Mining Order-preserving Submatrices under Data Uncertainty: A Possible-world Approach and Efficient Approximation Methods

JI CHENG, Department of Computer Science and Engineering, HKUST, Hong Kong
DA YAN, Department of Computer Science, The University of Alabama at Birmingham, United States
WENWEN QU, Shanghai Key Laboratory of Trustworthy Computing, ECNU, China
XIAOTIAN HAO, Department of Computer Science and Engineering, HKUST, Hong Kong
CHENG LONG, School of Comput. Sci. and Engineering, Nanyang Technological University, Singapore
WILFRED NG, Department of Computer Science and Engineering, HKUST, Hong Kong
XIAOLING WANG, Software Engineering Institute, East China Normal University, China

Given a data matrix D, a submatrix S of D is an order-preserving submatrix (OPSM) if there is a permutation of the columns of S, under which the entry values of each row in S are strictly increasing. OPSM mining is widely used in real-life applications such as identifying coexpressed genes and finding customers with similar preference. However, noise is ubiquitous in real data matrices due to variable experimental conditions and measurement errors, which makes conventional OPSM mining algorithms inapplicable. No previous work on OPSM has ever considered uncertain value intervals using the well-established possible world semantics.

We establish two different definitions of significant OPSMs based on the *possible world semantics*: (1) expected support-based and (2) probabilistic frequentness-based. An optimized dynamic programming approach is proposed to compute the probability that a row supports a particular column permutation, with a closed-form formula derived to efficiently handle the special case of uniform value distribution and an accurate cubic spline approximation approach that works well with any uncertain value distributions. To efficiently check the probabilistic frequentness, several effective pruning rules are designed to efficiently prune insignificant OPSMs; two approximation techniques based on the Poisson and Gaussian distributions, respectively, are proposed for further speedup. These techniques are integrated into our two OPSM mining algorithms, based on prefix-projection and Apriori, respectively. We further parallelize our prefix-projection-based mining algorithm using PrefixFPM, a recently proposed framework for parallel frequent pattern mining, and we achieve a good speedup with the number of CPU cores. Extensive experiments on real microarray

This work is partially supported by NSF OAC-1755464, NSF OAC-2106461, NSF DGE-1723250, NSFC grants (Nos. 62136002 and 61972155), the Science and Technology Commission of Shanghai Municipality (20DZ1100300) and Shanghai Knowledge Service Platform Project (No. ZF1213).
Authors' addresses: J. Cheng, X. Hao, and W. Ng, Department of Computer Science and Engineering, The Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; emails: {jchengac, xhao, wilfred}@cse.ust.hk; D. Yan, Department of Computer Science, The University of Alabama at Birmingham, University Hall 4105, 1402 10th Ave. S. Birmingham, AL 35294, United States; email: yanda@uab.edu; W. Qu and X. Wang, Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, 3663 N. Zhongshan Rd., Shanghai 200062, China; emails: wenwenqu@stu.ecnu.edu.cn, xlwang@cs.ecnu.edu.cn; C. Long, School of Computer Science and Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore 639798, Singapore; email: c.long@ntu.edu.sg.

0362-5915/2022/05-ART7 $15.00
https://doi.org/10.1145/3524915

data demonstrate that the OPSMs found by our algorithms have a much higher quality than those found by existing approaches.

CCS Concepts: • **Information systems** → **Uncertainty**; **Data mining**; • **Mathematics of computing** → **Probabilistic algorithms;**

Additional Key Words and Phrases: Order-preserving submatrices, OPSM, possible world semantics, expected support, probabilistic frequentness, data mining

ACM Reference format:
Ji Cheng, Da Yan, Wenwen Qu, Xiaotian Hao, Cheng Long, Wilfred Ng, and Xiaoling Wang. 2022. Mining Order-preserving Submatrices under Data Uncertainty: A Possible-world Approach and Efficient Approximation Methods. *ACM Trans. Database Syst.* 47, 2, Article 7 (May 2022), 57 pages.
https://doi.org/10.1145/3524915

1 INTRODUCTION

Order-preserving submatrix (OPSM) mining is an important data mining problem which given a data matrix, discovers a subset of attributes (columns) over which a subset of tuples (rows) exhibit a similar pattern of rises and falls in the tuples' values. It is useful in many real applications such as bioinformatics and customer segmentation.

For example, in bioinformatics, when analyzing gene expression data from microarray experiments, genes (rows) with simultaneous rises and falls of mRNA expression levels across different time points (columns) may share the same cell-cycle related properties [33]; columns may also represent different experimental conditions as in Reference [12]. In this application, an OPSM represents co-expressed patterns for large sets of genes, shared by a population of patients in a particular stage of a disease, or with the same drug treatment, and so on [4]. In fact, OPSM is well-known as the first bi-clustering method to overcome the drawback of clustering, which cannot identify patterns that are common to only a part of the expression data matrix [4].

As another example, in a customer-product rating matrix from a recommender system, each row (respectively, column) represents a customer (respectively, a product), and an OPSM represents a group of users with a similar product preference; mining such user groups enables the making of more effective advertising strategies. OPSM mining has also been successfully applied for analyzing indoor location tracking data [12], where visitors wearing RFID tags are tracked by RFID readers, and an OPSM represents a group of visitors who likely share a common visiting subroute.

Formally, OPSM mining considers a data matrix $D = (G, T)$ with a set of rows (e.g., genes) G and a set of columns (e.g., microarray tests) T. Each entry $D[g][t]$ of the matrix is a numerical value, e.g., the expression level of gene $g \in G$ under test $t \in T$. Consider the data matrix D shown in Figure 1. For simplicity, let us denote $D[g_i][t_j]$ by D_{ij}; then for row g_1, we have $D_{12} < D_{11} < D_{14} < D_{13}$, i.e., column value order is $t_2 < t_1 < t_4 < t_3$. Note that column value orders are also shown in Figure 1. Given a column permutation $t_1 < t_2 < t_3$, we can see from Figure 1 that rows g_2, g_3, and g_4 support this permutation while g_1 does not (since $D_{12} < D_{11}$).

An OPSM of an $n \times m$ matrix D is given by a pair (G', P), where G' is a subset of G, and $P = (t_{i_1}, t_{i_2}, \ldots, t_{i_\ell})$ is a permutation of a subset of T, such that for any $g_j \in G'$, $D_{ji_1} < D_{ji_2} < \cdots < D_{ji_\ell}$. Here, we say that g supports P and call P as the *pattern* of the OPSM. In Figure 1, (G', P) is an OPSM for $G' = \{g_2, g_3, g_4\}$ and $P = (t_1 < t_2 < t_3)$.

We are interested in those OPSMs with long patterns supported by sufficient rows, which exhibit statistical significance rather than occurring by chance. Given a row threshold τ_{row} and a column threshold τ_{col}, an OPSM (G', P) with $P = (t_{i_1} < t_{i_2} < \cdots < t_{i_\ell})$ is *significant* if $|G'| \geq \tau_{row}$ and $\ell \geq \tau_{col}$. We call ℓ as the length of pattern P. In other words, a significant OPSM has at least τ_{row} rows and τ_{col} columns.

	t_1	t_2	t_3	t_4	
g_1	49	38	115	82	$t_2 < t_1 < t_4 < t_3$
g_2	67	96	124	48	$t_4 < t_1 < t_2 < t_3$
g_3	65	67	132	95	$t_1 < t_2 < t_4 < t_3$
g_4	81	115	133	62	$t_4 < t_1 < t_2 < t_3$

Fig. 1. Gene expression matrix without noise.

	t_1	t_2	t_3	t_4
g_1	49, 55, 80	38, 51, 81	115, 101, 79	82, 110, 50
g_2	67, 54, 130	96, 85, 82	124, 92, 94	48, 37, 32
g_3	65, 49, 62	67, 39, 28	132, 119, 83	95, 89, 64
g_4	81, 83, 105	115, 110, 87	133, 108, 105	62, 52, 51

Fig. 2. Matrix with repeated measurements.

OPSM Mining on Noisy Matrices. Real data are often noisy. For example, in microarray tests, each value in the matrix is a physical measurement that is subject to measurement errors, variable experimental conditions, and instrumental limitations [12]. Also, a customer usually rates a product using discrete scores (e.g., 1–5 stars), and even if two products both gain 4 stars, a customer may prefer one over another, as each score actually represents a range of scores (e.g., [3.5, 4.5)).

Traditional OPSM mining algorithms are sensitive to such noise. For example, in Figure 1, if the value of D_{31} is slightly increased from 65 to 69, then g_3 will no longer support pattern $t_1 < t_2 < t_3$. One method to combat noise is **to sample each entry multiple times**, e.g., each microarray test can be repeated to record multiple measurements. Figure 2 illustrates a dataset with three repeated measurements (or replicates).

To handle such an expression data matrix, biologists usually take the average expression levels as the values in the matrix, to strike for higher data quality. OPSMRM [33] takes all the replicates into account and produces higher-quality OPSMs than those mined from the averaged matrix. OPSMRM is based on the possible world semantics, which assumes that each matrix entry is a random variable taking the value of each replicate with equal probability. For example, in the matrix D shown in Figure 2, D_{11} is assumed to take value 49, 55, or 80 with $33.\dot{3}\%$ probability. The OPSM significance is defined only based on "expected support" (see Section 2).

However, the data model of OPSMRM is restrictive. For D_{11} in Figure 2, if test t_1 is conducted on gene g_1 to get another measurement, then the result is very likely to be a value between 49 and 80 (e.g., 60), but not any of 49, 55, and 80. To address this issue, Reference [12] proposes the POPSM model, which converts the replicates for each entry in the matrix into an interval, and produced higher-quality OPSMs than OPSMRM. Figure 3 shows this **interval-based data model** for the data matrix shown in Figure 2. The interval model is sometimes even the only choice for representing data uncertainty, such as in Reference [12]'s RFID location tracking application where each row g of a matrix represents a loop-free object trajectory, each column t represents an RFID reader (i.e., a location), and each entry $D[g][t]$ records the time interval when g is detected by reader t. There, location (or subroute) uncertainty is generated when two readers detect the same object at the same time.

	t_1	t_2	t_3	t_4
g_1	[49, 80]	[38, 81]	[79, 115]	[50, 110]
g_2	[54, 130]	[82, 96]	[92, 124]	[32, 48]
g_3	[49, 65]	[28, 67]	[83, 132]	[64, 95]
g_4	[81, 105]	[87, 115]	[105, 133]	[51, 62]

Fig. 3. Gene expression matrix with intervals.

In the domain of uncertain database, the **possible world semantics (PWS)** is a robust probability model proven to be effective in handling data uncertainty in various applications [10], which we aim to follow. In PWS, a database is viewed as a set of deterministic instances (called possible worlds). As an illustration, Figure 1 is a possible world for Figure 2, occurring with a probability of $\frac{1}{3^{12}}$. Figure 1 is also a possible world for Figure 3, but unlike Figure 2 with finite and countable number of possible worlds, the number of possible worlds in Figure 3 is infinite and uncountable. In PWS, results obtained from probabilistic data are also probabilistic, to reflect the confidence placed on the mining results. Intuitively, we can mine OPSMs over each and every possible world and then aggregate their results based on their occurring probabilities to obtain the confidence for each OPSM to determine its significance. However, POPSM (the only OPSM mining model that considers interval-based value uncertainty) is not defined based on the possible world semantics.

This article studies how to mine OPSMs with the **interval-based uncertain model** based on the well-established **possible world semantics**, which is shown to generate higher-quality OPSMs than POPSM (cf. Section 7). A key challenge that we tackle is how to efficiently evaluate the OPSM significance without directly working on the infinite number of possible worlds, the latter of which is infeasible. Our contributions are summarized as follows:

- This is the first work that studies OPSM mining when matrix entry is modeled with interval and pattern significance is evaluated based on possible world semantics. This model is more robust than both OPSMRM and POPSM and is shown to generate higher-quality OPSMs.
- Under the possible world semantics, we study two different definitions of OPSM significance: (S1) expected support and (S2) probabilistic frequentness. Note that OPSMRM only considers "(S1)," while POPSM does not even follow the possible world semantics.
- A basic operation in our mining problem is to compute the probability that a row g supports a pattern $P = (t_{i_1} < t_{i_2} < \cdots < t_{i_\ell})$, which is challenging, since intervals $D[g][t_i]$ may overlap. We propose a **dynamic programming (DP)** algorithm to efficiently compute this probability. We further design a smart pay-as-you-go method to reuse DP computation when growing patterns.
- We derive a closed-form formula for computing the above-mentioned support probability when the value distribution is uniform in its interval, which enables very efficient evaluation. For more general value distributions, we propose an approximation approach based on cubic spline that is both accurate and generally applicable.
- Once the above probability is computed for all rows, we then propose efficient algorithms to check the significance of a pattern P under both "(S1)" and "(S2)." A few efficient pruning rules are checked to prune insignificant pattern P before the more expensive significance check for P. Checking "(S2)" is non-trivial and an efficient algorithm is designed based on **Fast Fourier Transform (FFT)** to reduce the time complexity compared with a straightforward method. Two linear-time approximation techniques are proposed to further improve the efficiency.

- Using the above algorithms to check the significance of each pattern P, two pattern-growth-based OPSM mining algorithms are proposed using different techniques: (1) prefix-projection and (2) Apriori. Both algorithms have pros and cons, but they both output higher-quality OPSMs than existing approaches using comparable running time if not better, as verified in our experiments.
- We further parallelize our prefix-projection-based mining algorithm using PrefixFPM [29, 30], a recently proposed general-purpose framework for parallel frequent pattern mining. Experiments show that the parallel algorithm achieves a good speedup with the number of CPU cores.

This manuscript is the journal extension of our previous conference paper. Due to the double-blind review requirement, we hereby summarize the differences from our conference paper as follows:

- We added a cubic spline approximation approach in Section 3.4 to support any value distributions in an uncertain value interval, while our conference paper only considered uniform value distributions.
- We added a parallel prefix-projection-based mining algorithm that achieves a good speedup with the number of CPU cores.
- We added two approximation techniques in Section 4.4 to speed up the examination of pattern frequentness, while our conference paper only had the exact algorithms.
- In Section 7, we have run experiments using more biology datasets such as GDS2712, as well as an RFID user trace dataset, in addition to the datasets in our conference paper. We also reported more extensive results using various metrics, and on more parameters, which were not included due to the space limitation of our conference paper.
- Also due to the space limitation, we had to omit the proofs of all our theorems and pruning rules in our conference paper. This journal version now includes all proofs in our online appendix to allow a more complete reading experience.
- This journal extension provides a more comprehensive related work section in Section 8 to better position our work in the context of data uncertainty research.

Article Organization. The rest of this article is organized as follows: Section 2 formally defines the concept of significant OPSMs under our interval-based uncertain data model, using expected support and probabilistic frequentness. Section 3 presents our dynamic programming algorithm to compute row supporting probability for a pattern P and the pay-as-you-go technique for computation reuse. Given the row supporting probabilities for P, Section 4 presents our algorithm for evaluating the significance of pattern P, rules for pruning insignificant patterns, and linear-time approximation of probabilistic frequentness. Then, Section 5 introduces our complete mining algorithms that grow patterns and examines their significance, and Section 6 presents a parallel version of our prefix-projection-based mining algorithm. Finally, Section 7 empirically compares our algorithms with existing algorithms, Section 8 reviews the related work, and Section 9 concludes this article.

2 PROBLEM DEFINITION

We assume that different rows of a data matrix are independent, and for each row, its different column entries are independent; we shall justify these assumptions in Sections 3 and 4. We mine significant OPSMs in two steps: (1) finding the frequent patterns with length at least τ_{col} and (2) selecting the rows that support each frequent pattern.

Above all, we need to first define pattern frequentness under the interval-based uncertain model. Given a pattern $P = (t_{i_1} < t_{i_2} < \cdots < t_{i_\ell})$, for each matrix row g, let us denote the probability that

g supporting P by p_g:

$$p_g = Pr\{\text{row } g \text{ supports pattern } P\}.$$

We call p_g the *supporting probability* hereafter, and we will discuss the computation of p_g in Section 3. To decide whether pattern P is frequent, we evaluate it using (i) the supporting probabilities of all rowsand (ii) the row threshold τ_{row}.

We define pattern significance using "expected support" and "probabilistic frequentness," two well-established semantics for defining pattern frequentness in mining uncertain data [26]. They both follow the possible world semantics.

Expected Support. Let us consider the expected number of rows that support pattern P. For each row g, we define a random variable X_g as follows:

$$X_g = \begin{cases} 1 & \text{if } g \text{ supports } P \\ 0 & \text{otherwise.} \end{cases}$$

Obviously, X_g follows the Bernoulli distribution and its expectation $E(X_g) = p_g$. The number of rows that support pattern P is a random variable $X = \sum_{g \in G} X_g$, and we have

$$E(X) = E\left(\sum_{g \in G} X_g\right) = \sum_{g \in G} E(X_g) = \sum_{g \in G} p_g.$$

Therefore, the expected support is simply the summation of p_g for all rows $g \in G$, and pattern P is frequent if and only if its expected support is not smaller than τ_{row}.

Probabilistic Frequentness. "Expected support" does not consider the distribution of X but merely its expectation. To be more accurate, "**probabilistic frequentness**" **(*p-frequentness*)** considers the **probability mass function (PMF)** of X.

Given a matrix D with row set $G = \{g_1, g_2, \ldots, g_n\}$, the support of P is depicted by the PMF of X, denoted as $f_P(c)$ where $c = 0, 1, \ldots, n$:

$$f_P(c) = Pr\{c \text{ rows in } D \text{ support pattern } P\}.$$

The PMF $f_P(c)$ can be computed using p_g of all rows $g \in G$ with the realistic assumption that rows are independent of each other, and we shall present more details in Section 4.

Let us denote the **cumulative distribution function (CDF)** by $F_P(c) = \sum_{i=0}^{c} f_P(i)$. Given a user-specified probability confidence threshold τ_{prob}, pattern P is probabilistically frequent (or *p-frequent*) if and only if

$$Pr\{X \geq \tau_{row}\} \geq \tau_{prob}, \tag{1}$$

where the L.H.S. can be represented as

$$Pr\{X \geq \tau_{row}\} = \sum_{c=\tau_{row}}^{n} f_P(c) = 1 - F_P(\tau_{row} - 1). \tag{2}$$

If we find that a pattern P is frequent, then we only output P if its length is at least τ_{col}.

Row Selection. The second step of OPSM mining is to select the rows that support each frequent pattern. Since the rows are considered as independent to each other, row g is favored over row g' if supporting probability $p_g > p_{g'}$.

There are several possible methods for selecting rows into an OPSM: (a) selecting k rows whose supporting probabilities are the highest; (b) selecting all rows whose supporting probabilities are at least τ_{cut}, where τ_{cut} is a user-specified "inclusion threshold" (different from τ_{prob} in

Table 1. Notations

Symbol	Meaning
$D = (G, T)$	Data matrix with row set G and column set T
$g \in G$, $t \in T$	A row of D, a column of D
$g_i \in G$, $t_j \in T$	The ith row of D, the jth column of D
$D_{ij} = D[g_i][t_j]$	The entry of D at Row i and Column j
n, m	The number of rows in D, the number of columns in D
P	A pattern of the form $P = (t_{i_1} < t_{i_2} < \cdots < t_{i_\ell})$
ℓ	The length of pattern P
S	A submatrix of D, or a subset of D's rows when P is given
τ_{row}, τ_{col}	The minimum numbers of rows and columns required for a result OPSM
p_g or $p_g(P)$	The probability that row g supports a given pattern P
τ_{prob}	The probability confidence threshold for a pattern to be probabilistically frequent
τ_{cut}	The inclusion threshold that selects all rows whose supporting probability is at least τ_{cut} to an OPSM
$\{I_i\}_{i=1}^{s}$	A subinterval partitioning for a row g (see Figure 4 for an illustration)
$I_i = [\ell(I_i), r(I_i)]$	The ith subinterval for a row g
$\Delta_i = r(I_i) - \ell(I_i)$	The length of subinterval I_i
$f_t(.)$	The probability density function (PDF) of $D[g][t]$ given a row g
$P_{I_k}(t_{i_1} < \cdots < t_{i_\ell})$	The probability of the event $t_{i_1} < t_{i_2} < \cdots < t_{i_\ell}$ with $t_{i_1}, \ldots, t_{i_\ell}$ located in I_k
$A[j][k]$	The probability of the event $t_{i_1} < \cdots < t_{i_j}$ with $t_{i_1}, \ldots, t_{i_j}$ located in the first k subintervals, i.e., $\bigcup_{i=1}^{k} I_i$
A	The dynamic programming array (DP-array)
X_g	The indicator variable on whether row g supports P
$X = \sum_{g \in G} X_g$	The support of P, which is a random variable
D_i	The first i rows of the data matrix D
$f_i(c)$	The probability mass function (PMF) of the support of P in D_i
S	A row subset of the rows of D
X^S	The support of P in S, which is a random variable
$f^S(c)$, $F^S(c)$	The PMF and (cumulative distribution function) CDF of the support of P in row set S
$cnt(P)$	Cardinality of the set $\{g \in G \mid p_g(P) > 0\}$

"p-frequentness"). Like Reference [12], we adopt the latter approach for row selection, since it is difficult to ask end-users to set a proper k for the former approach. For example, setting k as the expected support $\sum_{g \in G} p_g$ or as the row threshold τ_{row} is often an insufficient underestimate, since each of the k rows has $p_g(P) < 100\%$, while both support and τ_{row} are defined with respect to a deterministic possible world; however, it is unclear how much larger k should be set to be. Section 7.4 will compare the two row selection methods.

Notations. Table 1 summarizes the notations used throughout this article for a quick reference. While D and S denote the data matrix and a submatrix of it, respectively, when the context of a pattern $P = (t_{i_1} < t_{i_2} < \cdots < t_{i_\ell})$ is clear, we also use D and S to denote all the rows of the data matrix and a row subset, respectively.

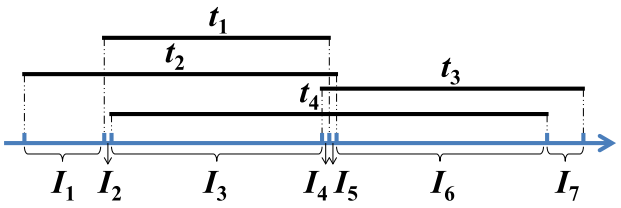


Fig. 4. Preprocessing of row g_1 in Figure 3.

3 SUPPORTING PROBABILITY COMPUTATION

In this section, we introduce how we compute the probability that a row g supports a pattern P, i.e., $p_g(P)$. We abbreviate it as p_g when P is clear from the context.

We compute $p_g(P)$ by dynamic programming. For ease of presentation, let us first assume that each element value in a data matrix follows uniform distribution within an interval. We will extend our dynamic programming method to handle arbitrary value distribution at the end of this section.

3.1 Preprocessing

Before applying the dynamic programming algorithm, we first need to preprocess each row $g = \langle[\ell_1, r_1], [\ell_2, r_2], \ldots, [\ell_m, r_m]\rangle$ to obtain a set of subintervals demarcated by ℓ_i, r_i $(i = 1, \ldots, m)$. These subintervals, denoted by $I_1, \ldots, I_s$, are ordered in increasing order of value. Note that $s \leq 2m$. We also denote the interval of I_i as $[\ell(I_i), r(I_i)]$.

Figure 4 shows the subintervals obtained by preprocessing row g_1 in the data matrix in Figure 3: $I_1 = [38, 49]$, $I_2 = [49, 50]$, $I_3 = [49, 79]$, $I_4 = [79, 80]$, $I_5 = [80, 81]$, $I_6 = [81, 110]$, and $I_7 = [110, 115]$.

Obviously, for each row g, given an interval element $D[g][t]$ and a subinterval I_i, we must have: either (1) $I_i \subseteq [\ell_t, r_t]$ or (2) $I_i \cap [\ell_t, r_t] = \emptyset$ or $\{\ell_t\}$ or $\{r_t\}$. For example, in Figure 4, $I_2 \subseteq t_1$, $I_2 \subseteq t_2$, $I_2 \cap t_3 = \emptyset$, and $I_2 \cap t_4 = r(I_2) = \ell_4 = \{50\}$.

Since each $D[g][t]$ is assumed to be uniform, we have:

Property 1. *Let $f_t(x)$ be the PDF of $D[g][t]$ on subinterval I_i, then we have*

$$f_t(x) = \begin{cases} 1/(r_t - \ell_t) & \text{if } I_i \subseteq [\ell_t, r_t], \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

Therefore, we can obtain the constant probability density of $D[g][t]$ on any subinterval I_i in $O(1)$ time by checking whether $I_i \subseteq [\ell_t, r_t]$. Since $f_t(x)$ is constant on I_i, we abbreviate it as f_t from now on. For example, in Figure 4, consider g_1 and subinterval I_2: $f_1 = 1/31$, since $I_2 \subseteq [49, 80]$, while $f_4 = 0$ except at the boundary 50 (which is immaterial for continuous distribution). As we shall see next, Property 1 is critical to the efficiency of computing supporting probability.

Assume data matrix D is $n \times m$, preprocessing each row into intervals $I_1, \ldots, I_s$ requires sorting $s \leq 2m$ values with the cost of $O(s \log s)$ time. Overall, processing the n rows of D takes $O(ns \log s) = O(nm \log m)$.

3.2 Dynamic Programming Formulation

Given a row g, a pattern $P = (t_{i_1} < t_{i_2} < \cdots < t_{i_\ell})$, and the subintervals $I_1, I_2, \ldots, I_s$ by preprocessing g, we compute $p_g(P)$ using **dynamic programming (DP)** as follows. We abuse the notation t_i to mean the interval $D[g][t_i]$, since g is given.

The DP algorithm first creates a 2D array A with ℓ rows and s columns. The element $A[j][k]$ denotes the probability of the event $t_{i_1} < \cdots < t_{i_j}$ with $t_{i_1}, \ldots, t_{i_j}$ located in the interval consisting of the first k subintervals, i.e., $\bigcup_{i=1}^{k} I_i$. Note that the value of $A[\ell][s]$ is exactly $p_g(P)$.

Our algorithm assumes that different column intervals of a row are independent, which is natural, since different microarray tests or RFID readers are usually independent. This assumption is used in many places below, such as the Proof of Theorem 3.1 and the last term's product in Equation (6).

Probability Evaluation on One Subinterval. Before describing the recursive formula for computing $A[j][k]$, we first explain how to compute the probability of the event $t_{i_1} < t_{i_2} < \cdots < t_{i_\ell}$ with $t_{i_1}, \ldots, t_{i_\ell}$ located in I_k. We denoted this probability by $P_{I_k}(t_{i_1} < \cdots < t_{i_\ell})$.

Theorem 3.1. *Given an interval $[\ell, r]$, let $\Delta = r - \ell$. If we have a set of random variables $x_1, x_2, \ldots, x_n$, where each variable x_i has constant probability density p_i on $[\ell, r]$, then $Pr\{(x_1, \ldots, x_n \in [\ell, r]) \wedge (x_1 < x_2 < \cdots < x_n)\} = \left(\prod_{i=1}^{n} p_i\right) \cdot \frac{\Delta^n}{n!}$.*

Proof. See Appendix A.1. □

Theorem 3.1 implies the following corollary for row g:

Corollary 3.2. *Let us define $\Delta_k = r(I_k) - \ell(I_k)$, and let f_{t_i} be the probability density of $D[g][t_i]$ on subinterval I_k as computed by Equation (3). Then,*

$$P_{I_k}(t_{i_1} < t_{i_2} < \cdots < t_{i_j}) = \left(\prod_{z=1}^{j} f_{t_{i_z}}\right) \cdot \frac{\Delta_k^j}{j!}. \tag{4}$$

Evaluation of $A[j][k]$. Now, we are ready to present the recursive formula for computing $A[j][k]$. Let us first consider the base case when $k = 1$. In this case,

$$A[j][1] = P_{I_1}(t_{i_1} < t_{i_2} < \cdots < t_{i_j}), \tag{5}$$

which can be computed using Equation (4).

When $k > 1$, the event "$t_{i_1} < t_{i_2} < \cdots < t_{i_j}$ with $t_{i_1}, t_{i_2}, \ldots, t_{i_j} \in \bigcup_{i=1}^{k} I_i$" can be decomposed into the following disjoint events:

- $t_{i_1} < \cdots < t_{i_j}$ with $t_{i_1}, \ldots, t_{i_j} \in \bigcup_{i=1}^{k-1} I_i$;
- $t_{i_1} < \cdots < t_{i_j}$ with $t_{i_1}, \ldots, t_{i_j} \in I_k$;
- $t_{i_1} < \cdots < t_{i_z}$ with $t_{i_1}, \ldots, t_{i_z} \in \bigcup_{i=1}^{k-1} I_i$, and $t_{i_{z+1}} < \cdots < t_{i_j}$ with $t_{i_{z+1}}, \ldots, t_{i_j} \in I_k$, where z can take values $1, 2, \ldots, j-1$.

According to the above discussion, we obtain

$$\begin{aligned} A[j][k] \quad = \quad & A[j][k-1] + P_{I_k}(t_{i_1} < \cdots < t_{i_j}) + \\ & \sum_{z=1}^{j-1} A[z][k-1] \cdot P_{I_k}(t_{i_{z+1}} < \cdots < t_{i_j}). \end{aligned} \tag{6}$$

In fact, if we define $A[j][0] = 0$ for any j, then we can even compute $A[j][1]$ using Equation (6). Note that computing $A[j][k]$ involves:

- The values $A[1][k-1], \ldots, A[j][k-1]$, which should have already been computed;
- Computing $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$ for $z = 1, \ldots, j$.

According to Equation (4), computing $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$ takes $O(j-z+1)$ time. Thus, computing $A[j][k]$ takes $\sum_{z=1}^{j} O(j - z + 1) = O(j^2)$ time if we compute each $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$ individually.

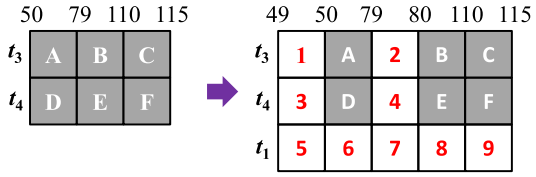


Fig. 5. Pay-as-you-go DP-array evaluation.

We can actually compute $A[j][k]$ in $O(j)$ time as follows: From Equation (4), we can derive the following recursive formula for computing $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$:

$$P_{I_k}(t_{i_z} < \cdots < t_{i_j}) = \begin{cases} f_{t_{i_j}} \cdot \Delta_k & \text{if } z = j \\ \frac{f_{t_{i_z}} \cdot \Delta_k}{j-z+1} \cdot P_{I_k}(t_{i_{z+1}} < \cdots < t_{i_j}) & \text{if } z < j. \end{cases} \tag{7}$$

Therefore, if we compute $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$ with z from j down to 1, then each $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$ can be computed from $P_{I_k}(t_{i_{z+1}} < \cdots < t_{i_j})$ using Equation (7) in O(1) time. Thus, computing $A[j][k]$ takes $\sum_{z=1}^{j} O(1) = O(j)$ time.

Accordingly, computing $p_g(P)$ requires computing all elements in the $\ell \times s$ array A, which takes $\sum_{j=1}^{\ell} \sum_{k=1}^{s} O(j) = O(\ell^2 s) = O(\ell^2 m)$ time (recall that $s \leq 2m$). We can further optimize the computation: If we find $f_{t_{i_z}} = 0$ when evaluating $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$, then we can terminate early, since according to Equation (4), $P_{I_k}(t_{i_{z'}} < \cdots < t_{i_j}) = 0$ for any $z' \leq z$.

3.3 The Pay-as-you-go Approach

From now on, let us call the array for dynamic programming as DP-array. We mine patterns by pattern-growth, i.e., pattern $t_{i_1} < \cdots < t_{i_j} < t_{i_{j+1}}$ is checked after pattern $t_{i_1} < \cdots < t_{i_j}$.

We now consider how to **reuse** the DP-array for $t_{i_1} < \cdots < t_{i_j}$ to compute the DP-array for $t_{i_1} < \cdots < t_{i_j} < t_{i_{j+1}}$.

In the base case, the pattern is a singleton t_i. Assume that the interval of $D[g][t_i]$ is $[\ell_i, r_i]$, then there is only one subinterval $I_1 = [\ell_i, r_i]$ and the DP-array is a 1×1 array with $A[1][1] = Pr_{I_1}(t_i) = 1$.

We now consider how to incrementally compute the DP-array of a pattern grown by one more interval. Referring to the data matrix in Figure 3 again, let us focus on the computation of $p_{g_1}(t_3 < t_4 < t_1)$.

Figure 5 illustrates the evaluation process, where the array on the left is the DP-array for pattern $t_3 < t_4$ that is already computed, and the array on the right is the DP-array for pattern $t_3 < t_4 < t_1$ that is to be computed. The split points of subintervals are also marked above the DP-arrays. Note that $A[j][k] = Pr\{(t_{i_1} < \cdots < t_{i_j}) \wedge (t_{i_1}, \ldots, t_{i_j} < r(I_k))\}$. For example, the value of cell E in the left array is the probability of the event $t_3 < t_4$ with $t_3, t_4 < 110$. Obviously, the value of cell E in the right array is exactly the same. In fact, we can copy the values of cells A–F in the left array directly into the corresponding cells in the right array.

However, for pattern $t_3 < t_4 < t_1$, the introduction of t_1 with interval $[49, 80]$ adds two more split points. For example, $I_2 = [79, 110]$ of the left array is now split into two subintervals for the right array: $I_3 = [79, 80]$ and $I_4 = [80, 110]$. However, the value of cell 4 (the probability of the event $t_3 < t_4$ with $t_3, t_4 < 80$), for example, is not computed in the left array, and therefore it has to be computed using Equation (6). In fact, the values of cells 1–9 have to be computed over the right array using Equation (6).

ALGORITHM 1: Computing DP-Array for $P = (t_{i_1} < \cdots < t_{i_j})$

Input: DP-Array A_0 for pattern $t_{i_1} < \cdots < t_{i_{j-1}}$; Split point list L_0 of array A_0; $D[g][t_{i_j}] = [\ell_{i_j}, r_{i_j}]$
Output: DP-Array A_1 for pattern $t_{i_1} < \cdots < t_{i_j}$; Split point list L_1 of array A_1

1: $\{L_0 = \{\ell(I_1^0), r(I_1^0), r(I_2^0), \ldots, r(I_{s_0}^0)\}$, starting with index 0$\}$
2: $L_1 \leftarrow$ merge_ordered_list$(L_0, \{\ell_{i_j}, r_{i_j}\})$
3: $\{L_1 = \{\ell(I_1^1), r(I_1^1), r(I_2^1), \ldots, r(I_{s_1}^1)\}\}$
4: {Compute the first $(j-1)$ rows of A_1}
5: **for** $row = 1$ **to** $j-1$ **do**
6: $\quad pos \leftarrow 1$ {Current pivot position in $A_0[row]$}
7: $\quad$ **for** $col = 1$ **to** s_1 **do**
8: $\quad\quad$ **if** $L_1[col] = L_0[pos]$ **then**
9: $\quad\quad\quad A_1[row][col] \leftarrow A_0[row][pos]$
10: $\quad\quad\quad pos \leftarrow pos + 1$
11: $\quad\quad$ **else**
12: $\quad\quad\quad$ Compute $A_1[row][col]$ using Equations (5) or (6)
13: {Compute the last row of A_1}
14: **for** $col = 1$ **to** s_1 **do**
15: $\quad$ Compute $A_1[j][col]$ using Equations (5) or (6)

Algorithm 1 shows how we compute the DP-array A_1 for pattern $t_{i_1} < \cdots < t_{i_j}$ from the DP-array A_0 for pattern $t_{i_1} < \cdots < t_{i_{j-1}}$ and the interval $[\ell_{i_j}, r_{i_j}]$ for t_{i_j}. First, the new ordered list of split points, L_1, is constructed by merging $\{\ell_{i_j}, r_{i_j}\}$ with the old split point list L_0 in Line 2, which takes $O(s_1)$ time ($s_1 = |L_1|$). Then, Lines 5–12 compute the first $(j-1)$ rows of A_1: If the column-right-marks of the elements $A_1[row][col]$ and $A_0[row][pos]$ are the same (Line 8), then the value of $A_1[row][col]$ is copied from $A_0[row][pos]$; otherwise, $A_1[row][col]$ has to be computed using the dynamic programming formula (Line 12). Finally, since A_0 does not have the jth row, elements $A_1[j][col]$ have to be computed using dynamic programming (Lines 14–15).

We now analyze the cost of this incremental computation of $p_g(t_{i_1} < \cdots < t_{i_j})$. Since at most two new split points ℓ_{i_j} and r_{i_j} are introduced when a pattern is grown with t_{i_j}, for each of the first $(j-1)$ rows in the DP-array, there are at most two elements to compute using dynamic programming. Together with the new jth row, there are totally $2(j-1) + (s-1) = O(j)$ elements (note that the number of split points $s < 2j$) to compute using dynamic programming, which takes $O(j^2)$ time. The remaining $(j-1) \cdot s$ elements are copied from the old DP-array, which takes $O(j \cdot s)$ time. Therefore, the total time complexity is $O(j^2)$, quadratic to the pattern length j.

For a pattern P of length ℓ, time complexity of computing $p_g(P)$ is reduced from $O(\ell^2 m)$ in Section 3.2 to $O(\ell^2)$ here.

3.4 Extension to Arbitrary Distributions

We have been assuming that each matrix entry conforms to a uniform distribution defined over its interval, which is a proper assumption to avoid inductive bias when there are only several replicates.

When there are sufficient number of replicates, we can infer the underlying distribution (e.g., Gaussian distribution) and learn the parameters from the replicates or fit the underlying distribution by kernel density estimation. Then, we can discretize the PDF using an equi-width or equi-depth histogram. Note that our dynamic programming framework is still applicable here, though each entry may now introduce more than two split points. In fact, if we discretize each distribution

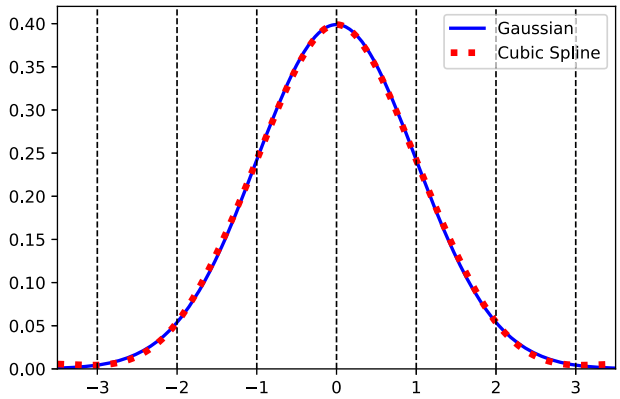


Fig. 6. Gaussian vs. cubic spline.

into n_s ranges (i.e., $n_s + 1$ split points), then for a pattern P of length ℓ, time complexity of computing $p_g(P)$ is increased from $O(\ell^2)$ in Section 3.3 to $O(n_s^2 \cdot \ell^2)$ here. The weakness of histogram-based approximation is that, the probability density within each range is treated as uniform, so we need to discretize a PDF with many small ranges to reach reasonable accuracy, but the time complexity of computing $p_g(P)$ grows quadratically with n_s.

We consider an alternative solution that fits a PDF with a cubic spline, which is a spline constructed of piecewise third-order polynomials passing through a set of control points. This approach is generally applicable to the PDF of an arbitrary distribution. We next illustrate this approach using the Gaussian distribution that is commonly used to model noises in measurements, as well as the exponential distribution as a second example. We also provide a discussion on how to choose appropriate control points for a general distribution.

Control Points for the Gaussian Distribution. Given a set of replicates for a matrix entry, we compute the sample mean μ and variance σ^2 to obtain a Gaussian distribution $\mathcal{N}(\mu, \sigma^2)$; we then fit its PDF with a cubic spline using critical points at $\mu - 4\sigma$, $\mu - 3\sigma$, $\mu - 2\sigma$, $\mu - \sigma$, 0, $\mu + \sigma$, $\mu + 2\sigma$, $\mu + 3\sigma$, $\mu + 4\sigma$. Figure 6 illustrates the PDF of $\mathcal{N}(0, 1)$ as well as the cubic spline fit from the PDF's nine critical points; we can see that the cubic spline matches tightly with the Gaussian PDF, so the probability approximation is much more accurate. Also, since 99.73% values in a Gaussian distribution lies within $[\mu - 3\sigma, \mu + 3\sigma]$, we only consider the six intervals $[\mu - 3\sigma, \mu - 2\sigma]$, $[\mu - 2\sigma, \mu - \sigma]$, $[\mu - \sigma, 0]$, $[0, \mu + \sigma]$, $[\mu + \sigma, \mu + 2\sigma]$, $[\mu + 2\sigma, \mu + 3\sigma]$ when computing $p_g(P)$, but we also use $\mu \pm 4\sigma$ as critical points to ensure smooth fitting on the border of $\mu \pm 3\sigma$. For each matrix entry $D[g][t_j]$, we now have seven split points $\mu - 3\sigma$, $\mu - 2\sigma$, $\mu - \sigma$, 0, $\mu + \sigma$, $\mu + 2\sigma$, $\mu + 3\sigma$, and the PDF between any two consecutive split points is determined by a unique third-order polynomial $p^{(j)}(x) = a_0^{(j)} + a_1^{(j)}x + a_2^{(j)}x^2 + a_3^{(j)}x^3$.

Control Points for the Exponential Distribution. Given a set of replicates for a matrix entry, $\{v_1, v_2, \ldots, v_k\}$, we compute shift parameter $min = \min_i\{v_i\}$, shifted mean $\mu = \frac{1}{N}\sum_i(v_i - min)$, and rate parameter $\lambda = \frac{1}{\mu}$ to obtain an exponential distribution $Exp(\lambda, min)$ whose PDF is given by $\lambda e^{-\lambda(x-min)}$ $(x \geq min)$. Figure 7 illustrates the PDF of a standard exponential distribution $Exp(1, 0)$ as well as the cubic spline fit from the PDF's six critical points at 0, μ, 2μ, 3μ, 4μ, and 5μ. Note that 99% values of an exponential distribution lies within $[0, 4.61\mu]$, so we only consider five intervals $[0, \mu]$, $[\mu, 2\mu]$, $[2\mu, 3\mu]$, $[3\mu, 4\mu]$, and $[4\mu, 5\mu]$ when computing $p_g(P)$. For each matrix entry $D[g][t_j]$, we now have six split points min, $min + \mu$, $min + 2\mu$, $min + 3\mu$, $min + 4\mu$, $min + 5\mu$, and the

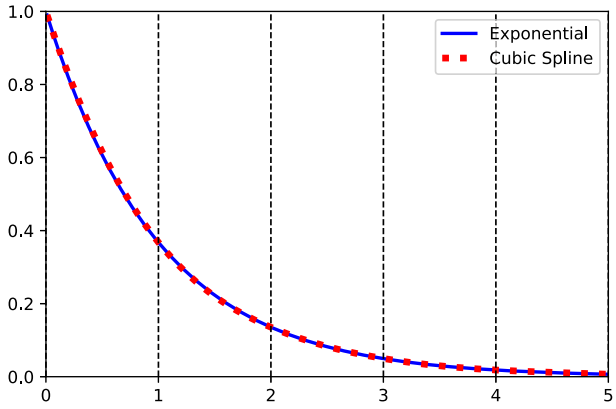


Fig. 7. Exponential vs. cubic spline.

PDF between any two consecutive split points is determined by a unique third-order polynomial $p^{(j)}(x) = a_0^{(j)} + a_1^{(j)}x + a_2^{(j)}x^2 + a_3^{(j)}x^3$.

Guidelines on Choosing Control Points for a General Distribution. Now that we have seen how to choose good control points for Gaussian and exponential distributions, we are ready to discuss how to choose proper control points for a general distribution. We remark that in our context, the most reasonable distributions are unimodal distributions where most probability density is clustered around the ground-truth value of a matrix entry, such as the exponential family that comprises the natural sets of distributions to consider in reality. For such a general distribution, the probability density drops quickly as the value increases beyond the mean value, so we only need to focus on fitting the limited value range that accounts for the vast majority of the value possibility (e.g., 99% or more), since ignoring the rest would not compromise much accuracy. For this value range considered, we can either divide the range evenly as we do for Gaussian and exponential distributions or by selecting intervals in sequence where each interval is selected by increasing its length till the point when a goodness-of-fit metric drops below a user-defined threshold (binary search applies here for efficiency). The number of intervals or the goodness-of-fit threshold can be adjusted according to users' accuracy needs, but there is an efficiency-accuracy tradeoff that users need to consider. However, as we have shown in Figures 6 and 7, a small number of five to six intervals is often sufficient to achieve very good fitting quality for a common distribution falling into the exponential family.

Computing $p_g(P)$. We can still use the same computation framework of Algorithm 1 in Section 3.3 when $D[g][t_j] \sim \mathcal{N}(\mu, \sigma^2)$ (or when $D[g][t_j] \sim Exp(\lambda, min)$). One difference is that Line 2 now merges the split points in L_0 with the seven (or six) split points of $D[g][t_j]$ to obtain the new split point list L_1. For any two consecutive split points ℓ, r in L_1, the PDF on the interval $[\ell, r]$ is defined by a unique third-order polynomial for each of $t_{i_1}, t_{i_2}, \ldots, t_{i_j}$.

The other difference in Algorithm 1 is the computation of $A_1[row][col]$ as specified in Lines 12 and 15, which uses Equations (5) or (6) where the key operation is to compute the probability $P_{I_k}(t_{i_1} < t_{i_2} < \cdots < t_{i_j})$ on subinterval $I_k = [\ell(I_k), r(I_k)]$, with $\ell(I_k)$ and $r(I_k)$ being two consecutive points in L_1. We next discuss its computation assuming $p^{(j)}(x) = a_0^{(j)} + a_1^{(j)}x + a_2^{(j)}x^2 + a_3^{(j)}x^3$:

$$\begin{aligned} & Pr\{(x_1, \ldots, x_n \in [\ell, r]) \wedge (x_1 < x_2 < \cdots < x_n)\} \\ = & \int_\ell^r \left[\int_\ell^{x_n} \left(\cdots \int_\ell^{x_2} p^{(1)}(x_1)\, dx_1 \cdots \right) p^{(n-1)}(x_{n-1})\, dx_{n-1} \right] p^{(n)}(x_n)\, dx_n. \end{aligned} \tag{8}$$

Even though we cannot derive a closed form formula as Equation (4) to evaluate Equation (8), Equation (8) can still be computed analytically to avoid expensive numerical evaluation. Specifically, let us define an order-k polynomial $poly(x, \mathbf{a}) = \sum_{i=0}^{k} a_i x^i$, which we implement as a class *Polynomial* that maintains an array of coefficients $\mathbf{a} = [a_0, a_1, \ldots, a_k]$ and that supports two operations: (1) multiplying another polynomial (which generates a new polynomial with a new coefficient array) and (2) computing the following integral where y is a variable while ℓ is a constant:

$$\begin{aligned}\int_{\ell}^{y} poly(x, \mathbf{a})\, dx &= \int_{\ell}^{y} \left(\sum_{i=0}^{k} a_i x^i\right) dx = \sum_{i=0}^{k} \left(a_i \int_{\ell}^{y} \frac{d\, x^{i+1}}{i+1}\right) \\ &= \sum_{i=0}^{k} \left(\frac{a_i}{i+1} \int_{\ell}^{y} d\, x^{i+1}\right) = \sum_{i=0}^{k} \frac{a_i \cdot (y^{i+1} - \ell^{i+1})}{i+1} \\ &= \sum_{i=0}^{k} \left(b_{i+1} \cdot y^{i+1} - c_i\right) = b_0 + \sum_{i=1}^{k+1} b_i \cdot y^i, \end{aligned} \tag{9}$$

where b_i and c_i are constants. In other words, the integral over $poly(x, \mathbf{a}) = \sum_{i=0}^{k} a_i x^i$ gives a new order-$(k+1)$ polynomial $poly(y, \mathbf{b}) = \sum_{i=0}^{k+1} b_i y^i$.

Using Equation (9), we can compute Equation (8) recursively from x_1 all the way till x_n as follows:

- **Base Case:** $\int_{\ell}^{x_2} p^{(1)}(x_1)\, dx_1 = \int_{\ell}^{x_2} (\sum_{i=0}^{3} a_i^{(1)} x_1^i)\, dx_1$ gives an order-4 polynomial $poly(x_2, \mathbf{b}^{(2)}) = \sum_{i=0}^{4} b_i^{(2)} x_2^i$ according to Equation (9).
- **Inductive Step:** assume that we already obtained ($j = 2$ in the base case):

$$\int_{\ell}^{x_j} \left(\cdots \int_{\ell}^{x_2} p^{(1)}(x_1)\, dx_1 \cdots\right) p^{(j-1)}(x_{j-1})\, dx_{j-1} = \sum_{i=0}^{4(j-1)} b_i^{(j)} x_j^i,$$

where $b_i^{(j)}$ are constants. Then, we have

$$\begin{aligned} &\int_{\ell}^{x_{j+1}} \left[\int_{\ell}^{x_j} \left(\cdots \int_{\ell}^{x_2} p^{(1)}(x_1)\, dx_1 \cdots\right) p^{(j-1)}(x_{j-1})\, dx_{j-1}\right] p^{(j)}(x_j) dx_j \\ = &\int_{\ell}^{x_{j+1}} \left(\sum_{i=0}^{4(j-1)} b_i^{(j)} x_j^i\right)\left(\sum_{i=0}^{3} a_i^{(j)} x_j^i\right) dx_j = \int_{\ell}^{x_{j+1}} \left(\sum_{i=0}^{4j-1} c_i^{(j)} x_j^i\right) dx_j \end{aligned} \tag{10}$$

$$= \sum_{i=0}^{4j} b_i^{(j+1)} x_{j+1}^i, \tag{11}$$

where $c_i^{(j)}$ in Equation (10) are constants, and Equation (11) is derived according to Equation (9). Finally, setting $j = n$ in Equation (11) and using the fact that $x_{n+1} = r$ in Equation (8), we can compute

$$Pr\{(x_1, \ldots, x_n \in [\ell, r]) \wedge (x_1 < x_2 < \cdots < x_n)\} = \sum_{i=0}^{4n} b_i^{(n+1)} x_{n+1}^i = \sum_{i=0}^{4n} b_i^{(n+1)} r^i. \tag{12}$$

Regarding the time complexity, evaluating the polynomial multiplication in Equation (10) takes $O([4(j-1)+1] \times [3+1]) = O(j)$ time, and evaluating the integral in Equation (10) to obtain Equation (11) also takes $O(j)$ time. So, evaluating the probability in Equation (12) takes $O(n)$ time,

so according to Equation (6), computing $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$ takes $O(j-z+1)$ time. Thus, computing $A[j][k]$ takes $\sum_{z=1}^{j} O(j-z+1) = O(j^2)$ time, as we compute each $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$ individually.

Accordingly, computing $p_g(P)$ requires computing all elements in the $\ell \times s$ array A, which takes $\sum_{j=1}^{\ell}\sum_{k=1}^{s} O(j^2) = O(\ell^3 s) = O(\ell^3 m)$ time, where the number of split points $s \le 7m$ in our Gaussian approximation scheme in Figure 6, and $s \le 6m$ in our exponential approximation scheme in Figure 7. We can further optimize the computation:If I_k is outside the $[-3\sigma, 3\sigma]$-interval (of the Gaussian distribution) or $[0, 5\mu]$-interval (of the exponential distribution) of some column value t_{i_z} when we evaluate $P_{I_k}(t_{i_z} < \cdots < t_{i_j})$, then we can terminate early, since we have $P_{I_k}(t_{i_{z'}} < \cdots < t_{i_j}) = 0$ for any $z' \le z$.

4 PATTERN SIGNIFICANCE CHECKING

In Section 3, we have already described how to compute the supporting probability of row g for pattern P (i.e., $p_g(P)$). This section explains how to evaluate the frequentness of a pattern P given the supporting probabilities all rows.

As discussed in Section 2, it is straightforward to check whether a pattern P is frequent in terms of *expected support*: We just check whether $\sum_{g \in G} p_g(P) \ge \tau_{row}$, which takes $O(n)$ time. Therefore, this section focuses on explaining how we determine whether a pattern P is *probabilistically frequent*.

We assume that different rows (e.g., genes, customers/visitors) in the data matrix are independent of each other. This is a reasonable assumption similar to tuple independence in uncertain databases and data point independence in machine learning and simplifies probability computations.

4.1 Pattern Support PMF Computation

We define the support of a pattern P in data matrix $D = (G, T)$ as the number of rows in G that supports P. Since each row g supports P only with probability $p_g(P)$, the support of P is a random variable, denoted as X.

We now consider how to compute the PMF of X, using the supporting probabilities $p_g(P)$ of all rows $g \in G$.

Naïve Method. Let $f_i(c)$ be the PMF of the support of P in matrix D_i, which consists of the first i rows in D, where $c = 0, 1, \ldots, i$ ($f_i(c) = 0$ for other values of c). Then, we have the following recursive formula:

$$f_{i+1}(c) = p_{g_{i+1}} \cdot f_i(c-1) + (1 - p_{g_{i+1}}) \cdot f_i(c). \tag{13}$$

This is because P's support in D_{i+1} is c, iff (1) P's support in D_i is $(c-1)$ and g_{i+1} supports P, or (2) P's support in D_i is c and g_{i+1} does not support P. Note that Equation (13) holds only for $c = 1, \ldots, i$, and the products of probabilities are due to row independence (the same for later equations and thus omitted).

When $c = 0$, we have

$$f_{i+1}(0) = (1 - p_{g_{i+1}}) \cdot f_i(0), \tag{14}$$

since the support of P in D_{i+1} is 0, if and only if the support of P in D_i is 0 and g_{i+1} does not support P.

When $c = i + 1$, we have

$$f_{i+1}(i+1) = p_{g_{i+1}} \cdot f_i(i), \tag{15}$$

since the support of P in D_{i+1} is $(i+1)$, if and only if the support of P in D_i is i and g_{i+1} supports P.

Furthermore, we have the base case

$$f_0(0) = 1, \tag{16}$$

since there is no row in D_0 and the support of P must be 0.

For a data matrix D with n rows, the PMF of X, denoted by $f_P(c)$, is equal to $f_n(c)$ $(c = 0, \ldots, n)$, since $D = D_n$. To compute the PMF $f_P(c)$, we start with $f_0(c)$ and recursively compute $f_{i+1}(c)$ from $f_i(c)$ until $f_n(c)$ is computed. According to Equations (13), (14), and (15), it takes $O(i)$ time to compute f_i and thus $O(n^2)$ time to compute $f_P(c)$.

Divide-and-conquer Algorithm. The naïve method takes time quadratic to the number of rows, which does not scale well. We now describe another algorithm for computing the support PMF $f_P(c)$, which achieves better time complexity by the divide-and-conquer strategy.

Given a set S of rows, let us define $f^S(c)$ as the PMF of the support of P in row set S, then our ultimate goal is to compute $f^D(c)$. To compute $f^S(c)$, we first divide the rows in S into two sets S_1 and S_2 of equal size. Let us denote $|S|$ by n, then S_1 contains the first $\lfloor \frac{n}{2} \rfloor$ rows, and S_2 contains the remaining $\lceil \frac{n}{2} \rceil$ rows.

Assume that $f^{S_1}(c)$ and $f^{S_2}(c)$ are already computed, then $f^S(c)$ can be obtained by the following formula:

$$f^S(c) = \sum_{i=0}^{c} f^{S_1}(i) \times f^{S_2}(c-i), \tag{17}$$

since the event that "the support of P in S is c" can be decomposed into the disjoint events "the support of P in S_1 is i, and the support of P in S_2 is $(c-i)$" for $i = 0, \ldots, c$.

Note that $|S| = n$, while $|S_1| = \lfloor \frac{n}{2} \rfloor$ and $|S_2| = \lceil \frac{n}{2} \rceil$. In Equation (17), we define $f^{S_1}(c) = 0$ for $c > \lfloor \frac{n}{2} \rfloor$, and define $f^{S_2}(c) = 0$ for $c > \lceil \frac{n}{2} \rceil$.

According to Equation (17), f^S is the convolution of f^{S_1} and f^{S_2}. Therefore, f^S can be computed from f^{S_1} and f^{S_2} in $O(n \log n)$ time using ***Fast Fourier Transform*** **(FFT)** [8].

Our divide-and-conquer algorithm for computing f^S is described as follows: S is first divided into two row sets S_1 and S_2 of equal size; then, PMFs f^{S_1} and f^{S_2} are computed by recursion; finally, f^S is computed as the convolution of f^{S_1} and f^{S_2} using FFT. Since each recursion step takes $O(n \log n)$ time (due to FFT), the overall time complexity of computing $f^D(c)$ is $O(n \log^2 n)$.

The base case for recursion is when $S = \{g\}$, in which case, we directly return f^S where $f^S(0) = 1 - p_g$ and $f^S(1) = p_g$. In reality, we find that recursion down to $|S| = 1$ does not provide the best performance. The most efficient configuration is to stop recursion when $|S| \leq 500$ and compute f^S directly using the naïve method. We adopted this implementation.

4.2 Early Frequentness Validation

In the previous subsection, we described how to compute the PMF of the support of pattern P in data matrix D. Once the PMF is computed, we can decide whether pattern P is p-frequent using Equations (1) and (2) in Section 2.

However, this two-step approach is time-consuming, since it requires to compute the whole PMF vector $f_P(c)$, $c = 0, \ldots, n$. In fact, to determine whether pattern P is frequent, it is not always necessary to compute f_P to the end. The theorem below states that pattern P is frequent in D, as long as it is found to be frequent in a subset of D. As a result, in our divide-and-conquer algorithm, in each recursion step (that computes f^S), we will check the frequentness of P over S using Equations (1) and (2), and if it is found to be frequent, we terminate the frequentness checking immediately and conclude that P is frequent.

Theorem 4.1. *Suppose that pattern P is p-frequent in $S' \subseteq S$, then P is also p-frequent in S.*

Proof. See Appendix A.2. □

4.3 Pattern Pruning

The frequentness checking operations described above is still very expensive (i.e., $O(n \log^2 n)$ time). We now present three pruning rules for pruning infrequent patterns. These rules can be checked efficiently in $O(n)$ time, and if any rule determines that a pattern P is infrequent, we do not need to do the expensive frequentness checking for P. The proofs of these rules can be found in Appendix A.3.

(1) Count-prune. Let $cnt(P) = |\{g \in G \mid p_g(P) > 0\}|$, then pattern P is not p-frequent if $cnt(P) < \tau_{row}$.

(2) Markov-prune. Pattern P is not p-frequent if

$$\sum_{g \in G} p_g(P) = E(X) < \tau_{row} \times \tau_{prob}.$$

(3) Exponential-prune. Let $\mu = E(X)$ and $\delta = \frac{\tau_{row} - \mu - 1}{\mu}$. When $\delta > 0$, pattern P is not p-frequent if

(1) $\delta \geq 2e - 1$, and $2^{-\delta\mu} < \tau_{prob}$, or

(2) $0 < \delta < 2e - 1$, and $e^{-\frac{\delta^2 \mu}{4}} < \tau_{prob}$.

4.4 PMF Approximation

To further improve the efficiency of examining pattern frequentness using probabilistic frequentness, instead of computing the exact PMF vector, we can use the Poisson or Gaussian distribution to approximate the PMF, which reduces the time complexity from $O(n \log^2 n)$ to $O(n)$. We next describe the two PMF approximation techniques using Poisson and Gaussian distributions, respectively.

4.4.1 Approximation by Poisson Distribution. Given an uncertain data matrix with n rows, each OPSM pattern P is associated with n supporting probabilities $p_g(P)$, one for each row $g \in G$. Here, each probability $p_g(P)$ conforms to an independent Bernoulli distribution representing if the row g supports pattern P. Since the rows are independent to each other, the n events {row g supports pattern P} for all rows $g \in G$ represent n Poisson trials. Let us denote the support of P by random variable X, then X follows a Poisson-binomial distribution.

Our probabilistic frequentness of an OPSM is given by

$$Pr\{X \geq \tau_{row}\} = 1 - Pr\{X \leq \tau_{row} - 1\}, \tag{18}$$

where $Pr\{X \leq \tau_{row} - 1\}$ is a Poisson-binomial cumulative distribution of random variable X. The Poisson-binomial distribution can be approximated by the Poisson distribution, and Reference [18] provided upper bound formulas on the difference between the two distributions in its Theorem 2; the detailed form is a bit complicated and thus omitted here. In Section 7, we will also empirically show that the performance of this approximation is close to the exact method.

Let us denote the Poisson distribution by $f(k, \lambda) = \frac{\lambda^k e^{-\lambda}}{k}$, and its cumulative distribution by $F(k, \lambda)$, $F(k, \lambda)$ can be written as

$$F(k, \lambda) = \frac{\Gamma(k + 1, \lambda)}{k!} = \frac{\int_\lambda^\infty t^{(k+1)-1} e^{-t} dt}{k!},$$

and $F(k, \lambda)$ is monotonically decreasing w.r.t. λ, since we have

$$\frac{\partial F(k,\lambda)}{\partial \lambda} = \frac{\partial}{\partial \lambda}\left(\frac{\int_{\lambda}^{\infty} t^{(k+1)-1}e^{-t}dt}{k!}\right) = \frac{1}{k!}\cdot\frac{\partial}{\partial \lambda}\left(\int_{\lambda}^{\infty} t^{k}e^{-t}dt\right)$$
$$= \frac{1}{k!}\cdot\left(-\lambda^{k}e^{-\lambda}\right) = -f(k,\lambda) \leq 0.$$

We propose an approximation algorithm that uses the Poisson cumulative distribution $F(\tau_{row} - 1, \mu)$, where the expected support μ is given by $\mu = \sum_{g \in G} p_g(P)$. Specifically, the Poisson approximation of the probabilistic frequentness of P is given by

$$Pr\{X \geq \tau_{row}\} \approx 1 - F(\tau_{row} - 1, \mu). \tag{19}$$

This approximation function of $Pr\{X \geq \tau_{row}\}$ increases monotonically with μ, since $F(\tau_{row} - 1, \mu)$ decreases monotonically with μ. Based on the above discussion, we can calculate the minimum expected support threshold μ_m by solving

$$1 - F(\tau_{row} - 1, \mu) = \tau_{prob}. \tag{20}$$

We use *exponential search* to solve for μ_m, and an OPSM is frequent if $\mu = \sum_{g \in G} p_g(P) \geq \mu_m$. We denote this probabilistic frequentness approximation that uses the Poisson distribution by ***PF-P***.

Computing the threshold μ_m using exponential search takes logarithmic time, while computing $\mu = \sum_{g \in G} p_g(P)$ takes $O(n)$ time, so *PF-P* takes only $O(n)$ time to judge OPSM frequentness according to $\mu \geq \mu_m$.

4.4.2 Approximation by Gaussian Distribution. We can also use the normal approximation method with continuous correction as proposed by Reference [27] to approximate the CDF of a Poisson binomial distribution, which is based on the *Central Limiting Theorem*. Specifically, we can approximate the CDF of Poisson binomial (denoted by $CDF(k)$) as [27]:

$$CDF(k) \approx G\left(\frac{k - \frac{1}{2} - \mu}{\delta}\right),$$

where $G(t) = \int_{-\infty}^{t} e^{-\frac{x^2}{2}} dx$ is the Gaussian CDF.

We now consider how to approximate the PMF of support X, using the supporting probabilities $p_g(P)$ of all rows $g \in G$. Specifically, the expected support μ is given by $\mu = \sum_{g \in G} p_g(P)$, and the standard deviation δ can be computed as

$$\delta = \sqrt{\sum_{g \in G}\left\{p_g(P)\cdot(1 - p_g(P))\right\}}, \tag{21}$$

and therefore the Gaussian approximation of the probabilistic frequentness of pattern P is given by

$$Pr\{X \geq \tau_{row}\} \approx 1 - G\left(\frac{\tau_{row} - \frac{1}{2} - \mu}{\delta}\right). \tag{22}$$

This method has a good approximation ratio whose error upper bound is given by Reference [27]:

$$\sup_{\tau_{row}}\left\{\left|Pr(X \leqslant \tau_{row} - 1) - G\left(\frac{\tau_{row} - \frac{1}{2} - \mu}{\delta}\right)\right|\right\} \leqslant c\delta^{-2},$$

where c is a constant and the proof can be found in Reference [27]. The approximation ratio is tighter for larger uncertain databases.

The approximation function for $Pr\{X \geq \tau_{row}\}$ is monotonically decreasing as t increases, since we have

$$\frac{\partial}{\partial t}(1 - G(t)) \;=\; -\frac{\partial}{\partial t}G(t) \;=\; -\frac{\partial}{\partial t}\int_{-\infty}^{t} e^{-\frac{x^2}{2}}dx \;=\; -e^{-\frac{t^2}{2}} \leqslant 0,$$

where $t = \frac{\tau_{row}-\frac{1}{2}-\mu}{\delta}$. Based on the above discussion, we can calculate the maximum t (i.e., t_m) as the verification threshold, and the formula is given by

$$1 - G(t_m) = \tau_{prob} \;\Rightarrow\; t_m = G^{-1}(1 - \tau_{prob}).$$

An OPSM is considered frequent if $t = \frac{\tau_{row}-\frac{1}{2}-\mu}{\delta} \leqslant t_m$. We denote this probabilistic frequentness approximation that uses the Gaussian distribution by ***PF-G***.

This method is expected to be more robust than the PF-P method, since it verifies the probabilistic frequentness of an OPSM using both the expected support and the standard deviation (i.e., μ and δ), while FP-P only uses the expected support.

Computing the threshold t_m using exponential search takes logarithmic time, while computing μ and δ (recall Equation (21)) takes $O(n)$ time, so *PF-G* takes only $O(n)$ time to judge OPSM frequentness according to $t = \frac{\tau_{row}-\frac{1}{2}-\mu}{\delta} \leqslant t_m$.

5 MINING ALGORITHMS

In this section, we first propose a method for filtering out the rows g such that $p_g(P) = 0$. Then, we describe our two OPSM mining algorithms that integrate the techniques we presented.

5.1 Row Filtering

Consider the matrix in Figure 3; it is obvious that g_2 can never support pattern $t_3 < t_4$, i.e., $p_{g_2}(t_3 < t_4) = 0$. We now describe how to determine whether $p_g(P) = 0$ efficiently without actually evaluating the supporting probability.

Given a pattern $P = (t_{i_1} < t_{i_2} < \cdots < t_{i_j})$, we define its *valid interval* as the interval $[\ell_P, r_P]$ such that $p_g(P) > 0$ if and only if $t_{i_j} \in [\ell_P, r_P]$. We now consider how to compute the valid interval of a pattern P.

In the base case when $P = t_{i_1}$, its valid interval is exactly the interval $D[g][t_{i_1}] = [\ell_{i_1}, r_{i_1}]$.

For pattern $P = (t_{i_1} < \cdots < t_{i_j})$ $(j > 1)$, we compute its valid interval using that of pattern $P' = (t_{i_1} < \cdots < t_{i_{j-1}})$. Since row g supports P' if and only if $t_{i_{j-1}} \in [\ell_{P'}, r_{P'}]$, row g would support P if and only if $\exists t_{i_{j-1}} \in [\ell_{P'}, r_{P'}]$ such that $t_{i_{j-1}} < t_{i_j}$, or equivalently, $t_{i_j} \in [\max\{\ell_{i_j}, \ell_{P'}\}, r_{i_j}] \triangleq [\ell_P, r_P]$ (where $[\ell_{i_j}, r_{i_j}]$ is the interval $D[g][t_{i_j}]$). Note that $[\ell_P, r_P] = \emptyset$ if $\ell_P > r_P$, and in this case $p_g(P) = 0$ and g can be filtered.

In our mining algorithm when checking pattern P', for each row g, we maintain the following information: (1) its valid interval $[\ell_{P'}, r_{P'}]$, (2) the ordered list of split points, and (3) the DP-array. Since our mining algorithm checks patterns by pattern-growth, P will be checked after P'. To process g for pattern P, we will first compute $[\ell_P, r_P] = [\max\{\ell_{i_j}, \ell_{P'}\}, r_{i_j}]$. If $\ell_P > r_P$, then we drop g from further consideration, since it does not contribute to the support of P. Otherwise, we will compute the DP-array with that of g for P', using the algorithm of Section 3.3, to obtain $p_g(P)$.

5.2 Algorithms

Pattern Anti-monotonicity. Suppose pattern P' is a sub-pattern of pattern P, then we have the following conclusions: (1) For any row g, $g_p(P) < g_p(P')$. This is because in any possible world instantiation of g, P' must be supported if P is supported. (2) When pattern frequentness is defined using expected support, if P' is infrequent, then P must be infrequent. This is because

ALGORITHM 2: Expected-support-based Frequentness Checking

1: Filter the rows in $G_{P'}$ with their valid intervals, to obtain G_P along with the updated valid intervals.
2: If $|G_P| < \tau_{row}$, mark P as infrequent and return.
3: $sum \leftarrow 0$, $sum_0 \leftarrow \sum_{g \in G_{P'}} p_g(P')$
4: **for each** $g \in G_P$ **do**
5: Compute the DP-array to obtain $p_g(P)$
6: $sum \leftarrow sum + p_g(P)$, $sum_0 \leftarrow sum_0 - p_g(P')$
7: If $sum + sum_0 < \tau_{row}$, mark P as infrequent and return.
8: **if** $sum \geq \tau_{row}$ **then**
9: Mark P as frequent and return.
10: **else**
11: Mark P as infrequent and return.

ALGORITHM 3: Probabilistic Frequentness Checking

1: Filter the rows in $G_{P'}$ with their valid intervals, to obtain G_P along with the updated valid intervals.
2: If $|G_P| < \tau_{row}$, mark P as infrequent and return.
3: $sum \leftarrow 0$, $sum_0 \leftarrow \sum_{g \in G_{P'}} p_g(P')$
4: **for each** $g \in G_P$ **do**
5: Compute the DP-array to obtain $p_g(P)$
6: $sum \leftarrow sum + p_g(P)$, $sum_0 \leftarrow sum_0 - p_g(P')$
7: **if** $sum + sum_0 < \tau_{row} \times \tau_{prob}$ **then**
8: Mark P as infrequent and return.
9: Apply Exponential-Prune and return when P is pruned.
10: **if** P is validated as frequent by divide-and-conquer **then**
11: Mark P as frequent and return.
12: **else**
13: Mark P as infrequent and return.

$\sum_{g \in G} g_p(P) < \sum_{g \in G} g_p(P')$. (3) If P' is probabilistically infrequent, then P must be probabilistically infrequent. This is because in any possible world of D, the support of P' is at least the support of P.

Frequentness Checking. Since our mining algorithms check patterns by pattern-growth, when checking pattern $P = (t_{i_1} < \cdots < t_{i_j})$, we make use of the information of the rows for computation when checking $P' = (t_{i_1} < \cdots < t_{i_{j-1}})$.

Given a pattern P, **let G_P be the set of rows $g \in G$ with $p_g(P) > 0$**, where each row is associated with its valid interval, split point list and DP-array.

The expected-support-based frequentness checking of pattern P is described by Algorithm 2. In Line 2, we prune P using the fact that $p_g(P) \leq 1$ and thus $\sum_{g \in G_P} p_g(P) \leq |G_P|$. Furthermore, we maintain two variables sum and sum_0. Let C be the set of rows already processed, then $sum = \sum_{g \in C} p_g(P)$ and $sum_0 = \sum_{g \in (G_P - C)} p_g(P')$. Line 7 prunes P using the fact that $\sum_{g \in G_P} p_g(P) = sum + \sum_{g \in (G_P - C)} p_g(P) \leq sum + sum_0$.

Algorithm 3 checks the p-frequentness of pattern P, where the pruning rules described in Section 4.3 are used: Line 2 applies Count-prune, Lines 7–8 apply Markov-prune, and Line 9 applies Exponential-prune.

Mining by Prefix-projection. Our first mining algorithm is based on the idea of prefix-projection used by sequential pattern mining [37], where the current pattern $P = (t_{i_1} < \cdots < t_{i_j})$ is processed using the projected row set $G_{P'}$ of $P' = (t_{i_1} < \cdots < t_{i_{j-1}})$.

The recursive algorithm, denoted as $DFS(P, G_{P'})$,[1] finds all the frequent patterns with prefix P from $G_{P'}$. Specifically, $DFS(P, G_{P'})$ first checks whether P is frequent using $G_{P'}$ (Algorithm 2 or 3). If so, then it recursively calls $DFS(t_{i_1} < \cdots < t_{i_j} < t, G_P)$ for all $t \in T - \{t_{i_1}, \ldots, t_{i_j}\}$. The mining algorithm starts by calling $DFS(\emptyset, G)$.

For each recursion that processes P, G_P is maintained so the subsequent recursions for $t_{i_1} < \cdots < t_{i_j} < t$ may use it. The maintenance of G_P incurs a space cost of $O(n \cdot j^2)$, since a DP-array of size $j \times s = O(j^2)$ is maintained for each row in G_P, and the space can only be released after its corresponding recursion is done.

Since our algorithm works in a depth-first manner, at most m projected row sets are maintained at any time, with the total space cost $\sum_{j=1}^{m} O(n \cdot j^2) = O(n \cdot m^3)$.

For a pattern P of length ℓ, computing $p_g(P)$ for all rows g using the pay-as-you-go algorithm of Section 3.3 takes $O(n\ell^2)$ time; then frequency checking takes $O(n)$ (respectively, $O(n \log^2 n)$) for expected support (respectively, p-frequentness with FFT computation). If C patterns are checked, then the time cost is bounded by $C \cdot O(n\ell^2 + n) = O(Cnm^2)$ (respectively, $C \cdot O(n\ell^2 + n \log^2 n) = O(Cn(m^2 + \log^2 n))$. Additionally, row preprocessing of Section 3.1 takes $O(nm \log m)$ time.

We remark that the above analysis is very loose, and m can be replaced by ℓ_{max}, the length of the longest pattern checked.

The benefit of this algorithm is that, for any pattern P checked, its DP-arrays (of the rows in G_P) is computed exactly once; however, it can be expensive when the column set T is large, since the recursions have to be called for all $t \in T - \{t_{i_1}, \ldots, t_{i_j}\}$ and column candidate pruning is lacking.

Mining by Apriori. The Apriori algorithm works as follows: Starting with the set of all length-1 patterns, we construct length-j frequent patterns from the set of all length-$(j-1)$ frequent patterns, until there is no frequent pattern left.

We organize the set of length-j frequent patterns using a prefix-tree T_j, like the FP-tree proposed in Reference [15]. When computing length-j frequent patterns $P = (t_{i_1} < \cdots < t_{i_j})$, we need $G_{P'}$ for $P' = (t_{i_1} < \cdots < t_{i_{j-1}})$.

However, the space cost is prohibitive if we maintain $G_{P'}$ for each frequent length-$(j-1)$ pattern P' due to the breadth-first search order. Therefore, we choose to recompute $G_{P'}$ over the prefix-tree T_{j-1} in a depth-first manner (like the prefix-projection method without pattern pruning), and when recursing to the last tree-node $t_{i_{j-1}}$, we check whether $P = (t_{i_1} < \cdots < t_{i_{j-1}} < t_{i_j})$ is frequent for all possible column candidates $t_{i_j} \in C$ (we will discuss how to compute the candidate set C later), and if P is checked to be frequent, then it is inserted to the new prefix-tree T_j.

Unlike the prefix-projection method, we need to recompute G_P when checking all prefix-trees T_z $(z \geq j)$. However, the computation in the same tree is still shared. For example, $G_{t_1 < t_2}$ is used for computing both $G_{t_1 < t_2 < t3}$ and $G_{t_1 < t_2 < t_4}$.

We now consider how to determine t_{i_j}'s candidate set C.

Theorem 5.1. *For any column t that is not a child of tree-node $t_{i_{j-2}}$ in T_{j-1}, pattern $P = (t_{i_1} < \cdots < t_{i_{j-1}} < t)$ is infrequent.*

Proof. Since $P' = (t_{i_1} < \cdots < t_{i_{j-2}} < t)$ is infrequent, its super-pattern P should also be infrequent according to pattern anti-monotonicity. □

[1] We name the algorithm as DFS due to its depth-first recursion nature.

Thus, we choose C to include all child nodes of node $t_{i_{j-2}}$ in T_{j-1}. Compared with the prefix-projection method, C is much smaller than $(T - \{t_{i_1}, \ldots, t_{i_{j-1}}\})$ and thus the recursion has a much smaller fan-out.

The pattern anti-monotonicity also allows for the following pattern pruning:

THEOREM 5.2. *For any length-j pattern P, if not all of the length-$(j-1)$ sub-pattern of P exist in T_{j-1}, then P is infrequent.*

Theorem 5.2 is efficient to check and is thus examined before the more expensive frequentness checking of Algorithms 2 or 3, which requires computing the DP-array for all rows in G_P.

In the Apriori algorithm, computing $p_g(P)$ of patterns of length ℓ requires re-computing the DP-array for frequent patterns of length $< \ell$; this brings an additional factor of ℓ_{max} to our prefix-projection algorithm's time complexity, since each P is re-processed for at most ℓ_{max} times. In return, the algorithm enjoys a small recursion fan-out, which, however, is not reflected in time complexity analysis.

6 PARALLEL MINING WITH PREFIXFPM

As we have shown in Section 5.2, our serial mining algorithms can be very costly. A natural idea to speed up the mining is by utilizing parallelization, but our algorithms are based on pattern growth and are not embarrassingly parallel. Fortunately, a recent parallel programming framework called PrefixFPM [29, 30] was proposed to parallelize any frequent pattern mining algorithm that adopts the idea of prefix projection. PrefixFPM adopts a think-like-a-task computation model that is able to fully utilize the available CPU cores. To adapt our algorithm to run with PrefixFPM, we need to properly map our methods to PrefixFPM's task-based programming interface, by identifying the basic unit of a task, and how to create children tasks during pattern growth to create sufficient parallelism.

Interestingly, our problem can be regarded as a frequent pattern mining problem as targeted by PrefixFPM: Each pattern is given by $P = (t_{i_1} < \cdots < t_{i_j})$, and its frequency is evaluated either by expected support or probabilisitic frequentness. We remark that while PrefixFPM already has seven applications on top [30] for mining patterns such as sequences, trees, and subgraphs, none of them mines submatrix patterns or considers a probabilistic setting, and our new addition to PrefixFPM application pool fills this void.

We next describe the implementation of our prefix-projection algorithm on PrefixFPM. We associate each pattern P' in a pattern-growth search tree with a task $t_{P'}$, which checks the frequentness of P' using its projected database $G_{P'}$ and which grows the pattern by one more element t_{i_j} to generate the children patterns $\{P\}$ and their projected databases G_P (computed incrementally from $G_{P'}$ rather than from the entire D). These children patterns give rise to new tasks $\{t_P\}$ that are added to a shared task queue to be fetched by computing threads for further processing.

PrefixFPM runs a number of computing threads that fetch pattern-tasks from a shared task queue Q_{task} for concurrent processing. In PrefixFPM, (i) a depth-first task scheduling strategy is used to minimize the memory footprint of patterns in processing, (ii) tasks in a small pattern-growth search subtree are processed by a single thread to avoid frequent queue contention, and (iii) a timeout mechanism is used for task decomposition to avoid stragglers [30].

To write a PrefixFPM program, we need to specify the template arguments of some predefined base classes and implement their provided **user-defined functions (UDFs)** with the application logic.

One base class is *ProjTrans*, which specifies a row in the projected database G_P. In our implementation, a *ProjTrans* object keeps (1) the row ID of $g \in G$ that can be used to fetch information of g from D and (2) the DP-array A for computing $p_g(P)$.

Another base class is *Pattern*, for which we keep a sequence $[i_1, i_2, \ldots, i_j]$ for a pattern $P = (t_{i_1} < \cdots < t_{i_j})$, the projected database G_P and its expected support $\sum_{g \in G} p_g$.

The third class is *Task*, which contains two key UDFs: (1) *pre_check*(*fout*), which is called initially by task t_P to check if P is frequent, and if so, to output P to an output stream *fout*. In our implementation, we only output P if its length is at least τ_{col}; t_P does not need to check the condition for τ_{row} here, since, as we shall see below, we only create t_P if P satisfies the frequentness condition regarding τ_{row}; (2) *setChildren()*, which generates child-patterns $\{P''\}$ along with their projected databases $G_{P''}$ by scanning G_P. Here, we check the expected support or p-frequentness of each child-pattern P'', where we apply the various pruning rules we previously introduced. Only those frequent child-patterns are retained to create new tasks for mining, which effectively reduces the number of tasks created and is critical to the program efficiency according to our experimental tests.

The last base class *Worker* contains a UDF *setRoot*(Q_{task}) to indicate how to bootstrap mining by creating initial tasks into the task queue Q_{task}. In our case, we create the singleton patterns $P = t_i$ for all $i = 1, 2, \ldots, m$ as the initial tasks.

7 EXPERIMENTS

This section evaluates the performance of our proposed algorithms on real datasets and compares it with existing approaches POPSM [12], where each matrix entry is modeled as a uniformly distributed interval, and OPSMRM [33], where each matrix entry is represented directly by the replicates.

When the number of replicates for each matrix entry is limited, we assume a uniform value distribution by default, since it introduces a small inductive bias compared with a more sophisticated distribution. We denote our prefix-projection (respectively, Apriori)-based mining algorithm by "DFS" (respectively, "Apri"), for pattern frequentness, we denote expected support (respectively, p-frequentness) by "ES" (respectively, "PF"). Thus, we have four algorithm variants: *DFS-ES*, *Apri-ES*, *DFS-PF,* and *Apri-PF*. In addition, since the FFT algorithm for checking p-frequentness of a pattern is expensive, we further proposed two probability approximation algorithms using the Poisson and Gaussian distributions, respectively, to check the p-frequentness, which improves the cost from $O(n \log^2 n)$ to $O(n)$; we denote the Gaussian-based approximation algorithms by *DFS-PF-G* and *Apri-PF-G*, and the Poisson-based approximation algorithms by *DFS-PF-P* and *Apri-PF-P*.

Whether the mining algorithm is "DFS" or "Apri" does not impact the output. Therefore, when we evaluate result quality, we use *ES* for both *DPS-ES* and *Apri-ES*; *PF* for both *DPS-PF* and *Apri-PF*; *PF-P* for both *DFS-PF-P* and *Apri-PF-P*; and *PF-G* for both *DFS-PF-G* and *Apri-PF-G*.

To test the effectiveness of our cubic spline approach to approximate an arbitrary uncertain value distributions as described in Section 3.4, we also consider Gaussian value distributions. As we shall see in later experiments, using Gaussian value distribution can generate results of a higher quality, but it is two orders of magnitude slower than using uniform value distributions. The reasons are twofold: (1) we cannot use an elegant closed-form formula as in Corollary 3.2 to compute $P_{I_k}(t_{i_1} < t_{i_2} < \cdots < t_{i_j})$, but rather have to use the much more expensive recursive integral computation as specified in Equation (11) of Section 3.4; (2) we can no longer use the incremental computation scheme as specified by Equation (7) in Section 3.2, but rather have to compute each $P_{I_k}(t_{i_1} < t_{i_2} < \cdots < t_{i_j})$ from scratch. As a result, we run experiments with Gaussian value distributions only on our gene expression microarray datasets that have a relatively small number of columns, and for the other real datasets with many more columns, we only run experiments with uniform value distributions for the purpose of scalability. Note that the number of patterns grows exponentially with the number of columns, and so does the running time. Exponential distribution is a less-convincing assumption of the value distribution than Gaussian, but we also

Table 2. Datasets Used to Evaluate Our OPSM Model

ID	Rows	Columns	Replicates
GDS2712	7,826	7	3
GDS2002	5,474	10	3
GAL	205	20	4
MovieLens	943	100	N/A
RFID	105	80	N/A

report its performance on a representative dataset to show the generality of our cubic spline approach.

To indicate whether an algorithm assumes uniform value distributions or Gaussian value distributions, we append each algorithm name with "(U)" (respectively, "(G)") to indicate that uniform (respectively, Gaussian) distribution is adopted. For example, for DFS-PF, we have *DFS-PF(U)* and *DFS-PF(G)*. Similarly, we use "(E)" to indicate that exponential distribution is adopted.

All serial experiments were conducted on a PC with Intel i7-6700K quad core CPU at 4 GHz, 16 GB DDR4 memory and 120 GB SSD. Our parallel programs were run on a server with 64 CPU cores (IBM POWER8 CPU at 3,491 MHz) and 1 TB RAM. Our complete code is released at the following GitHub link: https://github.com/wenwenQu/OPSM.

7.1 Experimental Setup

Datasets. As Table 2 shows, five publicly available real datasets were used in our experiments, including three microarray gene expression datasets, a movie rating dataset, and an RFID user trace dataset.

Among the three biology datasets, GDS2712 and GDS2002[2] are microarray datasets of the baker's yeast *Saccharomyces cerevisiae* from the **Gene Expression Omnibus (GEO)** database [3], where each matrix entry has three replicates. We also tested some other datasets such as GDS2713, GDS2715, and GDS2003, and we found that the results are similar and hence omitted to avoid redundancy in presentation. For GDS2712, the 7,826 rows shown in Table 2 are obtained from preprocessing, where we removed those unidentified genes and control probes, since they cannot be used to calculate the biological significance p-value against the database of ground-truth gene functional categories.

The other biology dataset, GAL,[3] is regarding yeast galactose utilization [32], which was also used by References [12, 33]. As Table 2 shows, GAL contains 205 gene probes (rows) and 20 experimental conditions (columns) with four replicates for each entry in the matrix.

For the gene expression microarray datasets, we construct a uniform value interval for every matrix entry as $[min, max]$, where min (respectively, max) is the minimum (respectively, maximum) replicated value of the entry from repeated experiments. As for Gaussian value distribution, given a matrix entry with replicates, we compute their sample mean μ and sample variance σ and assume that the matrix entry has a value following $\mathcal{N}(\mu, \sigma^2)$. We fit its PDF with a cubic spline as we described in Section 3.4, and we only consider the six intervals $[\mu - 3\sigma, \mu - 2\sigma]$, $[\mu - 2\sigma, \mu - \sigma]$, $[\mu - \sigma, 0]$, $[0, \mu + \sigma]$, $[\mu + \sigma, \mu + 2\sigma]$, $[\mu + 2\sigma, \mu + 3\sigma]$ when computing $p_g(P)$. We similarly fit the PDF of exponential value distributions following our discussion in Section 3.4.

Besides the gene expression datasets, we also used a movie rating dataset MovieLens[4] with 100,000 ratings from 943 users on 1,682 movies [16] to evaluate the algorithms. The movie rating

[2] https://www.ncbi.nlm.nih.gov/sites/GDSbrowser.
[3] http://genomebiology.com/content/supplementary/gb-2003-4-5-r34-s8.txt.
[4] https://grouplens.org/datasets/movielens/100k/.

dataset is an incomplete matrix, and we used TensorFlow to conduct factorization-based matrix completion, which provides a complete 943 × 1,682 user rating matrix M, where M_{ij} estimates User i's rating towards Movie j. The loss function of factorization is given by

$$\text{reduce\_sum}\Big(\mathcal{P}_\Omega(|M^0 - UV|)\Big),$$

where M^0 is the observed rating matrix with missing values, $M = UV$ is the estimated rating matrix, $U \in \mathbb{R}^{943\times 10}$, $V \in \mathbb{R}^{10\times 100}$, $|X|$ takes the element-wise absolute value, reduce_sum(X) sums all elements of X, and $\mathcal{P}_\Omega(X)$ projects to a matrix where the (i, j)-th element equals X_{ij} if matrix entry (i, j) in M^0 is observed, and 0 otherwise. Put simply, the goal is to minimize the difference between M^0 and M for those elements that are observed in M^0. Implementation in TensorFlow is simple, with operators like tf.abs for taking the absolute value, tf.subtract for matrix subtraction, tf.gather to collect elements at observed matrix locations, and tf.reduce_sum to compute the sum of elements. We learn U and V by initializing them with truncated normal sampling, running stochastic gradient descent with a learning rate of 10^{-3} and a staircase decay of rate 0.96, and running for 100,000 steps.

To consider only the popular movies, we select the top-100 movies that get the most user ratings, which generates a 943 × 100 submatrix M_s of M. Since each new rating r is now a low-rank approximation, we introduced uncertainty to each rating r. A rating in the original data takes its value from $\{1, 2, 3, 4, 5\}$; after matrix completion, a rating is a real value like $r = 3.4$, in which case, we assign an interval $[3, 4]$ to the matrix entry, and for OPSMRM, we assign replicates $\{3, 4\}$. Other reasonable methods to assign uncertainty to the values can also be used. Note that user ratings are intrinsically uncertain: (1) the discrete rating values $\{1, 2, 3, 4, 5\}$ are coarse-grained and preferences among the movies with the same rating are not captured; (2) different users have different bias on the rating scale, with some giving rating 5 only to movies they like while others giving rating 4 also to such movies. Such uncertainty should be captured by relaxing the strict values in $\{1, 2, 3, 4, 5\}$.

Finally, we also prepared an RFID dataset,[5] which contains the traces of 12 voluntary users in a building with 195 RFID antennas installed. Only 80 antennas are active, i.e., they detect the trace of at least one user for at least one time. The trace of a user consists of a series of (period, antenna) pairs, which indicates the time period during which the user is continuously detected by the antenna. When a user is detected by the same antenna at different time periods, we split the raw trace into several new traces in which the user is detected by any antenna for at most one time. We then generate a matrix having 105 rows (i.e., user traces) and 80 columns (i.e., antennas), where the (i, j)-th element records the potential interval of time that User i is near Antenna j, and an OPSM thus records a group of users that visits a group of locations (captured by antennas) in the same time order. To run OPSMRM for comparison, the set of timestamps within each time period is enumerated to generate the corresponding set-valued matrix.

Evaluation Metrics. The following metrics are studied to evaluate result quality, time efficiency, and space efficiency, respectively:

(1) Result Quality: For the microarray gene expression datasets, we use the *biological significance* of the mined OPSMs to demonstrate the result quality of our proposed method. We adopt a widely used metric, p-value [12, 21, 33], which measures the association between OPSMs mined and the known gene functional categories. Specifically, a smaller p-value indicates a stronger association between an OPSM and gene categories, i.e., biologically more significant. Following Reference [12], we also consider four exponential-scale p-value ranges as *significance levels*, such as $[0, 10^{-40})$,

[5] http://lahar.cs.washington.edu/content/Download/RFIDData/rfidData.html.

$[10^{-40}, 10^{-30})$, $[10^{-30}, 10^{-20})$, and $[10^{-20}, \infty)$ (the actual ranges depend on the concrete datasets). To compare the result quality of our algorithms with those of the existing algorithms, we use the number and proportion of OPSMs mined at each significance level.

For the movie rating dataset, we define a ***Kendall tau score*** **(KTS)** motivated by the concept of Kendall tau distance: For a mined OPSM with movie order $t_1 < t_2 < \cdots < t_k$, we compute a KTS for each user g_i in the OPSM, which equals the fraction of all possible C_k^2 movie pairs (t_i, t_j) where the rating order is consistent with that in our 943×100 submatrix M_s; the KTS of the OPSM is then computed as the average KTS over all its users.

For the RFID dataset, each mined OPSM consists of a set of users sharing a common subroute passing a subset of antennas. We define a measure called the ***trace matching score*** **(TMS)** as follows: For each minded OPSM with subroute $P = (t_1 < t_2 < \cdots < t_k)$, we compute a TMS for each user g_i in the OPSM as follows:

$$TMS(g_i, P) = \frac{longest\_common\_subsequence(\mathcal{T}_{g_i}, P)}{|P|},$$

where $\mathcal{T}_{g_i}$ denotes the manually annotated ground-truth trace of g_i, which is also provided by the RFID dataset. The TMS of the OPSM is then computed as the average TMS over all its users.

(2) Time Efficiency: We evaluate the time efficiency of the algorithms by considering the following two metrics:

- *TR-Time*: total running time for mining the OPSMs;
- *SP-Time*: the average time for mining a single OPSM.

Here, *SP-Time* equals *TR-Time* divided by the number of patterns mined.

(3) Space Efficiency: We report the peak memory consumption of each program run in our experiments.

All experiments were repeated for 10 times and the reported metrics were averaged over the 10 runs (although the results observed from different runs are actually quite stable/similar).

For the purpose of succinct presentation, we abuse the term "more than" (similarly for "less than" and "fewer than"): When we say that A is k times more than B, we mean $A = k \cdot B$, not $A = (1+k) \cdot B$. The terms "more," "less," and "fewer" are meant to indicate the directions of comparison.

As an overview of our findings, our *ES* and *PF* algorithms are robust and output higher-quality OPSMs than *OPSMRM* and *POPSM*, and generally find more OPSMs. Our algorithms also consistently outperform *OPSMRM* and *POPSM* in terms of *SP-Time* in the context of uniform value distributions. *ES* consumes the least amount of memory, and *PF* consumes more memory than *ES* but generally less memory than *OPSMRM* and *POPSM*. *PF* performs the best w.r.t. result quality and quantity. Among other findings, using Gaussian value distributions as the uncertain model generally outperforms the uniform distribution model scheme, but the running time is two orders of magnitude longer; using exponential value distributions exhibits a similar performance. Finally, our parallel algorithm for prefix-projection-based mining scales well with the number of mining threads.

The rest of this section is organized as follows: Section 7.2 reports the results on GDS2712 and GDS2002 in terms of the result quality and quantity, running time, and peak memory cost. There, we use GDS2712 to demonstrate that our algorithms are efficient even when exponential value distribution is adopted. Due to space limitation, we report the results on GAL in Appendix B. Section 7.3 then uses the GDS2712 dataset to study the effect of parameters τ_{cut} and τ_{prob}, and Section 7.4 compares row selection strategies to justify the use of τ_{cut}. Subsequently, Section 7.5 reports the vertical scalability experiments of our parallel prefix-projection-based mining algorithm,

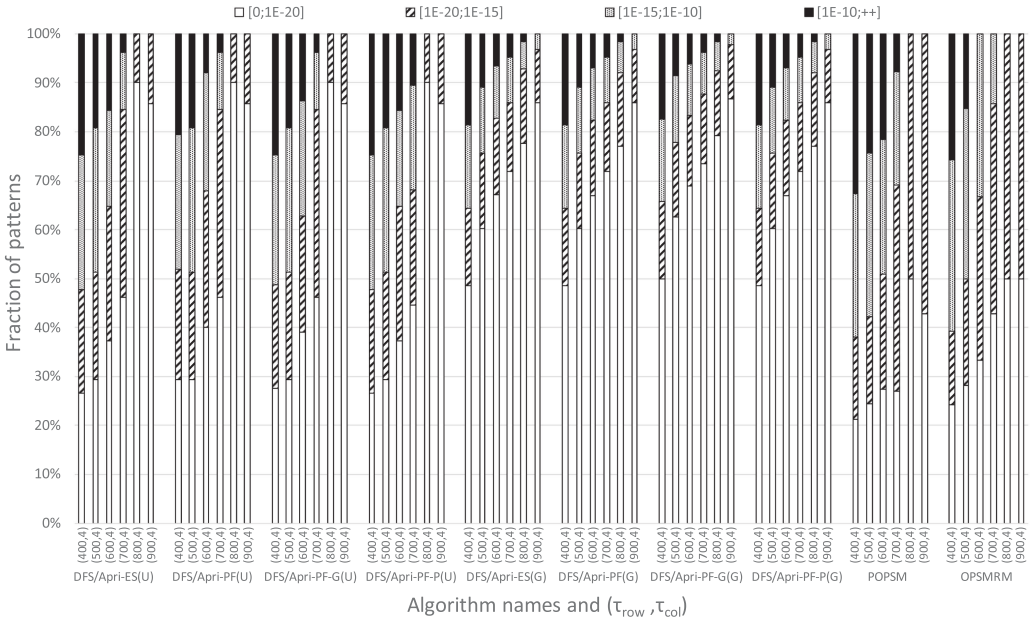


Fig. 8. Fractions of OPSMs at the four significance levels for different size thresholds on GDS2712.

with additional scalability results in Appendix C. Section 7.6 reports our results on MovieLens, and Section 7.7 replicates the MovieLens to generate datasets of different sizes to study the algorithm scalability. Finally, Section 7.8 reports the results on the RFID user trace dataset.

7.2 Performance Comparison on GDS2712 and GDS2002

Size Thresholds τ_{row} and τ_{col} for Testing. Both datasets are used in Reference [33] for evaluation, and following Reference [33], we fix $\tau_{cut} = 0.6$ for them. In fact, we also tested the other GDS datasets in Reference [33] such as GDS2713, GDS2715, and GDS2003, and the results are very similar and thus omitted. Since the results are similar for various τ_{prob} values, we only show results when $\tau_{prob} = 0.5$ to save space (see Section 7.3 for more results on the effect of τ_{prob}). While many (τ_{row}, τ_{col}) pairs are possible, we would like to show a limited number of typical combinations so they can fit in one figure to be readable. For GDS2712, we set $\tau_{col} = 4$ and vary τ_{row} among $\{400, 500, 600, 700, 800\}$ to show the effect τ_{row}. For GDS2002, we vary τ_{row} among $\{200, 300, 400\}$ but also vary τ_{col} among $\{2, 3\}$ to show the effect of τ_{col}.

Result Quality. Figure 8 presents the fraction of mined OPSMs that fall in each *significance level* for GDS2712. We see that our algorithms find larger fractions of high-quality OPSMs than POPSM and OPSMRM, as they have taller white bars (representing the highest significance level with p-value $\in [0, 10^{-20})$). For example, when $(\tau_{row}, \tau_{col}) = (800, 4)$, our proposed *ES(U)*, *PF(U)*, *PF-G(U)*, and *PF-P(U)* all have 90% patterns falling into the highest *significance level*, while *POPSM* and *OPSMRM* have only 50%. Moreover, our proposed *ES(U)*, *PF(U)*, *PF-G(U)*, and *PF-P(U)* discovered a significantly larger number of OPSM patterns compared with *POPSM* and *OPSMRM* as shown in Table 3. We can see that our algorithms using uniform value distributions generate consistently more OPSMs than *POPSM* and *OPSMRM* (and of higher quality as well). Furthermore, if we look at the numbers of OPSMs falling in the *highest significance level*, then when $(\tau_{row}, \tau_{col}) = (800, 4)$,

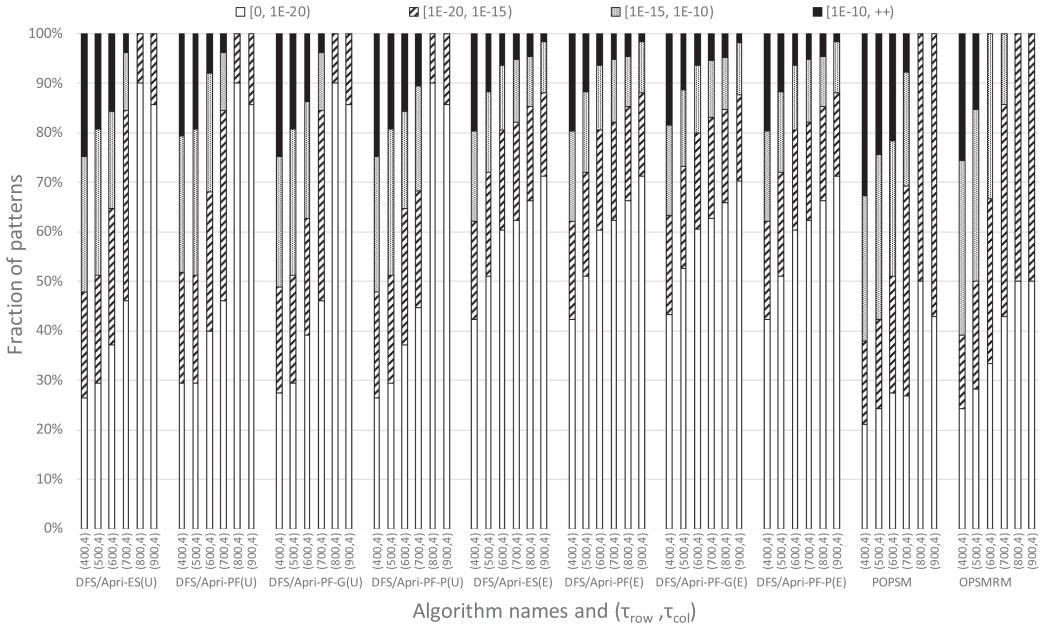


Fig. 9. Fractions of OPSMs at four significance levels (with exponential distributions) on GDS2712.

Table 3. Number of OPSMs w.r.t. Size Thresholds on GDS2712

	(400,4)	(500,4)	(600,4)	(700,4)	(800,4)	(900,4)
DFS/Apri-ES(U)	277	178	126	92	67	35
DFS/Apri-PF(U)	277	178	126	92	67	35
DFS/Apri-PF-G(U)	264	171	120	86	62	32
DFS/Apri-PF-P(U)	275	177	126	92	67	35
DFS/Apri-ES(G)	482	324	244	171	125	93
DFS/Apri-PF(G)	482	342	245	171	126	93
DFS/Apri-PF-G(G)	464	316	228	162	120	90
DFS/Apri-PF-P(G)	482	342	245	171	126	93
DFS/Apri-ES(E)	335	239	159	117	89	59
DFS/Apri-PF(E)	335	239	159	117	89	59
DFS/Apri-PF-G(E)	325	228	155	113	85	57
DFS/Apri-PF-P(E)	335	239	159	117	89	59
OPSMRM	74	46	21	7	2	2
POPSM	113	78	51	26	10	7

our proposed *ES(U)*, *PF(U)*, *PF-G(U)*, and *PF-P(U)* have 60, 60, 55, and 60 patterns falling into the highest *significance level*, respectively, while *POPSM* and *OPSMRM* have only 5 and 1, respectively.

As for our algorithms using Gaussian value distributions, while Figure 8 shows that they do not always have a taller white bar than the uniform distribution variants (e.g., much taller when $(\tau_{row}, \tau_{col}) = (400, 4)$ but shorter when $(\tau_{row}, \tau_{col}) = (800, 4)$), the absolute OPSM counts found in each significance level dominates those of the uniform distribution variants, as we can see from Table 3, showing that using the more expensive Gaussian distribution algorithm variants does pay off in result quality.

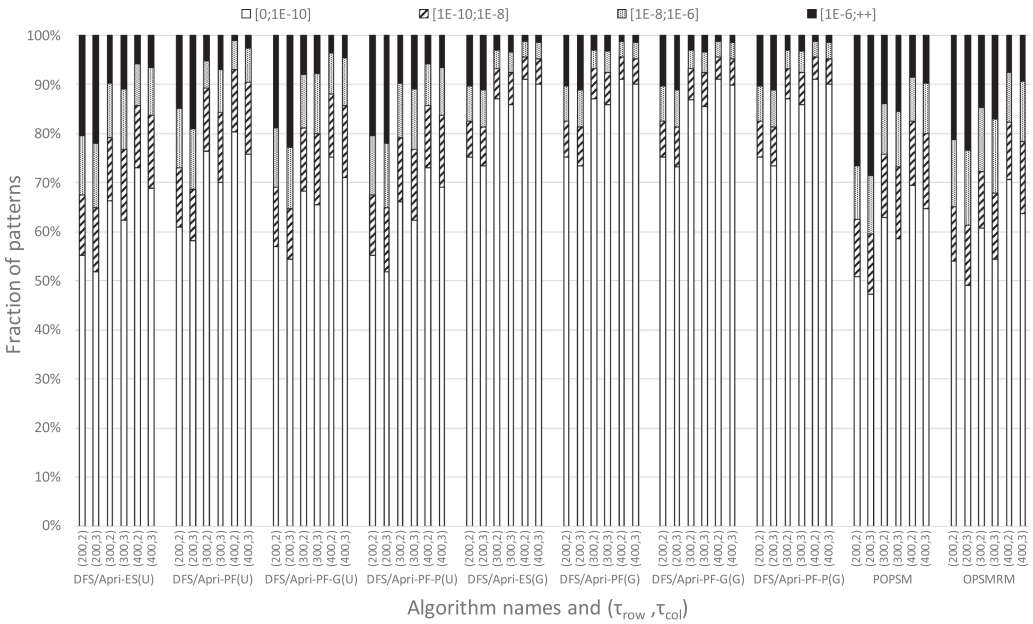


Fig. 10. Fractions of OPSMs at the four significance levels for different size thresholds on GDS2002.

Figure 9 presents the fraction of mined OPSMs that fall in each *significance level* for GDS2712, similar to Figure 8 except that we replace those algorithms adopting Gaussian value distributions with those adopting exponential value distributions. We can see that using exponential value distributions leads to much shorter white bars than using uniform distributions when $(\tau_{row}, \tau_{col}) = (800, 4)$, even though the absolute result counts are consistently higher, as shown in Table 3.

Comparing Figure 8 with Figure 9, we can see that using Gaussian value distributions is a better assumption than using exponential value distributions, since the former has a taller white bar. The same holds true when comparing absolute result counts as shown in Table 3, where using Gaussian value distributions consistently leads to more results for every (τ_{row}, τ_{col}) setting.

For the future experiments on the other datasets, we only consider uniform and Gaussian value distributions, since using exponential value distributions leads to a lower result quality than using Gaussian value distributions, while both are much more expensive than using uniform value distributions due to the need of applying the expensive cubic spline approach.

As Table 3 shows, among our algorithms, *PF* performs better than *ES*, as well as *PF-G* and *PF-P*. This verifies that considering PMF gives better results than considering only expectation. Also note that *PF-G* and *PF-P* produce results not far behind *PF*, and in fact, *PF-P(G)* gives the same results as *PF(G)*, and *PF-P(E)* gives the same results as *PF(E)*, showing that our approximations are accurate.

As for GDS2002, Figure 10 presents the fraction of mined OPSMs that fall in each *significance level*, while Table 4 shows the absolute counts. We observe similar results as on GDS2712. Specifically, Figure 10 shows that our algorithms find larger fractions of high-quality OPSMs than POPSM and OPSMRM, as they have taller white bars. The absolute counts in all the significance levels are also much higher, as Table 4 shows, with the absolute counts found by the Gaussian distribution algorithm variants in each significance level dominating those of the uniform distribution variants. This shows that using the more expensive Gaussian distribution algorithm variants does pay off in result quality. The comparative performances of the various algorithms also exhibit the same observations as those on GDS2712.

Table 4. Number of OPSMs w.r.t. Size Thresholds on GDS2002

	(200,2)	(200,3)	(300,2)	(300,3)	(400,2)	(400,3)
DFS/Apri-ES(U)	3,635	3,545	1,972	1,882	1,278	1,188
DFS/Apri-PF(U)	3,647	3,557	1,976	1,886	1,281	1,191
DFS/Apri-PF-G(U)	3,427	3,337	1,884	1,794	1,231	1,141
DFS/Apri-PF-P(U)	3,624	3,534	1,967	1,877	1,274	1,184
DFS/Apri-ES(G)	5,627	5,537	3,960	3,870	2,431	2,341
DFS/Apri-PF(G)	5,629	5,539	3,964	3,874	2,439	2,439
DFS/Apri-PF-G(G)	5,459	5,369	3,740	3,650	2,272	2,182
DFS/Apri-PF-P(G)	5,629	5,539	3,963	3,873	2,438	2,348
OPSMRM	908	818	659	569	477	387
POPSM	1,272	1,182	888	798	683	593

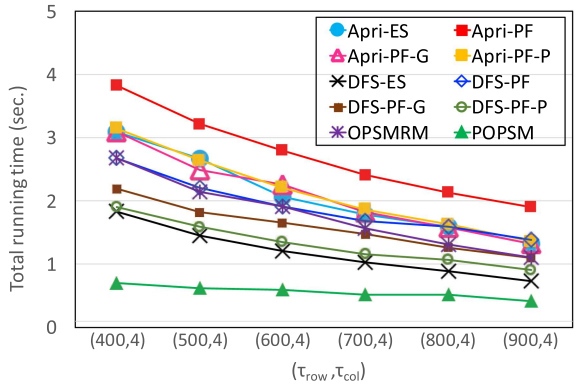


Fig. 11. Total runtime w.r.t. size thresholds (τ_{row}, τ_{col}) on GDS2712 (uniform value distributions).

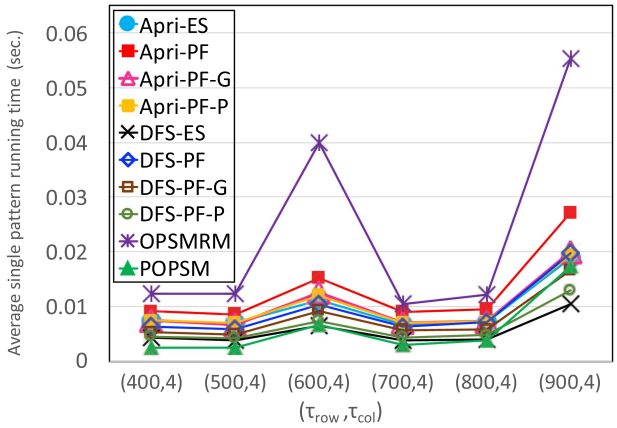


Fig. 12. Average single-pattern runtime w.r.t. (τ_{row}, τ_{col}) on GDS2712 (uniform value distributions).

Time Efficiency. We hereby report the experimental results on time efficiency for GDS2712. The time results on GDS2002 are very similar and thus omitted.

Figures 11 and 12 show the total running time *TR-Time* and average single-pattern running time *SP-Time*, respectively, of our proposed methods adopting uniform value distributions, as well as the

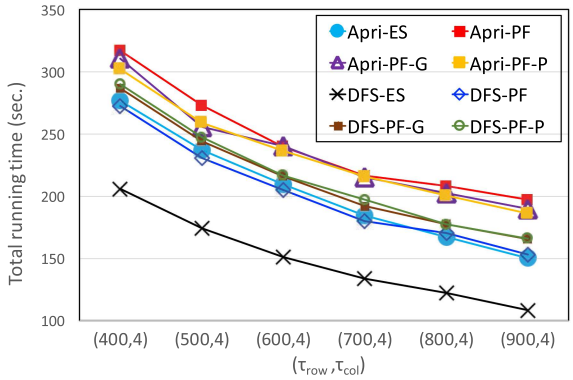


Fig. 13. Total runtime w.r.t. size thresholds (τ_{row}, τ_{col}) on GDS2712 (Gaussian value distributions).

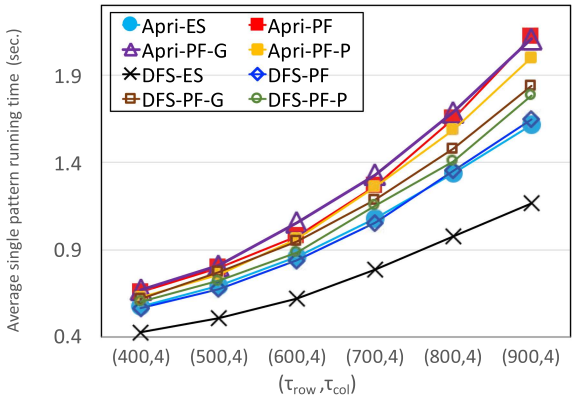


Fig. 14. Average single-pattern runtime w.r.t. (τ_{row}, τ_{col}) on GDS2712 (Gaussian value distributions).

existing algorithms *OPSMRM* and *POPSM* for different size thresholds (τ_{row}, τ_{col}). From Figure 11, we can see that *POPSM* runs faster than our methods, and *OPSMRM* has comparable total running time with our methods, but recall that both *POPSM* and *OPSMRM* mined significantly less OPSM patterns than our algorithms and their results are of poorer quality w.r.t. all different size thresholds (τ_{row}, τ_{col}). For example, as Table 3 shows, when $(\tau_{row}, \tau_{col}) = (800, 4)$, our *ES(U)* discovered 33.5 times more OPSMs than *OPSMRM* and 6.7 times more than *POPSM*.

According to Figure 12, *OPSMRM* has the worst *SP-Time*, and our methods have comparable average **single-pattern running time (*SP-time*)** with *POPSM*. Among our methods, *PF-G* and *PF-P* algorithms are always faster than *PF*, and in some cases the *DFS* algorithms run faster than our *Apri* algorithms: For example, *DFS-ES* runs faster than *Apri-ES*. This is because the GDS2712 dataset is small with merely 7 columns, so many of our pruning techniques are not effective. As we shall see later, on the GAL dataset with 20 columns and MovieLens dataset with 100 columns, Apriori-based algorithms are faster than DFS-based ones.

Figures 13 and 14 show the total running time *TR-Time* and average single-pattern running time *SP-Time*, respectively, of our proposed methods adopting Gaussian value distributions. We can see that the comparative performances of our algorithms are very similar to those in Figures 11 and 12 in the context of uniform value distributions. However, algorithms using Gaussian value distributions are two orders of magnitude more expensive.

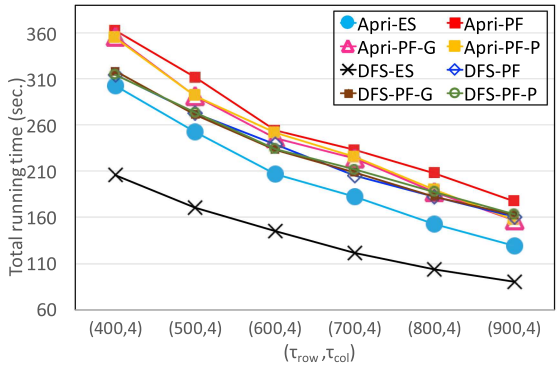


Fig. 15. Total runtime w.r.t. size thresholds (τ_{row}, τ_{col}) on GDS2712 (exponential distributions).

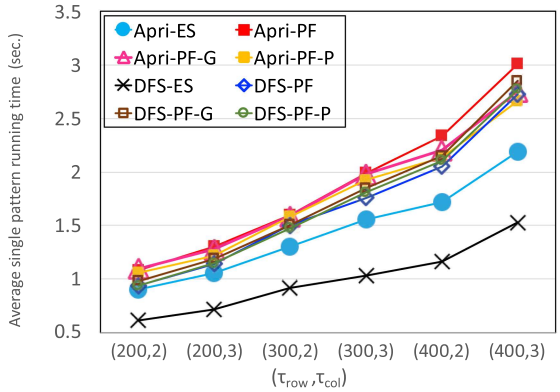


Fig. 16. Average single-pattern runtime w.r.t. (τ_{row}, τ_{col}) on GDS2712 (exponential distributions).

Figures 15 and 16 show the total running time *TR-Time* and average single-pattern running time *SP-Time*, respectively, of our proposed methods adopting exponential value distributions. We can see that the performances of our algorithms are very similar to those in Figures 13 and 14 in the context of Gaussian value distributions, actually slightly more expensive but comparable. This shows that our cubic spline approach is efficient and can handle various value distributions.

Memory Efficiency. We hereby report the experimental results on memory efficiency for GDS2712. The memory cost results on GDS2002 are very similar and thus omitted.

Figure 17 shows the peak memory usage of various algorithms using uniform value distributions. We can see that our algorithms have a consistently lower peak memory consumption than *OPSMRM*. As an illustration, when $(\tau_{row}, \tau_{col}) = (400, 4)$, our *Apri-ES* algorithm consumes 6.6 times less memory than *OPSMRM*. Our less memory usage does not compromise the number of mined OPSMs, as Table 3 already indicates. our algorithms discovered many times more OPSMs than *OPSMRM* and *POPSM*. For example, when $(\tau_{row}, \tau_{col}) = (400, 4)$, our *DFS/Apri-ES* algorithms discovered 2.5 times more OPSMs than *OPSMRM* and 3.7 times more OPSMs than *POPSM*.

Our *DFS/Apri-PF* algorithms consume more memory than *DFS/Apri-ES* due to the need of processing PMF vectors (cf. Section 4.2), but they generate higher-quality OPSMs (in terms of p-value) than *ES* (cf. Figure 8). However, even *DFS/Apri-PF* algorithms have a much lower memory consumption compared with *OPSMRM*. While *POPSM* tends to consume slightly less memory than our algorithms, it outputs much fewer OPSMs.

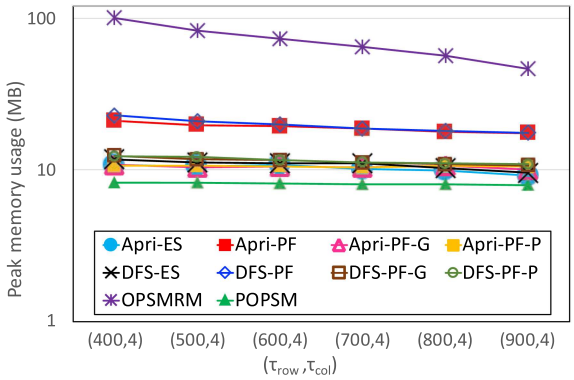


Fig. 17. Peak memory w.r.t. size thresholds (τ_{row}, τ_{col}) on GDS2712 (uniform value distributions).

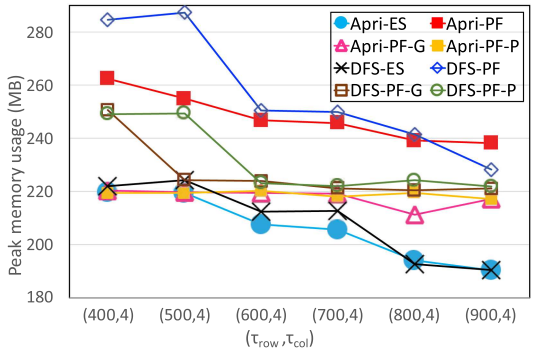


Fig. 18. Peak memory w.r.t. size thresholds (τ_{row}, τ_{col}) on GDS2712 (Gaussian value distributions).

Figure 18 shows the peak memory usage of our algorithms using Gaussian value distributions, and compared with Figure 17, we can see that their memory cost is one order of magnitude higher. This is reasonable, because each matrix entry now introduces seven split points (cf. Figure 6) rather than two as in the uniform interval case, and the recursive integral computation as specified in Equation (11) now requires the maintenance of a high-order polynomial for probability computation.

Figure 19 shows the peak memory usage of our algorithms using exponential value distributions, and compared with Figure 18, we can see that the memory cost is very similar.

7.3 Effect of Probability Threshold and Inclusion Threshold

We next report the experiments on the effect of τ_{cut} and τ_{prob} using the GDS2712 dataset. The results on the other datasets are similar and thus omitted.

Effect of Inclusion Threshold τ_{cut}. We fix $\tau_{row} = 400$ and $\tau_{col} = 3$, since various methods generate comparable number of OPSMs with these parameters. We also fix $\tau_{prob} = 0.5$ for *DFS-PF* and *Apri-PF*, but vary τ_{cut} among $\{0.2, 0.3, 0.4, 0.5, 0.6\}$. Figure 20 shows the fraction of OPSMs in each significance level for every algorithm with different inclusion threshold τ_{cut}. Table 5 presents the number of OPSMs mined with different inclusion threshold τ_{cut}, where we can see that our algorithms consistently find more OPSMs than *POPSM* and *OPSMRM* at various values of τ_{cut} in all significance levels, and those with Gaussian value distributions are better than those with uniform value distributions. In terms of the distribution of OPSMs among various significance levels, we

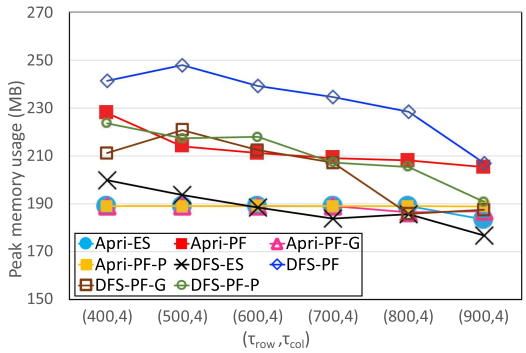


Fig. 19. Peak memory w.r.t. size thresholds (τ_{row}, τ_{col}) on GDS2712 (exponential distributions).

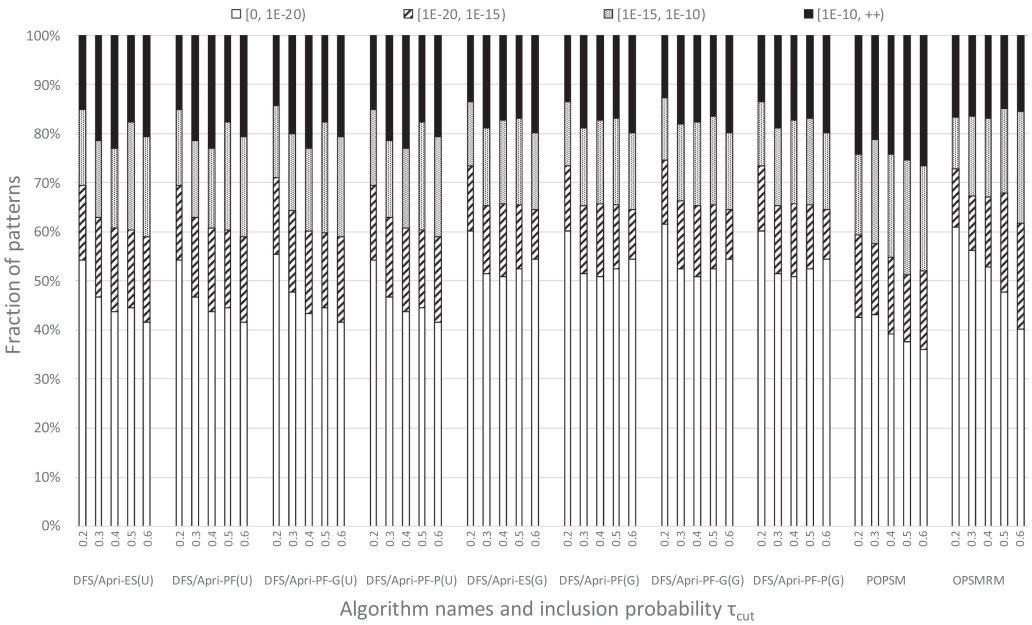


Fig. 20. OPSM result distribution w.r.t. τ_{cut} on GDS2712.

can see from Figure 20 our algorithms consistently have a tall white bar (with those using Gaussian distributions being taller), while OPSMRM is only competitive when τ_{cut} is very low (however, the absolution count is much lower, as shown in Table 5).

Effect of Probability Threshold τ_{prob}. We study the effect of τ_{prob} used in *PF* by fixing ($\tau_{row} = 600, \tau_{col} = 4$) and $\tau_{cut} = 0.2$; and varying τ_{prob} from 0.2 to 0.95 with a step length of 0.05.

First consider *PF(U)* algorithms. Table 6 shows the number of OPSMs mined at each value of τ_{prob}, and Figure 21 shows their distribution in different significance levels. We can see that our approach is very robust, not sensitive to parameter τ_{prob}, and has consistent performance in terms of result quality. Figure 22 shows the *TR-Time* of *PF*, *PF-G*, and *PF-P* algorithms w.r.t. diffierent τ_{prob}. We can see that our DFS-based algorithms are faster than Apriori-based ones, which is often the case when the number of results are small. However, we tested that when the number of results reach thousands, our Apriori-based algorithms are actually faster, since the additional pruning

Table 5. Number of OPSMs w.r.t. Inclusion Thresholds τ_{cut} on GDS2712

	0.2	0.3	0.4	0.5	0.6
DFS/Apri-ES(U)	478	478	430	339	281
DFS/Apri-PF(U)	478	478	430	339	281
DFS/Apri-PF-G(U)	463	463	427	339	281
DFS/Apri-PF-P(U)	478	478	430	339	281
DFS/Apri-ES(G)	692	676	559	429	347
DFS/Apri-PF(G)	692	676	559	429	347
DFS/Apri-PF-G(G)	674	662	559	429	347
DFS/Apri-PF-P(G)	692	676	559	429	347
POPSM	430	340	281	236	192
OPSMRM	192	171	155	128	97

Table 6. Number of OPSMs w.r.t. τ_{prob} on GDS2712 (Uniform Value Distributions)

	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
DFS/Apri-PF	130	128	127	126	124	123	120	120

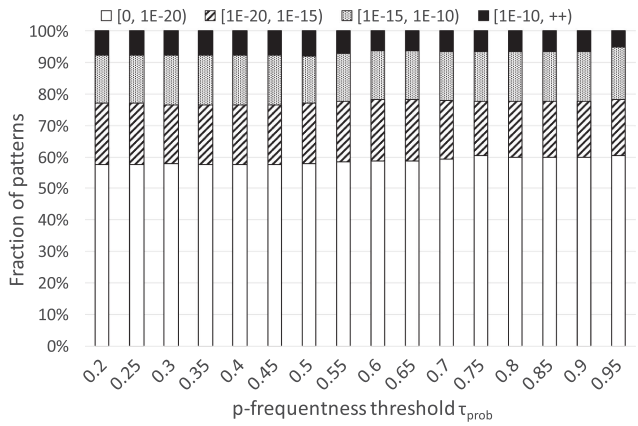


Fig. 21. OPSM result distribution w.r.t. τ_{prob} on GDS2712 (uniform value distributions).

power of the Apriori-based algorithms outweighs the pruning cost. Recall, for example, Figures 36 and 37 for GAL. So, both algorithms have their merits.

We also tested *PF(G)* algorithms, with Table 7 showing the number of OPSMs mined at each value of τ_{prob}, Figure 23 showing their distribution in different significance levels, and Figure 24 showing the *TR-Time* of *PF*, *PF-G*, and *PF-P* algorithms w.r.t. different τ_{prob}. We can observe similar results, except that the *TR-Time* is two orders of magnitude slower (compare Figure 24 with Figure 22), and the result quality is higher, as can been seen from the taller white bars in Figure 23 than in Figure 21, and the larger number of results in Table 7 than in Table 6.

7.4 Comparing Row Selection Strategies

Recall that after (Step 1) mining patterns $P = (t_{i_1}, t_{i_2}, \ldots, t_{i_\ell})$ that are deemed significant, we (Step 2) select those rows whose supporting probabilities are at least τ_{cut} into the OPSM for each pattern P.

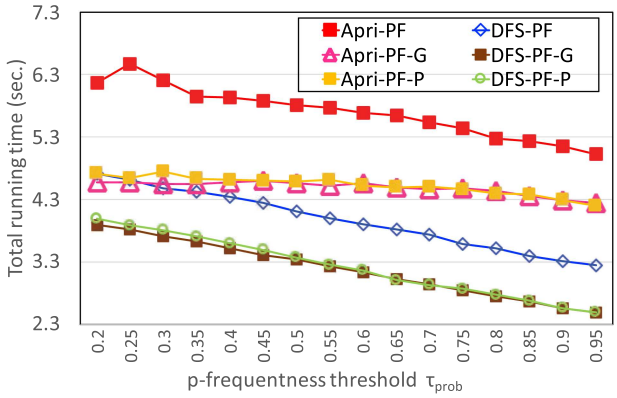


Fig. 22. Total running time w.r.t. τ_{prob} on GDS2712 (uniform value distributions).

Table 7. Number of OPSMs w.r.t. τ_{prob} on GDS2712 (Gaussian Value Distributions)

	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
DFS/Apri-PF	251	249	248	244	236	234	228	223

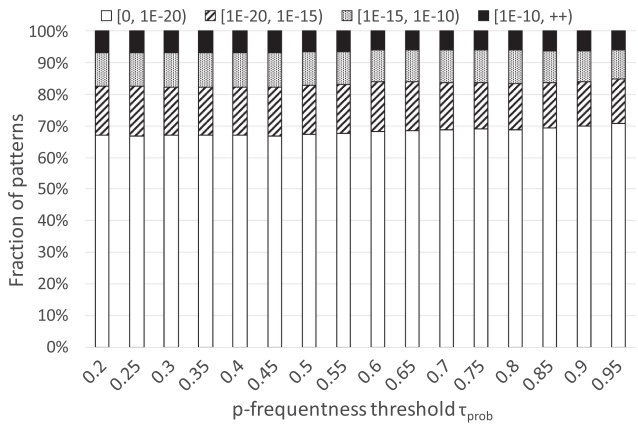


Fig. 23. OPSM result distribution w.r.t. τ_{prob} on GDS2712 (Gaussian value distributions).

We remark that our row selection strategy in Step 2 is reasonable, since τ_{cut} directly reflects the confidence level of supporting probability, which is intuitive to end-users. An alternative approach is to select k rows whose supporting probabilities are the highest, but it is difficult to ask end-users to set a proper k. One way is to set k as τ_{row}, but this tends to be an underestimate, since each of the k rows has $p_g(P) < 100\%$ while τ_{row} is defined with respect to each deterministic possible world.

To compare these two strategies, Figure 25 shows our previous experiments on GDS2712 assuming uniform value distributions, where algorithms ending with "(U)" are those that select rows with $p_g \geq \tau_{cut} = 0.6$ into the OPSMs, and algorithms ending with "(k)" are those that select rows with $k = \tau_{row}$ highest values of p_g into the OPSMs. We can see that the latter row selection method consistently delivers fewer highly significant OPSMs, as the white bars are much shorter. Also, Figure 26 shows our previous experiments on GDS2712 assuming Gaussian value distributions,

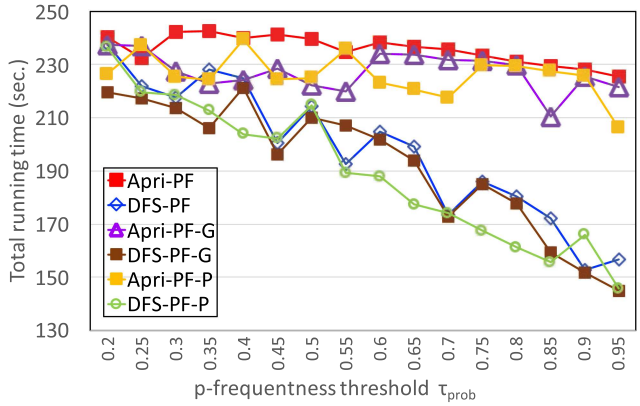


Fig. 24. Total running time w.r.t. τ_{prob} on GDS2712 (Gaussian value distributions).

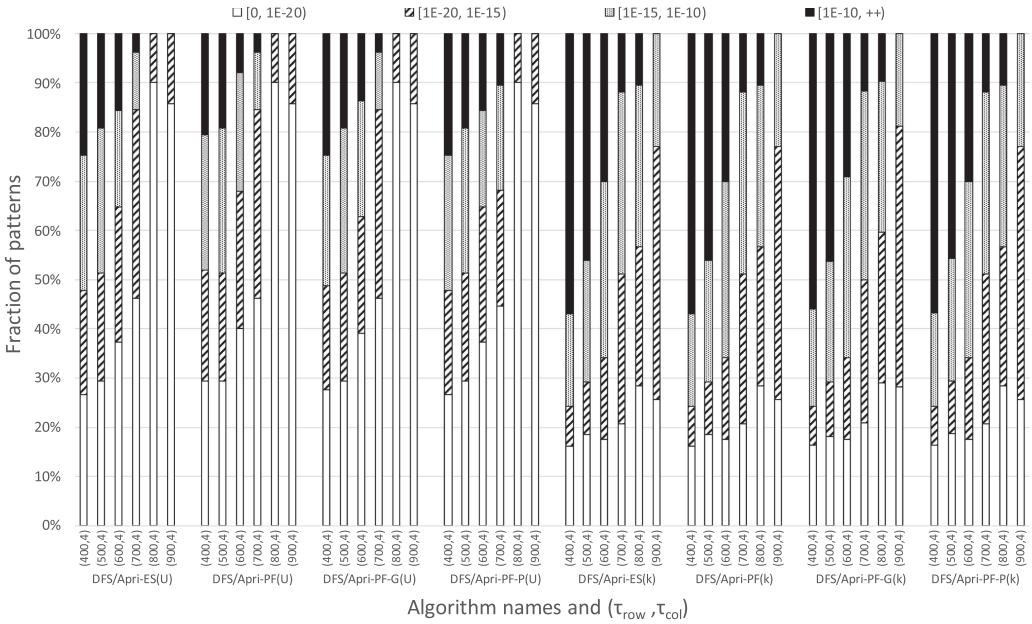


Fig. 25. OPSMs at the four levels for different row selection strategies (uniform value distribution).

and we can obtain the same observation that selecting rows with $k = \tau_{row}$ highest values of p_g is inferior to our adopted strategy of selecting rows by τ_{cut} in terms of result quality.

7.5 Vertical Scalability of Parallel Mining

Recall that Section 6 presents a parallel version of our prefix-projection-based mining algorithm. We now explore its vertical scalability by varying the number of mining threads to be 1, 2, 4, 8, 16, 32, and 64. Figure 27 shows the scalability curve for our four algorithms with uniform value distributions on GDS2712 when $(\tau_{row}, \tau_{col}) = (400, 4)$. Figure 28 shows the scalability curve for our four algorithms with uniform value distributions on GDS2002 when $(\tau_{row}, \tau_{col}) = (200, 2)$. We can see that the running time clearly reduces as we increase the number of mining threads. For

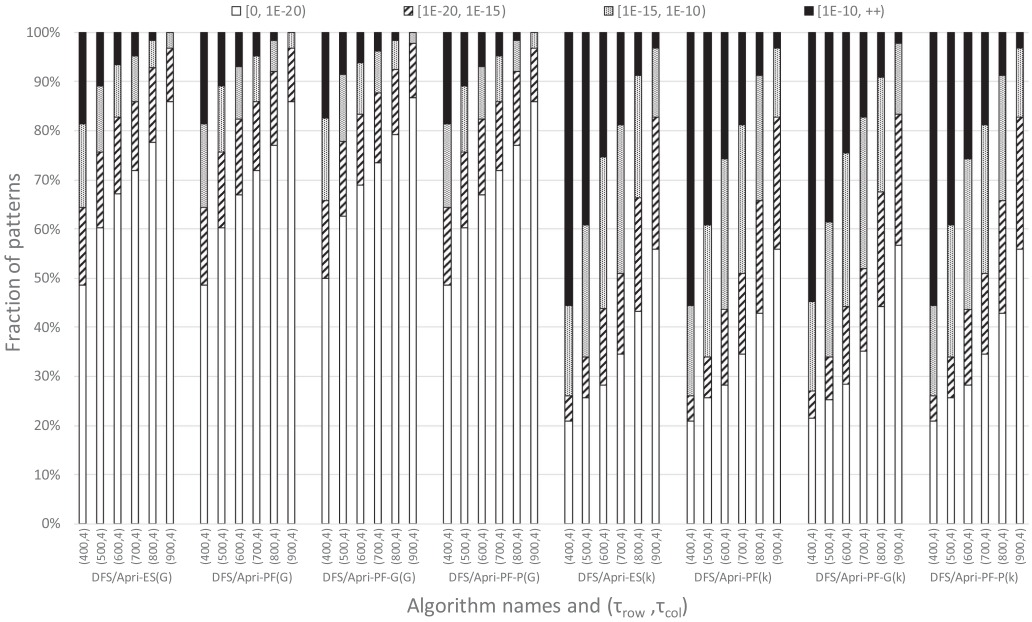


Fig. 26. OPSMs at the four levels for different row selection strategies (Gaussian Value distribution).

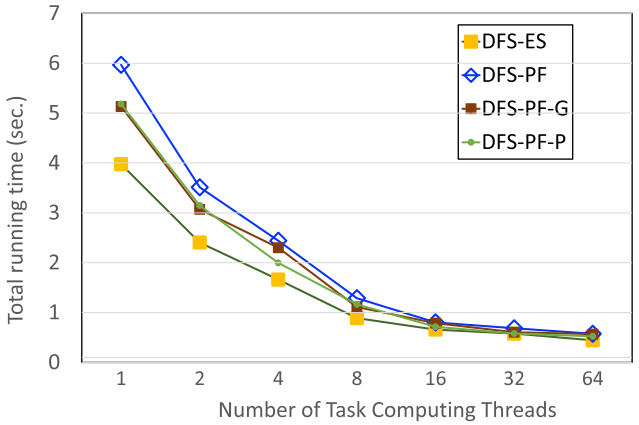


Fig. 27. Total running time w.r.t. # of threads on GDS2712 (uniform value distributions).

example, in Figure 28, our parallel *PF* takes 56.27 seconds when mining with one thread, but the time reduces to 3.80 seconds when mining with 32 threads, and the time reduces to 2.89 seconds when mining with 64 threads. Appendix C reports the results for our algorithms with Gaussian value distributions, where we observe even better speedups. We also report the speedups of our parallel algorithm over the other datasets in Appendix C, as well as the peak memory usage results.

7.6 Experiments on the Movie Rating Dataset

Recall from Section 7.1 that we use a completed movie rating matrix to find OPSMs where users share the same preference order over a set of movies, and that a ***Kendall tau score* (KTS)** is defined to judge how well an OPSM pattern is reflected in its users' movie ratings. We now report

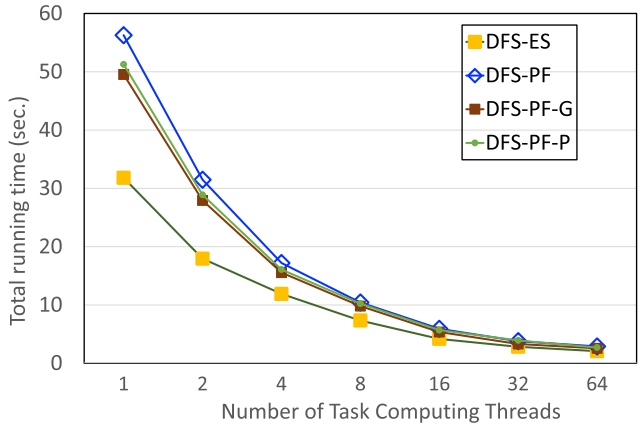


Fig. 28. Total running time w.r.t. # of threads on GDS2002 (uniform value distributions).

Table 8. Total Running Time (sec.) w.r.t. Size Thresholds on the Movie Rating Dataset

	(200, 4)	(200, 5)	(300, 4)	(300, 5)	(400, 4)	(400, 5)
Apri-ES	1,604	1,286	262	225	61	59
Apri-PF	2,871	2,261	473	440	123	125
Apri-PF-G	1,420	1,138	221	189	52	50
Apri-PF-P	1,512	1,212	241	207	58	54
DFS-ES	2,358	1,938	412	370	100	96
DFS-PF	6,345	5,640	1,155	1,104	300	297
DFS-PF-G	4,998	4,107	937	841	252	241
DFS-PF-P	3,678	3,022	808	726	211	202
POPSM	OOM	OOM	14,643	14,209	5,133	5,200
OPSMRM	235	235	216	216	197	197

the results on the movie rating dataset where we added uncertain intervals to the discrete scale ratings.

Time Efficiency. Table 8 shows the *TR-Time* of various algorithms with different (τ_{row}, τ_{col}) (without loss of generality, we fix $\tau_{cut} = 0.3$ and $\tau_{prob} = 0.6$), where *OOM* means out of memory. Note that when $\tau_{row} = 200$, *POPSM* always runs out of memory after 10 hours. *OPSMRM* has the shortest *TR-Time* but it failed to produce any OPSMs that satisfy the τ_{col} threshold. We see that our methods are tens of times faster than *POPSM*, and ES methods are faster than *PF*, as they do not process PMF vectors. Also, our Apriori-based algorithms are faster than their *DFS*-based counterparts due to the effective pattern pruning. Finally, approximation algorithms *PF-P* and *PF-G* provide reasonable speedup to *PF*.

Effectiveness. Without loss of generality, we consider $\tau_{cut} = 0.3$, $\tau_{prob} = 0.6$, $\tau_{row} = 300$, and vary $\tau_{col} = 5$ and visualize the top-10 OPSMs mined by *DFS/Apri-ES*, *DFS/Apri-PF*, and *POPSM* with the highest KTS. We remark that in this set of experiments, our *DFS-ES* consumes 2,582× less memory than *POPSM* and 3,294× less memory than *OPSMRM*.

Table 9(a) lists the top-10 OPSM patterns with the highest KTS produced by *ES*, Table 9(b) by *PF*, Table 9(c) by *PF-P*, Table 9(d) by *PF-G*, and Table 9(e) by *POPSM*. To be space-efficient, we use

Table 9. Movie Pattern Visualization

(a) Top-10 OPSMs by *DFS/Apri-ES*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.52%
TI	ID	MB	IO	MPHG	99.41%
TI	ID	SL	IO	MPHG	99.39%
TI	RK	AN	DW	MPHG	99.37%
TI	ID	JM	IO	MPHG	99.35%
TI	ID	SL	CA	MPHG	99.35%
ET	TK	AN	TT	MPHG	99.27%
TI	MI	AFO	IO	MPHG	99.26%
TI	ID	US	IO	MPHG	99.24%
TI	ID	SS	IO	MPHG	99.20%

(b) Top-10 OPSMs by *DFS/Apri-PF*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.52%
TI	ID	MB	IO	MPHG	99.41%
TI	ID	SL	IO	MPHG	99.39%
TI	ID	JM	IO	MPHG	99.35%
TI	ID	SL	CA	MPHG	99.35%
TI	ID	Jaws	CA	MPHG	99.26%
TI	R	RK	AN	MPHG	99.24%
TI	R	RK	AN	PB	99.18%
TI	R	Jaws	CA	MPHG	99.17%
TI	ET	SL	CA	MPHG	99.17%

(c) Top-10 OPSMs by *DFS/Apri-PF-P*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.52%
TI	ID	MB	IO	MPHG	99.41%
TI	ID	SL	IO	MPHG	99.39%
TI	RK	AN	DW	MPHG	99.37%
TI	ID	JM	IO	MPHG	99.35%
TI	ID	SL	CA	MPHG	99.35%
TI	MI	AFO	IO	MPHG	99.24%
TI	ID	US	IO	MPHG	99.24%
TI	ID	SS	IO	MPHG	99.19%
TI	ID	G	IO	MPHG	99.18%

(d) Top-10 OPSMs by *DFS/Apri-PF-G*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.52%
TI	ID	MB	IO	MPHG	99.41%
TI	ID	SL	IO	MPHG	99.39%
TI	ID	JM	IO	MPHG	99.35%
TI	ID	SL	CA	MPHG	99.35%
TI	ID	Jaws	CA	MPHG	99.26%
TI	R	RK	AN	MPHG	99.24%
TI	R	RK	AN	PB	99.18%
TI	R	Jaws	CA	MPHG	99.17%
TI	ET	SL	CA	MPHG	99.17%

(e) Top-10 OPSMs by *POPSM*

t_1	t_2	t_3	t_4	t_5	KTS
TI	ID	AFO	IO	MPHG	99.32%
TI	RK	AN	CA	MPHG	99.24%
R	RK	AN	PB	MPHG	99.19%
R	RK	AN	IO	MPHG	99.19%
TI	RK	MB	IO	MPHG	99.19%
ET	RK	AN	S	MPHG	99.19%
TI	ID	SL	IO	MPHG	99.19%
TI	ID	MB	IO	MPHG	99.19%
TI	RK	AN	PB	MPHG	99.16%
TI	ID	SL	CA	MPHG	99.15%

abbreviations for movie names, and their meanings are listed in Table 10. We can see that the OPSMs mined by our methods generally have a higher KTS.

Memory Efficiency. Table 11 shows the peak memory usage of our algorithms, *POPSM* and *OPSMRM* w.r.t. size thresholds (τ_{row}, τ_{col}), where *OOM* represents *Out of Memory*. When $(\tau_{row}, \tau_{col}) = (200, 4)$ or $(200, 5)$, *POPSM* ran out of memory after running for 10 hours (or more accurately, 603 minutes). It is clear that our **Expected Support (*ES*)**-based mining algorithms, namely, *Apri-ES* and *DFS-ES*, have a consistently much lower peak memory consumption than all the other algorithms. As an illustration, when $(\tau_{row}, \tau_{col}) = (300, 5)$, *DFS-ES* consumes 2,582 times less memory than *POPSM* and 3,294 times less memory than *OPSMRM*, and our *Apri-ES* consumes 720 times less memory than *POPSM* and 918 times less memory than *OPSMRM*. Also, our PMF

Table 10. Movie Names and Their Abbreviations

Titanic (1997)	TI	Ransom (1996)	R
Independence Day (1996)	ID	Rock, The (1996)	RK
Air Force One (1997)	AFO	Apocalypse Now (1979)	AN
In & Out (1997)	IO	Chasing Amy (1997)	CA
Princess Bride, The (1987)	PB	Schindler's List (1993)	SL
Sting, The (1987)	S	Men in Black (1997)	MB
Dances with Wolves (1990)	DW	Terminator, The (1984)	TT
Mission: Impossible (1996)	MI	Jerry Maguire (1996)	JM
Usual Suspects, The (1995)	US	Sense and Sensibility (1995)	SS
Game, The (1997)	G	E.T. the Extra-Terrestrial (1982)	ET
Monty Python and the Holy Grail (1974)			MPHG

Table 11. Peak Memory w.r.t. Size Thresholds on the Movie Rating Dataset

	(200, 4)	(200, 5)	(300, 4)	(300, 5)	(400, 4)	(400, 5)
Apri-ES	66.9MB	65.1MB	18.6MB	16.5MB	8.9MB	8.3MB
Apri-PF	8.6GB	8.6GB	2.2GB	2.2GB	0.63GB	0.63GB
Apri-PF-G	68.3MB	67.2MB	19.3MB	17.1MB	9.2MB	8.9MB
Apri-PF-P	69.1MB	67.4MB	19.2MB	17.4MB	9.4MB	9.0MB
DFS-ES	4.9MB	5.1MB	4.5MB	4.6MB	4.3MB	4.1MB
DFS-PF	8.6GB	8.6GB	2.2GB	2.2GB	0.63GB	0.63GB
DFS-PF-G	10.1MB	10.1MB	8.4MB	8.1MB	7.7MB	7.7MB
DFS-PF-P	5.2MB	5.4MB	5.4MB	5.5MB	5.7MB	5.4MB
POPSM	OOM	OOM	11.9GB	11.6GB	4.5GB	4.5GB
OPSMRM	14.8GB	14.8GB	14.8GB	14.8GB	14.8GB	14.8GB

approximation-based PF algorithms also use memory comparable to (or more accurately, slightly larger than) the ES counterparts, and the memory usage is three orders of magnitude less than their exact PF counterparts. This shows that our PMF approximation allows PF-based mining algorithms to scale to much larger datasets given the same memory budget, even though the speedup ratio is not as significant.

7.7 Scalability Experiments

To examine how well our algorithms scale to the number of rows and columns of a data matrix D, we duplicate the rows and columns of our 943×100 movie rating submatrix M_s to generate larger datasets for running scalability experiments.

To test row scalability, we duplicated the rows of M_s for 100, 200, 300, 400, and 500 times, and ran the various algorithms on them. Without loss of generality, we set $\tau_{row} = 0.6 * n$ (where n is the row number) and fix $\tau_{col} = 5$ and $\tau_{prob} = 0.6$. Unfortunately, *OPSMRM* runs out of memory even on the smallest data with M_s's rows duplicated for 100 times, so we cannot report its result. The results for the other algorithms are shown in Figure 29, where we observe that POPSM is much slower than our algorithms, and that *Apri-ES*, *Apri-PF-P*, and *Apri-PF-G* are much faster than the other algorithms. Overall, the time of all algorithms scale linearly with row number n, which matches our derived time complexity $O(Cn(m^2 + \log^2 n))$ in Section 5.2 (i.e., almost linear to n and quadratic to m).

To test column scalability, we duplicated the columns of M_s for 2, 4, 8, and 16 times and ran the various algorithms on them. Here, we set $\tau_{row} = 700, \tau_{col} = 5$, and $\tau_{prob} = 0.6$. The results

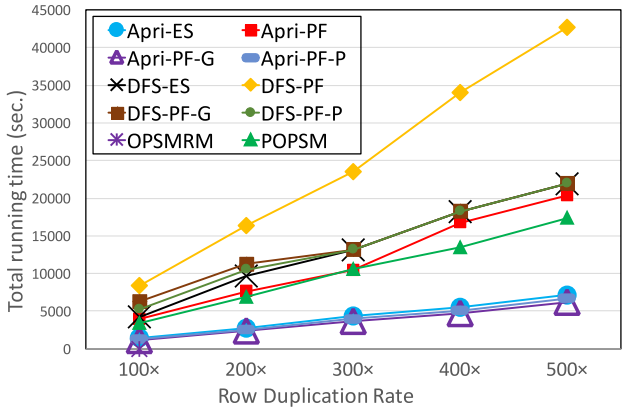


Fig. 29. Scalability to the number of rows on the movie rating dataset.

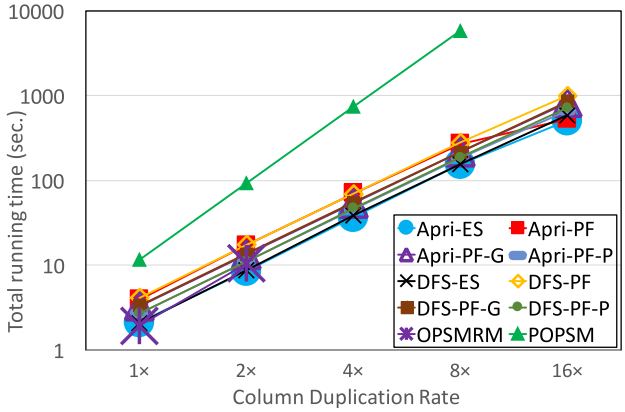


Fig. 30. Scalability to the number of columns on the movie rating dataset.

are shown in Figure 30, where we observe that the running time of various algorithms increase quickly with column number m, which aligns with our analysis. Although *OPSMRM* has a similar running time to our algorithms, the memory consumption rises quickly with more columns. Also, *OPSMRM* runs out of memory when the columns are duplicated for four times and thus, we only plot two points for it in Figure 30. Finally, POPSM is not only slower than all our algorithms, its curve slope is also steeper.

7.8 Experiments on RFID User Trace Dataset

Recall from Section 7.1 that we use an RFID user trace dataset to find OPSM patterns that reveal a group of users who travel to a sequence of locations in the same time order, and that a **trace matching score (TMS)** is defined to judge how well an OPSM pattern is reflected in its users' actual travel trajectories that are manually annotated. We now report the results on the RFID user trace dataset where uniform time intervals are imposed at each visit location (i.e., antenna). We also compare our methods with POPSM and OPSMRM. However, OPSMRM always runs out of memory in this set of experiments and is thus not reported.

In this set of experiments, we use $\tau_{prob} = 0.8$ and $\tau_{cut} = 0.6$ for PF algorithms. Figure 31 shows the fraction of OPSMs in each significance level for our algorithms and *POPSM* with different

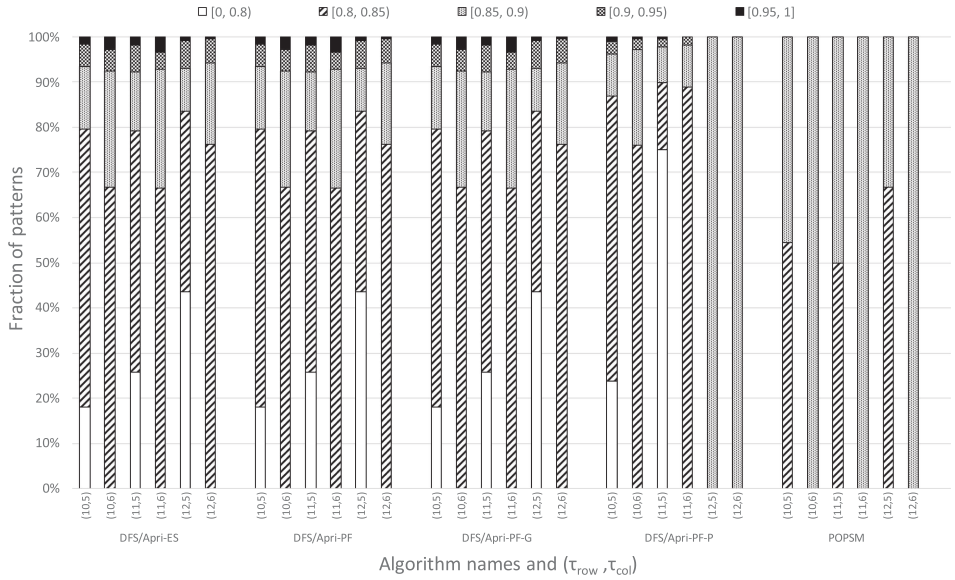


Fig. 31. Fractions of OPSMs at the five significance levels for different size thresholds on RFID.

Table 12. Number of OPSMs w.r.t. Size Thresholds on RFID

	(10, 5)	(10, 6)	(11, 5)	(11, 6)	(12, 5)	(12, 6)
DFS/Apri-ES	8,201	2,787	5,713	2,161	3,381	938
DFS/Apri-PF	8,201	2,787	5,713	2,161	3,381	938
DFS/Apri-PF-G	8,102	2,749	5,588	2,103	3,274	884
DFS/Apri-PF-P	1,422	384	452	54	45	23
POPSM	11	5	8	4	6	2

(τ_{row}, τ_{col}), and Table 12 presents the number of OPSMs mined. We can see that *POPSM* mines very few patterns compared with our algorithms, and all OPSMs are with TMS in the range of $[0.8, 0.9)$, while our algorithms are able to find OPSMs with TMS > 0.9 (i.e., black bars). Moreover, *PF-G* still achieves performance close to that of *PF* even when *PF-P* fails to get close. This verifies that PF-G is more robust than PF-P, thanks to its use of both the expected support and the standard deviation (i.e., μ and δ) while FP-P only uses the expected support, as we have indicated at the end of Section 4.4.2.

Figures 32 and 33 show the total running time *TR-Time* and average single-pattern running time *SP-Time*, respectively, of our algorithms and *POPSM* for different size thresholds (τ_{row}, τ_{col}). From Figure 32, we can see that *POPSM* runs faster than our methods, but recall that *POPSM* mined significantly less OPSM patterns than our algorithms and their results are of poorer quality. Also, *PF-P* algorithms are also fast but recall that the number of OPSMs found is much smaller than our other algorithms. According to Figure 33, *POPSM* has the worst *SP-Time*, followed by *PF-P* algorithms, due to their small number of OPSMs found.

Figure 34 shows the peak memory usage of our algorithms and *POPSM*. We can see that the memory cost of *PF* algorithms is much higher than the other algorithms. In this set of experiments *ES* and *PF-G* have similar result quantity and quality, showing that *ES* is effective in certain datasets, and *PF-G* is a good approximation method that reduces the cost significantly while retaining similar results as with *PF*.

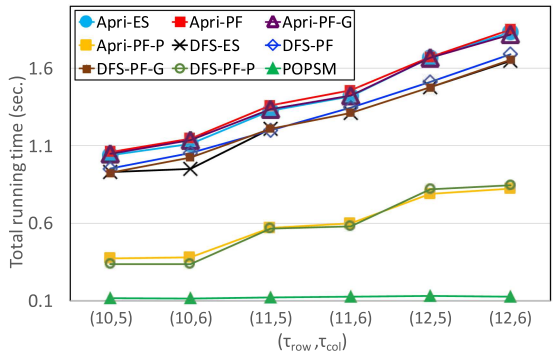


Fig. 32. Total runtime w.r.t. size thresholds (τ_{row}, τ_{col}) on RFID.

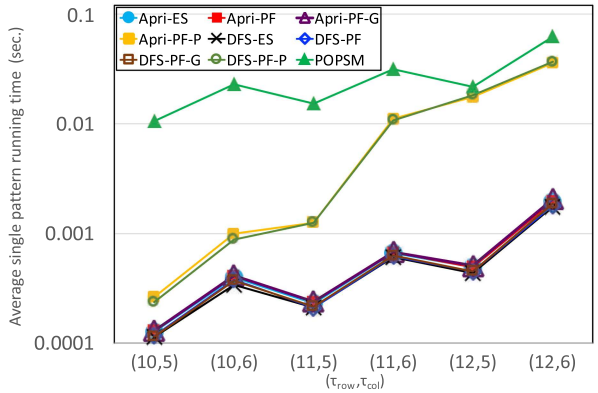


Fig. 33. Average single-pattern runtime w.r.t. (τ_{row}, τ_{col}) on RFID.

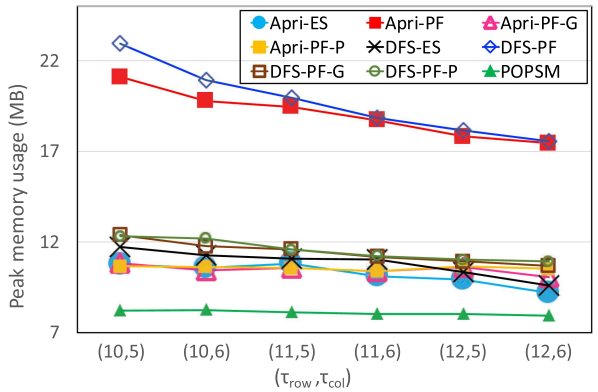


Fig. 34. Peak memory w.r.t. size thresholds (τ_{row}, τ_{col}) on RFID.

8 RELATED WORK

In this section, we review the most related studies, i.e., OPSM mining (especially over noisy matrices). Appendix D discusses additional related works on ranking uncertain data and frequent pattern mining over uncertain data.

The existing works that are closely related to ours are References [4, 12, 33]. The OPSM mining problem is a kind of biclustering problem [2] and is first introduced in Reference [4] to analyze gene expression data. Ben-Dor et al. [4] prove that the problem is NP-hard and propose a heuristic mining algorithm to mine significant OPSMs. Later, Cheung et al. [6] develop a more efficient mining algorithm based on a new data structure, the head-tail trees. Gao et al. [13] define the concept of twig clusters, which are OPSMs with large numbers of columns (and thus low supports), and propose the KiWi framework to efficiently mine the twig clusters.

Although the above studies are able to discover interesting biological associations between genes and tests, they are too restrictive in practice, since real gene expression data are noisy. OP-clustering [21] generalizes the OPSM model by grouping attributes into equivalent classes and uses a data structure, OPC-tree, to mine the generalized OPSMs. More recently, other more noise-tolerant submatrix models are proposed by relaxing the order requirement, such as **Approximate Order-Preserving Cluster (AOPC)** [34] and **Relaxed Order-Preserving Submatrix (ROPSM)** [11].

The OPSMRM Model. Yip et al. [33] combat noise in microarray analysis by letting each test be repeated to obtain several measurements, forming possible worlds with discrete probability distribution. Only expected support is used to evaluate pattern significance, and for each significant pattern, all rows whose supporting probability is at least the inclusion threshold τ_{cut} are selected to compose a submatrix. However, as Section 7 has shown, *PF* generates higher-quality OPSMs than *ES* due to considering the whole PMF, and the interval model delivers better results.

The POPSM Model. Fang et al. [12] attempt to model the underlying distributions that generate the observed measurements, and thus, the value in each matrix entry is given by an interval with its associated continuous probability distribution. However, instead of defining pattern frequentness according to the possible world semantics, POPSM adopts a simple requirement that every row in an output OPSM has its supporting probability no less than a user-defined threshold τ_{cut}, which is unclear how to set properly due to the lack of semantics from the perspective of probability theory.

9 CONCLUSION

We studied the problem of probabilistically-frequent OPSM mining over a data matrix with continuous value uncertainty, following the well-established possible world semantics. To our knowledge, this is the first OPSM mining work that combines both possible world semantics and interval-based data model. We proposed many techniques to efficiently determine pattern significance (including approximation methods) and to prune unpromising patterns. Experiments show that our algorithms find OPSMs with much higher quality than existing approaches, and the time efficiency and scalability (e.g., memory usage) is also competitive.

APPENDIX

A PROOFS

A.1 Proof of Theorem 3.1

Proof.

$$\begin{aligned} & Pr\{(x_1, \ldots, x_n \in [\ell, r]) \wedge (x_1 < x_2 < \cdots < x_n)\} \\ = & \int_\ell^r \left[\int_\ell^{x_n} \left(\cdots \int_\ell^{x_2} p_1 dx_1 \cdots \right) p_{n-1} dx_{n-1} \right] p_n dx_n \\ = & \left(\prod_{i=1}^{n} p_i \right) \cdot \int_\ell^r dx_n \int_\ell^{x_n} dx_{n-1} \cdots \int_\ell^{x_2} dx_1 \end{aligned}$$

Let us define $x_{n+1} = r$. Then, if we can prove that the following equation holds for any positive integer i

$$\int_{\ell}^{x_{i+1}} dx_i \int_{\ell}^{x_i} dx_{i-1} \cdots \int_{\ell}^{x_2} dx_1 = \frac{(x_{i+1} - \ell)^i}{i!}, \tag{23}$$

then Theorem 3.1 is proved by setting $i = n$.

We prove Equation (23) by induction:

- **Base Case:** when $i = 1$, we have

$$\int_{\ell}^{x_2} dx_1 = x_2 - \ell = \frac{(x_2 - \ell)^1}{1!};$$

- **Inductive Step:** Suppose that

$$\int_{\ell}^{x_i} dx_{i-1} \cdots \int_{\ell}^{x_2} dx_1 = \frac{(x_i - \ell)^{i-1}}{(i-1)!},$$

then

$$\begin{aligned}
&\int_{\ell}^{x_{i+1}} dx_i \int_{\ell}^{x_i} dx_{i-1} \cdots \int_{\ell}^{x_2} dx_1 \\
= &\int_{\ell}^{x_{i+1}} \frac{(x_i - \ell)^{i-1}}{(i-1)!} dx_i \\
= &\frac{1}{(i-1)!} \int_{\ell}^{x_{i+1}} (x_i - \ell)^{i-1} d(x_i - \ell) \\
= &\frac{1}{(i-1)!} \int_{\ell}^{x_{i+1}} d\frac{(x_i - \ell)^i}{i} = \frac{(x_{i+1} - \ell)^i}{i!}.
\end{aligned}$$

□

A.2 Proof of Theorem 4.1

Proof. Suppose that the rows in set S is divided into two sets S_1 and S_2. It is sufficient to prove that, when P is p-frequent in S_1, it is also p-frequent in S.

Let us denote the support of P in a set S by X^S, and denote the PMF (and CDF) of X^S by $f^S(c)$ (and $F^S(c)$). When P is p-frequent in S_1, according to Equation (2), we have

$$1 - F^{S_1}(\tau_{row} - 1) = Pr\{X^{S_1} \geq \tau_{row}\} \geq \tau_{prob}. \tag{24}$$

According to Equation (24), $F^{S_1}(\tau_{row}-1) \leq 1-\tau_{prob}$. If we can prove $F^S(\tau_{row}-1) \leq F^{S_1}(\tau_{row}-1)$, then we are done, since this implies $F^S(\tau_{row} - 1) \leq F^{S_1}(\tau_{row} - 1) \leq 1 - \tau_{prob}$, or equivalently, $Pr\{X^S \geq \tau_{row}\} = 1 - F^S(\tau_{row} - 1) \geq \tau_{prob}$.

We now prove $F^S(\tau_{row} - 1) \leq F^{S_1}(\tau_{row} - 1)$. Let us denote $\tau'_{row} = \tau_{row} - 1$. Then, we obtain

$$\begin{aligned}
F^S(\tau'_{row}) &= \sum_{i+j=0}^{\tau'_{row}} f^{S_1}(i) \times f^{S_2}(j) \\
&= \sum_{i=0}^{\tau'_{row}} \sum_{j=0}^{\tau'_{row}-i} f^{S_1}(i) \times f^{S_2}(j) \\
&= \sum_{i=0}^{\tau'_{row}} f^{S_1}(i) \times \sum_{j=0}^{\tau'_{row}-i} f^{S_2}(j)
\end{aligned}$$

$$= \sum_{i=0}^{\tau'_{row}} f^{S_1}(i) \times F^{S_2}(\tau'_{row} - i)$$

$$\leq \sum_{i=0}^{\tau'_{row}} f^{S_1}(i) = F^{S_1}(\tau'_{row}) \quad \square$$

A.3 Proof of Pattern Pruning Rules in Section 4.3

(1) Count-Prune. Let $cnt(P) = |\{g \in G \mid p_g(P) > 0\}|$, then pattern P is not p-frequent if $cnt(P) < \tau_{row}$.

Proof. When $cnt(P) < \tau_{row}$, $Pr\{X \geq \tau_{row}\} \leq Pr\{X > cnt(P)\} = 0$. $\square$

(2) Markov-Prune. Pattern P is not p-frequent if

$$\sum_{g \in G} p_g(P) = E(X) < \tau_{row} \times \tau_{prob}.$$

Proof. According to Markov's inequality, $E(X) < \tau_{row} \times \tau_{prob}$ implies that $Pr\{X \geq \tau_{row}\} \leq E(X)/\tau_{row} < \tau_{prob}$. $\square$

(3) Exponential-Prune. Let $\mu = E(X)$ and $\delta = \frac{\tau_{row} - \mu - 1}{\mu}$. When $\delta > 0$, pattern P is not p-frequent if

(1) $\delta \geq 2e - 1$, and $2^{-\delta\mu} < \tau_{prob}$, or

(2) $0 < \delta < 2e - 1$, and $e^{-\frac{\delta^2\mu}{4}} < \tau_{prob}$.

Proof. According to Chernoff Bound, we have

$$Pr\{X > (1+\delta)\mu\} < \begin{cases} 2^{-\delta\mu}, & \delta \geq 2e - 1 \\ e^{-\frac{\delta^2\mu}{4}}, & 0 < \delta < 2e - 1 \end{cases},$$

and if we set $\delta = \frac{\tau_{row} - \mu - 1}{\mu}$, i.e., $(1+\delta)\mu = \tau_{row} - 1$, we have $Pr\{X > (1+\delta)\mu\} = Pr\{X \geq \tau_{row}\}$. $\square$

B PERFORMANCE COMPARISON ON GAL

Effect of Size Thresholds τ_{row} and τ_{col}. We follow [12]'s parameter settings of τ_{row} and τ_{col} for evaluating of GAL by fixing $\tau_{cut} = 0.5$ and $\tau_{prob} = 0.95$. This is because with $\tau_{prob} = 0.95$ the number of OPSMs mined by different algorithms are similar to each other, which makes it fair to compare the percentage of patterns in each significance level. Also, since the results are similar for various τ_{cut} values, we only show results when $\tau_{cut} = 0.5$ to save space (see Section 7.3 for more results on the effect of τ_{cut}).

Result Quality. We vary τ_{row} among $\{40, 50, 60\}$ and vary τ_{col} among $\{4, 5\}$. Figure 35 presents the fraction of mined OPSMs that fall in each *significance level*. For example, the first bar "(40, 4), DFS-ES/Apri-ES(U)" represents the distribution of the patterns mined by *DFS-ES(U)* or *Apri-ES(U)* with $\tau_{row} = 40, \tau_{col} = 4$. We see that our algorithms that assume Gaussian value distributions find apparently larger fractions of high-quality OPSMs, than uniform ones, followed by POPSM and then by OPSMRM. This can be seen from how tall white bars are (representing the highest significance level with p-value $\in [0, 10^{-40})$). For example, when $(\tau_{row}, \tau_{col}) = (60, 4)$, our proposed *ES(G)* and *PF(G)* have 60.2% and 68.2% patterns falling into the highest significance level, respectively; our proposed *ES(U)* and *PF(U)* have 51.4% and 61.7% patterns falling into this level, respectively; while *POPSM* and *OPSMRM* have only 36.9% and 39.7% falling into this level, respectively. Among

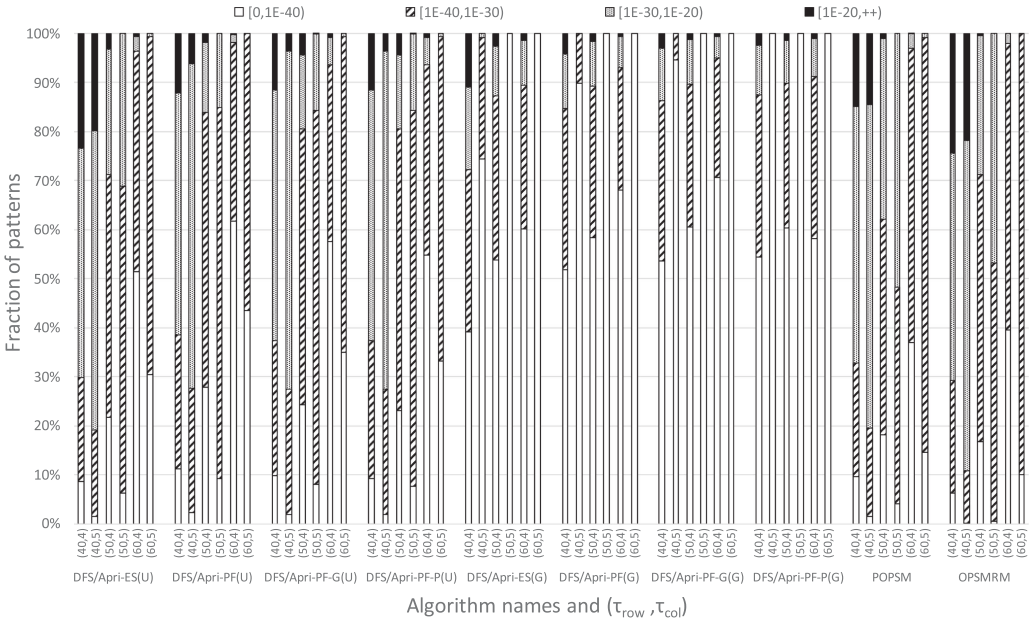


Fig. 35. OPSM Result Distribution w.r.t. Size Thresholds on GAL.

Table 13. Number of OPSMs w.r.t. Size Thresholds on GAL

	(40, 4)	(40, 5)	(50, 4)	(50, 5)	(60, 4)	(60, 5)
DFS/Apri-ES(U)	6,296	3,246	2,537	834	1,033	144
DFS/Apri-PF(U)	4,867	2,253	1,967	562	804	94
DFS/Apri-PF-G(U)	4,179	1,842	1,751	452	658	50
DFS/Apri-PF-P(U)	4,394	2,265	1,771	582	720	100
DFS/Apri-ES(G)	4,626	693	1,916	92	749	7
DFS/Apri-PF(G)	2,533	274	1,030	36	333	4
DFS/Apri-PF-G(G)	2,136	168	874	23	286	4
DFS/Apri-PF-P(G)	1,772	80	566	17	103	4
POPSM	5,687	3,177	3,003	1,290	1,475	355
OPSMRM	2,989	887	1,141	169	406	10

our algorithms, *PF* performs the best in terms of result distribution in significant levels, followed by *PF-G* and *PF-P* and then by *ES*.

Table 13 presents the number of OPSMs mined, and we can see that *ES(U)* has more OPSMs than *PF(U)* and *POPSM*, followed by *PF-P(U)* and *PF-G(U)*, and then by *OPSMRM*. We can see that *ES* and *POPSM* may work better than *PF* for small datasets (GAL has only 205 rows compared with GDS2712 with 7,826 rows and GDS2002 with 5,474 rows), but OPSMRM is always the worst showing the need of a continuous uncertain model. When the number of rows is large, *PF* is better because the p-frequentness computation more accurately captured due to the larger number of samples for the underlying Poission-binomial distribution.

The above comparisons are for algorithms that assume a uniform PDF or PMF. When Gaussian value distributions are adopted, the number of patterns is reduced compared with a uniform distribution assumption, as can be seen from Table 13, which is different from the observations on

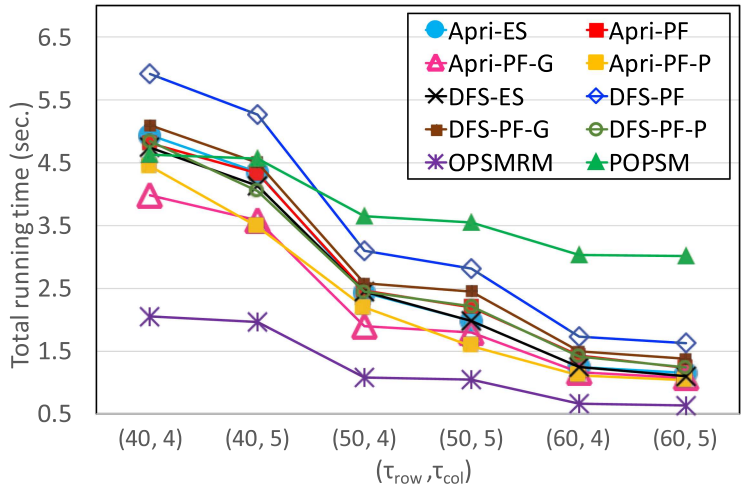


Fig. 36. Total Runtime w.r.t. Size Thresholds (τ_{row}, τ_{col}) on GAL (Uniform Value Distributions).

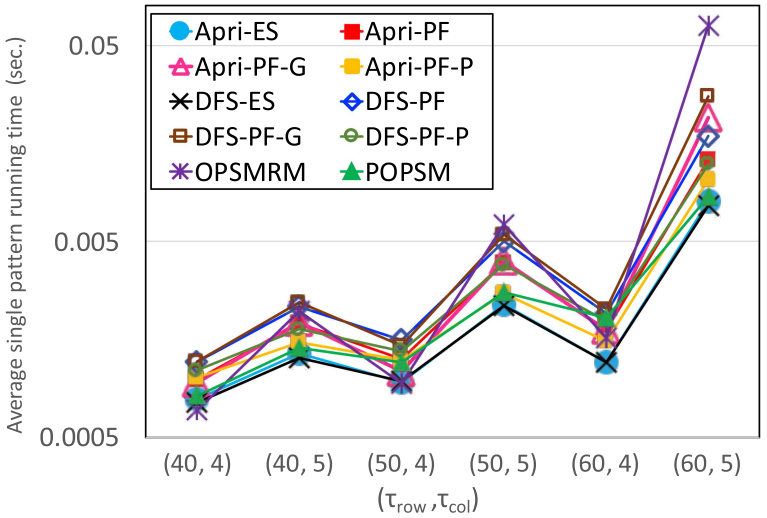


Fig. 37. Average Single-Pattern Runtime w.r.t. (τ_{row}, τ_{col}) on GAL (Uniform Value Distributions).

GDS datasets. This is likely because most probability mass in Gaussian is concentrated around the mean, making the support probability much lower for those patterns P that deviate slightly from the order of the mean entry values. Apparently, Gaussian distributions are too strict and tend to miss such patterns (i.e., the recall is not high) on GAL, but recall from Figure 35 that the fraction of significant OPSMs found is high (i.e., the precision is high). We expect that if more replicates are observed for each entry, we can better capture the value distribution, and the recall of using Gaussian distributions will improve and has the potential to beat the algorithm variants using uniform distributions.

Time Efficiency. Figures 36 and 37 show the total running time *TR-Time* and average single-pattern running time *SP-Time*, respectively, of our proposed methods adopting uniform value distributions, as well as the existing algorithms *OPSMRM* and *POPSM* for different (τ_{row}, τ_{col}). From

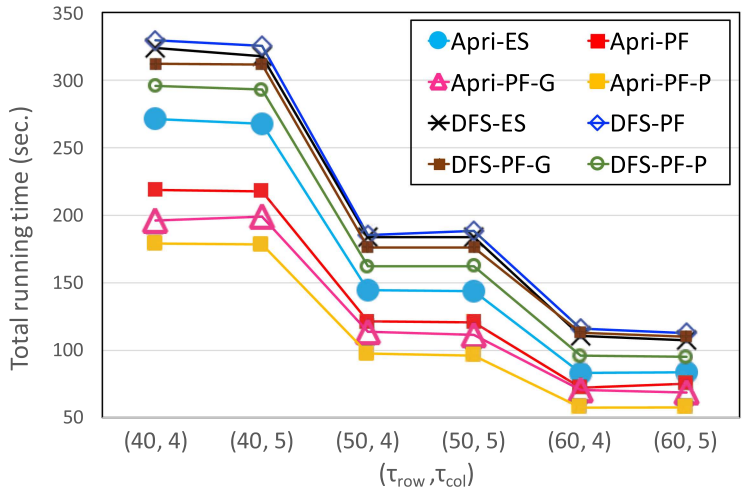


Fig. 38. Total Runtime w.r.t. Size Thresholds (τ_{row}, τ_{col}) on GAL (Gaussian Value Distributions).

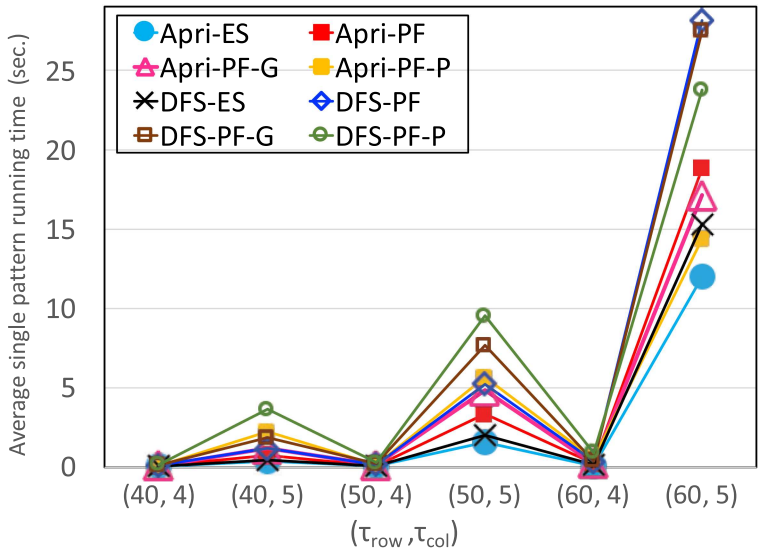


Fig. 39. Average Single-Pattern Runtime w.r.t. (τ_{row}, τ_{col}) on GAL (Gaussian Value Distributions).

Figure 36, we can see that our fast Apriori-based mining algorithms *Apri-ES* and *Apri-PF* have a shorter total running time compared with *POPSM*, except when $(\tau_{row}, \tau_{col}) = (40, 4)$ where *Apri-ES* is 0.3 second slower than *POPSM* (but discovers 611 more OPSMs). *OPSMRM* has the shortest running time among all the methods, but it discovers the fewest number of OPSMs. In fact, according to Figure 37, *OPSMRM* has the worst *SP-Time*. *Apri-ES* and *Apri-PF* perform better than POPSM in terms of *SP-Time*, as they discovered more OPSMs. *PF-P* and *PF-G* algorithms are always faster than *PF*, and while *Apri-PF-P* is faster than *Apri-ES*, *DFS-PF-G* is slightly slower than *DFS-ES*.

Figures 38 and 39 show the total running time *TR-Time* and average single-pattern running time *SP-Time*, respectively, of our proposed methods adopting Gaussian value distributions. We can see that the comparative performances of our algorithms are very similar to those in Figures 36

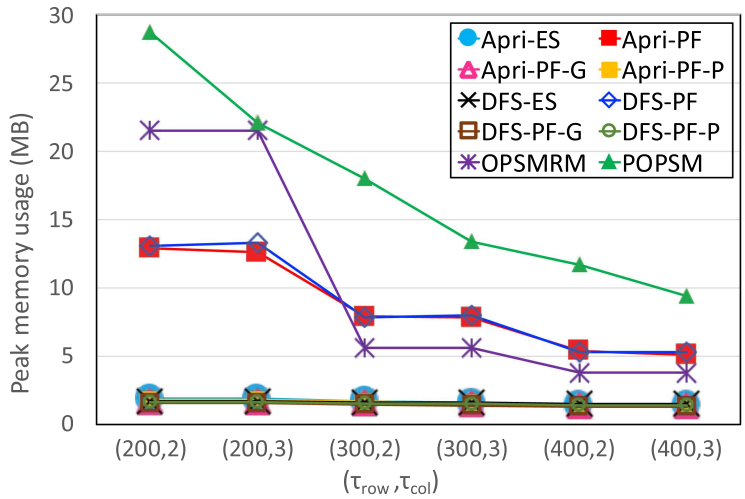


Fig. 40. Peak Memory w.r.t. Size Thresholds (τ_{row}, τ_{col}) on GAL (Uniform Value Distributions).

and 37 in the context of uniform value distributions. However, algorithms using Gaussian value distributions are two orders of magnitude more expensive.

Memory Efficiency. Figure 40 shows the peak memory usage of various algorithms using uniform value distributions. We can see that *Apri-ES*, *DFS-ES*, *Apri-FP-G*, *DFS-FP-G*, *Apri-FP-P*, and *DFS-FP-P* have a much lower memory consumption compared with the other algorithms. For example, when $(\tau_{row}, \tau_{col}) = (40, 4)$, *DFS-ES* consumes 12.6 times less memory than *OPSMRM*, and 16.9 times less memory than *POPSM*. As Table 13 shows, our less memory usage does not compromise the number of mined OPSMs: for example, when $(\tau_{row}, \tau_{col}) = (40, 4)$, *ES* discovers 2.1 times more OPSMs than *OPSMRM* and 1.1 times more OPSMs than *POPSM*. Our *PF* algorithms use more memory than *ES* due to the need of processing PMF vectors (cf. Section 4.2), but they generate higher-quality OPSMs (in terms of p-value) than *ES* (cf. Figure 35). Compared with *OPSMRM* and *POPSM*, *PF* algorithms have a much lower memory consumption than *OPSMRM* and *POPSM* especially when $(\tau_{row}, \tau_{col}) = (40, 4)$ and $(40, 5)$ where the number of OPSMs mined are large. In fact, as Figure 40 shows, POPSM consistently uses the most memory under different parameter settings.

Figure 41 shows the peak memory usage of our algorithms using Gaussian value distributions, and compared with Figure 40, we can see that their memory cost is one order of magnitude higher. This is reasonable because each matrix entry now introduces 7 split points (cf. Figure 6) rather than 2 as in the uniform interval case, and the recursive integral computation as specified in Equation (11) now requires the maintenance of a high-order polynomial for probability computation.

C ADDITIONAL EXPERIMENTS ON SCALABILITY OF PARALLEL MINING

Figure 42 shows the scalability curve for our four algorithms with Gaussian value distributions on GDS2712 when $(\tau_{row}, \tau_{col}) = (400, 4)$. Figure 43 shows the scalability curve for our four algorithms with Gaussian value distributions on GDS2002 when $(\tau_{row}, \tau_{col}) = (200, 2)$. We can see that the running time clearly reduces as we increase the number of mining threads.

Table 14 reports the running times of our serial prefix-projection-based mining algorithms, their parallel counterparts with 1 thread and 32 threads, respectively, and the speedup achieved with 32 threads over 1 thread, when using uniform value distributions. Note that our parallel algorithm running with 1 thread takes around twice the time of our serial algorithm. This is because the serial

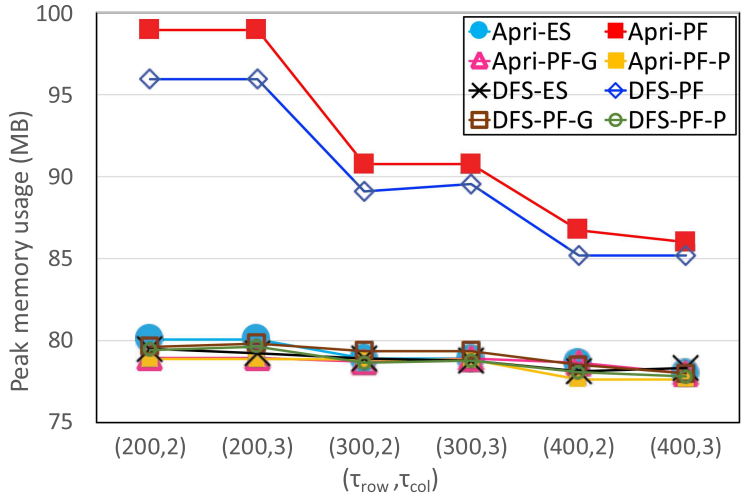


Fig. 41. Peak Memory w.r.t. Size Thresholds (τ_{row}, τ_{col}) on GAL (Gaussian Value Distributions).

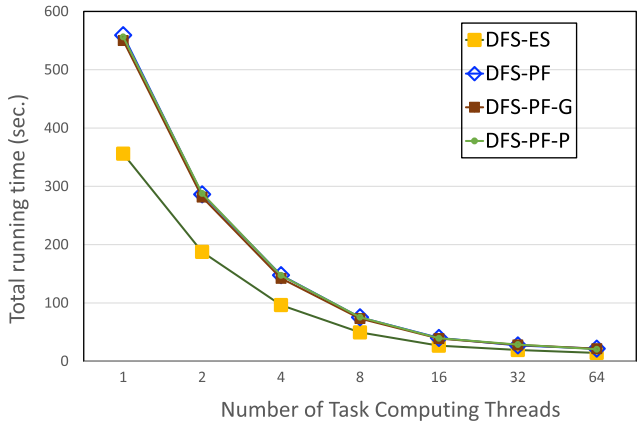


Fig. 42. Total Running Time w.r.t. # of Threads on GDS2712 (Gaussian Value Distributions).

mining algorithm was run on a PC with 4GHz CPU, but our parallel mining algorithm was run on a server with 3,491 MHz CPUs. Compared with the serial program, our parallel implementation also introduces some additional overheads when running with 1 mining thread, such as task creation and scheduling cost, but overall the performance is comparable. In contrast, increasing the number of threads in our parallel program can often lead to over one order of magnitude of speedup, which shows the effectiveness of our parallelization in improving the mining efficiency.

Table 15 reports the running times of our serial and parallel prefix-projection-based mining algorithms, when using Gaussian value distributions. We can observe an even better speedup often exceeding 20× when running with 32 threads, which is because using Gaussian value distributions leads to a much heavier mining workload, allowing the tasks to be more evenly distributed among the mining threads.

Tables 16 and 17 report the peak memory consumption of our parallel mining programs, where we can see that while the memory usage increases with the number of threads, it does not increase

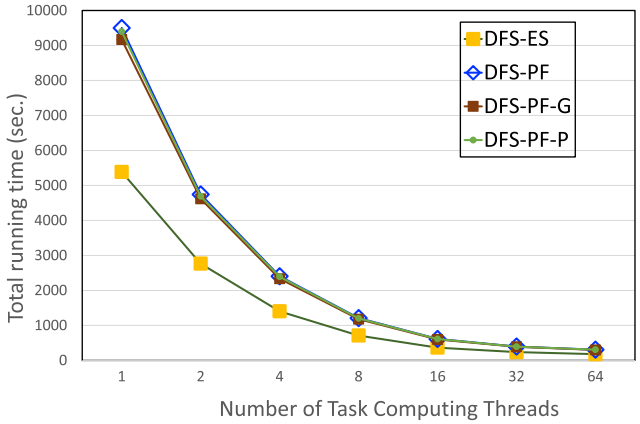


Fig. 43. Total Running Time w.r.t. # of Threads on GDS2002 (Gaussian Value Distributions).

much thanks to the depth-first task scheduling strategy that prioritizes deeper-expanded patterns for processing to keep the number of active patterns/tasks effectively under control.

D ADDITIONAL RELATED WORK

This section discusses additional related works on ranking uncertain data, and frequent pattern mining over uncertain data.

D.1 Ranking Uncertain Data

The problem of top-k queries (or ranking queries) has been extensively studied over uncertain databases, in which each tuple is associated with an uncertain score, and the objective is to find top-k tuples with the largest ranking scores in the probabilistic sense.

Many semantics for top-k queries on uncertain data are proposed in the literature, and they can be categorized into 3 classes, based on the uncertain data models they use:

- **Tuple-Level Uncertain Data Model.** Figure 44(a) illustrates an uncertain database conforming to the tuple-level uncertain data model, where the tuples record the velocity measurements of cars detected by different radars at a location on a highway. Due to device limitations, the detection is not 100% accurate and each tuple is thus associated with an occurrence probability. Furthermore, different tuples may refer to the same car (e.g., tuples 2 and 6), but only one of them may occur (e.g., the velocity of *Car* 2 is 120 with 70% probability and is 80 with 30% probability). Many ranking semantics are proposed in this context, including References [9, 14, 17, 20, 24, 28, 36].
- **Discrete Attribute-Level Uncertain Data Model.** Figure 44(b1) illustrates an uncertain database conforming to the attribute-level uncertainty model, for the same velocity monitoring application mentioned above. In this database, each tuple corresponds to a unique car, and its speed (i.e., ranking score) is described by a probability mass function (PMF). Cormode et al. [9] study top-k queries in this context.
- **Continuous Attribute-Level Uncertain Data Model.** Figure 44(b2) illustrates an uncertain database conforming to the attribute-level uncertainty model, where the attribute value is represented by an interval. The distribution of the attribute value over the interval is usually specified by a probability density function (PDF), such as that of a uniform or Gaussian distribution. Li et al. [19] and [23] study top-k queries in this context.

Table 14. Total Runtime (Seconds) on Different Datasets (Uniform Value Distributions)

Dataset	Algorithm	Serial	1 Thread	32 Threads	Speedup
GDS2002	DFS-ES	13.43	31.81	2.83	11.24
	DFS-PF	20.89	56.26	3.79	14.82
	DFS-PF-G	18.11	49.53	3.30	14.98
	DFS-PF-P	23.77	51.27	3.92	13.06
GDS2712	DFS-ES	1.83	3.97	0.57	6.89
	DFS-PF	2.68	5.96	0.68	8.71
	DFS-PF-G	2.2	5.13	0.60	8.47
	DFS-PF-P	1.9	5.18	0.58	8.94
GAL	DFS-ES	4.74	6.15	0.51	12.1
	DFS-PF	5.91	5.81	0.44	13.35
	DFS-PF-G	5.09	5.12	0.41	12.26
	DFS-PF-P	4.84	4.71	0.35	13.16
Movie	DFS-ES	2,358	3,961.88	458.56	8.64
	DFS-PF	6,345	15,792.4	715.01	22.09
	DFS-PF-G	4,998	14,574.7	653.73	22.29
	DFS-PF-P	3,678	15,606	710.85	21.95
RFID	DFS-ES	1.65	2.72	0.35	7.76
	DFS-PF	1.69	2.84	0.33	8.58
	DFS-PF-G	1.65	2.73	0.31	8.75
	DFS-PF-P	0.85	1.43	0.19	7.39

Table 15. Total Runtime (Seconds) on Different Datasets (Gaussian Value Distributions)

Dataset	Algorithm	Serial	1 Thread	32 Threads	Speedup
GDS2002	DFS-ES	2,971.16	5,386.68	237.28	22.7
	DFS-PF	4,808.64	9,500.81	396.22	23.98
	DFS-PF-G	4,764.08	9,170.09	388.91	23.58
	DFS-PF-P	4,828.17	9,397.34	393.65	23.87
GDS2712	DFS-ES	205.88	355.86	19.24	18.49
	DFS-PF	272.39	559.29	27.18	20.57
	DFS-PF-G	286.66	550.15	28.08	19.59
	DFS-PF-P	289.93	556.25	28.62	19.43
GAL	DFS-ES	323.68	582.2	27.04	21.52
	DFS-PF	329.37	598.78	28.88	20.73
	DFS-PF-G	311.99	566.93	24.87	22.79
	DFS-PF-P	295.60	540.38	25.11	21.52

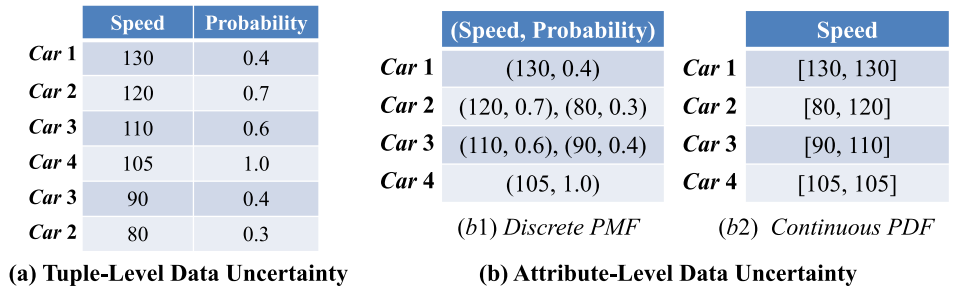
The problem of ranking uncertain tuples is relevant to our OPSM mining problem, since both problems consider the order relationships of uncertain numerical values. In fact, the uncertain database considered in the ranking problem is equivalent to a row of the data matrix in our OPSM mining problem. OPSMRM [33] adopts the discrete attribute-level uncertain data model, while we study OPSM mining under the continuous attribute-level uncertain data model.

Table 16. Peak Memory (MB) w.r.t. # of Threads on Different Datasets (Uniform Value Distributions)

Thread #:	1	2	4	8	16	32	64
GAL	211	213	199	194	207	216	225
GDS2002	742	736	707	757	843	873	1,252
GDS2712	111	102	128	177	205	321	330
Movie	110,114	110,164	109,710	116,534	110,207	110,238	110,439
RFID	15	16	39	27	28	23	37

Table 17. Peak Memory (MB) w.r.t. # of Threads on Different Datasets (Gaussian Value Distributions)

Thread #:	1	2	4	8	16	32	64
GAL	2,027	2,122	2,032	2,050	2,019	1,991	2,047
GDS2002	6,390	6,528	6,693	7,175	8,060	9,962	13,908
GDS2712	1,199	1,273	1,414	1,769	2,478	3,755	5,389



(a) Tuple-Level Data Uncertainty

(b1) *Discrete PMF* (b2) *Continuous PDF*

(b) Attribute-Level Data Uncertainty

Fig. 44. Uncertain Data Model.

D.2 Mining Uncertain Data

We focus on the problem of mining frequent patterns (such as itemsets and sequences) over uncertain data in this subsection.

Frequent Itemset Mining. The problem of frequent itemset mining has been extensively studied in the context of uncertain data. Earlier works use expected support to measure pattern frequentness [1, 7]. However, [5, 35] find that the use of expected support may render important patterns missing, and thus, recent research focuses more on using p-frequentness [5, 25]. See the excellent survey of [26] for a more complete overview of the many important studies in this area.

In fact, the transaction database for frequent itemset mining can also be represented as a data matrix D, where each row corresponds to a transaction and each column corresponds to an item. If transaction g contains item t, then $D[g][t] = 1$; otherwise, $D[g][t] = 0$.

Sequential Pattern Mining. The problem of sequential pattern mining has also been studied over uncertain data: Liu et al. [22] evaluate pattern frequentness based on expected support, while Zhao et al. [37] evaluate pattern frequentness based on p-frequentness. Yang et al. [31] study the problem of mining long sequential patterns in a noisy environment.

The problem of OPSM mining can be reduced into a sequential pattern mining problem: each row in the data matrix can be treated as a sequence (called *row sequence* hereafter) as Figure 1 illustrates, where the elements are the columns $t_1, \ldots, t_m$. However, unlike general sequential pattern mining

problems, in OPSM mining, each element can appear at most once in a row sequence, and it appears exactly once if there is no missing data.

Another difference between OPSM mining and sequential pattern mining is that, the identities of the supporting row sequences are important in OPSM mining but are immaterial in sequential pattern mining. Recall that an OPSM may refer to a set of coexpressed genes, or a group of users with similar preferences.

REFERENCES

[1] Charu C. Aggarwal, Yan Li, Jianyong Wang, and Jing Wang. 2009. Frequent pattern mining with uncertain data. In *SIGKDD*. 29–38.

[2] Rakesh Agrawal, Johannes Gehrke, Dimitrios Gunopulos, and Prabhakar Raghavan. 1998. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD*. 94–105.

[3] Tanya Barrett, Dennis B. Troup, Stephen E. Wilhite, Pierre Ledoux, Dmitry Rudnev, Carlos Evangelista, Irene F. Kim, Alexandra Soboleva, Maxim Tomashevsky, Kimberly A. Marshall, et al. 2009. NCBI GEO: Archive for high-throughput functional genomic data. *Nucleic Acids Res.* 37, suppl 1 (2009), D885–D890.

[4] Amir Ben-Dor, Benny Chor, Richard M. Karp, and Zohar Yakhini. 2003. Discovering local structure in gene expression data: The order-preserving submatrix problem. *J. Computat. Biol.* 10, 3/4 (2003), 373–384.

[5] Thomas Bernecker, Hans-Peter Kriegel, Matthias Renz, Florian Verhein, and Andreas Züfle. 2009. Probabilistic frequent itemset mining in uncertain databases. In *SIGKDD*. 119–128.

[6] Lin Cheung, David W. Cheung, Ben Kao, Kevin Y. Yip, and Michael K. Ng. 2007. On mining micro-array data by order-preserving submatrix. *Int. J. Bioinf. Res. Applic.* 3, 1 (2007), 42–64.

[7] Chun Kit Chui, Ben Kao, and Edward Hung. 2007. Mining frequent itemsets from uncertain data. In *PAKDD*. 47–58.

[8] Thomas H. Cormen et al. 2009. *Introduction to Algorithms*. The MIT Press.

[9] Graham Cormode, Feifei Li, and Ke Yi. 2009. Semantics of ranking queries for probabilistic data and expected ranks. In *ICDE*. 305–316.

[10] Nilesh N. Dalvi and Dan Suciu. 2004. Efficient query evaluation on probabilistic databases. In *VLDB*. Morgan Kaufmann, 864–875.

[11] Qiong Fang, Wilfred Ng, and Jianlin Feng. 2010. Discovering significant relaxed order-preserving submatrices. In *SIGKDD*. 433–442.

[12] Qiong Fang, Wilfred Ng, Jianlin Feng, and Yuliang Li. 2014. Mining order-preserving submatrices from probabilistic matrices. *ACM Trans. Datab. Syst.* 39, 1 (2014), 6:1–6:43.

[13] Byron J. Gao, Obi L. Griffith, Martin Ester, and Steven J. M. Jones. 2006. Discovering significant OPSM subspace clusters in massive gene expression data. In *SIGKDD*. 922–928.

[14] Tingjian Ge, Stanley B. Zdonik, and Samuel Madden. 2009. Top-k queries on uncertain data: On score distribution and typical answers. In *SIGMOD*. 375–388.

[15] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. In *SIGMOD*. 1–12.

[16] F. Maxwell Harper and Joseph A. Konstan. 2016. The MovieLens datasets: History and context. *ACM Trans. Interact. Intell. Syst.* 5, 4 (2016), 19:1–19:19.

[17] Ming Hua, Jian Pei, Wenjie Zhang, and Xuemin Lin. 2008. Ranking queries on uncertain data: A probabilistic threshold approach. In *SIGMOD*. 673–686.

[18] Lucien Le Cam et al. 1960. An approximation theorem for the poisson binomial distribution. *Pacific J. Math.* 10, 4 (1960), 1181–1197.

[19] Jian Li and Amol Deshpande. 2010. Ranking continuous probabilistic datasets. *PVLDB* 3, 1 (2010), 638–649.

[20] Jian Li, Barna Saha, and Amol Deshpande. 2009. A unified approach to ranking in probabilistic databases. *PVLDB* 2, 1 (2009), 502–513.

[21] Jinze Liu and Wei Wang. 2003. OP-cluster: Clustering by tendency in high dimensional space. In *ICDM*. 187–194.

[22] Muhammad Muzammal and Rajeev Raman. 2011. Mining sequential patterns from probabilistic databases. In *PAKDD*. 210–221.

[23] Mohamed A. Soliman and Ihab F. Ilyas. 2009. Ranking with uncertain scores. In *ICDE*. 317–328.

[24] Mohamed A. Soliman, Ihab F. Ilyas, and Kevin Chen-Chuan Chang. 2007. Top-k query processing in uncertain databases. In *ICDE*. 896–905.

[25] Liwen Sun, Reynold Cheng, David W. Cheung, and Jiefeng Cheng. 2010. Mining uncertain data with probabilistic guarantees. In *SIGKDD*. 273–282.

[26] Yongxin Tong, Lei Chen, Yurong Cheng, and Philip S. Yu. 2012. Mining frequent itemsets over uncertain databases. *PVLDB* 5, 11 (2012), 1650–1661.

[27] A. Yu Volkova. 1996. A refinement of the central limit theorem for sums of independent random indicators. *Theor. Probab. Applic.* 40, 4 (1996), 791–794.
[28] Da Yan and Wilfred Ng. 2011. Robust ranking of uncertain data. In *DASFAA*. 254–268.
[29] Da Yan, Wenwen Qu, Guimu Guo, and Xiaoling Wang. 2020. PrefixFPM: A parallel framework for general-purpose frequent pattern mining. In *ICDE*. IEEE, 1938–1941.
[30] Da Yan, Wenwen Qu, Guimu Guo, Xiaoling Wang, and Yang Zhou. 2022. PrefixFPM: A parallel framework for general-purpose mining of frequent and closed patterns. *VLDB J.* 31, 2 (2022), 253–286. https://dl.acm.org/doi/abs/10.1007/s00778-021-00687-0.
[31] Jiong Yang, Wei Wang, Philip S. Yu, and Jiawei Han. 2002. Mining long sequential patterns in a noisy environment. In *SIGMOD*. 406–417.
[32] Ka Yee Yeung, Mario Medvedovic, and Roger E. Bumgarner. 2003. Clustering gene-expression data with repeated measurements. *Genome Biol.* 4, 5 (2003), R34.
[33] Kevin Y. Yip, Ben Kao, Xinjie Zhu, Chun Kit Chui, Sau Dan Lee, and David W. Cheung. 2013. Mining order-preserving submatrices from data with repeated measurements. *IEEE Trans. Knowl. Data Eng.* 25, 7 (2013), 1587–1600.
[34] Mengsheng Zhang, Wei Wang, and Jinze Liu. 2008. Mining approximate order preserving clusters in the presence of noise. In *ICDE*. 160–168.
[35] Qin Zhang, Feifei Li, and Ke Yi. 2008. Finding frequent items in probabilistic data. In *SIGMOD*. 819–832.
[36] Xi Zhang and Jan Chomicki. 2008. On the semantics and evaluation of top-k queries in probabilistic databases. In *DBRank*. 556–563.
[37] Zhou Zhao, Da Yan, and Wilfred Ng. 2012. Mining probabilistically frequent sequential patterns in uncertain databases. In *EDBT*. 74–85.

Received September 2020; revised January 2022; accepted March 2022