model are higher than the traditional OPSM model. In contrast, while OPSM-avg and OPSM-med combine the information of multiple replicates, they are only marginally better than when only one of the replicates is considered. This comparison shows that OPSM-RM is able to better utilize the information provided by the repeated experiments than applying OPSM on an averaged dataset.

Since longer patterns are observed to be more biologically relevant, in the following we concentrate on the results of the last iteration that all models contain frequent patterns. For example, for the set of results corresponding to Table 4, only the odd ratios of iteration 4 will be shown.

Figure 4 and Figure 5 show the fraction of significant patterns based on PPI and DAVID, respectively. Again we see that the patterns obtained by the OPSM-RM approach are more biologically relevant. This is an encouraging result since the default functional categories used by DAVID do not include protein physical interactions, and thus the two types of analysis are largely independent, yet the main conclusions are consistent.



(a) 50 gene sets (b) 100 gene sets

Fig. 4. Fraction of significant patterns mined from GDS2003 at $\rho = 10\%n$, evaluated by PPI.

It is then natural to ask if the patterns from OPSM-RM are also more biologically relevant than the traditional OPSM model when other datasets, support thresholds, and protein interaction sets are used. We first fix the other parameters, and change the microarray dataset. Figure 6 shows the resulting odd ratios.

The odd ratios from OPSM-RM are in general larger

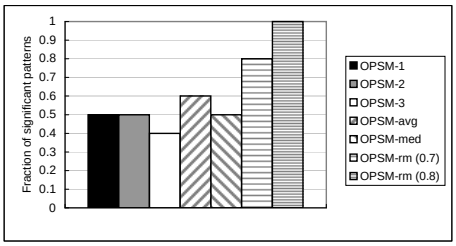


Fig. 5. Fraction of significant patterns mined from GDS2003 at $\rho = 10\%n$, evaluated by DAVID

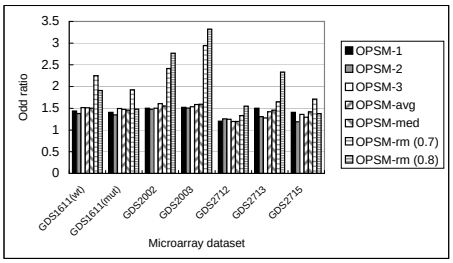


Fig. 6. Odd ratios of patterns mined from various microarray datasets at $\rho = 10\%n$, evaluated by BioGRID-10

than those from traditional OPSM, but the differences are dataset-dependent. More substantial differences are observed for GDS1661(wt), GDS2002 and GDS2003 than the other datasets. We conclude that while the quality of mined patterns highly depends on the specific dataset, OPSM-RM has the potential to mine better patterns.

Next we vary the support threshold to see its effect on the odd ratios. The results are shown in Figure 7.

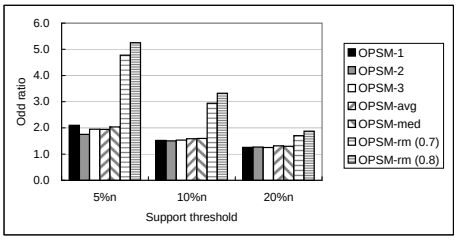


Fig. 7. Odd ratios of patterns mined from GDS2003 at various support thresholds, evaluated by BioGRID-10

We again observe the general trend that OPSM-RM gives patterns with larger odd ratios. The difference is more apparent for smaller support thresholds. The main reason is that when the support threshold is small, the frequent patterns can grow longer, and longer patterns are more likely to contain proteins that interact as we observed earlier. This result indicates that while a larger support threshold is intuitively equivalent to a more stringent requirement, the pattern quality is not necessarily higher due to shorter pattern lengths. In practical use of OPSM-RM, one should thus try multiple support thresholds, and look for reasonably stringent values that give sufficiently long patterns.

Finally, we test the effect of the protein interaction set for evaluation. Figure 8 shows the resulting odd ratios.

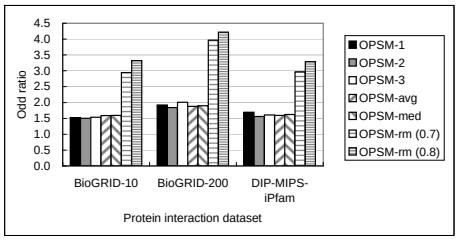


Fig. 8. Odd ratios of patterns mined from GDS2003 at $\rho = 10\%n$, evaluated by various protein interaction sets

The results confirm that patterns from OPSM-RM have higher odd ratios than OPSM regardless of the precision and coverage of the evaluating protein interaction set.

Due to space limitation, we leave the results for other parameter combinations and evaluation methods to our technical report [18]. The general conclusions drawn from the above observations are supported by most of the results. For the DAVID analysis, since the DAVID API imposes limitations on the number of genes per gene set and the total number of gene sets per day, we resorted to performing the analysis by manually entering the gene lists one by one. This time-consuming process prohibited us from performing a complete evaluation of all patterns, methods and parameter choices. Instead, for each method we could only sample 10 patterns mined from GDS2003 at support threshold $0.1n$. We hope that in the future there will be simple ways to perform large-scale DAVID evaluations efficiently.

7.1.3 Handling replicates by statistical tests

We have shown that if we summarize the values from different replicates by a single statistic (such as mean and median), the resulting mined frequent patterns are not as biologically relevant as the ones obtained by the OPSM-RM approach. To further explore other possible ways to handle replicates, we have also studied a method based on statistical tests. For each row, we define the order $a < b$ for columns a and b if the values at a are significantly smaller than those at b based on t-test and a p-value cutoff. Similarly, we define the order $a > b$ if the values at b are significantly smaller than those at a. If the values at the two columns are not significantly different, no ordering is defined between the columns, and the row would not support any pattern that involves an ordering of a and b. This results in a variation of the original OPSM mining problem, with each row allowed to refuse supporting both the patterns $\langle a, b\rangle$ and $\langle b, a\rangle$.

We compared the resulting frequent patterns at different p-value cutoffs with those returned by OPSM-RM. In general, the patterns based on this statistical test approach are slightly more biologically relevant, but the number of frequent patterns is much smaller even at a loose significance level such as 0.1. There is thus a tradeoff between precision and coverage.

This statistical test approach is equivalent to the ANOVA F-test for two samples. For a pattern with more than two columns, we need to perform a t-test for every pair of adjacent columns in the pattern, and a row supports the pattern only if the results of all the tests are statistically significant. This requirement could be too strict as in some cases statistically insignificant changes are still biologically meaningful, especially when the number of replicates is small. It is possible to derive a method to test all involved columns at the same time, but this cannot be done by a standard ANOVA analysis, as it does not involve the total order of all the columns. We leave the detailed investigation of more advanced statistical test approaches to a future study.

7.2 Mining efficiency

7.2.1 Setup

After showing the potential biological significance of the patterns from OPSM-RM, our next concern is the mining efficiency. In particular, we would want to test the speed performance of our algorithm in terms of four scalability parameters, namely the number of rows, columns, replicates, and the support threshold. We are interested in both the absolute performance of our final algorithm with HTBound, and the relative performance as compared to other bounding techniques described.

In order to test the scalability of our algorithm, we start with a small microarray dataset, and generate more rows, columns, and replicates based on the original data distribution as follows.

We choose the microarray dataset that was also used in some previous studies on mining data with repeated measurements [30], [31]. 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 baker's yeast [32][1]. 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 from one of the replicates.

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 were then sampled from the Gaussian. New columns are synthesized by randomly drawing an existing column, fitting Gaussians as described, and sampling values from it. This way of construction mimics the addition of knockout experiments of genes in the same sub-pathways of the original ones. Finally, new rows were synthesized as in the synthesis of new columns, but with

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

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 to co-occurrence in the same downstream pathways.

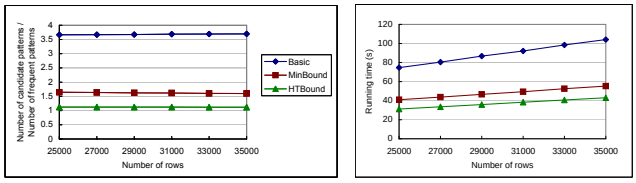
We compare the performance of three methods: (1) **Basic**, which applies the basic Apriori algorithm (see Figure 2) with the counting array data structure and data compression, (2) **MinBound**, which is the Basic method plus candidate pruning using MinBound, and (3) **HTBound**, which is the Basic method plus candidate pruning using HTBound.

We use two different performance metrics: the number of candidate patterns, and the actual running time. The former shows the effectiveness of the bounding techniques, while the later also takes into account the overhead of computing the bounds.

Our programs are written in C. The experiments are performed on a machine with 2GHz CPU, 16GB memory, and Red Hat Linux as the operating system.

7.2.2 Results

We first test the speed performance with various numbers of rows. Figure 9 shows the results.



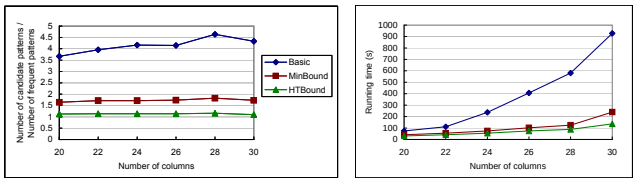
(a) Ratio of candidate patterns (b) Running time

Fig. 9. Speed performance with respect to the number of rows, with 20 columns, 4 replicates and $\rho = 20\%n$

Figure 9(a) shows how many candidate patterns are generated as a multiple of the actual number of frequent patterns. A smaller ratio indicates more effective pruning, and an ideal algorithm that prunes away all infrequent candidate patterns before the verification step would have a ratio of 1. We observe that the bounding techniques are very effective. HTBound, in particular, has a ratio constantly very close to 1 for all numbers of rows. Figure 9(b) further suggests that this saving in support verification justify the extra overhead incurred by computing the bounds. The actual running time of the mining algorithm improves by using better bounds. For instance, with HTBound, the mining time is consistently less than half that without using bounding techniques.

The absolute running time is also reasonable. For 25,000 rows, which is approximately the number of genes in the human genome, the total mining time is only half a minute. This suggests that the OPSM-RM model can be practically applied to new datasets of date.

Next, we test the speed performance with various numbers of columns. As more datasets are produced and incorporated into single analyzes, being able to scale well with respect to the number of columns is crucial to the applicability of an analysis method. Figure 10 shows the speed performance of our algorithm.

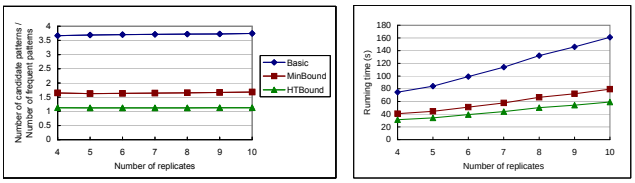


(a) Ratio of candidate patterns (b) Running time

Fig. 10. Speed performance with respect to the number of columns, with 25,000 rows, 4 replicates and $\rho = 20\%n$

Again, we observe strong pruning power of the bounding methods, with the candidate to frequent pattern ratios for HTBound always close to 1. The running time clearly reveals the importance of the bounding methods when there is a large number of columns. For instance, at 30 columns, the algorithm takes only two minutes to complete with HTBound, while the basic approach requires eight times more time.

The next set of tests concerns the number of replicates. The results are shown in Figure 11.

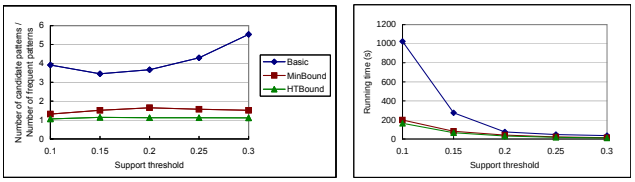


(a) Ratio of candidate patterns (b) Running time

Fig. 11. Speed performance with respect to the number of replicates, with 25,000 rows, 20 columns and $\rho = 20\%n$

The general trends remain the same as before, with the importance of the bounding methods clearly shown. We observe that with HTBound, the running time remains reasonable even with 10 replicates. As in real experiments it is very rare to exceed this number of replicates, practically the algorithm remains applicable.

Finally, we test the efficiency with respect to the support threshold ρ. The results are shown in Figure 12.



(a) Ratio of candidate patterns (b) Running time

Fig. 12. Speed performance with respect to the support threshold ρ (as a fraction of the total number of rows, n), with 25,000 rows, 20 columns and 4 replicates

The running time is longer when a smaller support threshold is used, as there are more patterns qualified as frequent. While the basic approach takes almost 1,000

seconds to complete when the support threshold is equal to 10% of the total number of rows, with HTBound it requires only 1/5 of the time.

In summary, we have observed that our mining algorithm with HTBound is efficient in practical settings, and it is scalable with respect to the number of rows, columns, replicates, and the support threshold. For very large datasets, our algorithms could be quite demanding in terms of memory usage. How efficient disk access can be incorporated in the algorithms is an important follow-up work for this study.

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 a new problem, OPSM-RM that takes into account the repeated measurements, 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 proposed the counting array data structure and 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 biological validity of the OPSM-RM model, the effectiveness of the pruning methods, and the scalability of the algorithm.

As sequencing-based methods have become more popular, the noise level in new gene expression datasets is expected to decrease and more distinct states of expression can be identified. How this will affect the advantages of OPSM-RM over OPSM is to be studied when more sequencing datasets become available.

References

[1] C. K. Chui, B. Kao, K. Y. Yip, and S. D. Lee, "Mining order-preserving submatrices from data with repeated measurements," in *Eighth IEEE International Conference on Data Mining (ICDM'08)*, 2008, pp. 133–142.

[2] 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*, vol. 9, no. 12, pp. 3273–3297, 1998.

[3] 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*, vol. 10, no. 3-4, pp. 373–384, 2003.

[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*, vol. 3, no. 1, pp. 42–64, 2007.

[5] 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*, vol. 97, no. 18, pp. 9834–9839, 2000.

[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*, 2006, pp. 922–928.

[7] S. Bleuler and E. Zitzler, "Order preserving clustering over multiple time course experiments," in *EvoWorkshops 2005*, ser. LNCS, vol. 3449, 2005, pp. 33–43.

[8] 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*, 2003, pp. 187–194.

[9] H. Wang, W. Wang, J. Yang, and P. S. Yu, "Clustering by pattern similarity in large data sets," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, 2002, pp. 394–405.

[10] Y. Cheng and G. M. Church, "Biclustering of expression data," in *Proceedings of the 8th International Conference on Intelligent Systems for Molecular Biology*, 2000, pp. 93–103.

[11] L. Lazzeroni and A. Owen, "Plaid models for gene expression data," *Statistica Sinica*, vol. 12, pp. 61–86, 2002.

[12] R. Agrawal and R. Srikant, "Mining sequential patterns," in *Proceedings of the Eleventh International Conference on Data Engineering*, 1995, pp. 3–14.

[13] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining closed sequential patterns in large databases," in *Proceedings of the Third SIAM International Conference on Data Mining*, 2003, pp. 166–177.

[14] A. Prelic, S. Bleuler, P. Zimmermann, A. Wille, P. Buhlmann, W. Gruissem, L. Hennig, L. Thiele, and E. Zitzler, "A systematic comparison and evaluation of biclustering methods for gene expression data," *Bioinformatics*, vol. 22, no. 9, pp. 1122–1129, 2006.

[15] K.-O. Cheng, N.-F. Law, W.-C. Siu, and A. W.-C. Liew, "Identification of coherent patterns in gene expression data using an efficient biclustering algorithm and parallel coordinate visualization," *BMC Bioinformatics*, vol. 9, no. 210, 2008.

[16] 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*, 1994, pp. 487–499.

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

[18] K. Y. Yip, B. Kao, X. Zhu, C. K. Chui, S. D. Lee, and D. W. Cheung, "Mining order-preserving submatrices from data with repeated measurements," HKU CS, Tech. Rep. TR-2011-04, May 2011, http://www.cs.hku.hk/research/techreps/document/TR-2011-04.pdf.

[19] T. Barrett, D. B. Troup, S. E. Wilhite, P. Ledoux, D. Rudnev, C. Evangelista, I. F. Kim, A. Soboleva, M. Tomashevsky, K. A. Marshall, K. H. Phillippy, P. M. Sherman, R. N. Muertter, and R. Edgar, "NCBI GEO: Archive for high-throughput functional genomic data," *Nucleic Acids Research*, vol. 37, pp. D885–D890, 2009.

[20] Y. Okada, K. Okubo, P. Horton, and W. Fujibuchi, "Exhaustive search method of gene expression modules and its application to human tissue data," *IAENG International Journal of Computer Science*, vol. 34, no. 16, 2007.

[21] X. Liu and L. Wang, "Computing the maximum similarity bi-clusters of gene expression data," *Bioinformatics*, vol. 23, no. 1, pp. 50–56, 2007.

[22] N. Bhardwaj and H. Lu, "Correlation between gene expression profiles and protein-protein interactions within and across genomes," *Bioinformatics*, vol. 21, no. 11, pp. 2730–2738, 2005.

[23] S. D. Bodt, S. Proost, K. Vandepoele, P. Rouze, and Y. V. de Peer, "Predicting protein-protein interactions in arabidopsis thaliana through integration of orthology, gene ontology and co-expression," *BMC Genomics*, vol. 10, no. 288, 2009.

[24] H. Ge, Z. Liu, G. M. Church, and M. Vidal, "Correlation between transcriptome and interactome mapping data from saccharomyces cerevisiae," *Nature Genetics*, vol. 29, no. 4, pp. 482–486, 2001.

[25] R. Jansen, D. Greenbaum, and M. Gerstein, "Relating whole-genome expression data with protein-protein interactions," *Genome Research*, vol. 12, no. 1, pp. 37–46, 2002.

[26] A. K. Ramani, Z. Li, G. T. Hart, M. W. Carlson, D. R. Boutz, and E. M. Marcotte, "A map of human protein interactions derived from co-expression of human mRNAs and their orthologs," *Molecular Systems Biology*, vol. 4, no. 180, 2008.

[27] K. Y. Yip and Mark Gerstein, "Training set expansion: An approach to improving the reconstruction of biological networks from limited and uneven reliable interactions," *Bioinformatics*, vol. 25, no. 2, pp. 243–250, 2009.

[28] D. W. Huang, B. T. Sherman, and R. A. Lempicki, "Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources," *Nature Protocols*, vol. 4, no. 1, pp. 44–57, 2009.
[29] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: a practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society. Series B*, vol. 57, no. 1, pp. 289–300, 1995.
[30] M. Medvedovic, K. Y. Yeung, and R. E. Bumgarner, "Bayesian mixture model based clustering of replicated microarray data," *Bioinformatics*, vol. 20, no. 8, pp. 1222–1232, 2004.
[31] K. Y. Yeung, M. Medvedovic, and R. E. Bumgarner, "Clustering gene-expression data with repeated measurements," *Genome Biology*, vol. 4, no. R34, 2004.
[32] 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*, vol. 292, no. 5518, pp. 929–934, 2001.

Kevin Y. Yip is an assistant professor in the Department of Computer Science and Engineering at the Chinese University of Hong Kong. He received his B.Engg. in Computer Engineering and M.Phil. in Computer Science from the University of Hong Kong, in 1999 and 2004, respectively. He received his Ph.D. in Computer Science from Yale University in 2009. His research interest is in bioinformatics, with a special focus on biological network inference and analysis using data mining and machine learning techniques.

Ben Kao received the B.Sc. degree in computer science from the University of Hong Kong in 1989 and the Ph.D. degree in computer science from Princeton University in 1995. He is currently a professor in the Department of Computer Science at the University of Hong Kong. From 1989 to 1991, he was a teaching and research assistant at Princeton University. From 1992 to 1995, he was a research fellow at Stanford University. His research interests include database management systems, data mining, real-time systems, and information retrieval systems.

Xinjie Zhu is a Research Assistant at the University of Hong Kong. He is expected to receive his M.Phil degree from the University of Hong Kong in 2011. He is interested in the research area of data mining, uncertain data management and bioinformatics.

Chun Kit Chui is a Ph.D. candidate at the University of Hong Kong (HKU). His research interests include data mining, data warehousing, cloud computing and uncertain data management.

Sau Dan Lee is a Post-doctoral Fellow at the University of Hong Kong. He received his Ph.D. degree from the University of Freiburg, Germany in 2006 and his M.Phil. and B.Sc. degrees from the University of Hong Kong in 1998 and 1995. He is interested in the research areas of data mining, machine learning, uncertain data management and information management on the WWW. He has also designed and developed backend software systems for e-Business and investment banking.

David W. Cheung received the M.Sc. and Ph.D. degrees in computer science from Simon Fraser University, Canada, in 1985 and 1989, respectively. Since 1994, he has been a faculty member of the Department of Computer Science in The University of Hong Kong. His research interests include database, data mining, database security and privacy. Dr. Cheung was the Program Committee Chairman of PAKDD 2001, Program Co-Chair of PAKDD 2005, Conference Chair of PAKDD 2007 and 2011, and the Conference Co-Chair of CIKM 2009.

A PROOFS

Lemma 1

Proof: Let H be a head array, and T_1 and T_2 be two arrays with $|T_1| = |T_2|$, where T_2 is produced by a push-right operation that moves a positive value v from the x-th entry of T_1 to the y-th entry, with $x < y$. Then,

$$\begin{aligned}
& \textit{HT-sum}(H, T_2) - \textit{HT-sum}(H, T_1) \\
= & \sum_{p=1}^{|H|} \sum_{q=1}^{H[p]} T_2[|T_2| - q + 1] - \sum_{p=1}^{|H|} \sum_{q=1}^{H[p]} T_1[|T_1| - q + 1] \\
= & \sum_{p=1}^{|H|} \left(\sum_{q=1}^{H[p]} T_2[|T_1| - q + 1] - \sum_{q=1}^{H[p]} T_1[|T_1| - q + 1] \right) \\
= & \sum_{p=1}^{|H|} \begin{cases} 0 & \textit{if } |T_1| - H[p] + 1 < x \\ 0 & \textit{if } |T_1| - H[p] + 1 > y \\ v & \textit{otherwise} \end{cases} \\
\geq & \ 0
\end{aligned}$$

Therefore the HT-sum is not reduced. □

Lemma 2

Proof: Let T' be an array with non-increasing entries, H_1 be a head array, and H_2 be an array with $|H_1| = |H_2|$, where H_2 is produced by a push-left operation that moves one from the x-th entry of H_1 to the y-th entry, with $x > y$. Due to the push-left operation and the non-increasing property of head arrays,

$$H_2[x] = H_1[x] - 1 < H_1[x] \leq H_1[y] < H_1[y] + 1 = H_2[y]$$

We have:

$$\begin{aligned}
& H_1[x] < H_2[y] \\
\Rightarrow \ & |T'| - H_1[x] + 1 > |T'| - H_2[y] + 1 \\
\Rightarrow \ & T'[|T'| - H_1[x] + 1] \leq T'[|T'| - H_2[y] + 1] \\
\Rightarrow \ & -T'[|T'| - H_1[x] + 1] + T'[|T'| - H_2[y] + 1] \geq 0
\end{aligned}$$

where the third line is due to the non-increasing property of T'. Now,

$$\begin{aligned}
& \textit{HT-sum}(H_2, T') - \textit{HT-sum}(H_1, T') \\
= & \sum_{p=1}^{|H_2|} \sum_{q=1}^{H_2[p]} T'[|T'| - q + 1] - \sum_{p=1}^{|H_1|} \sum_{q=1}^{H_1[p]} T'[|T'| - q + 1] \\
= & \sum_{p=1}^{|H_1|} \left(\sum_{q=1}^{H_2[p]} T'[|T'| - q + 1] - \sum_{q=1}^{H_1[p]} T'[|T'| - q + 1] \right) \\
= & \left(\sum_{q=1}^{H_2[x]} T'[|T'| - q + 1] - \sum_{q=1}^{H_1[x]} T'[|T'| - q + 1] \right) + \\
& \left(\sum_{q=1}^{H_2[y]} T'[|T'| - q + 1] - \sum_{q=1}^{H_1[y]} T'[|T'| - q + 1] \right) \\
= & -T'[|T'| - H_1[x] + 1] + T'[|T'| - H_2[y] + 1] \\
\geq & \ 0
\end{aligned}$$

Therefore the HT-sum is not reduced. □

Lemma 3

Proof: Without loss of generality, let us compare the averages of the first and second intervals. Each entry in the second interval corresponds to an occurrence of P[3..k-1] after the second occurrence of $P[2]$, which is in turn after the first occurrence of $P[2]$. Therefore each entry in the second interval has a corresponding entry in the first interval with the same value. The first interval may contain additional entries, corresponding to occurrences of P[3..k-1] where the $P[3]$ is before the second occurrence of $P[2]$. Since the entries are in lexicographic order, these additional entries must be the leftmost entries of the first interval. Let us call the additional entries the leading group and the remaining ones the trailing group. We will prove that the average of the leading group is no smaller than that of the trailing group, which is sufficient to show that the average of the first interval is not smaller than that of the second interval. We prove this proposition by mathematical induction.

Base case: k-1=3. As discussed, the entries in the leading group all have their P[k-1]=P[3] before the second occurrence of $P[2]$ while the entries in the trailing group all have their P[k-1] after it. Since the value of an entry equals the number of P[k]'s after its P[k-1], each entry in the leading group must be not smaller than every entry in the trailing group. The average of the leading group must therefore be not smaller than the average of the trailing group.

Inductive case: Now assume the proposition is true up to k-1=l, for some $l \geq 3$. For k-1=l+1, we transform the sequence by keeping only elements after the first occurrence of $P[2]$, and then remove all occurrences of $P[2]$ in the resulting subsequence. Then each entry in the first interval of the original sequence corresponds to the number of occurrences of $P[k]$ after a P[3..k-1] in this transformed sequence. We again partition the transformed sequence into intervals by grouping entries that share the same occurrence of P[3] together. If we can show the averages of these intervals are non-increasing, then certainly the average of the leading group, which is composed of the leftmost intervals, must be not smaller than the average of the trailing group, which is composed of the rightmost intervals. But this is exactly the inductive assumption. Therefore by mathematical induction, the proposition is true for all $k \geq 4$. □

Lemma 4(b)

Proof: We will prove that for each interval of T, we can use push-right operations to obtain the corresponding interval of T'. Again, we will use mathematical induction.

Base case: k-1=3. As proved in the base case of Lemma 3, the entries in the interval are non-increasing in T. We repeat the following: take the leftmost entry in the interval that is larger than the average, and use a push-right operation to move the difference to the next entry. The resulting interval will have all entries equal to the average, which is the same as the corresponding

interval in T'.

Inductive case: Now assume the proposition is true up to k-1=l, for some $l \geq 3$. For k-1=l+1, we first partition the entries in the interval of T into sub-intervals according to which $P[3]$ they refer to. As proved in the inductive case of Lemma 3, the averages of these sub-intervals are non-increasing. We use push-right operations to make them all have the same average as follows: repeatedly we pick the leftmost sub-interval with an average larger than the average of the whole interval. Then we move the difference from the last entry of the sub-interval to the first entry of the next sub-interval. After that, the sub-intervals all have an average equal to the average of the corresponding sub-intervals of T'. Therefore by the inductive assumption, each of these sub-intervals of T' can be obtained by push-right operations of the corresponding sub-interval of T. □

Lemma 7

Proof: For any pattern $P[1..k]$ and each row i, MinBound is composed of two parts due to the head $P[1..k\text{-}1]$ and the tail $P[2..k]$, with values $s_i(P[1..k\text{-}1])$ and $s_i(P[2..k])$ respectively. The part due to the head assumes the extreme case that each occurrence of the head is followed by $r(P[k])$ occurrences of $P[k]$ later in the sequence. It is interesting that this part of the bound, $s_i(P[1..k\text{-}1])$, can be obtained from HT-sum$(H, T^\dagger)$, where H is the actual head array of P and $T^\dagger$ is an array with the same number of entries as the actual tail array of P, but every entry takes the maximum allowed value $r(P[k])$ of the array:

$$\begin{aligned}
\text{HT-sum}(H, T^\dagger) &= \sum_{p=1}^{|H|} \sum_{q=1}^{H[p]} T^\dagger[|T^\dagger| - q + 1] \\
&= \sum_{p=1}^{|H|} \sum_{q=1}^{H[p]} r(P[k]) \\
&= r(P[k]) \sum_{p=1}^{|H|} H[p] \\
&= r(P[k]) sn_i(P[1..k-1])
\end{aligned}$$

where the last line is due to the sum constraint of the head array. Normalizing the HT-sum by the number of replicate combinations, we get the part of MinBound due to the head:

$$\begin{aligned}
& \frac{\text{HT-sum}(H, T^\dagger)}{\prod_{j=1}^{k} r(P[j])} \\
= & \frac{r(P[k]) sn_i(P[1..k-1])}{\prod_{j=1}^{k} r(P[j])} \\
= & \frac{sn_i(P[1..k-1])}{\prod_{j=1}^{k-1} r(P[j])} \\
= & s_i(P[1..k-1])
\end{aligned}$$

Since the bounding tail array T^* cannot contain any entry larger than the maximum, HT-sum(H^*, T^*) must not be larger than HT-sum$(H, T^\dagger)$:

$$\begin{aligned}
\text{HT-sum}(H^*, T^*) &= \sum_{p=1}^{|H^*|} \sum_{q=1}^{H^*[p]} T^*[|T^*| - q + 1] \\
&\leq \sum_{p=1}^{|H^*|} \sum_{q=1}^{H^*[p]} r(P[k]) \\
&= r(P[k]) \sum_{p=1}^{|H^*|} H^*[p] \\
&= r(P[k]) sn_i(P[1..k-1]) \\
&= \text{HT-sum}(H, T^\dagger)
\end{aligned}$$

Therefore the corresponding bound for $s_i(P)$ is also not larger than that from the part of MinBound due to the head.

Similarly, the part of MinBound due to the tail assumes the extreme case that each occurrence of the tail is preceded by $r(P[1])$ occurrences of $P[1]$ earlier in the sequence. This part of the bound, $s_i(P[2..k])$, can be obtained from HT-sum$(H^\dagger, T)$, where T is the actual tail array of P and $H^\dagger$ is an array with the same number of entries as the actual head array of P, but every entry takes the maximum allowed value $sn_i(P[2..k\text{-}1])$ of the array, which is also equal to the number of entries of T:

$$\begin{aligned}
\text{HT-sum}(H^\dagger, T) &= \sum_{p=1}^{|H^\dagger|} \sum_{q=1}^{H^\dagger[p]} T[|T| - q + 1] \\
&= \sum_{p=1}^{|H^\dagger|} \sum_{q=1}^{sn_i(P[2..k-1])} T[|T| - q + 1] \\
&= \sum_{p=1}^{|H^\dagger|} sn_i(P[2..k]) \\
&= r(P[1]) sn_i(P[2..k])
\end{aligned}$$

where the third line is due to the sum constraint of the tail array and the fourth line is due to the size constraint of the head array.

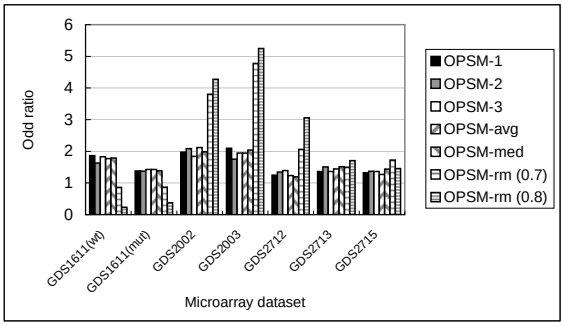
Again, we can show that it is no better (smaller) than HT-sum(H^*, T^*) :

$$\begin{aligned}
\text{HT-sum}(H^*, T^*) &= \sum_{p=1}^{|H^*|} \sum_{q=1}^{H^*[p]} T^*[|T^*| - q + 1] \\
&\leq \sum_{p=1}^{|H^*|} \sum_{q=1}^{sn_i(P[2..k-1])} T^*[|T^*| - q + 1] \\
&= \sum_{p=1}^{|H^*|} sn_i(P[2..k]) \\
&= r(P[1]) sn_i(P[2..k]) \\
&= \text{HT-sum}(H^\dagger, T)
\end{aligned}$$

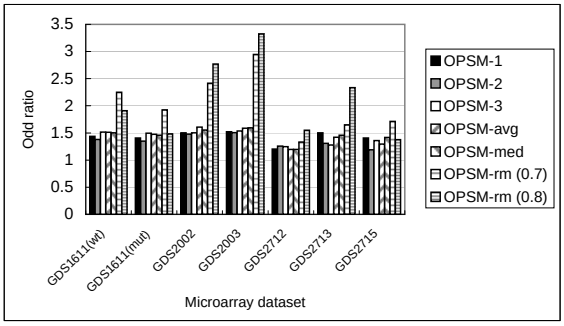
Combining the two parts of results, HTBound is always at least as tight as MinBound. □

B THE COMPLETE SET OF RESULTS FOR EVALUATING THE OPSM-RM MODEL

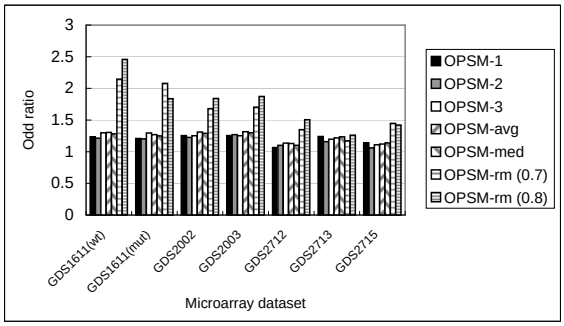
The following figures show the complete set of results with all combinations of microarray dataset, support threshold, evaluation method and protein interaction set. Notice that Figure 13(b) is the same as Figure 6.



(a) At $\rho = 5\%n$

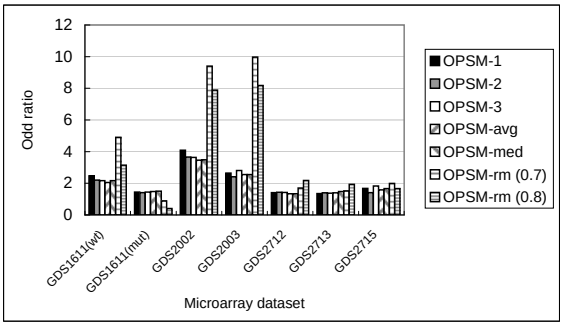


(b) At $\rho = 10\%n$

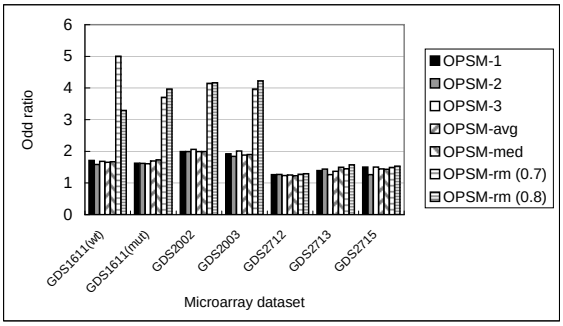


(c) At $\rho = 20\%n$

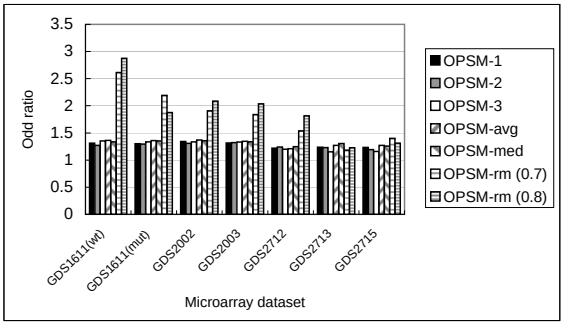
Fig. 13. Odd ratios of the patterns mined from various microarray datasets, evaluted by BioGRID-10



(a) At $\rho = 5\%n$

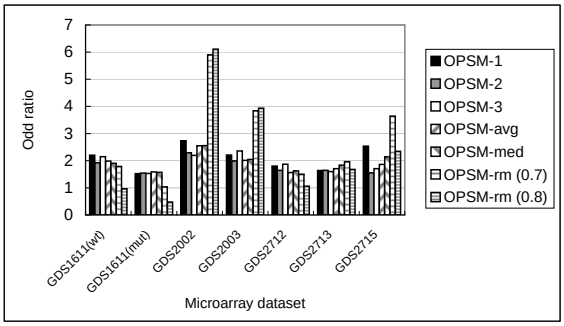


(b) At $\rho = 10\%n$

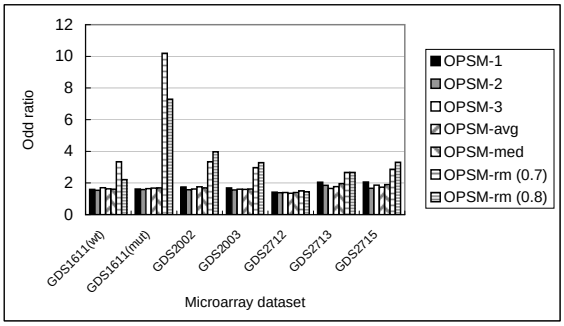


(c) At $\rho = 20\%n$

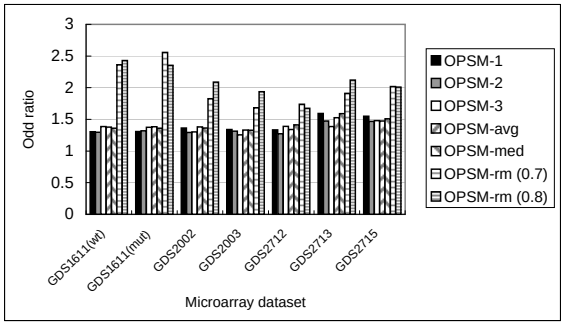
Fig. 14. Odd ratios of the patterns mined from various microarray datasets, evaluted by BioGRID-200



(a) At $\rho = 5\%n$

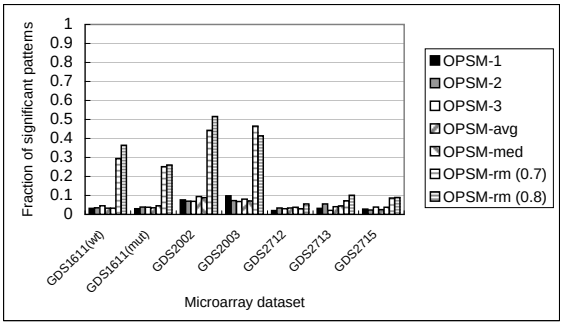


(b) At $\rho = 10\%n$

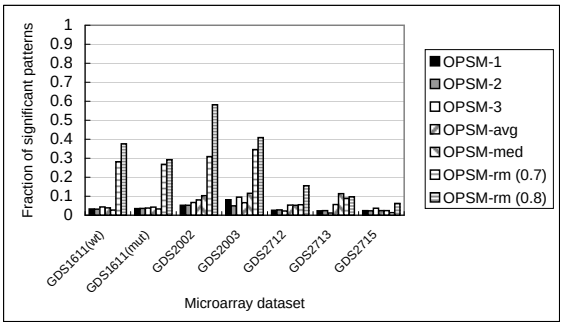


(c) At $\rho = 20\%n$

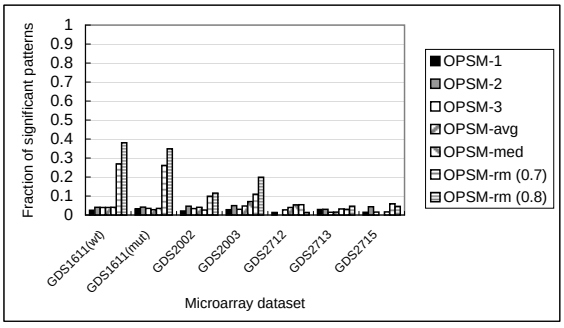
Fig. 15. Odd ratios of the patterns mined from various microarray datasets, evaluted by DIP-MIPS-iPfam



(a) At $\rho = 5\%n$

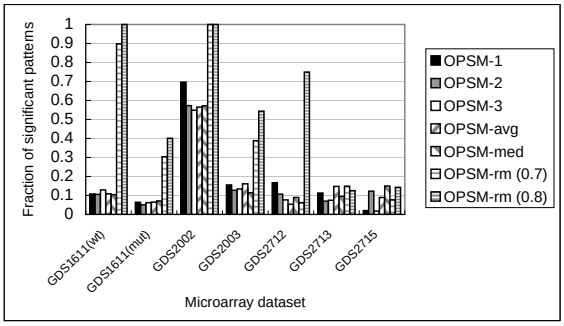


(b) At $\rho = 10\%n$

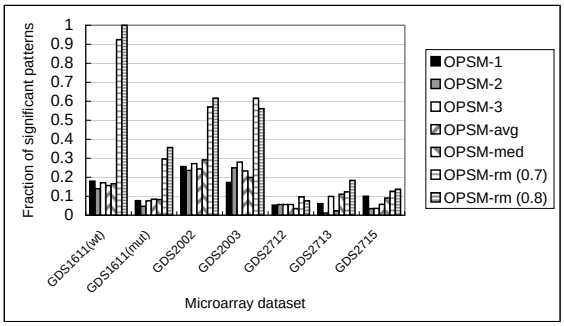


(c) At $\rho = 20\%n$

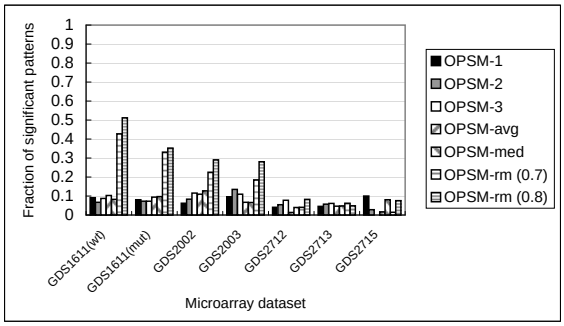
Fig. 16. Fraction of significant patterns mined from various microarray datasets, evaluted by BioGRID-10 with 50 genes sampled from each gene set



(a) At $\rho = 5\%n$

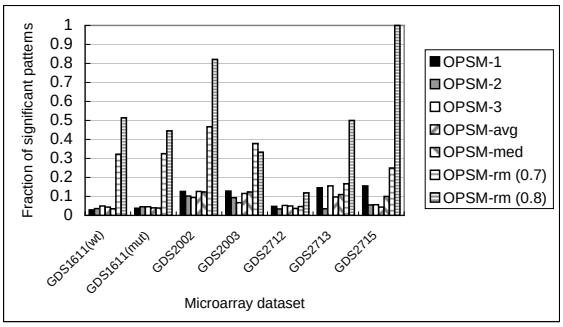


(b) At $\rho = 10\%n$

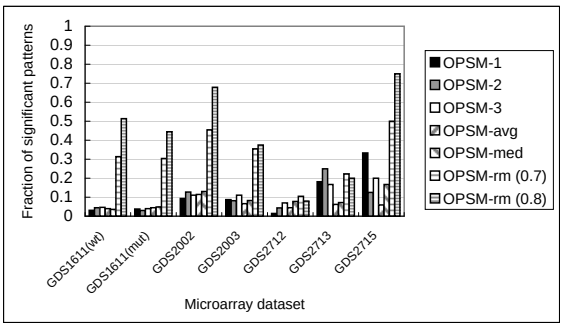


(c) At $\rho = 20\%n$

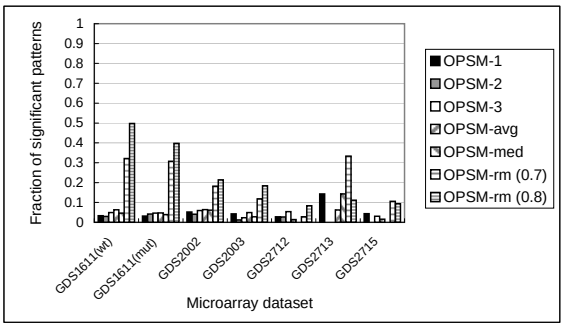
Fig. 17. Fraction of significant patterns mined from various microarray datasets, evaluted by BioGRID-200 with 50 genes sampled from each gene set



(a) At $\rho = 5\%n$

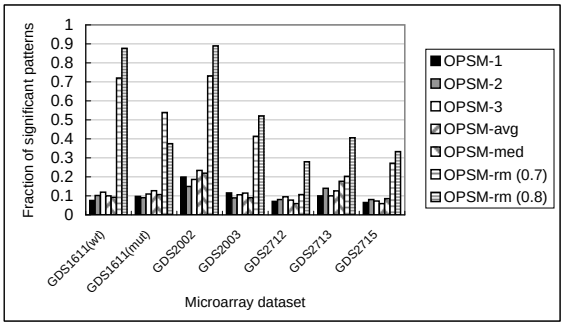


(b) At $\rho = 10\%n$

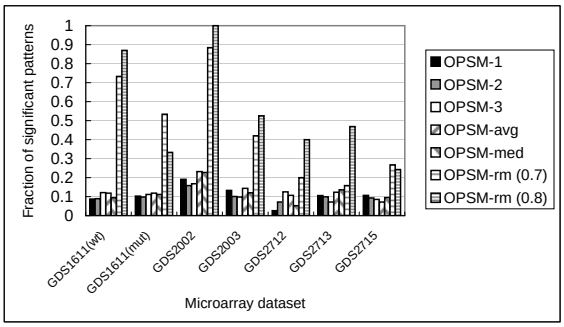


(c) At $\rho = 20\%n$

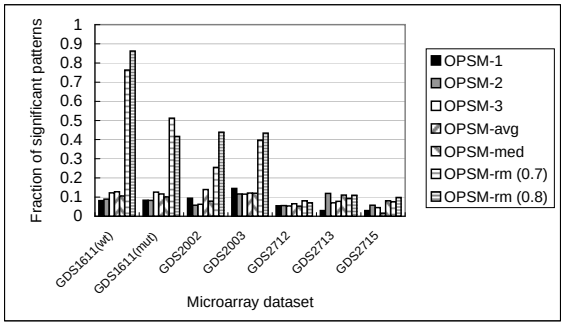
Fig. 18. Fraction of significant patterns mined from various microarray datasets, evaluted by DIP-MIPS-iPfam with 50 genes sampled from each gene set



(a) At $\rho = 5\%n$

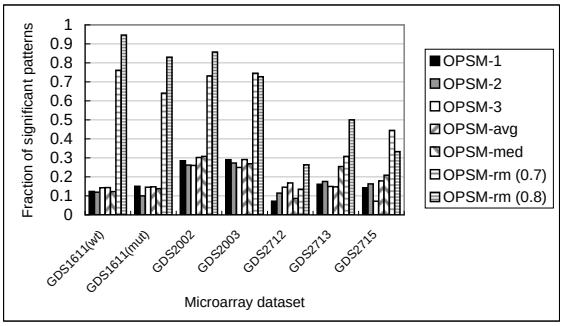


(b) At $\rho = 10\%n$

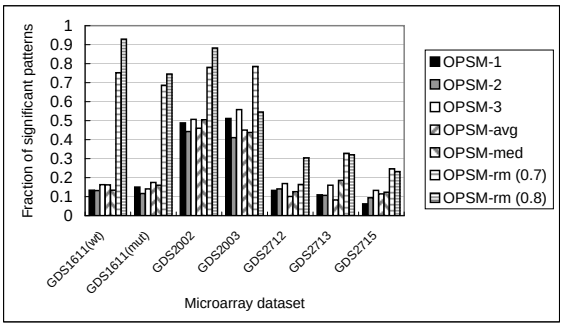


(c) At $\rho = 20\%n$

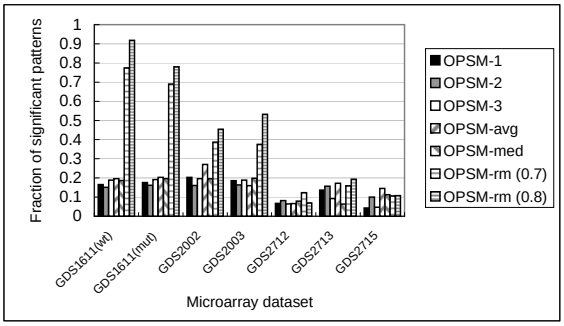
Fig. 19. Fraction of significant patterns mined from various microarray datasets, evaluted by BioGRID-10 with 100 genes sampled from each gene set



(a) At $\rho = 5\%n$

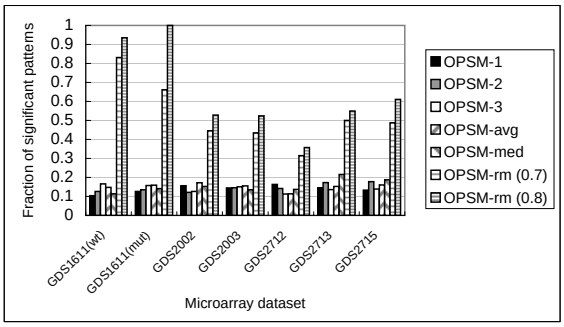


(b) At $\rho = 10\%n$

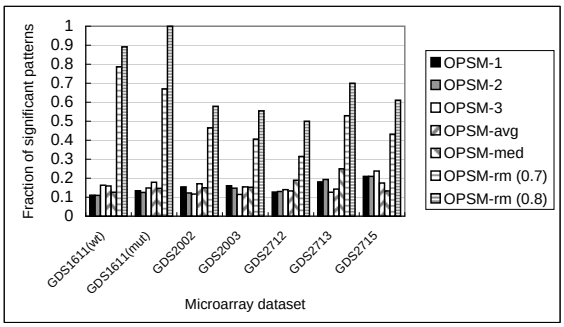


(c) At $\rho = 20\%n$

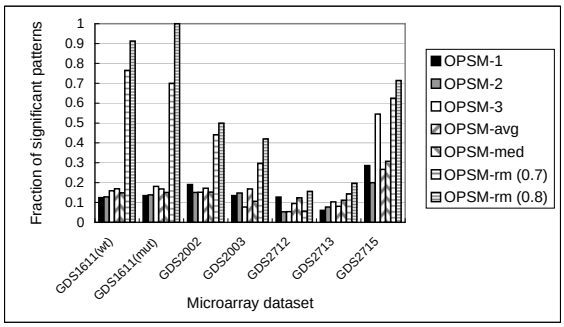
Fig. 20. Fraction of significant patterns mined from various microarray datasets, evaluted by BioGRID-200 with 100 genes sampled from each gene set



(a) At $\rho = 5\%n$



(b) At $\rho = 10\%n$



(c) At $\rho = 20\%n$

Fig. 21. Fraction of significant patterns mined from various microarray datasets, evaluted by DIP-MIPS-iPfam with 100 genes sampled from each gene set