RHPTree—Risk Hierarchical Pattern Tree for Scalable Long Pattern Mining

DANLU LIU, YU LI, WILLIAM BASKETT, DAN LIN, and CHI-REN SHYU,
University of Missouri-Columbia

Risk patterns are crucial in biomedical research and have served as an important factor in precision health and disease prevention. Despite recent development in parallel and high-performance computing, existing risk pattern mining methods still struggle with problems caused by large-scale datasets, such as redundant candidate generation, inability to discover long significant patterns, and prolonged post pattern filtering. In this article, we propose a novel dynamic tree structure, Risk Hierarchical Pattern Tree (RHPTree), and a top-down search method, RHPSearch, which are capable of efficiently analyzing a large volume of data and overcoming the limitations of previous works. The dynamic nature of the RHPTree avoids costly tree reconstruction for the iterative search process and dataset updates. We also introduce two specialized search methods, the extended target search (RHPSearch-TS) and the parallel search approach (RHPSearch-SD), to further speed up the retrieval of certain items of interest. Experiments on both UCI machine learning datasets and sampled datasets of the Simons Foundation Autism Research Initiative (SFARI)—Simon's Simplex Collection (SSC) datasets demonstrate that our method is not only faster but also more effective in identifying comprehensive long risk patterns than existing works. Moreover, the proposed new tree structure is generic and applicable to other pattern mining problems.

CCS Concepts: • **Information systems** → **Association rules**; • **Theory of computation** → **Sorting and searching**; **Massively parallel algorithms**; • **Applied computing** → **Health informatics;**

Additional Key Words and Phrases: Risk pattern mining, contrast mining, odds ratio, indexing structure, parallel computing

ACM Reference format:
Danlu Liu, Yu Li, William Baskett, Dan Lin, and Chi-Ren Shyu. 2022. RHPTree—Risk Hierarchical Pattern Tree for Scalable Long Pattern Mining. *ACM Trans. Knowl. Discov. Data.* 16, 4, Article 63 (January 2022), 33 pages.
https://doi.org/10.1145/3488380

1 INTRODUCTION

Risk factors are variables which capture the differences of two groups through certain risk measurements such as risk difference, relative risk, and odds ratio [10]. Risk factors allow people to gain a better understanding of the intricate structure of a large dataset, which has a catalytic effect in decision-making and predictions. For decades, risk factors have been widely utilized in a broad range of applications, such as fraud detection [37], social bot detection [38, 57], and geospatial

Authors' address: D. Liu, Y. Li, W. Baskett, D. Lin, and C.-R. Shyu (corresponding author), University of Missouri-Columbia, Columbia, Missouri, 65211; emails: {dltb9, ylrbc, wibpp9}@mail.missouri.edu, {lindan, shyuc}@missouri.edu.

1556-4681/2022/01-ART63 $15.00
https://doi.org/10.1145/3488380

traffic intervention [55]. Most recently, risk factors have begun to play an important role in medical research as they contribute to identifying behavioral, environmental, and genetic factors that can increase the likelihood of developing a disease. Specifically, there has been a rising demand to identify the risk factors in precision medicine [40, 49] and preventive healthcare [39, 52] to better understand the etiology of diseases in order to tailor treatments for targeting patients. Risk factors and the corresponding outcomes are typically quantified using the statistic measurements such as risk difference, relative risk, and odds ratio, which indicate the difference of risks, the ratio of risks, and the ratio of odds between two groups, respectively.

A combination of several co-occurring risk factors, which is defined as a risk pattern, is critical in etiological research and disease intervention [8, 22, 51]. For example, the etiology of autism could include multiple mutated genes as risk factors. The majority of prevention research has focused on single or simple factor identification. However, a number of studies have shown that risk factors often coexist and interact mutually [9, 15]. For instance, a previous study demonstrated a gene-environment interaction between passive smoking and two HLA genes, which relates to higher multiple sclerosis risk [43]. There is a clear need to discover longer and more complex patterns for understanding risk factors in the abovementioned application areas. Unfortunately, only 8.8% of studies have considered multiple risk factors [54] and are still in a preliminary stage.

1.1 Motivation

To gain a better understanding of the current challenges, we first formally introduce the risk measurements with an example of a genetic mutation (m) as a risk factor between two groups of autistic patients with (G_i) and without (G_j) aggressive behavior. Within each group (G_i, G_j), the number of patients who have the mutation ($g_{m,i}, g_{m,j}$) or who do not have the mutation ($g_{n,i}, g_{n,j}$) are recorded. One can measure the risk of m between the two groups using (1) risk difference (or support difference) to capture the probability difference of having the mutation, $RD(G_i, G_j) = (|g_{m,i}|/|G_i|) - (|g_{m,j}|/|G_j|)$; (2) relative risk (or growth rate) to capture the ratio of the probability of having the mutation, $RR(G_i, G_j) = (|g_{m,i}|/|G_i|)/(|g_{m,j}|/|G_j|)$; and (3) odds ratio to capture the ratio of the percentage of patients with the mutation compared to those do not have the mutation, $OR(G_i, G_j) = (|g_{m,i}|/|g_{n,i}|)/(|g_{m,j}|/|g_{n,j}|)$.

Unlike identifying patterns containing a single risk factor, mining risk patterns, which are composed of multiple risk factors, is much more computationally expensive as the mining process must compare a potentially large number of candidate patterns in one group with every pattern in the other group. The search space expands rapidly due to combinatorial explosion caused by a large number of risk factors [4]. Although there have been several previous studies on risk pattern mining [12, 14, 21, 26, 47], they have mainly focused on two risk factors, i.e., risk difference and relative risk because these two measurements can take the advantage of a contrast mining algorithm. The efficiency of the contrast mining algorithm mainly originates from the border representation proposed by [12], which utilizes Apriori property [1] to effectively reduce the search space.

However, odds ratio pattern mining has not been studied as much in computational communities due to the fact that the border representation cannot be applied to odds ratio patterns. Unlike all sub-patterns of maximal frequent patterns which can be computed based on Apriori property, the odds ratios of all subsets of maximal odds patterns do not automatically meet odds threshold β. Therefore, the border cannot cover all sub-patterns with odds above β. This increases the complexity and difficulty for odds ratio pattern mining [12]. Similarly, for any other risk criteria that do not have the Apriori property, border representation will not be valid. Hence, a generalized approach is needed to handle risk criteria that are not restricted to following border representations.

1.2 Novelty of the Work

To build a general risk pattern mining algorithm, it is critical to utilize a persistent and dynamic data structure that allows search space reduction with various risk criteria and dynamic data updates without full reconstruction. Currently, the data structures used by most risk pattern mining methods are not suitable for odds ratio or other less studied risk criteria. Practically, discovering patterns with varied risk measurements is often performed manually by humans and has become increasingly challenging with the unprecedented growth of the data. For example, with different discovery criteria or database updates, repeatedly mining the database unnecessarily wastes resources and time [1, 31]. Also, it is common for studies to prioritize items of interest to the mining process based on observations and prior beliefs, and therefore it is important that the data structure accommodate for the target-orientated searches. Itemset tree [28] and FPTree [23] have been adapted for target-oriented search for frequent patterns [19, 29, 33, 48]. But they do not offer any possibility to avoid unnecessary search using risk measurements because the required statistics are not easily accessible. With these identified requirements and limitations, it is clear that a novel data structure is needed for the increased risk calculation complexity rooted in risk pattern mining, especially with odds ratio as the risk measurement.

In this work, we propose a dynamic indexing structure for general risk pattern mining that allows for data updates and target search. We also design a general mining method for the proposed data structure that effectively reduce the search space for various risk criteria. The proposed mining method can be further specified to provide incremental and target-oriented mining, and it can be scaled by performing parallel computation.

1.3 Contributions

To bridge the gaps stated above, this article proposes a novel data structure as well as a highly efficient search algorithm to identify significant risk patterns. Our main contributions are summarized as follows:

- —We proposed a new dynamic tree structure, namely **Risk Hierarchical Pattern Tree** (**RHPTree**), in which the number of nodes scales linearly to the cardinality of items in the data and thus handles large datasets effectively. The RHPTree also serves as a persistent data structure which supports insert, delete, and update operations to adjust well to database growth. Moreover, the RHPTree inherently indexes frequent and risk patterns, enabling efficient, targeted pattern mining, and inclusive mining when the search criteria vary iteratively.
- —We designed a top-down comprehensive search paradigm, called **Risk Hierarchical Pattern Search** (**RHPSearch**). This method is capable of tackling the challenging mining task that identifies long and complex risk patterns. Furthermore, we also developed an efficient target-oriented search algorithm RHPSearch-TS for applications that focus only on specific items. We have proved the completeness of the patterns returned by our algorithms, and also evaluated their performance with benchmark datasets and real-world data.
- —We developed a parallel search algorithm based on our proposed RHPSearch to further improve the overall pattern mining performance in a distributed computing environment.

The rest of this article is organized as follows. Section 2 discusses related work and provides the formal definition of the odds ratio pattern mining. Section 3 reports the new dynamic tree structure RHPTree and explores its frequency and bound properties for risk pattern mining. Section 4 introduces the definition and comprehensive analysis of search methods, including the RHPSearch, the target-oriented search, RHPSearch-TS, and the RHPSearch-SD for parallel computing. Section 5

evaluates the efficiency of our algorithms on both benchmark and health data sets. Finally, the article is concluded in Section 6 with future research directions.

2 RELATED WORK AND PROBLEM DEFINITION

2.1 Related Work

Risk pattern mining is related to frequent itemset mining, contrast mining, and equivalent class mining. In this section, we provide a brief overview of main approaches in these fields and compare our risk pattern mining algorithm with existing studies to provide a background for the proposed method.

2.1.1 Risk Pattern Selection Using Frequent Pattern Mining. Naturally, one straightforward solution to mine risk patterns is to implement a post-processing step after the frequent pattern mining to extract risk patterns from the frequent patterns. Most of the existing frequent pattern mining algorithms can be categorized into three main types: join-based approaches, growth-based approaches, and set-enumeration-based approaches. Join-based approaches start with candidate generation and then filter out candidates that do not satisfy the frequency threshold [25]. Join-based approaches, such as Apriori, do not have a tree structure to adjust changes in the database. Also, the candidate generation is time- and memory-consuming. Growth-based approaches, such as FP-Growth [23] and LP-Growth [45], implement a divide-and-conquer method to partition and project the dataset to a smaller conditional subset until all patterns are identified. The growth-based tree structure for FP-Growth and LP-Growth supports insert, delete, update operations based on transactions, but are not designed for a target-oriented search that looks for a specific pattern. Set-enumeration-based algorithms use a vertical representation of the dataset by scanning it once and generate the longer patterns by enumeration, such as LCM [53], PrePost+ [11], and negFIN [2]. The set-enumeration-based algorithms either do not have an indexing structure or do not support target-oriented search.

Most importantly, adding post-processing to existing frequent pattern mining is not an efficient method for risk pattern mining because a large number of frequent patterns that need to be evaluated by the post-processing are not risk patterns. Even though some frequent pattern mining methods utilize parallel implementations, such as EAFIM [46] based on Apriori and Pfp [33] based on FP-Growth, the exhaustive checks of the large amount of candidate patterns is still a waste of computing power. In our work, we propose a novel and parallel algorithm that is specially designed for risk pattern mining to avoid such unnecessary overhead.

2.1.2 Contrast Mining Using Growth Measurement. Contrast mining is a field which detects patterns that are used to distinguish two groups. Measurement growth, referred to as relative risk in this article, is often used in contrast mining. As stated previously, Dong and Li [12] proposed the border differential algorithm to reduce the search space when mining contrast patterns. Because contrast patterns are often used in classification, sub-contrast mining technologies have been developed to detect contrast patterns which have a strong classification ability, such as **jumping emerging patterns** (**JEPs**) and **strong emerging patterns** (**SJEPs**). These subsets of contrast patterns can be mined based on the tree structure. Bailey, Manoukian, and Kotagiri [3] proposed a component tree structure for mining JEPs based on the FP-Growth algorithm. The tree structure maps each transaction onto a path in the tree, and the search algorithm for this tree is based on the border differential algorithm. Fan and Kotagiri [14] proposed the **contrast pattern tree** (**CP-tree**) to discover the SJEPs which have a strong discriminating power. The CP-tree is also FP-tree based but only works using the growth statistic. Currently there is no generic tree structure that

is suitable for mining contrast patterns under other measurements such as odds ratios considered in our work.

2.1.3 Equivalent Class Mining. In addition to mining patterns using target statistics, several studies aim to find patterns with the same statistical measurements, which is called equivalent class mining [32, 34, 35]. Li et al. [32] proposed a method that mines the generators and closed patterns of equivalence classes at the same time, and then combines them to discover the odds ratio pattern. An equivalent class can be represented by a concise closure, which is denoted as $[g, c]$, where g is a generator and c is a closed pattern. A generator is an itemset which contains no subsets with the same support and a closed pattern is an itemset where there exists no superset that has the same support. A closure $[g, c]$ can be extended to include the patterns having the same transactions, and thus these patterns share the same statistical measurements. The concise closure avoids mining all frequent patterns by only mining generators and closed patterns to cover all frequent patterns that have the same support level. Equivalent class mining aims to discover all closures for each support level. After discovering all equivalent classes in the dataset, users need to calculate the risk measurement values for all closures and then select the ones which meet the chosen criteria.

Although equivalent class mining represents patterns with the same statistical measurements elegantly, extracting risk patterns from all equivalent classes is as computationally costly as getting risk patterns from frequent patterns. This is especially true when many equivalent classes exist, but only a few pass the risk threshold. Also, the method for mining generators or closed patterns does not support target-oriented search. Some existing methods can mine closed patterns [24, 50] or generators [56] in the incremental databases, but none of them are designed for generating a concise closure of equivalent classes. Therefore, these aforementioned methods for related problems cannot be easily applied to risk pattern mining. Inspired by the FHPTree for maximal pattern mining [44], we developed a new tree structure for efficiently mining risk patterns between two groups directly without further post-processing.

2.1.4 Target-orientated Mining. Target item mining is to discover the target itemsets that are useful for specific application, instead of mining all frequent itemsets. Several target itemset mining algorithms are based on itemset tree which stores transactions or their intersections as nodes [20, 28, 30]. While the itemset tree is able to mine the target frequent itemsets, its structure is designed for frequent pattern mining, which only involves the measurement of a single dataset. Our motivation is to have a single tree structure that can perform full search, target search, parallel search for risk patterns from two contrast collections of transactions.

2.1.5 high-utility Pattern Mining. Recently, high utility mining has been an active area of frequent pattern mining research with the goal of evaluating the usefulness of the item. The transaction database contains the quantity of items and the unit profit of each item. The aim of high utility mining is to discover the **high utility itemsets** (**HUIs**) that is able to generate the high profit that exceeds the predefined threshold (minutil). There are several methods that mines the high utility patterns by using advanced tree and list structures [6, 27, 36, 42, 58]. Although high utility pattern mining considers the profit of the item, it does not consider the cost to obtain the utility. Philippe Fournier-Viger [18] proposed the novel approaches to mine cost-effective patterns in event logs. While the purpose of finding risk patterns is different from finding high utility patterns, high utility concept can be added into the RHPTree. An example is discovering the druggability of a collection of genes. However, to have a clear goal of this project, we focus on the risk pattern tree and risk pattern part and leave the potential opportunities of high utility pattern mining as future work.

2.2 Risk Pattern Definition

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items, and a dataset D be a collection of transactions T, where each transaction is a set of items in I. A pattern P is a collection of co-occurring items. The support of P in D, $supp(P, D)$, is the ratio of the number of transactions containing P to the total number of transactions in D. A pattern is called a frequent pattern when its support is no less than a given support threshold α. Given two groups G_i and G_j in the dataset D, the risk difference RD of a pattern P in G_i, when comparing against G_j, is defined as the support difference of P between the groups G_i and G_j:

$$RD(P, G_i, G_j) = supp(P, G_i) - supp(P, G_j). \tag{1}$$

The relative risk RR of a pattern P in G_i, when comparing against G_j, is defined as the ratio of two supports:

$$RR(P, G_i, G_j) = \frac{supp(P, G_i)}{supp(P, G_j)}. \tag{2}$$

The odds ratio OR of a pattern P in G_i, when comparing against G_j, is defined as the ratio of the odds of P in the presence of G_i to the odds of P in the presence of G_j:

$$OR(P, G_i, G_j) = \frac{supp(P, G_i) * |G_i|/((1 - supp(P, G_i)) * |G_i|)}{supp(P, G_j) * |G_j|/((1 - supp(P, G_j)) * |G_j|)}, \tag{3}$$

where $supp(P, G) * |G|$ is the number of transactions containing P in G, and $(1 - supp(P, G)) * |G|$ is the number of transactions excluding P in G. P can be considered a risk pattern in G_i or G_j using either odds ratio, risk difference, or relative risk with a threshold β while being a frequent pattern in $G_i \cup G_j$. As discussed in the Motivation section, mining patterns using odds ratio measurement is more challenging than mining patterns using risk difference and relative risk due to the applicability of the border condition.

3 RISK HIERARCHICAL PATTERN TREE

Mining and searching odds ratio patterns require an efficient tree structure for varied risk thresholds and target-oriented search. Once constructed, this tree allows multiple mining processes under different thresholds with no additional risk computations. The tree can also be dynamically updated with new transactions without rebuilding. In this section, we first define the RHPTree and then discuss the tree building method as well as tree operations.

3.1 Definition

The basic elements of a RHPTree are tree nodes representing an item or a set of items and the relationships between a parent node and its two child nodes. Because risk pattern mining always involves comparing patterns between two groups, the tree structure is designed to contain the transactions as well as the group information for each item. To keep track of the transaction information of both G_i and G_j, each node stores the IDs of the transactions where the item occurs in groups G_i and G_j. To estimate the range of a risk measurement value for a pattern efficiently, we use the concept of exact transaction set eT and candidate transaction set cT as the lower and upper estimations of the number of transactions that contain a pattern. The exact transaction set of a node includes all transaction IDs that exactly contain the items in the node, while the candidate transaction set includes all transaction IDs that may contain the items. For each leaf node, which represents a single item, these two sets are the same. For each internal node, the exact transaction set is the intersection of both child nodes' exact transaction sets and the candidate transaction set is the union of both child nodes' candidate transaction sets. As one ascends the tree, the number of transactions in eT shrinks and the number of transactions in cT grows. One group's cT and the

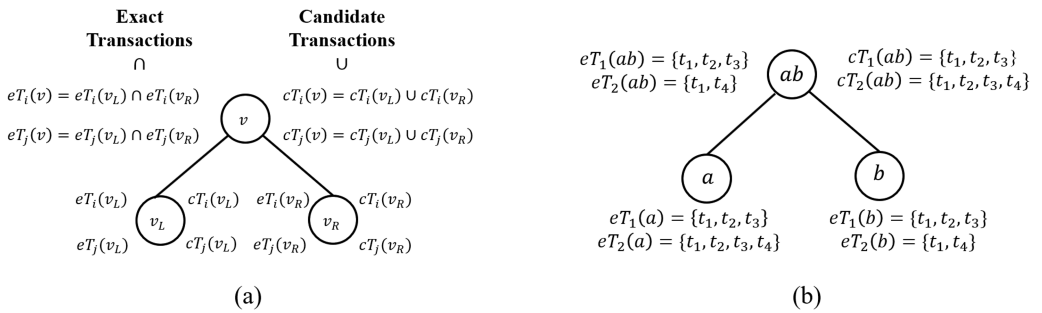

Fig. 1. (a) The hierarchical structure of a node and its children. (b) An example of hierarchical relation in the RHPTree node.

other group's eT can be used to calculate the upper and lower bounds of the risk measurement value. These bounds can be further used to generate qualified patterns that pass the threshold and to prune the unnecessary patterns that fall below the threshold. For the purpose of presentation clarity, we will present the process for finding odds ratio patterns only in G_i. The same principles can also be applied to the other group G_j. The basic element tree node is defined as follows:

Definition 3.1 (Tree Node). Each node v in RHPTree contains three parts as shown in Figure 1(a): (1) label: an item or a set of items that v represents, (2) exact transaction set eT: eT_i is a set of IDs of the transactions that **exactly** contain v in group G_i, and (3) candidate transaction set cT: cT_i is a set of IDs of the transactions that **possibly** contain v in group G_i.

The RHPTree consists of leaf nodes and internal nodes in a binary tree format. The relationships of the parent node to its child nodes are defined as follows:

Definition 3.2 (Parent and Child Node Relationships). Each leaf node in the RHPTree is a single frequent item which has $eT_i = cT_i$ as the true counts that can be obtained. Each internal node represents an itemset which is a union of itemsets from its children. eT_i of an internal node v is the intersection of its children's eT_i, and its cT_i is the union of its children's cT_i, i.e., $eT_i(v) = eT_i(v_L) \cap eT_i(v_R)$ and $cT_i(v) = cT_i(v_L) \cup cT_i(v_R)$, where v_L, v_R are children of the node v.

With these definitions, it is worth noting that a node's exact transaction set $eT_i(v)$ is a subset of its candidate transaction set $cT_i(v)$, i.e., $eT_i(v) \subseteq cT_i(v)$. Additionally, according to the parent and child node relationships, a node's exact transaction set is a subset of its children's exact transaction set, and the child's candidate transaction set is a subset of its parent's candidate transaction set, i.e., $eT_i(v) \subseteq eT_i(v_L)$, $eT_i(v) \subseteq eT_i(v_R)$ and $cT_i(v_L) \subseteq cT_i(v)$, $cT_i(v_R) \subseteq cT_i(v)$.

Example. As shown in Figure 1(b), the leaf node representing item a appears in the transactions t_1, t_2, t_3 in group G_i, and transactions t_1, t_2, t_3, t_4 in group G_j; the leaf node representing b appears in the transactions t_1, t_2, t_3 in group G_i, and transactions t_1, t_4 in group G_j. Their parent node ab has the following exact transaction sets and candidate transaction sets: $eT_i(ab) = eT_i(a) \cap eT_i(b) = \{t_1, t_2, t_3\}$, $eT_j(ab) = eT_j(a) \cap eT_j(b) = \{t_1, t_4\}$; $cT_i(ab) = cT_i(a) \cup cT_i(b) = \{t_1, t_2, t_3\}$, $cT_j(ab) = cT_j(a) \cup cT_j(b) = \{t_1, t_2, t_3, t_4\}$. Using the two basic tree elements, we will discuss the tree construction process for the RHPTree.

3.2 RHPTree Construction

The construction of the RHPTree follows a bottom-up method that is similar to hierarchical clustering. First, all single frequent items are made into leaf nodes where both cT and eT are the sets of

transactions in which the items occur, calculated separately in each group. Secondly, the algorithm selects two leaf nodes based on similarities of exact and candidate transaction sets and generates a parent node. This parent node will enter the next round of selection and parent node generation. The process repeats until only one node, the root of the tree, is left.

One key step in RHPTree construction is to identify the most comparable peer nodes which have similar transactions with similar supports and risk measurements. Nodes with similar transaction sets are more likely to form long odds ratio patterns, and thus should be grouped together. To accomplish this, we use the Jaccard distance in groups G_i and G_j to measure the distance between the exact and candidate transaction sets of two nodes:

$$\begin{aligned} d_{eT_i}(v_L, v_R) &= 1 - \frac{eT_i(v_L) \cap eT_i(v_R)}{eT_i(v_L) \cup eT_i(v_R)} \quad \text{and} \quad d_{eT_j}(v_L, v_R) = 1 - \frac{eT_j(v_L) \cap eT_j(v_R)}{eT_j(v_L) \cup eT_j(v_R)}, \\ d_{cT_i}(v_L, v_R) &= 1 - \frac{cT_i(v_L) \cap cT_i(v_R)}{cT_i(v_L) \cup cT_i(v_R)} \quad \text{and} \quad d_{cT_j}(v_L, v_R) = 1 - \frac{cT_j(v_L) \cap cT_j(v_R)}{cT_j(v_L) \cup cT_j(v_R)}, \end{aligned} \tag{4}$$

and then combine these distances linearly to get the distance function between two nodes.

Definition 3.3. The distance between two nodes v_L and v_R is defined as follows:

$$dist(v_L, v_R) = d_{eT_i}(v_L, v_R) + d_{eT_j}(v_L, v_R) + d_{cT_i}(v_L, v_R) + d_{cT_j}(v_L, v_R). \tag{5}$$

Each component in the distance function ranges from 0 to 1, thus the range of the $dist(v_L, v_R)$ is [0,4]. The pseudocode of the RHPTree construction is listed in Algorithm 1.

The most expensive operation in the RHPTree construction algorithm is the iterative pairwise comparison, which calculates the distance of each node pair based on their transaction sets. The pairwise comparison at each tree layer requires $O(n^2)$ comparisons, where n is the number of items in the dataset. Each layer contains roughly half the items from the previous layer, and there are $log(n)$ layers as RHPTree is a full binary tree. Thus, the upper bound for the number of comparisons during the RHPTree construction is:

$$n^2 \times \sum_{k=0}^{log(n)} \left(\frac{1}{2}\right)^{2k} < \frac{4}{3}n^2. \tag{6}$$

During each comparison, the algorithm does the bitset union and intersection on the transaction sets to calculate the distance. The time complexities of the union and intersection operations are linear to the transaction number. To store the tree structure, the worst-case scenario is that every item appears in every transaction, which means each node has to store all transaction IDs. Assuming there are m transactions and n items in the dataset, the size of the tree is $(2n-1) * 4m$, where $(2n-1)$ is the number of nodes in the RHPTree, and 4^m is the size of two exact transaction sets and two candidate transaction sets for each node. Thus, the space complexity is $O(nm)$.

Example. Figure 2 depicts a simplified example of the RHPTree construction process. Assume support threshold $\alpha = 0.5$, then the frequent items are a, b, c, d in the combination of G_i and G_j. By calculating the distance between the nodes using Equation (5), as the line 2 in the Algorithm 1 states, leaf nodes c, d have the smallest distance among all pairs formed by a, b, c, d. Next c, d are paired to form an internal node cd first as the line 3 in Algorithm 1. Then node cd is the new node for the next iteration with nodes a, b as shown in line 4. In the next iteration, nodes a, b are paired to form node ab. Finally, ab and cd are paired to form the root node $abcd$. Different orders of node parings will generate different tree structures which will ensure that the patterns of interest remain identical. Section 4.1 will provide a formal proof of the completeness in pattern search.

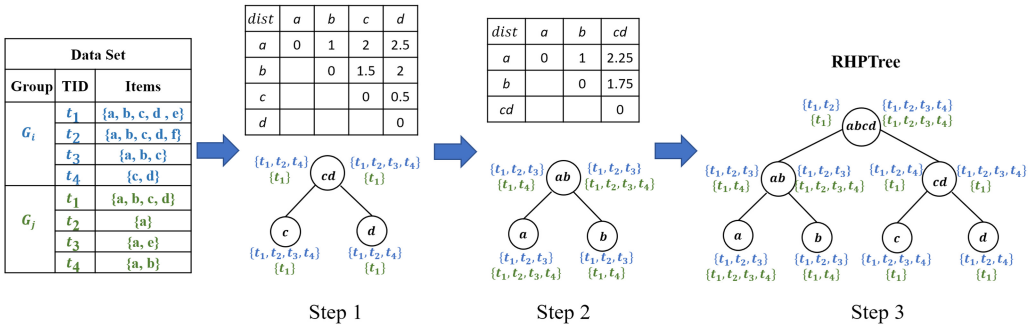

Fig. 2. An example of RHPTree construction.

ALGORITHM 1: RHPTree Construction

Input: A node set: $V = \{v_1, v_2, \ldots, v_n\}$, A distance function $dist(v_L, v_R)$
Output: RHPTree

1: **while** $V.size > 1$ **do**
2: v_L, v_R satisfy min $dist$ for all pairs in V
3: Create parent node v_p for v_L, v_R
4: $V = V - v_L - v_R + v_p$
5: **end while**
6: **return** $V.head$

3.3 Operations

The ever-growing data require a flexible data structure. The reusable tree structure is a requisite to iterative discovery and fault tolerance. The RHPTree features insert, delete, and update operations to dynamically adjust the tree structure as the dataset it represents is updated.

3.3.1 Insert. The insert operation is triggered when a new item is introduced to the dataset or a previous infrequent item is now frequent due to a reduced support threshold or newly added transactions. To integrate the new leaf node v into the RHPTree, the primary goal is to pair v with the most similar node in the RHPTree.

To optimize the tree structure after inserting v, we adopt the rotate technique as described in [41]. First, we pair v with the nearest leaf neighbor *Sib* and generate a new parent which replaces *Sib*'s position in the tree. The original sibling of *Sib* becomes the *aunt* of both v and *Sib*. Second, we check the distances between *Sib* and v, *Sib* and *aunt*. If the distance between *Sib* and *aunt* is less than the distance between *Sib* and v, then swap v with *aunt* so that the *Sib* and *aunt* can still pair together. The algorithm then repeatedly checks the distances between v and its new sibling *Sib*, *Sib* and *aunt*, and swaps v and *aunt* until $dist(v, Sib) < dist(Sib, aunt)$ or v's parent is the root of the RHPTree. Finally, it updates the transaction sets of v's ancestors. The pseudocode for the RHPTree insert is described in Algorithm 2. The insertion operation has two steps, one is to find the nearest neighbor of the new item, and the other is to update ancestors' transaction sets. Assuming there are n leaf nodes in the tree and m transactions in the dataset, the worst-case scenarios of finding nearest neighbor is $O(n)$ and updating all ancestor's transactions is $O(log(n) * m)$, where $log(n)$ is the height of the RHPTree.

Example. In Figure 3, a new node e is added into the RHPTree in Figure 2. First, the algorithm finds the nearest neighbor a among all leaf nodes and rotates b to become their *aunt* as lines 1–3 show in Algorithm 2. However, a and b have a smaller distance than a and e as the condition in

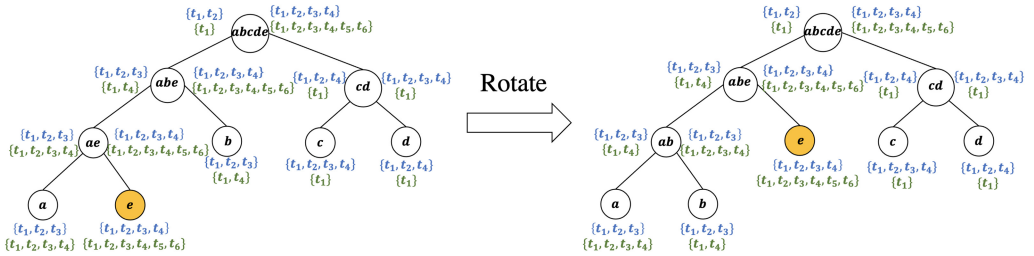

Fig. 3. An insert example of adding node e into an existing RHPTree.

ALGORITHM 2: RHPTree Insertion

Input: A new node v, RHPTree
Output: RHPTree

```
1: Sib=Nearest_Neighbor (v)
2: Replace Sib with a new parent of both Sib and v
3: aunt = v.aunt
4: while v.parent is not RHPTree.root or dist(Sib, aunt) < dist(Sib, v) do
5:     swap v and aunt
6:     Sib = v.sibling, aunt = v.aunt
7: end while
8: UpdateAncestorTransactions(v)
9: return RHPTree
```

line 4, so node e is swapped with node b to put a and b together again as lines 5–7 show. Because $dist(ab, cd) > dist(ab, e)$, no more swapping is needed.

3.3.2 Delete. The delete operation is triggered when an item is deleted from the dataset or a previously frequent item is now infrequent. The delete operation is a bottom-up approach. The algorithm deletes the node with its parent, and then uses its sibling to replace its former parent. Similar to the insert, delete also needs to update the transaction sets of its ancestors. The pseudocode for the delete operation is provided in Algorithm 3. The delete operation deletes the target leaf node and then updates the ancestors' transaction sets. Assuming there are n leaf nodes in the tree and m transactions in the dataset, updating all ancestor's transactions has a complexity of $O(log(n) * m)$.

Example. Figure 4 shows the process of deleting c from the RHPTree. The node c's sibling d replaces its original parent cd.

ALGORITHM 3: RHPTree Deletion

Input: A remove node v, RHPTree
Output: RHPTree

```
1: if v.sibling ≠ ∅ then
2:     grandparent=v.parent.parent
3:     v.sibling.parent=grandaparent
4:     RHPTree.delete(v.parent)
5: end if
6: UpdateAncestorTransactions(v.sibling)
7: return RHPTree
```

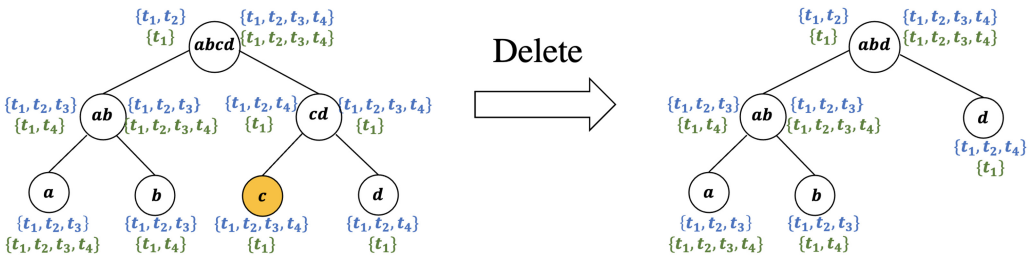

Fig. 4. Deleting the node c from the RHPTree.

3.3.3 Update. The update operation is designed to modify the label and the transaction sets of an existing node. Changing the label is straightforward. Updating the transaction sets is a bottom-up method, it starts from the leaf node and continuously updates the transaction sets of its ancestors as provided in Algorithm 4. The update operation triggers the insert or delete operation when the modified item becomes frequent or infrequent, respectively. As discussed previously, assuming there are n leaf nodes in the tree and m transactions in the dataset, updating all ancestor's transactions has a complexity of $O(log(n) * m)$.

ALGORITHM 4: RHPTree Update

Input: A leaf node v, new transaction sets eT, cT, RHPTree
Output: RHPTree

1: $p = v$
2: $p.eT = eT, p.cT = cT$
3: **while** p is not RHPTree.*root* **do**
4: $p.parent.eT_i = p.eT_i \cap p.sibling.eT_i, p.parent.eT_j = p.eT_j \cap p.sibling.eT_j$
5: $p.parent.cT_i = p.cT_i \cup p.sibling.cT_i, p.parent.cT_j = p.cT_j \cup p.sibling.cT_j$
6: $p = p.parent$
7: **end while**
8: **return** RHPTree

4 RHPSEARCH: RISK PATTERN TREE SEARCH

Based on the RHPTree structure, we describe an efficient, top-down, iterative search method ensuring the coverage of all item combinations that satisfy both support and odds ratio criteria.

4.1 The Naïve RHPSearch

We first describe the naïve search method for RHPTree without pruning methods to provide an intuitive introduction of the basic search strategy. We define a naïve search as an exhaustive depth-first search method that lists all possible patterns. The RHPSearch is a top-down search method that starts from the root node and iteratively searches downwards. Because the RHPTree is a binary tree, there are three paths requiring checking for a given node: the node itself, its left child, and its right child. For each search path, the tree checks the pattern combinations of nodes to find qualified risk patterns. With this naïve search strategy, the search paths continue to split as the tree is traversed downwards iteratively until all leaf nodes are covered. The iterative three-way split method ensures that all possible patterns are covered. The complete RHPSearch, which utilizes advantageous properties of RHPTree to prune many unnecessary search paths, will be given in Section 4.3.

As described above, RHPSearch starts the first iteration from the root node $V_0 = \{v_{root}\}$ and then in the second iteration splits into three sub-searches conducted on the collection of search nodes of the root node v_{root} and its children v_L and v_R, which are $V_1 = \{v_L, v_R\}$. $V_{1,L} = \{v_L\}$, and $V_{1,R} = \{v_R\}$. Each of these sub-searches will also iteratively yield three sub-searches. The mechanism to determine the three sub-searches first selects a non-leaf node v in the searching node collection V and launches the searches on its children. Assume that at the nth iteration, the algorithm is checking a collection $V_n = \{v_1, v_2, \ldots, v_k\}$, and that a non-leaf node v_i, which is picked for this iteration, has children $v_{i,L}$ and $v_{i,R}$. Then three sub-searches are launched:

$$V_{n+1} = (V_n - v_i) \cup \{v_{i,L}\} \cup \{v_{i,R}\}$$
$$V_{n+1,L} = (V_n - v_i) \cup \{v_{i,L}\}$$
$$V_{n+1,R} = (V_n - v_i) \cup \{v_{i,R}\}.$$

Algorithm 5 describes the searching steps. Starting at the root node, if there are non-leaf nodes inside the searching collection (lines 1–3), the first non-leaf node v_n is replaced by its two children $v_{n,L}$ and $v_{n,R}$ (lines 4–5), and then three sub-searches are launched (lines 6–12). Using the RHPTree in Figure 2 as an example, by following Algorithm 5, all derived paths are shown in Figure 5. The normal node collections contain non-leaf nodes, while the marked collections are the final searching collections that only contain leaf nodes and represent all possible item combinations. In Theorem 4.2, we prove that this search method covers a comprehensive list of patterns containing all items in the tree.

ALGORITHM 5: Naive RHPSearch

Input: A searching node collection: V
Output: Pattern collection: *RHPSet*

```
1:  leaves = {v|v ∈ V and v.children.size = 0}
2:  nonLeaves = V − leaves
3:  if nonLeaves.size > 0 then
4:      v_n = nonLeaves.head
5:      Nodes = (nonLeaves − v_n) + leaves
6:      Naive RHPSearch(Nodes + V_{n,R} + V_{n,L})
7:      if v_{n,R} ∉ RHPSet then
8:          Naive RHPSearch(Nodes + v_{n,R})
9:      end if
10:     if v_{n,L} ∉ RHPSet then
11:         Naive RHPSearch(Nodes + v_{n,L})
12:     end if
13: end if
14: RHPSet.add(V)
```

Before introducing Theorem 4.2, we define the set of patterns that each searching node collection V covers, which is called k-ary collection as follows. The k-ary collection of a searching node collection contains a representation of all derived patterns, which are also related to the pruning properties introduced in Section 4.2.

Definition 4.1 (k-ary Collection). Given a searching node collection $V_n = \{v_1, v_2, \ldots, v_k\}$, all possible patterns V_n covers are $P(V_n) = P^+(W_1) \times \cdots \times P^+(W_k) = \{w_1 \cup \cdots \cup w_k | w_i \in P^+(W_i)$ for every $v_i \in V_n\}$, where W_i is the set of all leaf nodes covered by v_i and $P^+(W_i)$ is the powerset of W_i excluding the empty set.

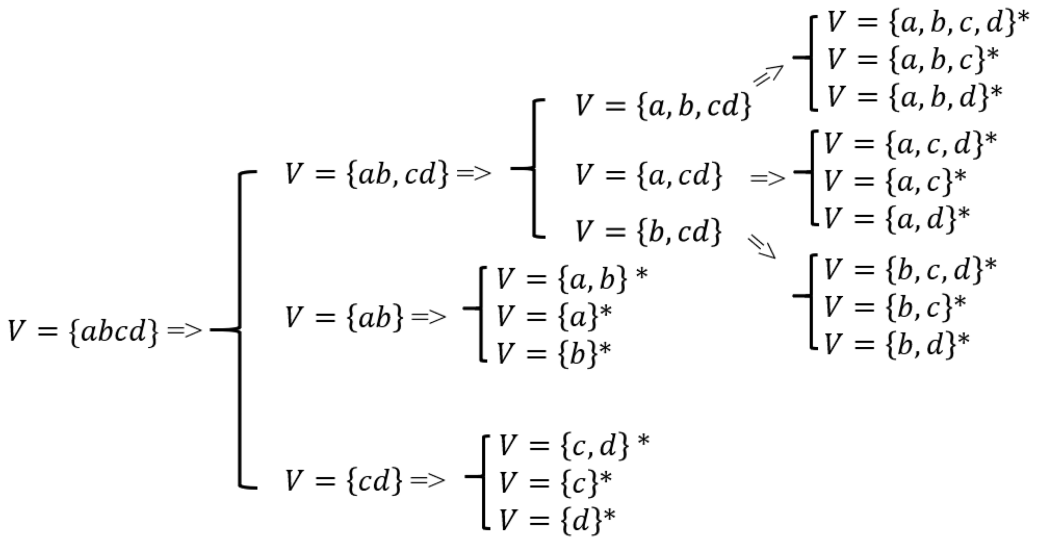

Fig. 5. The derived searching paths of an example RHPTree.

As shown by the RHPTree in Figure 2, assume the node collection is $V = \{ab, cd\}$, then the leaf node sets of ab and cd are $W_1 = \{a, b\}$ and $W_2 = \{c, d\}$, respectively. The powerset $P^+(W_1)$ is presented by $\{\{a\}, \{b\}, \{a, b\}\}$ and the powerset $P^+(W_2)$ is presented by $\{\{c\}, \{d\}, \{c, d\}\}$. The k-ary collection over V is $P(V) = P^+(W_1) \times P^+(W_2) = \{\{a, c\}, \{a, d\}, \{a, c, d\}, \{b, c\}, \{b, d\}, \{b, c, d\}, \{a, b, c\}, \{a, b, d\}, \{a, b, c, d\}\}$. From this definition, we can derive Theorem 4.2, which ensures that the splitting strategy in Naïve RHPSearch keeps the same patterns when launching the three sub-searches.

Theorem 4.2. *Given a searching node collection $V_n = \{v_1, v_2, \ldots, v_k\}$, in the Naïve RHPSearch, the patterns covered by subsequent sub-searches are the same as the patterns covered by V_n, which is $P(V_n) = P(V_{n+1}) \cup P(V_{n+1,L}) \cup P(V_{n+1,R})$.*

Proof (Left → Right). For any pattern $t = \{w_1 \cup w_2 \cup \cdots \cup w_k\} \in P(V_n)$, we want to prove that $t \in P(V_{n+1}) \cup P(V_{n+1,L}) \cup P(V_{n+1,R})$. Assume the node v_1 is the non-leaf node to split in the collection V_n, and $v_{1,L}, v_{1,R}$ are the children of v_1, then three launched sub-searches are on the collections $V_{n+1} = \{v_{1,L}, v_{1,R}, v_2, \ldots, v_k\}$, $V_{n+1,L} = \{v_{1,L}, v_2, \ldots, v_k\}$, and $V_{n+1,R} = \{v_{1,R}, v_2, \ldots, v_k\}$. Also, all item combinations covered by v_1 are $P^+(W_1) = P^+(W_{1,L}) \cup P^+(W_{1,R}) \cup (P^+(W_{1,L}) \times P^+(W_{1,R}))$. According to the k-ary collection definition, we know that $w_1 \in P^+(W_1)$ because of $P(V_n) = P^+(W_1) \times P^+(W_2) \times \cdots \times P(W_k)$. Thus, w_1 belongs in $P^+(W_{1,L})$ or $P^+(W_{1,R})$ or $(P^+(W_{1,L}) \times P^+(W_{1,R}))$.

Case 1: If $w_1 \in P^+(W_{1,L})$, then $t \in P^+(W_{1,L}) \times P^+(W_2) \times \cdots \times P^+(W_k)$. The definition of $P(V_{n+1,L})$ is $P^+(W_{1,L}) \times \cdots \times P^+(W_k)$, then $t \in P(V_{n+1,L})$.

Case 2: If $w_1 \in P^+(W_{1,R})$, then $t \in P^+(W_{1,R}) \times P^+(W_2) \times \cdots \times P^+(W_k)$. The definition of $P(V_{n+1,R})$ is $P^+(W_{1,R}) \times \cdots \times P^+(W_k)$, then $t \in P(V_{n+1,R})$.

Case 3: If $w_1 \in P^+(W_{1,L}) \times P^+(W_{1,R})$, then $t \in P^+(W_{1,L}) \times P^+(W_{1,R}) \times \cdots \times P^+(W_k)$. The definition of $P(V_{n+1}$ is $P^+(W_{1,L}) \times P^+(W_{1,R}) \times \cdots \times P^+(W_k)$, then $t \in P(V_{n+1})$.

According to three cases above, $t \in P(V_{n+1}) \cup P(V_{n+1,L}) \cup P(V_{n+1,R})$. Therefore, $P(V_n) \subseteq P(V_{n+1}) \cup P(V_{n+1,L}) \cup P(V_{n+1,R})$.

(Right → Left). Assume any $t' \in P(V_{n+1}) \cup P(V_{n+1,L}) \cup P(V_{n+1,R})$. We want to prove that $t' \in P(V_n)$. There also exists three cases.

Case 1: The definition of $P(V_{n+1,L})$ is $P^+(W_{1,L}) \times P^+(W_2) \times \cdots \times P^+(W_k)$. If $t' \in P(V_{n+1,L})$, then $t' \in P(V_n)$ because $P^+(W_{1,L}) \subseteq P^+(W_1)$.

Case 2: The definition of $P(V_{n+1,R})$ is $P^+(W_{1,R}) \times P^+(W_2) \times \cdots \times P^+(W_k)$. If $t' \in P(V_{n+1,R})$, then $t' \in P(V_n)$ because $P^+(W_{1,R}) \subseteq P^+(W_1)$.

Case 3: The definition of $P(V_{n+1})$ is $P^+(W_{1,L}) \times P^+(W_{1,R}) \times P^+(W_2) \times \cdots \times P^+(W_k)$. If $t' \in P(V_{n+1})$, then $t' \in P(V_n)$ because $P^+(W_{1,L}) \times P^+(W_{1,R}) \subseteq P^+(W_1)$.

According to three cases above, we know that $t' \in P(V_n)$. Therefore, $P(V_{n+1}) \cup P(V_{n+1,L}) \cup P(V_{n+1,R}) \subseteq P(V_n)$. Finally, we can conclude that $P(V_n) = P(V_{n+1}) \cup P(V_{n+1,L}) \cup P(V_{n+1,R})$. □

4.2 Pruning Strategies: Statistical Bound Properties for Searching RHPTree

The Naïve RHPSearch is able to search the entire tree and cover all possible combinations, which makes it capable of finding all risk patterns. However, the structure of the RHPTree has two important properties that allow us to prune search paths that contain unqualified patterns. As defined in Section 2.2, frequent odds ratio patterns are patterns that exceed both support and odds ratio thresholds. With two thresholds, the two properties of a complete RHPTree search are: (1) a frequency property using the support threshold and (2) a bounds property using the odds ratio threshold. The complete RHPSearch can efficiently discover all qualified risk patterns with no redundant searches. Before describing the search method, we first define these properties in this section.

The idea behind the frequency property is that if we already know that the search paths derived from the current search node are certainly frequent or certainly infrequent, there is no need to check the support of any pattern in the derived paths. In the first case, when the paths are certainly infrequent, it is easily understood that the subsequent patterns will also be infrequent, and that we can safely discard them. In the second case, when the paths are certainly frequent, we can also stop checking because all subsequent patterns will be frequent as well. For example, if either item a or item b falls below the frequency threshold, then there is no need to check further to see if $a \cap b$ meets the frequency threshold. On the other hand, if $a \cap b$ meets the frequency threshold, then both item a and item b certainly will too. By following this idea, our method is to decide whether the derived paths of the node collection V are definitely frequent or not. We then can define both the upper bound (definitely infrequent) and the lower bound (definitely frequent) of the frequency property.

As discussed in Section 3.2, the bottom-up building method makes the candidate transaction sets expand with the union operation while the exact transaction sets shrink with the intersection operation. The enlarged candidate transaction sets and the reduced exact transaction sets are both valuable to estimate the support values of the patterns. Before introducing the frequency property and the bounds property, we first define the preliminary definitions of intersection, which is the key operation in the tree pruning process to ensure a confined and efficient search. The intersection of the transaction sets of a node collection is used to estimate the frequency and the bounds of the collection.

Definition 4.3 (Intersection). Given a searching node collection $V = \{v_1, v_2, \ldots, v_k\}$, the exact intersection of all exact intersection sets of V in group G_i is marked as $eX_i(V) = eT_i(v_1) \cap eT_i(v_2) \cap \ldots \cap eT_i(v_k)$; the candidate intersection of all candidate intersection sets of V in group G_i is represented as $cX_i(V) = cT_i(v_1) \cap cT_i(v_2) \cap \cdots \cap cT_i(v_k)$.

For the RHPTree shown in Figure 2, assume $V = \{ab, d\}$, then in group G_1, there are $eX_1(V) = eT_1(ab) \cap eT_1(d) = \{t_1, t_2\}$ and $cX_1(V) = cT_1(ab) \cap cT_1(d) = \{t_1, t_2\}$; In group G_2, there are

$eX_2(V) = eT_2(ab) \cap eT_2(d) = \{t_1\}$ and $cX_2(V) = cT_2(ab) \cap cT_2(d) = \{t_1\}$. Also, based on Definition 3.1, for each node v_k, $eT_i(v_k)$ is the transaction set in group G_i that exactly contains v_k. Thus, it is worth noting that $eX(V)$ is the set of all the transactions that exactly contain all items in node collection V. After introducing the basic terminologies, we define the frequency property as follows, based on Definition 4.1 (k-ary collection) and Definition 4.3 (Intersection).

Theorem 4.4 (Frequency Property). *Given a searching node collection $V = \{v_1, v_2, \ldots, v_k\}$, (1) if $|cX_i(V)| + |cX_j(V)| < \alpha * (|G_i| + |G_j|)$, then all elements in the k-ary collection are infrequent; (2) if $|eX_i(V)| + |eX_j(V)| \geq \alpha * (|G_i| + |G_j|)$, then all elements in the k-ary collection are frequent.*

Proof. (1) To prove all elements in the k-ary collection $P(V)$ are infrequent, it is only necessary to prove that any element $w_1 \cup \cdots \cup w_k \in P(V)$ is infrequent. As defined in Definition 4.3 (Intersection), the frequency of the element $w_1 \cup \cdots \cup w_k \in P(V)$ is $|eX_1(w_1 \cup \cdots \cup w_k)| + |eX_2(w_1 \cup \cdots \cup w_k)|$. The proof now becomes that if $|cX_i(V)| + |cX_j(V)| < \alpha * (|G_i| + |G_j|)$ holds, then $(|eX_1(w_1 \cup \cdots \cup w_k)| + |eX_2(w_1 \cup \cdots \cup w_k)|) < \alpha * (|G_i| + |G_j|)$.

According to Definition 4.3 (Intersection), $|eX_i(w_1 \cup \cdots \cup w_k)|$ and $|cX_i(V)|$ can be obtained by:

$$|eX_i(w_1 \cup \cdots \cup w_k)| = |eT_i(w_1) \cap \cdots \cap eT_i(w_k)| \tag{7}$$

$$|cX_i(V)| = |cT_i(v_1) \cap \cdots \cap cT_i(v_k)| \tag{8}$$

In Definition 4.1, w_k is a subset of v_k's leaf node collection. According to Definition 3.2 (Parent and Child Node Relationships), in group G_i, $eT_i(w_k) \subseteq cT_i(w_k)$ and $cT_i(w_k) \subseteq cT_i(v_k)$ imply $eT_i(w_k) \subseteq cT_i(v_k)$. Thus, $|eT_i(w_1) \cap \cdots \cap eT_i(w_k)| \leq |cT_i(v_1) \cap \cdots \cap cT_i(v_k)|$. According to Equations (7) and (8), in group G_i, $|eX_i(w_1 \cup \cdots \cup w_k)| \leq |cX_i(V)|$ holds. Therefore, if $|cX_i(V)| + |cX_j(V)| < \alpha * (|G_i| + |G_j|)$ holds, then $(|eX_i(w_1 \cup \cdots \cup w_k)| + |eX_j(w_1 \cup \cdots \cup w_k)|) < \alpha * (|G_i| + |G_j|)$.

(2) Similarly, to prove all elements in k-ary collection $P(V)$ are frequent, we only need to prove that any element $w_1 \cup \cdots \cup w_k \in P(V)$ is frequent. The proof now becomes if $|eX_i(V)| + |eX_j(V)| \geq \alpha * (|G_i| + |G_j|)$ holds, then $(|eX_i(w_1 \cup \cdots \cup w_k)| + |eX_j(w_1 \cup \cdots \cup w_k)|) \geq \alpha * (|G_i| + |G_j|)$.

Based on Definition 3.2 ($eT_i(v_k) \subseteq eT_i(w_k)$), $|eT_i(w_1) \cap \ldots \cap eT_i(w_k)| \geq |eT_i(v_1) \cap \ldots \cap eT_i(v_k)|$, which means the number of intersections of the exact transactions from all child nodes is no less than the number of intersections of the exact transactions from their parent nodes. Using Equation (7) and Definition 4.3, $|eX_i(w_1 \cup \cdots \cup w_k)| \geq |eX_i(V)|$. Therefore, if $|eX_i(V)| + |eX_j(V)| \geq \alpha * (|G_i| + |G_j|)$ holds, then $(|eX_i(w_1 \cup \cdots \cup w_k)| + |eX_j(w_1 \cup \cdots \cup w_k)|) \geq \alpha * (|G_i| + |G_j|)$. □

As shown by the RHPTree in Figure 2, assuming $V = \{ab, d\}$ and support threshold hold $\alpha = 0.5$. There are $|cX_1(V)| + |cX_2(V)| = |\{t_1, t_2\}| + |\{t_1\}| = 3$, and $\alpha * (|G_1| + |G_2|) = 4$. From Theorem 4.4, we know all elements in the k-ary collection $P(V) = \{\{a, d\}, \{b, d\}, \{a, b, d\}\}$ are infrequent. On the contrary, assuming $V = \{ab, d\}$ and support threshold $\alpha = 0.3$ holds, we have $|eX_1(V)| + |eX_2(V)| = |\{t_1, t_2\}| + |\{t_1\}| = 3$, and $\alpha * (|G_1| + |G_2|) = 2.4$. From Theorem 4.4, we know all of the elements in the k-ary collection $P(V) = \{\{a, d\}, \{b, d\}, \{a, b, d\}\}$ are frequent.

The pruning process is based on the estimates of lower and upper bounds. The idea of the Bounds Property is similar to that of the Frequency Property, but targets the odds ratio threshold. In Definition 4.5, we give the new format of the odds ratio definition by utilizing the exact intersection eX and the candidate intersection cX of the node collection V. The lower and upper bounds, the decrease and amplification for the real odds ratio, are also defined by using eX and cX in Definition 4.6.

Definition 4.5 (Odds Ratio of the Searching Node Collection). Given a set of searching nodes $V = \{v_1, v_2, \ldots, v_k\}$, the real value of odds ratio of V in G_i in comparison to G_j is:

$$OR(V, G_i, G_j) = \frac{|eX_i(V)| * (|G_j| - |eX_j(V)|)}{|eX_j(V)| * (|G_i| - |eX_i(V)|)}. \tag{9}$$

Based on Equation (3), the odds ratio of the collection V is $\frac{supp(V,G_i)*|G_i|/((1-supp(V,G_i))*|G_i|)}{supp(V,G_j)*|G_j|/((1-supp(V,G_j))*|G_j|)}$. $supp(V, G) * |G|$, the number of transactions that contain V in G, is equal to $|eX(V)|$. Similarly, $(1 - supp(V, G)) * |G|$, the number of transactions in G that excludes V, is equal to $(|G| - |eX(V)|)$. Thus, $OR(V, G_i \cup G_j) = \frac{|eX_i(V)|*(|G_j|-|eX_j(V)|)}{|eX_j(V)|*(|G_i|-|eX_i(V)|)}$.

Definition 4.6 (Bounds). Given a set of searching nodes $V = \{v_1, v_2, ..., v_k\}$, the upper and lower bounds of odds ratio of V in G_i in comparison to G_j are:

$$\begin{aligned} Upperbound(V, G_i) &: \frac{|cX_i(V)| * (|G_j| - |eX_j(V)|)}{|eX_j(V)| * (|G_i| - |cX_i(V)|)}, \\ Lowerbound(V, G_i) &: \frac{|eX_i(V)| * (|G_j| - |cX_j(V)|)}{|cX_j(V)| * (|G_i| - |eX_i(V)|)}. \end{aligned} \tag{10}$$

The upper bound enlarges the odds ratio by replacing $|eX_i(V)|$ with $|cX_i(V)|$ as the numerator and $|G_i| - |eX_i(V)|$ with $|G_i| - |cX_i(V)|$ as the denominator in Equation (9). On the other hand, the lower bound shrinks the odds ratio by replacing $|G_j| - |eX_j(V)|$ to $|G_j| - |cX_j(V)|$ at the numerator and $|eX_j(V)|$ to $|cX_j(V)|$ at the denominator. After defining the upper and lower bounds, we introduce the Bounds Property, which estimates the odds ratio of all elements in k-ary collection over V based on the upper and lower bounds of V.

Theorem 4.7 (Bounds Property). *Given a set of searching nodes $V = \{v_1, v_2, ..., v_k\}$, (1) if the upper bound odds ratio of V in G_i over G_j is less than or equal to β, then all elements in the k-ary collection over V have odds ratio values less than or equal to β; (2) if its lower bound odds ratio is greater than or equal to β, then all elements in the k-ary collection over V have odds ratio values greater than or equal to β.*

Proof (Upper bound proof). To prove all elements in k-ary collection $P(V)$ have odds ratio values less than or equal to β in G_i over G_j, one only needs to prove that any element $w_1 \cup \cdots \cup w_k \in P(V)$ has $OR(w_1 \cup \cdots \cup w_k, G_i, G_j) \leq \beta$. The proof now becomes if $\frac{|cX_i(V)|*(|G_j|-|eX_j(V)|)}{|eX_j(V)|*(|G_i|-|cX_i(V)|)} \leq \beta$ holds, then $OR(w_1 \cup \cdots \cup w_k, G_i, G_j) = \frac{|eX_i(w_1 \cup \cdots \cup w_k)|*(|G_j|-|eX_j(w_1 \cup \cdots \cup w_k)|)}{|eX_j(w_1 \cup \cdots \cup w_k)|*(|G_i|-|eX_i(w_1 \cup \cdots \cup w_k)|)} \leq \beta$.

According to the proof in Frequency Property, in group G_j, $eT_j(v_k) \subseteq eT_j(w_k)$ implies $|eX_j(V)| \leq |eX_j(w_1 \cup \cdots \cup w_k)|$; in group G_i, $|eX_i(w_1 \cup \cdots \cup w_k)| \leq |cX_i(V)|$ holds. Thus, $|G_j| - |eX_j(w_1 \cup \cdots \cup w_k) \leq |G_j| - |eX_j(V)|$ and $|G_i| - |cX_i(V)| \leq |G_i| - |eX_i(w_1 \cup \cdots \cup w_k)|$. Therefore, $OR(w_1 \cup \cdots \cup w_k, G_i, G_j) = \frac{|eX_i(w_1 \cup \cdots \cup w_k)|*(|G_j|-|eX_j(w_1 \cup \cdots \cup w_k)|)}{|eX_j(w_1 \cup \cdots \cup w_k)|*(|G_i|-|eX_i(w_1 \cup \cdots \cup w_k)|)} \leq \frac{|cX_i(V)|*(|G_j|-|eX_j(V)|)}{|eX_j(V)|*(|G_i|-|cX_i(V)|)} \leq \beta$.

(Lower Bound Proof). Similarly, to prove all elements in k-ary collection over V have odds ratio values greater than or equal to β, we only need to prove that any element $w_1 \cup \cdots \cup w_k \in P(V)$ has $OR(w_1 \cup \cdots \cup w_k, G_i, G_j) \geq \beta$. The proof now becomes if $\frac{|eX_i(V)|*(|G_j|-|cX_j(V)|)}{|cX_j(V)|*(|G_i|-|eX_i(V)|)} \geq \beta$ holds, then $OR(w_1 \cup \cdots \cup w_k, G_i, G_j) = \frac{|eX_i(w_1 \cup \cdots \cup w_k)|*(|G_j|-|eX_j(w_1 \cup \cdots \cup w_k)|)}{|eX_j(w_1 \cup \cdots \cup w_k)|*(|G_i|-|eX_i(w_1 \cup \cdots \cup w_k)|)} \geq \beta$.

According to the proof in Frequency Property, $|eX_j(w_1 \cup \cdots \cup w_k)| \leq |cX_j(V)|$ and $|eX_i(V)| \leq |eX_i(w_1 \cup \cdots \cup w_k)|$ holds. Thus, $|G_j| - |cX_j(V)| \leq |G_j| - |eX_j(w_1 \cup \cdots \cup w_k)|$

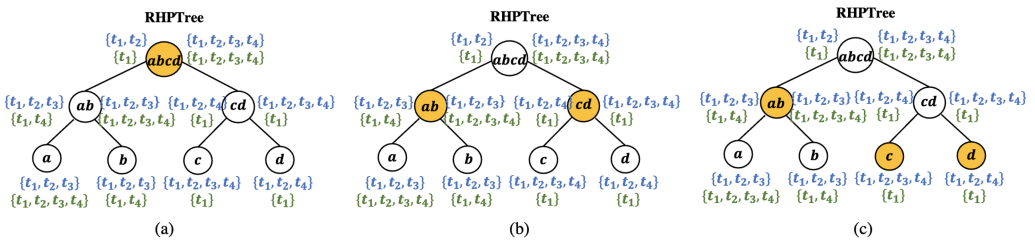

Fig. 6. The search paths derive from the root node.

and $|G_i| - |eX_i(w_1 \cup \cdots \cup w_k)| \leq |G_i| - |eX_i(V)|$. Therefore, $OR(w_1 \cup \cdots \cup w_k, G_i, G_j) = \frac{|eX_i(w_1 \cup \cdots \cup w_k)| * (|G_j| - |eX_j(w_1 \cup \cdots \cup w_k)|)}{|eX_j(w_1 \cup \cdots \cup w_k)| * (|G_i| - |eX_i(w_1 \cup \cdots \cup w_k)|)} \geq \frac{|eX_i(V)| * (|G_j| - |cX_j(V)|)}{|cX_j(V)| * (|G_i| - |eX_i(V)|)} \geq \beta$. □

As shown in Figure 2, given $V = \{ab\}$, the upper bound of V in $|G_i|$ is $\frac{|cX_i| * (|G_j| - |eX_j|)}{|eX_j| * (|G_i| - |cX_i|)} = 3$, then all elements in the k-ary collection over V have odds ratio values less than or equal to 3 in G_i; On the contrary, given $V = \{cd\}$, the lower bound of V in $|G_i|$ is $\frac{|eX_i| * (|G_j| - |cX_j|)}{|cX_j| * (|G_i| - |eX_i|)} = 9$, all elements in k-ary collection over V have odds ratio values greater than or equal to 9 in G_i.

4.3 RHPSearch

With the Frequency and Bounds Properties, the algorithm can prune unnecessary search paths that are evidently qualified or unqualified during the search process. Using the search principles of the Naïve RHPSearch in Algorithm 5 and the two properties, we can strategically search only the paths that fall between the upper and lower bounds, and these are only paths whose patterns cannot be safely discarded or qualified. Starting at the root node, we calculate all its upper and lower bounds, as well as the odds ratio values shown in line 1 of Algorithm 6. If the frequency and upper bound conditions are not met, the algorithm stops (lines 2–4). Otherwise, the lower bound will be checked. If the frequency and lower bound conditions are satisfied, the current nodes' collection is saved (lines 5–8). Otherwise, the algorithm checks the exact odds ratio value (lines 9–11), and then the three sub-searches from that node are launched as same as the Naïve RHPSearch in Algorithm 5.

Using the example shown in Figure 2, given the support threshold $\alpha = 0.5$ and the odds ratio threshold $\beta = 5$, the search process is depicted in Figure 6 and steps are given below from (1) to (3):

(1) Starts from the root node $\{abcd\}$, the $UpperBound(\{abcd\}) = \infty$ and the $LowerBound(\{abcd\}) = 0$. It is clear that the odds ratio of k-ary collection over $\{abcd\}$ is between 0 and infinity. Also, the exact odds ratio of $\{abcd\}$ is 3. Thus, the search continues.
(2) Based on the search strategy, the search splits into three sub-searches $\{ab, cd\}$, $\{ab\}$, $\{cd\}$. Their bounds are calculated as follows:
  (a) As $UpperBound(\{ab, cd\}) = 9$ and $LowerBound(\{ab, cd\}) = 3$, the odds ratio of k-ary collection over $\{ab, cd\}$ is between 3 and 9. The exact odds ratio of $\{ab, cd\}$ is 3. The search continues.
  (b) As $UpperBound(\{ab\}) = 3$ and $LowerBound(\{ab\}) = 0$, the odds ratio of k-ary collection over $\{ab\}$ is between 0 and 3. The search stops.
  (c) As $UpperBound(\{cd\}) = \infty$ and $LowerBound(\{cd\}) = 9$, the odds ratio of k-ary collection over $\{cd\}$ is between 9 and ∞. Then the search stops and saves the k-ary collection of $\{cd\}$.
(3) Continues the search over $\{ab, cd\}$ and splits it into three sub-searches $\{ab, c, d\}$, $\{ab, c\}$, $\{ab, d\}$. Their bounds are calculated as follows:

ALGORITHM 6: RHPSearch

Input: Suppport threshold: α, Odds Ratio threshold: β, Node collection: V
Output: Pattern collection $RHPSet$

1: Calculate eX_i, cX_j, $OR(V, G_i \cup G_j)$ and $OR(V, G_j \cup G_i)$
2: **if** $(cX_i.size < \alpha * (|G_i| + |G_j|)$ **or** $UpperBound(V, G_i) < \beta)$ **and** $(cX_j.size < \alpha * (|G_i| + |G_j|)$ **or** $UpperBound(V, G_j) < \beta)$ **then**
3: **return**
4: **end if**
5: **if** $(eX_i.size \geq \alpha * (|G_i| + |G_j|)$ **and** $LowerBound(V, G_i) \geq \beta)$ **or** $(eX_j.size \geq \alpha * (|G_i| + |G_j|)$ **and** $LowerBound(V, G_j) \geq \beta)$ **then**
6: $RHPSet+ = saveRHP(P(V))$
7: **return**
8: **end if**
9: **if** $(eX_i.size \geq \alpha * (|G_i| + |G_j|)$ **and** $OR(V, G_i, G_j) \geq \beta)$ **or** $(eX_j.size \geq \alpha * (|G_i| + |G_j|)$ **and** $OR(V, G_j, G_i) \geq \beta)$ **then**
10: $RHPSet+ = saveRHP(P(V))$
11: **end if**
12: $leaves = \{v | v \in V$ **and** $v.children.size = 0\}$
13: $nonLeaves = V - leaves$
14: **if** $nonLeaves.size > 0$ **then**
15: $v_n = nonLeaves.head$
16: $Nodes = (nonLeaves - v_n) + leaves$
17: $RHPSearch(Nodes + v_{n,R} + v_{n,L})$
18: **if** $v_{n,R} \notin RHPSet$ **then**
19: $RHPSearch(Nodes + v_{n,R})$
20: **end if**
21: **if** $v_{n,L} \notin RHPSet$ **then**
22: $RHPSearch(Nodes + v_{n,L})$
23: **end if**
24: **end if**

(a) As $UpperBound(\{ab, c, d\}) = 3$ and $LowerBound(\{ab, c, d\}) = 3$, the odds ratio of k-ary collection over $\{ab, c, d\}$ is 3. The search stops.
(b) As $UpperBound(\{ab, c\}) = 9$ and $LowerBound(\{ab, c\}) = 9$, the odds ratio of k-ary collection over $\{ab, c\}$ is 9. Then the search stops and saves the k-ary collection of $\{ab, c\}$.
(c) As $UpperBound(\{ab, d\}) = 3$ and $LowerBound(\{ab, d\}) = 3$, the odds ratio of k-ary collection over $\{ab, c\}$ is 3. The search stops.

Overall, the patterns that satisfy the preset threshold is k-ary collections over $\{cd\}$, $\{ab, c\}$, which is $\{c, d, cd, ac, bc, abc\}$.

4.4 Target Search and Sequential Delete Search

The RHPSearch, described in Section 4.3, provides a comprehensive search through the RHPTree to provide all the odds ratio patterns of a dataset. However, a full search is usually unnecessary when a study is only interested in patterns containing a subset of items. In such a situation, we only need to consider the paths which include the target item instead of going through all paths in the full search. The pseudocode of this target search is listed in Algorithm 7. In line 2 and line 3,

the algorithm keeps the ancestors of the target item and makes sure the searching node collection V contains the ancestor of the target item.

ALGORITHM 7: RHPSearch-TS

Input: Suppport threshold: α, $OddsRatio$ threshold: β, Node Collection: V
Output: Target pattern collection $TRHPSet$

Calculate the eX_i, cX_j, $OR(V, G_i, G_j)$ and $OR(V, G_j, G_i)$
$ancestors(T) = ancestors\ of\ T$
if $V \cap ancestors(T) = \emptyset$ **then**
  return
end if
if $(cX_i.size < \alpha * (|G_i| + |G_j|)$ **or** $UpperBound(V, G_i) < \beta)$ **and** $(cX_j.size < \alpha * (|G_i| + |G_j|)$ **or** $UpperBound(V, G_j) < \beta)$ **then**
  return
end if
if $(eX_i.size \geq \alpha * (|G_i| + |G_j|)$ **and** $LowerBound(V, G_i) \geq \beta)$ **or** $(eX_j.size \geq \alpha * (|G_i| + |G_j|)$ **and** $LowerBound(V, G_j) \geq \beta)$ **then**
  $TRHPSet+ = saveTRHP(P(V))$
  return
end if
if $(eX_i.size \geq \alpha * (|G_i| + |G_j|)$ **and** $OR(V, G_i, G_j) \geq \beta)$ **or** $(eX_j.size \geq \alpha * (|G_i| + |G_j|)$ **and** $OR(V, G_j, G_i) \geq \beta)$ **then**
  $TRHPSet+ = saveTRHP(P(V))$
end if
$leaves = \{v | v \in V$ **and** $v.children.size = 0\}$
$nonLeaves = V - leaves$
if $nonLeaves.size > 0$ **then**
  $v_n = nonLeaves.head$
  $Nodes = (nonLeaves - v_n) + leaves$
  $RHPSearch - TS(Nodes + v_{n,R} + v_{n,L})$
  if $v_{n,R} \notin TRHPSet$ **then**
    $RHPSearch - TS(Nodes + v_{n,R})$
  end if
  if $v_{n,L} \notin TRHPSet$ **then**
    $RHPSearch - TS(Nodes + v_{n,L})$
  end if
end if

The target search can be used to decompose the full search into multiple individual target searches. For example, the full search over $\{abcd\}$ can be replaced by target searches over the items $\{a\}$, $\{b\}$, $\{c\}$, $\{d\}$. However, performing a target search for every item sequentially is inefficient because some pattern's combinations will appear multiple times. For example, pattern $\{ab\}$ in the search of item a will also appear in the search of item b. To avoid redundancy, we refine the complete sequential target search and propose Sequential Delete Search, namely RHPSearch-SD, for parallel computing.

By utilizing high-performance computational resources, RHPSearch-SD (Algorithm 8) can be executed in parallel to speed up the mining processes. When executing in parallel, the number of items to be searched at once, or the batch size, needs to be specified. Assuming there are n

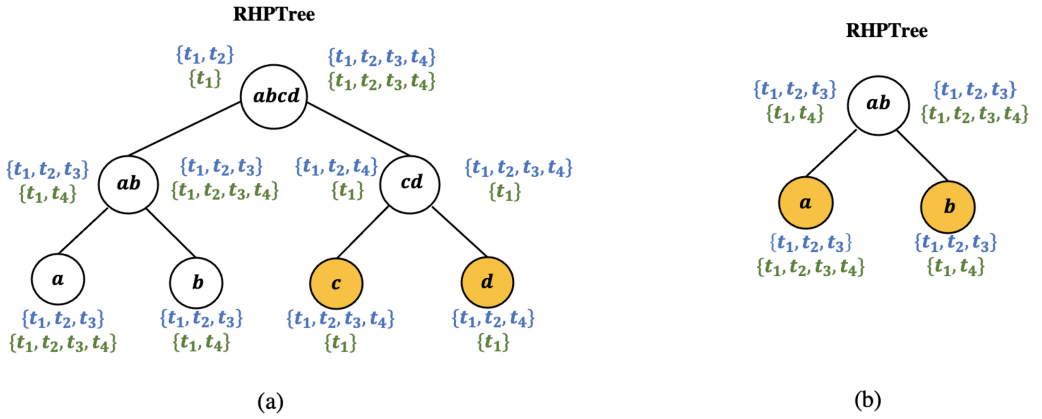

Fig. 7. Batch search and delete process.

ALGORITHM 8: RHPSearch-SD

Input: Batch size: b, Node collection: V
Output: Pattern collection: $RHPSet$

```
leaves = {v|v ∈ V and v.children.size = 0}
while leaves.size > 0 do
    ItemList = GetItems(leaves, b)
    for item ∈ ItemList do
        RHPSet=RHPSearch-TS(item) {Do in Parallel}
    end for
    RHPTree.delete(ItemList)
    leaves = leaves − ItemList
end while
return RHPSet
```

leaves in the RHPTree and m cores in a computational node. If each core searches for k items, then the batch size is $\lceil \frac{n}{m*k} \rceil$. During the search process, each core searches their assigned items on the same tree structure, which means there is no dependency issues across multiple cores. In our implementation, Scala parallelizes the search operation on each partition of the node collection and recombining all of the results that were completed in parallel [7]. After the user defines the batch size, multiple cores are launched to search the items in the batch at the same time. After finishing the batch, we delete those items from the RHPTree and then begin the next batch. The process continues until no item is left in the RHPTree. The batch search reduces the search time, but it can still potentially produce some duplicate patterns due to the parallelization. For example, if both a and b are searched in parallel, a and b are searched first and produce the pattern ab twice. The RHPSearch-SD could be optimized to reduce the occurrence of duplicate patterns in concurrent searches upon further research.

For the RHPTree shown in Figure 2, assume processing is taking place on a machine with two cores and the batch size is set to two. The support threshold $\alpha = 0.5$ and the odds ratio threshold $\beta = 2$. As shown in Figure 7, the parallel search follows: (1) search items c, d in parallel first; (2) delete items c, d from the RHPTree; (3) search items a, b in parallel; (4) delete items a, b from the RHPTree. The searches on the items c and d will have patterns $\{c, cd, ac, bc, abc\}$ and $\{d, cd\}$. After

Table 1. The Statistics of UCI Datasets

Dataset	Transaction Count	Item Count	Average Length Per Transaction	Class 1 Transaction Count	Class 2 Transaction Count
Chess	3,196	75	37	1,527	1,669
Connect-4	67,557	129	43	44,473	16,635
Pumsb	49,046	2,113	74	24,523	24,523
Accidents	340,183	468	33.8	170,092	170,091

deleting c, d from the RHPTree, the searches for items a, b start, and they end with "no pattern is identified." In summary, the patterns are $\{c, d, cd, ac, bc, abc\}$.

5 EXPERIMENTS

We evaluated our methods using UCI Machine Learning Repository datasets [13] and a sampled version of a real medical dataset from the Simons Foundation with **Single Nucleotide Polymorphisms** (**SNPs**) in **Autism Spectrum Disorder** (**ASD**) [16]. The performance of RHPSearch and RHPSearch-SD on the UCI datasets served as a baseline for efficiency comparison against other methods. The sampled ASD datasets are used to evaluate the scalability of the methods.

We compared our algorithms with two other algorithm categories as described in Section 2: (1) frequent itemset mining and (2) equivalent class mining. The concurrent frequent itemset mining category includes LCM-ver2 [53], PrePost+ [11], and negFIN [2]. The equivalent class category includes GC-growth-v2.1 [32], which also features odds ratio mining. Because frequent itemset mining and equivalent class mining are not designed directly for mining odds ratio patterns, an additional post-mining filtering process is required to produce desirable results for odds ratio patterns. For frequent pattern mining, the post-processing step calculates the odds ratio of each frequent pattern, while for equivalent class mining, it automatically calculates the odds ratio of each pattern closure. The performances with and without this post-processing are both shown for detailed comparison and discussion. All experiments were conducted on a single machine with Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40 GHz and 256 GB of RAM with 20 CPU cores available for fair comparisons.

5.1 UCI Data Experiments

The following experiments were conducted to evaluate the efficiency of using four UCI datasets including Chess, Connect, Accidents, and Pumsb, which are commonly used benchmark dataset for frequent pattern mining algorithms [17]. For the Chess and Connect-4 datasets, we used the class labels assigned to each sample. There is no class label in the Pumsb and Accidents datasets. Thus, we separated the data randomly into two equal-sized classes. The statistics of the four datasets are shown in Table 1.

5.1.1 RHPTree and Search Methods Evaluations. We evaluated our RHPTree and the three search methods, namely RHPSearch, RHPSearch-TS, and RHPSearch-SD, on the four datasets based on running time. The RHPTree was evaluated based on build time and time needed for insert and delete operations. Without bias, we included all the items in the dataset without any support threshold to build the tree.

To evaluate the tree building time, we used the procedure discussed in Section 4.2. Figure 8(a) shows the numbers of items and transactions that affect the RHPtree construction time. As depicted in Figure 8(a), the tree building time for the Pumsb dataset is far larger than that of the Connect-4 dataset despite their transaction counts being close. At the same time, the number of

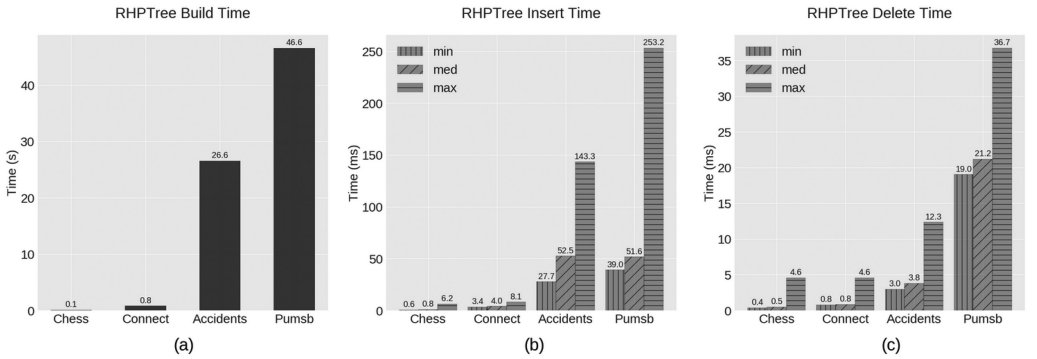

Fig. 8. Tree building time and the insert, delete time on four datasets.

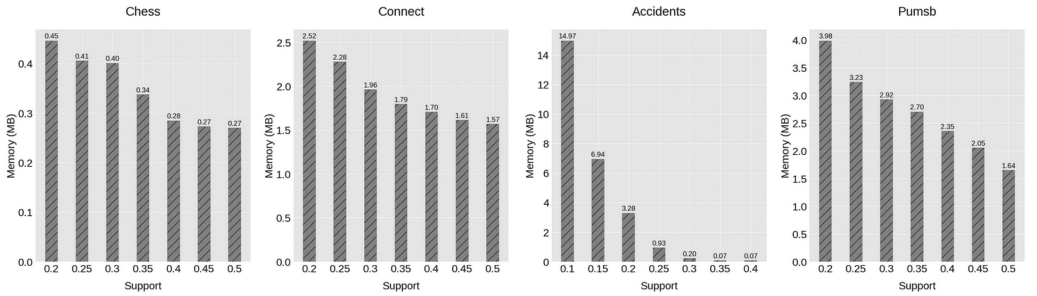

Fig. 9. The memory footprint for the RHPTree on different datasets.

transactions also has an impact on tree building time. The number of transactions in the Accidents dataset is about seven times that of the Pumsb dataset, and the running time of the Pumsb is only around twice that of the Accidents dataset although the number of items in the Pumsb dataset is approximately 4.5 times as many as the number in the Accidents dataset. We also provide the memory footprint in terms of different datasets as shown in Figure 9. As the support value decreases, more frequent items and their transaction sets are included in the RHPTree, which increases the size of the RHPTree. As we can see in Figure 9, for the Chess, Connect, and Pumsb datasets, the memory size is no more than 5 MB when the support value is 0.2, for the Accidents datasets, the memory size is no more than 15 MB when the support value is 0.1.

To evaluate the insertion time, we conducted a "leave-one-item-out" experiment where a selected item was removed from the dataset to build the RHPTree. We then inserted it back into the tree to measure the insertion time. This process is to mimic the real-world situation in which new items are added to the existing transactions. The results are shown in Figure 8(b). The insertion operation involves the nearest neighbor searching, node rotation, and transaction sets updating as described in Section 3.3. The efforts of these operations are affected by the size of RHPTree and the transaction number of each node in the tree. This is the main reason for the excessive running time for the Accidents and Pumsb datasets in comparison to the other two datasets.

To evaluate the deletion time, we began with a completely-built RHPTree and then deleted each item from the tree to measure the deletion time. This process was repeated for all items. The deletion times are shown in Figure 8(c). The delete operation involves updating of transaction sets, which depends on the size of the tree. The deletion time for the Pumsb dataset is more than deletion times for other datasets due to its large number of items and transactions. Also, the insert

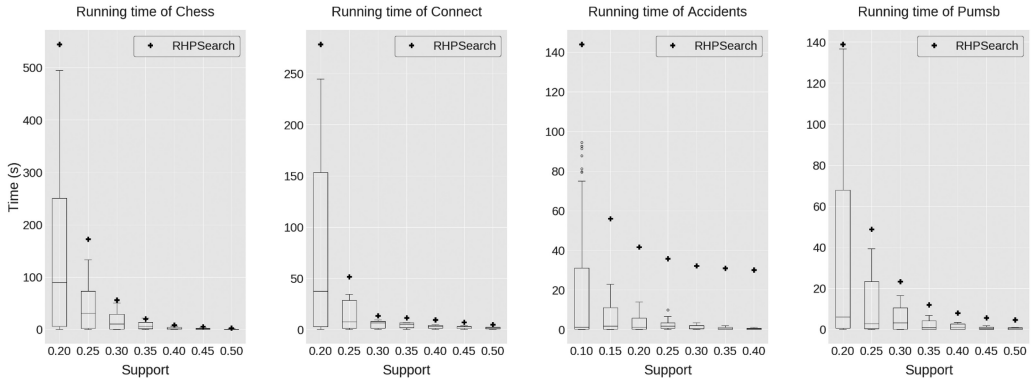

Fig. 10. The running time boxplots of the RHPSearch-TS compared with RHPSearch on four datasets.

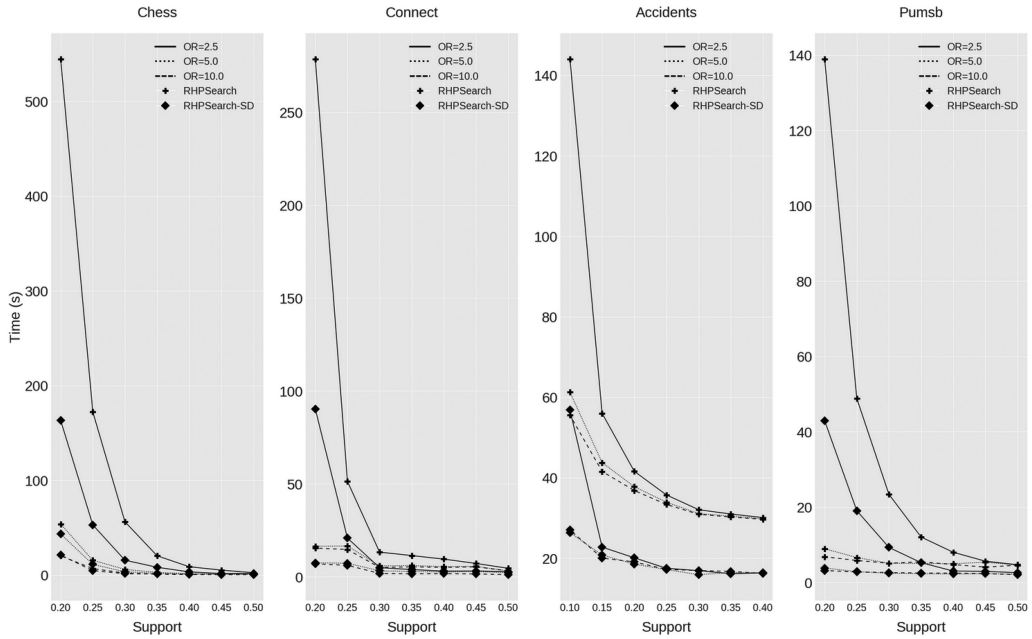

Fig. 11. The running time of RHPSearch and RHPSearch-SD with different odds ratio levels on four datasets.

time on the RHPTree is longer than the delete time. The main reason for this is that the insert continuously compares and rotates nodes while the delete simply uses the sibling to replace the parent node.

To evaluate the target search time for a specific item, we queried patterns containing a randomly selected item in the dataset with RHPSearch-TS. We set the odds ratio $\beta = 2.5$ with various support thresholds. Because the query time for each item is different, we recorded the search times for all items using the RHPSearch-TS. In Figure 10, for each dataset, the boxplot result shows the distribution of the search times under each support threshold. The maximal search time of RHPSearch-TS is comparable to the RHPSearch time, the median search time is much less than the RHPSearch, and the minimal search time is close to zero. The interquartile ranges of the boxplot,

Table 2. Number of Patterns with Different Support and Odds Ratio Thresholds

Chess							
β	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$	$\alpha = 0.5$
10	1,408,752	319,194	75,985	16,506	2,194	115	95
5	4,198,046	997,213	281,658	105,597	47,646	27,084	15,345
2.5	38,348,314	11,996,615	4,326,461	1,690,025	678,959	293,030	132,787
Connect-4							
β	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$	$\alpha = 0.5$
10	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
2.5	6,797,899	597,787	11	3	3	3	3
Accidents							
β	$\alpha = 0.1$	$\alpha = 0.15$	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$
10	63,106	4,648	384	16	0	0	0
5	63,106	4,648	384	16	0	0	0
2.5	169,672	22,150	1,780	86	0	0	0
Pumsb							
β	$\alpha = 0.2$	$\alpha = 0.25$	$\alpha = 0.3$	$\alpha = 0.35$	$\alpha = 0.4$	$\alpha = 0.45$	$\alpha = 0.5$
10	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0
2.5	3	3	3	3	3	3	3

$IQR = Q_3 - Q_1$, shows that 50% of items' search times are less than half of the RHPSearch's. In the figure, for each dataset, the search times and the number of frequent items share the same trend. The running times of both RHPSearch and RHPSearch-TS increase when the support threshold decreases because the algorithms included more frequent items and generated more patterns. It is worth noting that the search times soar when support thresholds are less than 0.35, 0.25, 0.2, 0.2 for the Chess, Connect, Accidents, and Pumsb, respectively. The reason for these increases is that the algorithm had to search deeper for the qualified patterns since the frequency property and the bounds properties defined in Section 4.2 were not activated during the searches at the top layers. At the same time, the search time also associates with the number of discovered patterns due to the computational time for the algorithm to generate the qualified patterns from the k-ary collection as discussed in Section 4.1. As listed in Table 2, when the odds ratio threshold $\beta = 2.5$, we observed that the increments of the number of discovered patterns were more than one million, five hundred thousand, and one thousand for the Chess, Connect, and Accidents at the abovementioned support thresholds separately. These increments of the number of patterns also contribute to the sharp increments of the running times.

To evaluate the search time using a distributed computing environment, we searched all patterns in the dataset by using the RHPSearch-SD with all 20 cores of the single server node listed previously. We set three odds ratio thresholds $\beta = 2.5$, 5.0, and 10.0. For each odds ratio threshold, there are $\lceil \frac{n}{20} \rceil$ iterations to complete the search, where n is the number of leaf nodes (items) in the tree. In Figure 11, RHPSearch-SD is significantly faster than RHPSearch when both the odds ratio and support thresholds decrease. We also evaluate the RHPSearch-SD algorithm in terms of the number of threads. The experiments were conducted on 5, 10, 15, and 20 cores with odds ratio equals to 2.5. Each core was assigned an item to search during each batch. In Figure 12, RHPSearch-SD time decreases significantly when the number of cores increases. The RHPSearch-SD utilized multiple cores to search and delete a batch of items. After each deletion, the tree size becomes smaller, which contributes to shorter search time in future searches. If multiple computational

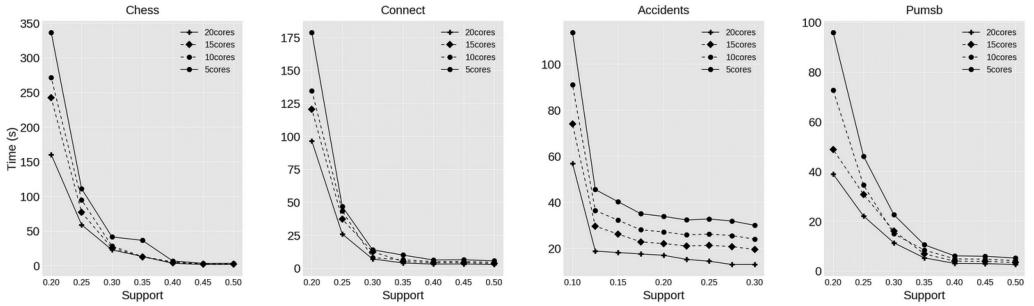

Fig. 12. Running time of RHPSearch-SD on four datasets with respect to the number of cores.

cores are available, it is beneficial to use RHPSearch-SD to speed up the search. This is a unique tree characteristic compared to other tree-based pattern indexing structures [19, 48].

5.1.2 Comparison with Other Methods. We compared our RHPSearch (including tree building time) methods with other state-of-the-art methods. Because all the compared methods require a post-processing step, i.e., filtering out patterns with $OR \geq \beta$, the experiments were separated into two phases: (1) before post-processing; (2) with post-processing. For frequent pattern mining and equivalent class mining, post-processing is to calculate the odds ratio of each frequent pattern or each pattern closure. We set a fixed odds ratio β to 2.5 for all datasets. We also set a range of support values between 0.2 and 0.5 with a 0.05 increment for the Chess, Connect and Pumsb datasets, and the range of support values between 0.1 and 0.4 with a 0.05 increment for the Accident dataset to show the clear differences between the compared running times.

In phase (1), all algorithms utilized a single core to create a fair comparison with other algorithms which were not designed for distributed computing. We directly ran the adapted frequent itemset algorithms LCM-ver2 [53], PrePost+ [11], and negFIN [2], equivalent class algorithm GC-growth-v2.1 [32], and our RHPSearch. We did not include the RHPSearch-SD algorithm in Phase (1) as it is designed for distributed computing. In Figure 13, the Phase (1) result shows that the frequent itemset mining algorithms are faster than other algorithms on the datasets which do not have a large number of frequent patterns, such as Chess and Accidents datasets. However, for the datasets with large numbers of frequent patterns, such as Connect-4, and Pumsb, the frequent itemset mining algorithms require more time to identify all frequent items. The GC-growth algorithm outperforms on the Connect dataset but suffers on the Pumsb dataset. The reason for the performance differences is that GC-growth mines the frequent equivalent classes instead of mining the frequent patterns directly. There are more frequent equivalent classes in Pumsb compared to the Connect. Thus, the running time of GC-growth for the Connect dataset is less than the Pumsb dataset. Our RHPSearch algorithm is much faster than any other algorithm on the Pumsb dataset because it detects the risk patterns directly without mining all frequent patterns. Also, the running time of RHPSearch on the Connect dataset is closer to the GC-growth. It is worth noting that, to get the odds ratio patterns, the frequent itemset mining algorithms and equivalent class algorithm must perform a post-processing step while RHPSearch does not. In phase (1), other algorithms had not done the post-processing while our RHPSearch had already found all qualified odds ratio patterns without post-processing.

In phase (2), we included the post-processing time for the frequent itemset algorithms and equivalent class algorithm. These algorithms utilized 20 cores for the post-processing step. Our RHPSearch-SD was included in Phase (2) with 20 cores, and the RHPSearch still used the single core since it does not require the post-processing time. In Figure 13, the Phase (2) results

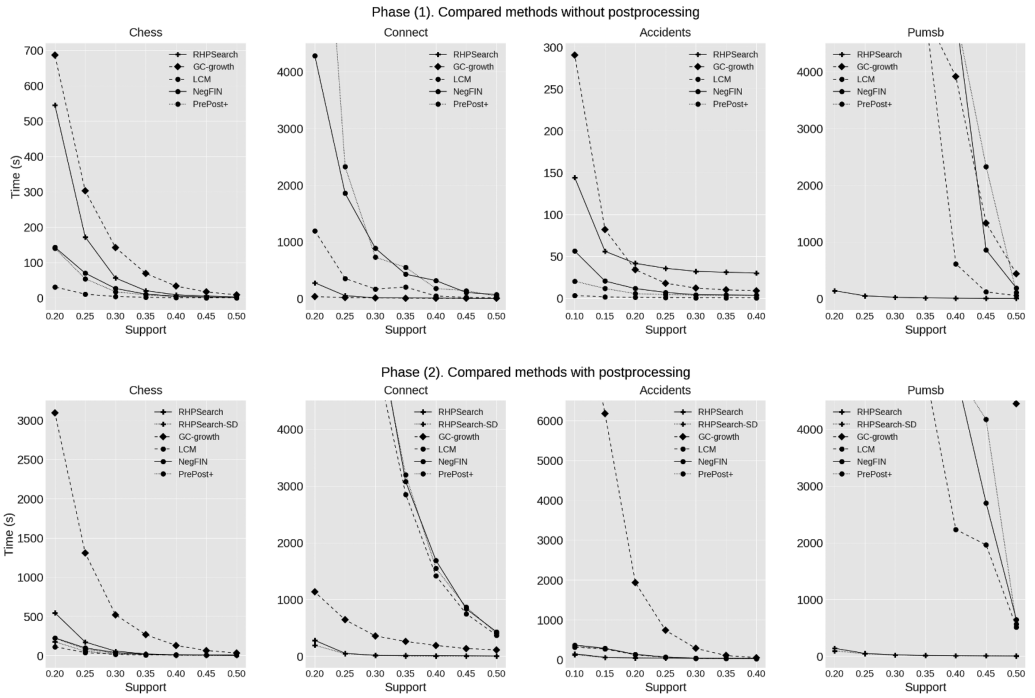

Fig. 13. Running time comparisons with other methods. Phase (1) is running time before post-processing, and Phase (2) is running time with post-processing.

show that the frequent itemset mining algorithm cannot finish in one hour on the Connect and Pumsb datasets. This is because there are a large number of frequent patterns generated by the algorithms. Consequently, more frequent items require longer post-processing time. The running time of the GC-growth algorithm increases dramatically when adding the post-processing time to the Chess, Pumsb, and Accidents datasets as it is time-consuming to calculate the odds ratio values and generate qualified odds ratio patterns from a large number of the frequent equivalent classes. Our RHPSearch and RHPSearch-SD algorithms outperform significantly on the Connect and Pumsb dataset. They also beat other algorithms on the Accident dataset. The running time of RHPSearch-SD is closer to the frequent pattern mining algorithms and much faster compared to the GC-growth.

Overall, the compared methods struggle when there are many frequent items or frequent equivalent classes exist in the dataset. Because the equivalent class mining and adapted frequent itemset mining algorithms cannot finish mining frequent patterns or frequent equivalent classes, then it is unfeasible to extract odds ratio patterns in a real-world applications. Using the RHPSearch methods, we can directly check the qualified risk patterns instead of wasting time on mining frequent patterns. In the experiment, the Connect dataset has more than 6 million patterns when $\beta = 2.5$ and support threshold $\alpha = 0.2$. The Accidents dataset does not have patterns when support threshold is more than 0.3. To obtain more patterns under a manageable execution time algorithm comparison, we chose 0.1 as the support threshold to ensure some results from other algorithms. When $\alpha = 0.1$ and $\beta = 2.5$, there are more than one hundred thousand patterns. For the Pumsb dataset, even by setting $\alpha = 0.4$, our algorithm is proven much faster than other algorithms as shown in Figure 13 although there are only three odds ratio patterns exist. In the experiment, even by setting lower

Table 3. The Description of the Sampled ASD Dataset

Dataset	Transaction Count	Item Count	Average Length Per Transaction	Class 1 Transaction Count	Class 2 Transaction Count
Sample 1	1,000	50	25	500	500
Sample 2	1,000	75	38	500	500
Sample 3	1,000	100	50	500	500
Sample 4	1,000	125	63	500	500
Sample 5	1,000	150	75	500	500

$\alpha = 0.15$ and $\beta = 2.5$, there are still the same three patterns listed in Table 2. The risk patterns can be used to interpret the potential factors that contribute to the outcomes. For the chess and connect game datasets which have the win and loss labels, the risk patterns can be used to analyze the potential causes for winning or losing the game. For example, the patterns (hdchk = f, wkna8 = f, skrxp = f, mulch = f, bxqsq = f, skach = f), (a1 = o, g6 = b, f6 = b, e6 = b) have winning odds ratios $OR = 10.32$ and $OR = 2.68$ in the Chess and Connect datasets, respectively. Such types of patterns may guide the players to take less risky moves to increase the chances to win the game if the next moves match with the patterns. The traffic accidents dataset records different circumstances where accidents have occurred. The risk patterns can be used to analyze the causes of different collisions, such as traffic condition, environmental conditions, and human conditions. The pumsb dataset contains census data for population, housing, and income. The researchers can use these patterns to analyze the differences between cohorts and areas for health disparity studies. These datasets serve as common datasets for performance comparisons.

5.2 Scalability Assessments

The sampled ASD dataset contains SNPs from two groups: people with and without autism. The scalability is tested with three variables: (1) the number of SNPs, (2) the number of data records, and (3) the odds ratio levels. In reality, the number of SNPs is much higher. However, due to comparisons with existing algorithms, we only use SNP numbers ranging from 50 to 150.

In the first part, we set the data records to be 1,000 (500 people in each group) with odds ratio $\beta = 2.5$ and varied the numbers of SNPs to be 50, 75, 100, 125, and 150. The data is described in Table 3. We compared RHPSearch and RHPSearch-SD by using 5, 10, 15, and 20 cores with other methods. Due to the performance issue with other methods, the running time is recorded without post-processing and their results are not odds patterns. We evaluated with different support thresholds and the running time is shown in Table 4. We can see that RHPSearch and RHPSearch-SD outperforms other methods by a large margin in efficiency and scalability. The LCM method is the best of the other compared algorithms. Upon further inspection, the output it produces is more than 1 TB when processing only 125 SNPs with a 0.7 support threshold. This shows the large amount of overhead required for generating frequent patterns instead of directly targeting odds ratio patterns. The GC-growth runs more than four hours when there are only 100 SNPs with a 0.7 support threshold. The exhaustive equivalent class generation drastically limits the performance of GC-growth. The huge number of mined patterns requires a large amount of post-processing time and unfinished jobs, causing failure for further odds ratio pattern extraction. In comparison, our algorithm can still extract the patterns within a reasonable time when the number of SNPs is 150. To evaluate the usefulness of the patterns, we found that there are 0.9% patterns containing genes that appear in AutDB, an evolving database for autism research community [5], by setting $\alpha = 0.5$ and $\beta = 2.5$. 14 patterns have all genes shown in AutDB. For example, the patterns (CLSTN2,

Table 4. The Running Time (Seconds) Comparison of 1,000 Data Records

Support (α)	Methods	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
0.9	LCM	0.08	0.64	38.93	1,517.02	NA*
	NegFIN	0.30	3.52	77.33	6,613.21	NA*
	PrePost+	0.40	3.14	141.25	6,945.20	NA*
	GC-growth	0.22	2.46	57.34	NA	NA
	RHPSearch	0.27	0.64	0.69	2.49	19.43
	RHPSearch-SD-5cores	0.26	0.62	0.64	2.23	18.25
	RHPSearch-SD-10cores	0.25	0.60	0.63	2.19	16.54
	RHPSearch-SD-15cores	0.22	0.58	0.6	1.98	14.72
	RHPSearch-SD-20cores	0.22	0.54	0.67	1.67	12.69
0.8	LCM	0.14	19.09	1,871.16	NA*	-
	NegFIN	1.57	78.52	5,340.15	NA*	-
	PrePost+	1.38	58.94	4,970.24	NA*	-
	GC-growth	1.49	78.14	2,948.38	NA	-
	RHPSearch	0.26	0.64	0.99	4.47	28.75
	RHPSearch-SD-5cores	0.26	0.62	0.92	3.97	26.65
	RHPSearch-SD-10cores	0.23	0.61	0.86	3.32	22.49
	RHPSearch-SD-15cores	0.21	0.57	0.67	2.98	18.67
	RHPSearch-SD-20cores	0.21	0.54	0.66	2.86	16.78
0.7	LCM	0.43	190.64	NA*	-	-
	NegFIN	2.80	579.12	NA*	-	-
	PrePost+	4.29	803.02	NA*	-	-
	GC-growth	5.17	805.38	NA	-	-
	RHPSearch	0.28	0.76	1.09	5.96	38.31
	RHPSearch-SD-5cores	0.27	0.72	0.97	5.23	35.78
	RHPSearch-SD-10cores	0.23	0.69	0.92	5.01	32.03
	RHPSearch-SD-15cores	0.21	0.61	0.84	4.90	27.09
	RHPSearch-SD-20cores	0.20	0.59	0.78	4.83	25.11
0.6	LCM	0.83	1,201.27	-	-	-
	NegFIN	3.90	4,231.46	-	-	-
	PrePost+	5.64	4,019.10	-	-	-
	GC-growth	10.44	4,511.93	-	-	-
	RHPSearch	0.29	0.72	1.44	8.28	50.39
	RHPSearch-SD-5cores	0.26	0.71	1.32	7.87	46.27
	RHPSearch-SD-10cores	0.24	0.67	1.02	6.24	41.37
	RHPSearch-SD-15cores	0.24	0.63	0.92	5.98	33.30
	RHPSearch-SD-20cores	0.21	0.63	0.79	5.88	28.03
0.5	LCM	1.12	3,627.87	-	-	-
	NegFIN	5.77	7,928.07	-	-	-
	PrePost+	7.51	12,222.63	-	-	-
	GC-growth	16.94	13,274.21	-	-	-
	RHPSearch	0.30	0.86	1.68	12.05	74.10
	RHPSearch-SD-5cores	0.29	0.82	1.24	10.97	69.30
	RHPSearch-SD-10cores	0.25	0.78	1.08	9.64	62.95
	RHPSearch-SD-15cores	0.23	0.75	0.92	8.44	56.08
	RHPSearch-SD-20cores	0.22	0.72	0.85	7.98	52.10

Note: "NA" marks jobs unfinished due to running time greater than four hours; "NA*" marks jobs unfinished due to output more than 1 TB; "-" marks jobs unfinished due to no more experimentation necessary based on the unfinished status of its previous setting.

HERC2, RNF38, RBFOX1) (Odds Ratio: 2.53) and (HERC2, SLC24A2, RNF38, RBFOX1) (Odds Ratio: 3.1) appear more frequently in autism patients. Other patterns could be potential new findings.

In the second part, we evaluated the performance of RHPSearch with different numbers of data records. To get the data records with different numbers, we did the sampling with the replacement

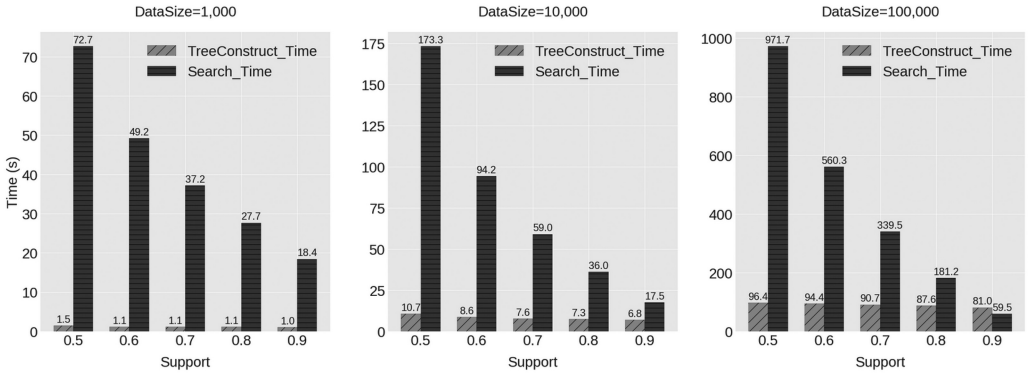

Fig. 14. The running time (seconds) of the tree construction and search with different data sizes and support thresholds.

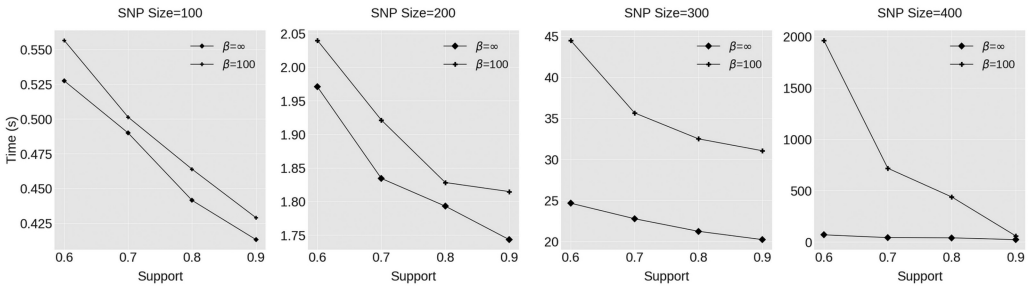

Fig. 15. The running time (seconds) of minimal pattern length is 20 under different feature sizes, support, and odds ratio thresholds.

on the autism and the non-autism groups, respectively. We set the number of SNPs to 150 with $\beta = 2.5$, the support threshold ranges from 0.9 to 0.5, and the numbers of data records are varied from 1,000, 10,000, to 100,000. We evaluated the times of tree building and search separately. As shown in Figure 14, the times of tree construction are only slightly affected by the support threshold in all data sizes. Both construction and search time increase as the data size grows, and the construction time exceeds search time at a 0.9 support threshold in the largest dataset size due to the increased complexity in node distance calculation. The effect of the support threshold on search time becomes progressively more significant with the growth of data size.

In the third part, we conducted more experiments to test the limitation of RHPSearch's scalabilities. In the first part, as shown in Table 4, RHPSearch does not perform well when the number of SNPs is beyond 200. There are three reasons: (1) we generated all patterns when detecting the k-ary collection during the search process, (2) the number of search branches bloated when setting a small threshold and the algorithm searches in deeper levels, and (3) the generation of the long patterns from the k-ary collection takes much computational time. Thus, we evaluated the performance using k-ary descendant collections to represent patterns directly instead of generating all patterns. We used 1,000 sampled data records and a minimum pattern length of 20 for our evaluation. The odds ratio thresholds were set to 100 and infinity, and the support thresholds ranged from 0.6 to 0.9. The number of SNPs ranged from 100 to 400. The running times are shown in Figure 15. The result shows that the running time of $\beta = \infty$ is less than the $\beta = 100$ in all four cases. The running time increases sharply when the number of SNPs is set to 400 and odds ratio

threshold is set to 100. The reason for the large time increment is the increased frequent items in the dataset and also the frequency property and the bounds property were not able to be used at the top layers of the RHPTree, and the algorithm had to split further to finish all inspection. However, the RHPSearch is able to discover the long odds ratio patterns in the dataset because of the top-down search strategy. In the experiment, the longest valid pattern in the 400 SNP dataset consisted of 34 SNPs, which is usually considered to be a long pattern in a real-world application.

6 CONCLUSION

In almost all epidemiology research, finding risk patterns is a critical process in identifying risks for diseases. In this article, we introduce the RHPTree, a dynamic hierarchical tree, and a collection of efficient search methods, RHPSearch. The RHPTree supports common insert, delete, and update operations to adapt to dataset evolvement and iterative search process without tree reconstruction. In addition, the RHPTree data structure is item-oriented and suitable for targeted searches for users interested in specific items. The RHPSearch-TS and paralleled search method RHPSearch-SD decompose the full search into item-based searches, which greatly speeds up the odds ratio pattern extraction process. Experimental results on the UCI datasets demonstrate that our method significantly reduces the running time when compared to the existing methods, negFIN, PrePost+, LCM-ver2, and GC-growth. Also, the target search is faster than the full RHPSearch when searching a specific item, thus offering an efficient alternative when there is a specific mining target. Experiments on the sampled genomic dataset demonstrate that our method outperforms other methods by a large margin, especially when many frequent patterns exist but only a few of them are qualified odds ratio patterns. Our approach is most advantageous when the data contains a large number of features which may form long patterns.

Our proposed approaches may open up new possibilities for a broad range of applications to healthcare research for discovering complex, interacting risk factors. Given the dynamic structure and flexibility of our approaches, they may offer advantages in other pattern mining areas, such as high-utility pattern mining, sequential pattern mining, and distributed pattern mining.

text

ACKNOWLEDGMENTS

The authors thank Dr. Matt Spencer for his contribution in pre-processing of the autism data set, and the University of Missouri Research Computing Support Services (RCSS) group for providing computing support and technical advice.

REFERENCES

[1] Charu C. Aggarwal, Mansurul A. Bhuiyan, and Mohammad Al Hasan. 2014. *Frequent Pattern Mining Algorithms: A Survey*. Springer International Publishing, Cham, 19–64. DOI: https://doi.org/10.1007/978-3-319-07821-2_2

[2] Nader Aryabarzan, Behrouz Minaei-Bidgoli, and Mohammad Teshnehlab. 2018. negFIN: An efficient algorithm for fast mining frequent itemsets. *Expert Systems with Applications* 105 (2018), 129–143. DOI: https://doi.org/10.1016/j.eswa.2018.03.041

[3] James Bailey, Thomas Manoukian, and Ramamohanarao Kotagiri. 2002. Fast algorithms for mining emerging patterns. In *Proceedings of the Principles of Data Mining and Knowledge Discovery*. Springer, Berlin, 39–50.

[4] Dennis L. Barbour. 2019. Precision medicine and the cursed dimensions. *npj Digital Medicine* 2, 1 (2019), 4. DOI: https://doi.org/10.1038/s41746-019-0081-5

[5] Saumyendra N. Basu, Ravi Kollu, and Sharmila Banerjee-Basu. 2008. AutDB: A gene reference resource for autism research. *Nucleic Acids Research* 37, Supplement 1 (Nov. 2008), D832–D836. DOI: https://doi.org/10.1093/nar/gkn835

[6] Huong Bui, Bay Vo, Tu-Anh Nguyen-Hoang, and Unil Yun. 2021. Mining frequent weighted closed itemsets using the WN-list structure and an early pruning strategy. *Applied Intelligence* 51, 3 (2021), 1439–1459. DOI: https://doi.org/10.1007/s10489-020-01899-7

[7] Luís Campos, Aaron Hawley, Olivier Blanvillain, and Heather Miller. 2013. Parallel Collections Overview. Retrieved September 30, 2010 from https://docs.scala-lang.org/overviews/parallel-collections/overview.html.

[8] Pauline Chaste and Marion Leboyer. 2012. Autism risk factors: Genes, environment, and gene-environment interactions. *Dialogues in Clinical Neuroscience* 14, 3 (2012), 281–292.

[9] Casey Crump, Jan Sundquist, Marilyn A. Winkleby, and Kristina Sundquist. 2016. Interactive effects of physical fitness and body mass index on the risk of hypertension. *JAMA International Medicine* 176, 2 (2016), 210–216. DOI: https://doi.org/10.1001/jamainternmed.2015.7444

[10] Peter Cummings. 2009. The relative merits of risk ratios and odds ratios. *Archives of Pediatrics & Adolescent Medicine* 163, 5 (2009), 438–445.

[11] Zhi-Hong Deng and Sheng-Long Lv. 2015. PrePost+: An efficient N-lists-based algorithm for mining frequent itemsets via Children'Parent Equivalence pruning. *Expert Systems with Applications* 42, 13 (2015), 5424–5432. DOI: https://doi.org/10.1016/j.eswa.2015.03.004

[12] Guozhu Dong and Jinyan Li. 1999. Efficient mining of emerging patterns: Discovering trends and differences. In *Proceedings of the 5th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 43–52.

[13] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. (2017). Retrieved 01 May 2020 from http://archive.ics.uci.edu/ml.

[14] H. Fan and Ramamohanarao Kotagiri. 2006. Fast discovery and the generalization of strong jumping emerging patterns for building compact and accurate classifiers. *IEEE Transactions on Knowledge and Data Engineering* 18, 6 (2006), 721–737. DOI: https://doi.org/10.1109/TKDE.2006.95

[15] Marie-Julie Favé, Fabien C. Lamaze, David Soave, Alan Hodgkinson, Héloïse Gauvin, Vanessa Bruat, Jean-Christophe Grenier, Elias Gbeha, Kimberly Skead, Audrey Smargiassi, Markey Johnson, Youssef Idaghdour, and Philip Awadalla. 2018. Gene-by-environment interactions in urban populations modulate risk phenotypes. *Nature Communications* 9, 1 (2018), 827. DOI: https://doi.org/10.1038/s41467-018-03202-2

[16] Gerald D. Fischbach and Catherine Lord. 2010. The Simons Simplex Collection: A resource for identification of autism genetic risk factors. *Neuron* 68, 2 (2010), 192–195. DOI: https://doi.org/10.1016/j.neuron.2010.10.006

[17] P. Fournier-Viger, A. Gomariz, T. Gueniche, A. Soltani, and C.-Wei Wu. 2014. Spmf: A java open-source pattern mining library. *Journal of Machine Learning Research* 15, 1 (2014), 3389–3393.

[18] Philippe Fournier-Viger, Jiaxuan Li, Jerry Chun-Wei Lin, Tin Truong Chi, and R. Uday Kiran. 2020. Mining cost-effective patterns in event logs. *Knowledge-Based Systems* 191 (2020), 105241. DOI: https://doi.org/10.1016/j.knosys.2019.105241

[19] Philippe Fournier-Viger, Espérance Mwamikazi, Ted Gueniche, and Usef Faghihi. 2013. MEIT: Memory efficient itemset tree for targeted association rule mining. In *Proceedings of the Advanced Data Mining and Applications*. Springer, Berlin, 95–106.

[20] Philippe Fournier-Viger, Espérance Mwamikazi, Ted Gueniche, and Usef Faghihi. 2013. MEIT: Memory efficient itemset tree for targeted association rule mining. In *Proceedings of the Advanced Data Mining and Applications*. Hiroshi Motoda, Zhaohui Wu, Longbing Cao, Osmar Zaiane, Min Yao, and Wei Wang (Eds.). Springer, Berlin, 95–106.

[21] M. Garcia-Borroto, O. Loyola-Gonzalez, J. F. Martinez-Trinidad, and J. A. Carrasco-Ochoa. 2017. Evaluation of quality measures for contrast patterns by using unseen objects. *Expert Systems with Applications* 83, C (2017), 104–113. DOI: https://doi.org/10.1016/j.eswa.2017.04.038

[22] Richard A. Goodman, Samuel F. Posner, Elbert S. Huang, Anand K. Parekh, and Howard K. Koh. 2013. Defining and measuring chronic conditions: Imperatives for research, policy, program, and practice. *Preventing Chronic Disease* 10, E66 (2013), E66. DOI: https://doi.org/10.5888/pcd10.120239

[23] Jiawei Han, Jian Pei, and Yiwen Yin. 2000. Mining frequent patterns without candidate generation. *SIGMOD Rec.* 29, 2 (2000), 1–12. DOI: https://doi.org/10.1145/335191.335372

[24] Nan Jiang and Le Gruenwald. 2006. CFI-Stream: Mining closed frequent itemsets in data streams. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 592–597. DOI: https://doi.org/10.1145/1150402.1150473

[25] Park Jong Soo, Chen Ming-Syan, and P. S. Yu. 1997. Using a hash-based method with transaction trimming for mining association rules. *IEEE Transactions on Knowledge and Data Engineering* 9, 5 (1997), 813–825. DOI: https://doi.org/10.1109/69.634757

[26] Bamba Kane, Bertrand Cuissart, and Bruno Crémilleux. 2015. Minimal jumping emerging patterns: Computation and practical assessment. In *Proceedings of the 19th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Vol. 9077. Springer International Publishing, 722–733. DOI: https://doi.org/10.1007/978-3-319-18038-0_56

[27] Heonho Kim, Unil Yun, Yoonji Baek, Jongseong Kim, Bay Vo, Eunchul Yoon, and Hamido Fujita. 2021. Efficient list based mining of high average utility patterns with maximum average pruning strategies. *Information Sciences* 543, 3 (2021), 85–105. DOI: https://doi.org/10.1016/j.ins.2020.07.043

[28] M. Kubat, A. Hafez, V. V. Raghavan, J. R. Lekkala, and Chen Wei Kian. 2003. Itemset trees for targeted association querying. *IEEE Transactions on Knowledge and Data Engineering* 15, 6 (2003), 1522–1534. DOI: https://doi.org/10.1109/TKDE.2003.1245290

[29] Jennifer Lavergne, Ryan Benton, and Vijay V. Raghavan. 2012. Min-max itemset trees for dense and categorical datasets. In *Proceedings of the Foundations of Intelligent Systems*. Springer, Berlin, 51–60.
[30] Jennifer Lavergne, Ryan Benton, and Vijay V. Raghavan. 2012. Min-max itemset trees for dense and categorical datasets. In *Proceedings of the Foundations of Intelligent Systems*. Li Chen, Alexander Felfernig, Jiming Liu, and Zbigniew W. Raś (Eds.). Springer, Berlin, 51–60.
[31] Carson Kai-Sang Leung, Quamrul I. Khan, Zhan Li, and Tariqul Hoque. 2007. CanTree: A canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems* 11, 3 (2007), 287–311. DOI: https://doi.org/10.1007/s10115-006-0032-8
[32] Haiquan Li, Jinyan Li, Limsoon Wong, Mengling Feng, and Yap-Peng Tan. 2005. Relative risk and odds ratio: A data mining perspective. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*. ACM, New York, NY, 368–377. DOI: https://doi.org/10.1145/1065167.1065215
[33] Haoyuan Li, Yi Wang, Dong Zhang, Ming Zhang, and Edward Y. Chang. 2008. Pfp: Parallel Fp-growth for query recommendation. In *Proceedings of the 2008 ACM Conference on Recommender Systems*. ACM, New York, NY, 107–114. DOI: https://doi.org/10.1145/1454008.1454027
[34] Jinyan Li, Guimei Liu, and Limsoon Wong. 2007. Mining statistically important equivalence classes and delta-discriminative emerging patterns. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 430–439. DOI: https://doi.org/10.1145/1281192.1281240
[35] Jinyan Li and Qiang Yang. 2007. Strong compound-risk factors: Efficient discovery through emerging patterns and contrast sets. *IEEE Transactions on Information Technology in Biomedicine* 11, 5 (2007), 544–552. DOI: https://doi.org/10.1109/TITB.2007.891163
[36] Chun-Wei Lin, Tzung-Pei Hong, and Wen-Hsiang Lu. 2011. An effective tree structure for mining high utility itemsets. *Expert Systems with Applications* 38, 6 (2011), 7419–7424. DOI: https://doi.org/10.1016/j.eswa.2010.12.082
[37] S. Liu, B. Hooi, and C. Faloutsos. 2019. A contrast metric for fraud detection in rich graphs. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2019), 2235–2248. DOI: https://doi.org/10.1109/TKDE.2018.2876531
[38] O. Loyola-González, R. Monroy, J. Rodríguez, A. López-Cuevas, and J. I. Mata-Sánchez. 2019. Contrast patteren-based classification for bot detection on Twitter. *IEEE Access* 7 (2019), 45800–45817. DOI: https://doi.org/10.1109/ACCESS.2019.2904220
[39] JoAnn E. Manson, Nancy R. Cook, I.-Min Lee, William Christen, Shari S. Bassuk, Samia Mora, Heike Gibson, Christine M. Albert, David Gordon, and Trisha Copeland. 2019. Marine n-3 fatty acids and prevention of cardiovascular disease and cancer. 380, 1 (2019), 23–32.
[40] Howard L. McLeod. 2015. Precision medicine to improve the risk and benefit of cancer care: Genetic factors in vincristine-related neuropathy. *JAMA* 313, 8 (2015), 803–804. DOI: https://doi.org/10.1001/jama.2015.1086
[41] Nicholas Monath, Ari Kobren, Akshay Krishnamurthy, Michael R. Glass, and Andrew McCallum. 2019. Scalable hierarchical clustering with tree grafting. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 1438–1448. DOI: https://doi.org/10.1145/3292500.3330929
[42] Hyoju Nam, Unil Yun, Bay Vo, Tin Truong, Zhi-Hong Deng, and Eunchul Yoon. 2020. Efficient approach for damped window-based high utility pattern mining with list structure. *IEEE Access* 8 (2020), 50958–50968. DOI: https://doi.org/10.1109/ACCESS.2020.2979289
[43] Tomas Olsson, Lisa F. Barcellos, and Lars Alfredsson. 2017. Interactions between genetic, lifestyle and environmental risk factors for multiple sclerosis. *Nature Reviews Neurology* 13, 1 (2017), 25–36. DOI: https://doi.org/10.1038/nrneurol.2016.187
[44] Michael Phinney. 2017. *Distributed Frequent Hierarchical Pattern Mining for Robust andEfficient Large-Scale Association Discovery*. Thesis.
[45] Gwangbum Pyun, Unil Yun, and Keun Ho Ryu. 2014. Efficient frequent pattern mining based on Linear Prefix tree. *Knowledge-Based Systems* 55 (2014), 125–139. DOI: https://doi.org/10.1016/j.knosys.2013.10.013
[46] Shashi Raj, Dharavath Ramesh, M. Sreenu, and Krishan Kumar Sethi. 2020. EAFIM: Efficient apriori-based frequent itemset mining algorithm on Spark for big transactional data. *Knowledge and Information Systems* 62, 4 (2020). DOI: https://doi.org/10.1007/s10115-020-01464-1
[47] D. Savage, X. Zhang, P. Chou, X. Yu, and Q. Wang. 2017. Distributed mining of contrast patterns. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 1881–1890. DOI: https://doi.org/10.1109/TPDS.2016.2637914
[48] Lior Shabtay, Philippe Fournier-Viger, Rami Yaari, and Itai Dattner. 2020. A guided FP-Growth algorithm for mining multitude-targeted item-sets and class association rules in imbalanced data. *Information Sciences* 553, (2020), 353–375. DOI: https://doi.org/10.1016/j.ins.2020.10.020
[49] Rama S. Singh and Bhagwati P. Gupta. 2020. Genes and genomes and unnecessary complexity in precision medicine. *npj Genomic Medicine* 5, 1 (2020), 21. DOI: https://doi.org/10.1038/s41525-020-0128-1
[50] M. Sreedevi and G. Vijay Kumar. 2014. Parallel and distributed approach for mining closed regular patterns on incremental databases at user thresholds. In *Proceedings of the 2014 International Conference on Information and*

Communication Technology for Competitive Strategies. ACM, New York, NY, Article 59, 5 pages. DOI: https://doi.org/10.1145/2677855.2677914

[51] Magdalena Szumilas. 2010. Explaining odds ratios. *Journal of the Canadian Academy of Child and Adolescent Psychiatry* 19, 3 (2010), 227–229.

[52] Sotirios Tsimikas, Ewa Karwatowska-Prokopczuk, Ioanna Gouni-Berthold, Jean-Claude Tardif, Seth J. Baum, Elizabeth Steinhagen-Thiessen, Michael D. Shapiro, Erik S. Stroes, Patrick M. Moriarty, and Børge G. Nordestgaard. 2020. Lipoprotein (a) reduction in persons with cardiovascular disease. 382, 3 (2020), 244–255.

[53] Takeaki Uno, Masashi Kiyomi, and Hiroki Arimura. 2004. LCM ver. 2: Efficient mining algorithms for frequent/closed/maximal itemsets. In *Proceedings of the IEEE ICDM Workshop on Frequent Itemset Mining Implementations*.

[54] Ashley J. Vargas, Sheri D. Schully, Jennifer Villani, Luis Ganoza Caballero, and David M. Murray. 2019. Assessment of prevention research measuring leading risk factors and causes of mortality and disability supported by the US national institutes of health. *JAMA Network Open* 2, 11 (2019), e1914718–e1914718. DOI: https://doi.org/10.1001/jamanetworkopen.2019.14718

[55] Xiaoting Wang, Christopher Leckie, Hairuo Xie, and Tharshan Vaithianathan. 2015. Discovering the impact of urban traffic interventions using contrast mining on vehicle trajectory data. In *Proceedings of the Advances in Knowledge Discovery and Data Mining*. Springer International Publishing, 486–497.

[56] Lijun Xu and Kanglin Xie. 2005. An incremental algorithm for mining generators representation. In *Proceedings of the 9th European Conference on European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Springer-Verlag, Berlin, 701–708. DOI: https://doi.org/10.1007/11564126_75

[57] Yuan Yuan, Sihong Xie, Chun-Ta Lu, Jie Tang, and Philip S. Yu. 2016. Interpretable and Effective Opinion Spam Detection Via Temporal Patterns Mining Across Websites. In *Proceedings of the 2016 IEEE International Conference on Big Data*, 96–105 pages. DOI: https://doi.org/10.1109/BigData.2016.7840593

[58] Unil Yun, Hyoju Nam, Gangin Lee, and Eunchul Yoon. 2019. Efficient approach for incremental high utility pattern mining with indexed list structure. *Future Generation Computer Systems* 95 (2019), 221–239. DOI: https://doi.org/10.1016/j.future.2018.12.029

Received March 2021; revised July 2021; accepted September 2021