IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication December 1, 2024, date of current version December 23, 2024

Design and Analysis of Algorithms for Top-K High-Utility Itemsets from Unstable Databases with Both Positive and Negative Utilities (December 2024)

Dang Nguyen. Le, Fellow, IEEE, Gia Huy. Vo, and Ngoc Tra My. Vu, Member, IEEE

Department of Computer Science, Faculty of Information Technology, Ton Duc Thang University, Ho Chi Minh City, Vietnam
Corresponding author: Dang Nguyen. Le (e-mail: 522k0020@student.tdtu.edu.vn).
This work was supported in part by the Vietnam Department of Computer Science.

ABSTRACT High-utility itemset mining (HUIM) plays a crucial role in data mining by uncovering valuable patterns within transaction databases. This process goes beyond traditional frequent itemset mining by considering both the quantity of items and their associated utilities, including scenarios with positive and negative utilities. Such scenarios frequently arise in practical applications, such as market basket analysis, where bundled promotions or discounts affect the profitability of individual items. However, the absence of the downward closure property and the vast search space pose significant challenges for efficient mining. To address these issues, this study introduces advanced pruning strategies and novel data structures, including utility-driven tree and list models. These structures ensure compact representation of transaction data and facilitate efficient search space reduction. Furthermore, the proposed algorithms leverage enhanced upper-bound techniques to minimize redundant computations while maintaining accuracy. Experimental evaluations on diverse datasets, both synthetic and real-world, demonstrate the significant advantages of these approaches in terms of memory efficiency and computational performance compared to state-of-the-art algorithms. These results highlight the potential applications of HUIM in various domains, such as retail decision-making, cross-marketing strategies, and dynamic inventory management. The proposed methods provide a practical framework for discovering actionable insights from large-scale transaction data, offering substantial contributions to the advancement of utility-based data mining research.

INDEX TERMS Data mining, frequent itemset mining, high-utility itemset mining, memory efficiency, negative utility, pattern discovery, pruning strategies, transaction databases, utility-driven algorithms.

CONTENTS

I Introduction **1**
A Background . . . 1
B Research Motivation . . . 1

II Literature Review **2**
A Mining Periodic High-Utility Itemsets with Both Positive and Negative Utilities by Lai et al. [1] . . . 2
B ITUFP: A Fast Method for Interactive Mining of Top-K Frequent Patterns from Uncertain Data by Razieh Davashi [2] . . . 2
C An Efficient Method for Mining High-Utility Itemsets from Unstable Negative Profit Databases by Tung et al. [3] . . . 3
D Discovering High Utility-Occupancy Patterns from Uncertain Data by Chien-Ming Chen et al. [4] . . . 3
E A Generic Algorithm for Top-K On-Shelf Utility Mining by Jiahui Chen et al. [5] . . . 3
F Compact Tree Structures for Mining High Utility Itemsets by Anup Bhat et al. [6] . . . 4
G A Survey on High Utility Itemsets Mining by Ninoria and Thakur [7] . . . 4
H An Evolutionary Model to Mine High Expected Utility Patterns from Uncertain Databases by Ahmed et al. [8] . . . 5
I Compact Tree Structures for Mining High Utility Itemsets by Bhat et al. [9] . . . 5
J C-HUIM: A Novel Framework for Customized High-Utility Itemset Mining by Sagare and Kodavade [10] . . . 5
K Incremental High Average-Utility Itemset Mining: Survey and Challenges by Jing Chen et al. [11] . . . 6
L Mining Frequent Itemsets from Streams of Uncertain Data by Leung and Hao [12] . . . 6
M Mining Periodic High-Utility Itemsets with Both Positive and Negative Utilities by Lai et al. [13] . . . 6
N UGMINE: Utility-Based Graph Mining by Alam et al. [14] . . . 7
O Mining Statistically Significant Patterns with High Utility by Tang et al. [15] . . . 7
P Targeted Mining of Top-k High Utility Itemsets by Huang et al. [16] . . . 8
Q Temporal Fuzzy Utility Maximization with Remaining Measure by Wan et al. [17] . . . 8
R Heuristically Mining the Top-k High-Utility Itemsets with Cross-Entropy Optimization by Song et al. [18] . . . 9
S Top-k High Utility Itemset Mining: Current Status and Future Directions by Kumar and Singh [19] 9
T Towards Target High-Utility Itemsets by Miao et al. [20] . . . 10
U High Utility Periodic Frequent Pattern Mining in Multiple Sequences by Chen et al. [21] . . . 10

III Problem's statements and related definitions **11**
A Problem's statements . . . 11
B Related definitions . . . 11
B.1 Objective . . . 12

IV Pseudocodes of Algorithms **13**
A Algorithm 1: High Utility Pattern Mining with Transaction Probability (HUPT) . . . 13
B Algorithm 2: Revised Fast Tree (RFT) Algorithm for High-Utility Top-k Mining . . . 16
C Algorithm 3: Pattern Tree Remaining Utility (PTRU) Algorithm . . . 18
D Algorithm 4: Trie-based Top-K Utility with UBU pruning (TTUBU) . . . 20
E Algorithm 5: PU List Search Algorithm (PUT) . . . 22

V Asymptotic Complexity Analysis **24**
A HUPT (High-Utility Pattern Tree) Algorithm . . . 24
B RFT (Revised Fast Tree) Algorithm . . . 25
C PTRU (Pattern Tree Remaining Utility) Algorithm . . . 25
D TTUBU (Trie-based Top Utility Bound Utility) . . . 26
E PUT (PU List Search) Algorithm . . . 27

VI Experiment Design **29**
A Experiment Design for HUPT (High-Utility Pattern Tree Algorithm) . . . 29
B Experiment Design for RFT (Revised Fast Tree) . . . 29

C Experiment Design for PTRU (Pattern Tree Remaining Utility) . . . 29
D Experiment Design for TTUBU (Trie-based Top Utility Bound Utility) . . . 30
E Experiment Design for PUT (PU List Search) . . . 30

VII Experiment Results and Analysis **32**
A Results for Retail Dataset . . . 32
B Results for Accident Dataset . . . 33
C Results for Kosarak Dataset . . . 34
D Results for Mushroom Dataset . . . 35
E Overall Conclusion . . . 36

VIII Question and Answering Session **38**
A Question 1: What is the main objective of the project? . . . 38
B Question 2: How are the algorithms implemented? . . . 38
C Question 3: What datasets were used in the experiments? . . . 38
D Question 4: How were the experiments conducted? . . . 38
E Question 5: What performance metrics were used for evaluation? . . . 38
F Question 6: What were the key findings from the experiments? . . . 38
G Question 7: What challenges were encountered during the project? . . . 38
H Question 8: How were the challenges addressed? . . . 39
I Question 9: How do the algorithms compare overall? . . . 39
J Question 10: What are the future directions for this project? . . . 39

IX Conclusion **40**

Authors **45**

LIST OF FIGURES

VII.1Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Retail Dataset. 32
VII.2Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Accident Dataset. 33
VII.3Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Kosarak Dataset. 34
VII.4Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Mushroom Dataset. 35

I. INTRODUCTION

A. BACKGROUND

Frequent pattern mining is a fundamental task in data mining that discovers itemsets appearing frequently in a transaction database [22]. While traditional algorithms like Apriori [23] and FP-Growth [24] only consider occurrence frequency, high-utility itemset mining (HUIM) was introduced to consider both quantity and utility of items [25], enabling the discovery of profitable patterns [26]. However, HUIM becomes challenging when dealing with uncertain databases, where item presence is probabilistic [27]. Uncertain data arises in scenarios such as sensor networks and privacy-preserving data publishing [28]. Moreover, items can have both positive and negative utilities, representing profits and costs [29]. Ignoring negative utilities can lead to biased results [30].

Recent research has focused on mining high-utility itemsets from uncertain databases with both positive and negative utilities [31]. However, most approaches aim to find all high-utility itemsets above a minimum utility threshold, which can be computationally expensive [32]. In practice, decision-makers often prefer a concise set of the top-k high-utility itemsets [33].

In this paper, we propose efficient algorithms for mining top-k high-utility itemsets from uncertain databases with positive and negative utilities. Our algorithms incorporate novel pruning strategies, utility-list structures, and probability-aware techniques to effectively reduce the search space and efficiently compute the expected utilities of itemsets.

B. RESEARCH MOTIVATION

Despite the progress made in mining high-utility itemsets from uncertain databases, several limitations and challenges remain. Existing approaches often rely on a minimum utility threshold to filter out low-utility itemsets, which can be difficult to set appropriately [34]. Setting the threshold too high may miss important patterns, while setting it too low may generate an overwhelming number of itemsets [35]. Moreover, most algorithms focus on finding all high-utility itemsets, which can be computationally expensive and impractical for large databases [36].

Real-world applications, such as market basket analysis, product recommendation, and risk management, often require the discovery of the most valuable or top-k high-utility itemsets [33]. For example, in cross-selling strategies, identifying the top-k high-utility product combinations can help retailers optimize their product placement and promotional offers [37]. In fraud detection, discovering the top-k high-utility patterns can assist in identifying the most significant fraud cases for investigation [38].

However, mining top-k high-utility itemsets from uncertain databases with both positive and negative utilities poses several challenges. The probabilistic nature of uncertain data requires the computation of expected utilities, which can be complex and time-consuming [39]. The presence of negative utilities further complicates the mining process, as traditional pruning strategies based on positive utilities may not be directly applicable [29]. To address these challenges and meet the demands of real-world applications, there is a pressing need for efficient algorithms that can effectively mine top-k high-utility itemsets from uncertain databases with both positive and negative utilities [31]. Such algorithms should be able to handle large - scale databases, efficiently compute expected utilities, and incorporate effective pruning strategies to reduce the search space and improve performance [28].

In this paper, we propose a suite of efficient algorithms, namely HUPT, RFT, PTRU, TTUBU, and PUT, for mining top-k high-utility itemsets from uncertain databases with positive and negative item utilities.

- **HUPT (High Utility Pattern Mining with Transaction Probability)**: A pattern growth approach integrating probability-based utility computation.
- **RFT (Revised Fast Tree)**: A tree-based algorithm leveraging hierarchical storage for efficient pattern generation.
- **PTRU (Pattern Tree Remaining Utility)**: A method combining pattern tree structures with remaining utility calculations for flexible and memory-efficient mining.
- **TTUBU (Trie Tree Utility-Based Uncertain Miner)**: A Trie Tree-based approach employing UBU pruning and prefix-sharing to enhance scalability.
- **PUT (Probability Utility Threshold)**: An algorithm focused on utility thresholds and probabilistic pruning strategies.

Our algorithms leverage novel data structures, such as utility-lists and prefix-trees, to efficiently compute expected utilities and prune the search space. We also introduce probability-aware pruning techniques to further optimize the mining process. The proposed algorithms aim to provide a balance between efficiency and effectiveness, enabling the discovery of the most valuable itemsets while considering the uncertainty and dual utility nature of the data.

II. LITERATURE REVIEW

High-Utility Itemset Mining (HUIM) has emerged as a vital research area in data mining, focusing on identifying patterns that not only occur frequently but also contribute significantly to the overall utility in transactional databases. Unlike traditional frequent itemset mining, HUIM considers both the quantity and utility of items, offering insights that are more actionable in various real-world applications, such as retail analytics, inventory management, and decision-making processes. However, the lack of downward closure property, the complexity of mixed utilities (positive and negative), and the challenges posed by uncertain or dynamic data significantly complicate the mining process.

In recent years, researchers have proposed numerous algorithms, data structures, and frameworks to address these challenges. These advancements range from tree-based structures for efficient search space representation to dynamic pruning strategies and evolutionary models for mining high-utility patterns in uncertain and periodic scenarios. Additionally, techniques for incremental mining, targeted mining, and statistical significance analysis have expanded the scope of HUIM to diverse datasets and application domains.

This section reviews key contributions to HUIM, including methodologies for mining high-utility itemsets in databases with both positive and negative utilities, interactive mining of Top-K patterns, and utility-based mining in uncertain and temporal datasets. The selected works provide a comprehensive view of the evolution of HUIM, emphasizing novel algorithms, frameworks, and strategies for addressing computational and practical challenges.

The following subsections discuss each paper in detail, summarizing their approaches, key contributions, and relevance to the field.

A. MINING PERIODIC HIGH-UTILITY ITEMSETS WITH BOTH POSITIVE AND NEGATIVE UTILITIES BY LAI ET AL. [1]

Pattern mining has been a fundamental research area in data mining, with early algorithms like Apriori [40] and FP-Growth [41] focusing on discovering frequent patterns. However, these traditional approaches only considered binary occurrence and ignored quantitative values and utilities of items. This limitation led to the development of High-Utility Itemset Mining (HUIM), which incorporates both quantities and utilities into the mining process [42].

Chan et al. [43] introduced the concept of utility mining in 2003, proposing a level-wise algorithm for mining high-utility patterns. Liu et al. [44] later developed the Two-Phase algorithm that introduced Transaction-Weighted Utilization (TWU) for pruning the search space. Several tree-based algorithms were subsequently proposed, including IHUP [45], UP-Growth [46], and its enhancement UP-Growth+ [47], which improved mining efficiency through various pruning strategies.

A significant advancement came with HUI-Miner [48], which introduced a list-based structure to avoid multiple database scans. Fournier-Viger et al. [49] further enhanced this approach with the FHM algorithm, introducing the EUCP (Estimated Utility Co-occurrence Pruning) strategy. The EFIM algorithm [50] combined projected database and transaction merging techniques to achieve better performance.

An important extension to HUIM is handling negative utility values, which is crucial for real-world applications. Tseng et al. [51] proposed an efficient algorithm for mining high-utility itemsets with negative item values. FHN [52] and HUPM-N [53] further advanced this area by introducing specialized data structures and pruning strategies for handling negative utilities.

Recent research has focused on developing hybrid frameworks and incorporating various constraints. The UFH algorithm [54] proposed a hybrid framework combining tree-based and inverted-list-based approaches. Additionally, algorithms like TopHUI [55] have been developed to handle top-k high-utility itemset mining with negative utilities more efficiently.

B. ITUFP: A FAST METHOD FOR INTERACTIVE MINING OF TOP-K FREQUENT PATTERNS FROM UNCERTAIN DATA BY RAZIEH DAVASHI [2]

Pattern mining has emerged as a fundamental field in data mining research, with early algorithms like Apriori [40] and FP-Growth [41] focusing on discovering frequent patterns. However, real-world data often contains uncertainty, leading to the development of methods specifically designed for uncertain data mining. These approaches need to handle both the presence/absence uncertainty of items and their utility values [56].

The introduction of uncertainty in pattern mining posed new challenges that traditional algorithms could not address. Initial solutions like U-Apriori [56] extended the Apriori approach to handle uncertain data, but suffered from candidate generation and multiple database scans. To overcome these limitations, tree-based approaches like UF-growth [57] were proposed, which could process uncertain data more efficiently. However, these methods still faced challenges with large tree structures and memory requirements.

A significant advancement came with list-based structures for mining uncertain patterns. The LUNA algorithm [58] introduced an efficient approach using list structures to avoid multiple database scans. The TUFP algorithm [59] further improved this by introducing strategies for mining top-k uncertain frequent patterns. However, these approaches still required rebuilding their data structures when parameters changed, making them less suitable for interactive environments.

The incorporation of negative utilities added another

dimension of complexity to uncertain pattern mining. Tseng et al. [51] proposed an algorithm specifically designed to handle negative utility values, while maintaining efficient mining capabilities. Later approaches like FHN [52] and HUPM-N [53] introduced specialized data structures and pruning strategies for handling negative utilities more effectively.

C. AN EFFICIENT METHOD FOR MINING HIGH-UTILITY ITEMSETS FROM UNSTABLE NEGATIVE PROFIT DATABASES BY TUNG ET AL. [3]

Frequent Itemset Mining (FIM) algorithms like Apriori [40] and FP-Growth [41] have been foundational in pattern mining research. However, traditional FIM only considers binary occurrence, ignoring quantitative values and utilities that are crucial in real-world applications. High-Utility Itemset Mining (HUIM) was introduced to address this limitation by incorporating both quantities and utilities into the mining process [43]. Early HUIM algorithms like Two-Phase [**?**] pioneered the use of Transaction Weighted Utilization (TWU) for pruning, while later approaches like HUI-Miner [48] improved efficiency through list-based structures.

A significant challenge in HUIM is managing negative utility values, which commonly arise in retail scenarios through discounts or promotions. The HUINIV-Mine algorithm [60] was among the first to address this, though it suffered from excessive candidate generation. More efficient approaches were later developed, including FHN [52] and EHIN [61], which introduced specialized data structures and pruning strategies for negative utilities. The EHMIN algorithm [62] further improved performance through list-based processing.

While existing algorithms can handle negative utilities, they typically assume constant utility values across transactions. This assumption fails in real-world scenarios where an item's utility may fluctuate between positive and negative values depending on sales context or promotional strategies. Current approaches like EHIN and EHMIN cannot effectively process such unstable profit patterns, leading to either missed high-utility itemsets or computational inefficiencies.

Recent research has expanded HUIM to address various practical scenarios. This includes algorithms for top-k high utility pattern mining [63], handling uncertain data [8], and processing dynamic databases [64]. However, the challenge of efficiently mining high-utility itemsets from databases with unstable negative profits remains largely unaddressed in the literature.

D. DISCOVERING HIGH UTILITY-OCCUPANCY PATTERNS FROM UNCERTAIN DATA BY CHIEN-MING CHEN ET AL. [4]

High-Utility Itemset Mining (HUIM) extends traditional Frequent Itemset Mining (FIM) by incorporating utility measures, such as profit, importance, and weight, to discover patterns with high utility contributions in transactional databases. Foundational algorithms, such as *Apriori* and *FP-Growth*, effectively leverage the anti-monotonic property of support for efficient candidate pruning but do not consider utility measures, which limits their applicability in real-world scenarios. HUIM introduced algorithms like *Two-Phase*, *HUI-Miner*, and *HUP-Growth* to address these challenges by employing Transaction Weighted Utility (TWU) and utility list structures to prune the search space, enabling more efficient pattern mining. These methods identify patterns exceeding a predefined utility threshold, offering applications in domains like retail, marketing, and recommendation systems.

Incorporating negative utilities into HUIM addresses real-world complexities such as discounts, losses, and negative profits. Algorithms like *HUINIV-Mine* and *FHN* introduced strategies to handle negative utilities by partitioning patterns and applying advanced pruning techniques [51]. Despite these advancements, existing methods assume stable utility contributions, limiting their applicability for databases with fluctuating profits.

Modern databases often exhibit *uncertainty* due to noise or incomplete information, making traditional deterministic mining approaches insufficient. In this context, the work by Chen et al. (2020) proposes the **UHUOPM (High Utility-Occupancy Pattern Mining in Uncertain Databases)** algorithm, which integrates utility occupancy, probability, and frequency measures to mine Potential High Utility-Occupancy Patterns (PHUOPs). The authors address the challenges of uncertainty in data by introducing innovative data structures, such as the Probability-Utility-Occupancy List (PUO-List) and the Probability-Frequency-Utility Table (PFU-Table). These structures enable the algorithm to efficiently prune the search space and reduce memory consumption while ensuring robust pattern discovery.

The UHUOPM algorithm introduces a Support-Count Tree (SC-Tree) to navigate the search space effectively, employing pruning strategies based on utility occupancy and probability thresholds. Experiments conducted on real-life and synthetic datasets demonstrate the algorithm's superiority in terms of runtime, memory usage, and pattern discovery compared to state-of-the-art methods like *HUOPM*. This makes the UHUOPM algorithm highly suitable for applications in uncertain data environments, such as IoT-based systems and financial transaction analysis.

E. A GENERIC ALGORITHM FOR TOP-K ON-SHELF UTILITY MINING BY JIAHUI CHEN ET AL. [5]

High Utility Itemset Mining (HUIM) extends traditional Frequent Itemset Mining (FIM) by incorporating utility measures such as profit, importance, and weight to identify patterns with high contributions. Foundational

algorithms, such as *Apriori* and *FP-Growth*, are efficient in frequent itemset mining but fail to consider utility measures, which limits their real-world applicability. HUIM introduced utility-based measures and algorithms, including *Two-Phase* [44], *HUI-Miner* [48], and *UP-Growth* [65], which employ Transaction Weighted Utility (TWU) and utility-list structures to prune the search space and optimize mining efficiency. These methods ensure that discovered patterns exceed user-defined utility thresholds, making them highly relevant in domains like retail, marketing, and resource optimization.

To address limitations in traditional HUIM methods, such as overlooking item availability periods, On-Shelf Utility Mining (OSUM) was introduced. OSUM incorporates time periods into utility calculations, ensuring that itemsets with high utility during specific selling periods are identified. Algorithms like *TP-HOUN* [66], *OSUMI* [67], and *OHUQI* [68] advanced OSUM by handling both positive and negative utilities and introducing more efficient data structures and pruning strategies. However, these algorithms still rely on user-defined minimum utility thresholds (*minUtil*), which can be challenging to set correctly, especially in varying database densities.

Top-K Utility Mining addresses the issue of setting minimum utility thresholds by allowing users to specify K, the number of desired high-utility patterns. Existing algorithms, such as *TKU* and *TKO* [69], and *REPT* [70], adopt techniques like utility-lists and tree structures for efficient pattern discovery. However, challenges remain, including memory overhead and computational inefficiencies when handling datasets with negative utilities.

To overcome these limitations, the authors of this paper propose **TOIT**, a generic algorithm for Top-K On-Shelf Utility Mining. TOIT introduces novel strategies like Subtree Utility and Local Utility to prune the search space, and Real Item Utility (RIU) to dynamically adjust the utility threshold during mining. Experiments on real-world datasets demonstrate the superior performance of TOIT in terms of runtime and memory efficiency compared to state-of-the-art algorithms such as KOSHU [71]. This makes TOIT a robust solution for mining high-utility patterns in dynamic and time-sensitive domains.

F. COMPACT TREE STRUCTURES FOR MINING HIGH UTILITY ITEMSETS BY ANUP BHAT ET AL. [6]

High Utility Itemset Mining (HUIM) extends traditional Frequent Itemset Mining (FIM) by incorporating utility measures such as profit, importance, and weight, to identify patterns with high contributions. Foundational algorithms, such as *Apriori* [40] and *FP-Growth* [41], efficiently mine frequent itemsets using the anti-monotonic property of support. However, these methods do not consider utility measures like quantities and unit profit, limiting their real-world applicability. HUIM addresses this limitation by introducing utility-based measures.

Algorithms like *Two-Phase* [44] and *HUP-Growth* [72] efficiently prune the search space using Transaction Weighted Utility (TWU) and utility-lists, ensuring the discovery of patterns exceeding a predefined utility threshold. These methods have been widely applied in domains such as retail, inventory management, and recommendation systems.

Tree-based methods offer compact and efficient representations of transactional data, enabling recursive mining operations. Notable examples include the *IHUP* algorithm [45], which avoids unpromising item elimination by constructing trees without discarding any items. However, IHUP and its variants suffer from reordering overhead and memory inefficiencies.

The *UP-Growth* and *UP-Growth+* algorithms improved tree-based methods by introducing strategies to prune unpromising nodes and reduce the search space during recursive mining. Nevertheless, these methods require costly tree reconstructions when the utility threshold (*minUtil*) changes [63].

In this paper, Bhat et al. propose novel tree structures—*Utility Prime Tree (UPT)*, *Prime Cantor Function Tree (PCFT)*, and *String-based Utility Prime Tree (SUPT)*—to address the challenges of memory efficiency and tree reconstruction. Unlike traditional item-based prefix trees, the proposed structures compactly encode transaction-level information in a single node, ensuring higher abstraction and reduced redundancy.

- **Utility Count Tree (UCT)**: Ensures prefix-sharing by constructing a node for every item in a transaction while accumulating utility values.
- **Utility Prime Tree (UPT)**: Utilizes prime numbers to compactly encode item and utility information, enabling faster traversal and storage efficiency.
- **Prime Cantor Function Tree (PCFT)**: Applies Cantor Function mapping to resolve utility and item combinations while maintaining completeness.
- **String-based Utility Prime Tree (SUPT)**: Reduces large numerical encodings by concatenating prime numbers as strings, enhancing prefix-sharing and memory efficiency.

Experiments conducted on real and synthetic datasets demonstrate that UCT outperforms existing methods such as *IHUP* and *UP-Growth* in execution time, while SUPT achieves superior memory efficiency, especially in sparse datasets. These results highlight the scalability and applicability of the proposed tree structures in large transactional databases.

G. A SURVEY ON HIGH UTILITY ITEMSETS MINING BY NINORIA AND THAKUR [7]

This survey provides a comprehensive overview of High Utility Itemset Mining (HUIM), which extends Frequent Itemset Mining (FIM) by incorporating utility measures such as profit, cost, or user preferences. Unlike FIM,

HUIM identifies itemsets that provide higher utility rather than focusing solely on frequency.

The paper reviews seminal HUIM techniques, from foundational approaches like Two-Phase [44] to advancements like UP-Growth and HUI-Miner, which enhance efficiency and scalability. It discusses key challenges, including pruning strategies for non-downward-closed properties and mining dynamic or incremental datasets. The survey highlights recent trends, such as constraint-based mining and applications in diverse domains, providing insights into the advantages and limitations of current methodologies. This work serves as a valuable reference for understanding the evolution of HUIM and its applications.

H. AN EVOLUTIONARY MODEL TO MINE HIGH EXPECTED UTILITY PATTERNS FROM UNCERTAIN DATABASES BY AHMED ET AL. [8]

This paper addresses the challenge of mining high expected utility patterns (HEUPs) in uncertain environments, a critical issue in the era of IoT and big data. Traditional HUIM algorithms often fail to account for data uncertainty, leading to unreliable patterns. This study introduces a Multi-Objective Evolutionary Approach (MOEA-HEUPM) that simultaneously considers utility and uncertainty to discover meaningful patterns without predefined thresholds.

The proposed MOEA-HEUPM utilizes evolutionary computation to optimize multiple objectives, addressing trade-offs between utility and uncertainty. Key contributions include two encoding methodologies for effective pattern representation and robust performance across varied datasets. Experimental results demonstrate the model's superiority in convergence and computational efficiency compared to existing methods. This approach paves the way for robust decision-making in uncertain data environments.

I. COMPACT TREE STRUCTURES FOR MINING HIGH UTILITY ITEMSETS BY BHAT ET AL. [9]

The field of Frequent Itemset Mining (FIM) began with foundational methods like Apriori [40] and FP-Growth [41], which focused on identifying patterns based on item frequency. While these methods advanced the analysis of transactional data, they failed to account for item utility, leading to the emergence of High Utility Itemset Mining (HUIM). HUIM, with early models like Two-Phase [**?**] and UP-Growth [65], incorporated utility metrics like profit and cost, addressing the limitations of FIM's binary treatment of item importance.

Tree-based approaches, such as IHUP and UP-Growth, enhanced memory efficiency and reduced computational overhead but required reconstruction for threshold changes. This study builds on these advancements by introducing Utility Prime Tree (UPT), Prime Cantor Function Tree (PCFT), and String-based Utility Prime Tree (SUPT). These structures optimize data representation by encoding transaction-level information compactly, reducing memory usage and enabling dynamic adjustments without tree reconstruction. The proposed methodologies mark a significant step forward in efficient and scalable HUIM.

J. C-HUIM: A NOVEL FRAMEWORK FOR CUSTOMIZED HIGH-UTILITY ITEMSET MINING BY SAGARE AND KODAVADE [10]

This paper presents a novel framework, C-HUIM (Customized High-Utility Itemset Mining), designed to address the limitations in neglecting item importance and quantity [40], [41] of traditional High-Utility Itemset Mining (HUIM) techniques, which often lack the flexibility to adapt to custom constraints, such as time windows or correlation thresholds. The framework introduces the C-HUIM algorithm, a powerful tool for mining high-utility itemsets tailored to user-defined constraints.

Key contributions of the paper include:

- **Customizable Mining Framework:** The proposed framework allows users to specify constraints, such as time windows or correlation thresholds, to discover high-utility itemsets that meet specific needs. This flexibility makes it applicable to a wide range of real-world datasets and business scenarios.
- **C-HUIM Algorithm:** The C-HUIM algorithm incorporates innovative preprocessing techniques, such as customized data structure creation and parameter-based pruning strategies, to efficiently mine customized high-utility itemsets (C-HUIs).
- **Time-Window and Correlation Modules:** Two key modules, a time-window constraint module and a correlation module, are introduced to refine the mining process. These modules ensure that mined itemsets not only have high utility but also satisfy temporal and relational requirements.
- **Experimental Validation:** The authors evaluate the C-HUIM framework against the LHUI-Miner algorithm, showing improved performance in terms of runtime and the number of high-utility itemsets discovered. The evaluation uses real-world datasets, including retail and e-commerce, to validate the algorithm's effectiveness.
- **Practical Applications:** The framework demonstrates significant potential in fields like retail analytics and inventory management, where decision-making often requires customized analysis of transactional data.

This study contributes a robust and flexible approach to HUIM, addressing critical gaps in adaptability and user-specific constraints. Future work could extend the framework to handle dynamic datasets, explore parallel computing techniques, and integrate additional customization options for broader applicability.

K. INCREMENTAL HIGH AVERAGE-UTILITY ITEMSET MINING: SURVEY AND CHALLENGES BY JING CHEN ET AL. [11]

This paper provides a comprehensive survey on Incremental High Average-Utility Itemset Mining (iHAUIM), a variation of High Utility Itemset Mining (HUIM) that focuses on the average utility of itemsets within dynamically updated databases. The study emphasizes the importance of incremental mining approaches in addressing the limitations of static algorithms, particularly for applications in dynamic environments like market basket analysis, streaming data, and real-time analytics.

Key contributions of the paper include:

- **Apriori-Based Algorithms:** These use breadth-first search and frequent pattern upper bounds for candidate pruning.
- **Tree-Based Algorithms:** These employ hierarchical structures for efficient data representation and pattern discovery.
- **Utility-List-Based Algorithms:** These optimize memory usage and processing efficiency by storing itemset-specific utilities.

This survey offers a thorough evaluation of state-of-the-art iHAUIM methods, providing valuable insights into their practical applications and limitations. Algorithms such as FUP [11] introduced incremental updates, while tree-based models like IHAUI-Tree emphasized compact data structures for scalability. Despite these advancements, the paper serves as a foundation for future advancements in utility mining, with a focus on developing scalable and efficient algorithms for dynamic environments.

L. MINING FREQUENT ITEMSETS FROM STREAMS OF UNCERTAIN DATA BY LEUNG AND HAO [12]

This paper focuses on the challenge of mining frequent itemsets from data streams of uncertain data. Traditional frequent itemset mining techniques handle static databases with precise data, but in many real-world scenarios, transaction data is uncertain, and streams of data flow continuously. The need to handle uncertain data led to the development of U-Apriori [73] and UF-Growth [74], which incorporated existential probabilities into the mining process. The authors propose two tree-based algorithms, UF-streaming and SUF-growth, to address this gap, combining methods for managing data streams and uncertain data.

Key contributions of the paper include:

- **UF-Streaming Algorithm:** An approximate algorithm that uses a UF-tree structure to store and manage the streaming data. It employs a lower threshold, *preMinsup*, to prevent premature pruning of potentially frequent itemsets.
- **SUF-Growth Algorithm:** An exact algorithm utilizing a SUF-tree structure, designed for delayed mining. Unlike UF-streaming, it avoids false positives and negatives by maintaining only truly frequent itemsets based on a specified *minsup*.
- **Efficient Data Structures:** The UF-tree and SUF-tree structures enable compact representation and efficient updating of frequent itemsets in uncertain and streaming environments.
- **Experimental Validation:** Experiments on synthetic and real-world datasets demonstrate the efficiency of the proposed algorithms in terms of runtime and scalability. The SUF-growth algorithm is particularly effective for delayed mining in large datasets, while UF-streaming responds quickly to user requests for frequent patterns.
- **Handling Data Properties:** The algorithms address challenges of unbounded and non-uniform data streams by focusing on sliding windows and maintaining recent data for mining.

This paper is significant for bridging the gap between stream mining and uncertain data mining. It highlights practical applications in domains such as real-time analytics and IoT, where both uncertainty and data volume are critical considerations. Future work could explore distributed implementations of the proposed methods to further enhance scalability.

M. MINING PERIODIC HIGH-UTILITY ITEMSETS WITH BOTH POSITIVE AND NEGATIVE UTILITIES BY LAI ET AL. [13]

This paper addresses the novel challenge of mining periodic high-utility patterns from transactional databases containing both positive and negative utilities. In real-world scenarios, such as market basket analysis, mixed utilities often arise from marketing strategies like bundling, where some items may have negative profits while others yield positive gains. This situation presents unique computational challenges for traditional utility mining methods. The need to handle uncertain data led to the development of [73] and UF-Growth [74], which incorporated existential probabilities into the mining process.

Key contributions of the paper include:

- **PHMN Algorithm:** The authors propose the PHMN algorithm, which is based on a vertical data structure and effectively discovers periodic high-utility patterns (PHUPs) in mixed utility databases. This algorithm is the first to address periodic high-utility pattern mining in such a context.
- **Pruning Strategies:** A novel upper-bound pruning technique is introduced to reduce the search space. The authors also improve efficiency by proposing PHMN+, which incorporates the Dynamical Upper (DU) bound and pruning strategies for further optimization.
- **Experimental Validation:** The performance of PHMN and PHMN+ is evaluated across four datasets with varying characteristics. Results show significant improvements in execution time and memory

utilization compared to existing methods, demonstrating the algorithms' scalability and practical applicability.

This study represents a significant advancement in utility mining by integrating periodicity and handling both positive and negative utilities. The paper highlights the relevance of periodic mining in identifying stable, long-term patterns, which are crucial for business strategies and inventory management. However, databases often include negative utilities, such as discounted or bundled items, requiring methods like FHN [52] and HUPNU [75] to address both positive and negative utilities. Additionally, periodic pattern mining, such as PF-tree [76], advanced the field by identifying patterns with recurring behavior. The combination of these approaches to mine periodic high-utility patterns in mixed utility contexts remains an underexplored area, which this study addresses with the PHMN algorithm. This work contributes by proposing novel upper-bound pruning strategies and experimentally validating the method's efficiency. Future research directions could explore dynamic data streams and real-time periodic high-utility mining for IoT applications.

N. UGMINE: UTILITY-BASED GRAPH MINING BY ALAM ET AL. [14]

This paper introduces a comprehensive framework and algorithm, UGMINE, for mining high-utility subgraph patterns from graph databases. While traditional frequent graph mining focuses on patterns with high occurrence frequencies, it often fails to capture patterns with higher utility or importance to the user. UGMINE addresses this limitation by integrating both internal and external utilities into the graph mining process, providing a more meaningful analysis of graph-based data.

Key contributions of the paper include:

- **Utility-Based Graph Mining Framework:** A complete framework is proposed for mining subgraphs with high utility, incorporating internal utility (quantitative measures) and external utility (qualitative importance) for both nodes and edges in graphs.
- **UGMINE Algorithm:** The authors develop UGMINE, an algorithm designed to efficiently discover high-utility subgraphs while addressing challenges like subgraph isomorphism and candidate explosion.
- **RMU Pruning Technique:** A novel pruning strategy, Rightmost Utility (RMU), is introduced to eliminate false candidates and reduce the search space. RMU achieves tighter pruning compared to Graph Weighted Utility (GWU), significantly improving runtime performance and scalability.
- **Experimental Evaluation:** Experiments on various datasets, including graph-based datasets from PubChem and NCI1, demonstrate the effectiveness of UGMINE. The results show that RMU pruning reduces the number of candidates and improves runtime by up to 50% compared to GWU pruning.
- **Applications:** The framework is applicable to a wide range of domains, including chemical compound analysis, web access log mining, and social network analysis.

Frequent graph mining methods, such as gSpan [77], efficiently identified subgraph patterns but ignored the utility of graph components. UGMINE proposes a novel framework and algorithm for utility-based subgraph mining, addressing challenges like the lack of downward closure in utility metrics. This study represents a significant advancement in graph mining by incorporating utility measures, enabling the discovery of patterns with real-world relevance. Future directions include exploring larger datasets, integrating dynamic or streaming data, and extending the framework to handle multi-graph environments.

O. MINING STATISTICALLY SIGNIFICANT PATTERNS WITH HIGH UTILITY BY TANG ET AL. [15]

This paper explores the novel problem of mining patterns that are not only high in utility but also statistically significant, termed High Utility and Significant Patterns (HUSPs). Unlike traditional High-Utility Itemset Mining (HUIM), which focuses solely on the utility of patterns, this study integrates statistical significance to ensure the meaningfulness of the discovered patterns. The authors propose a new framework, HUSP-Mining-Test, combining utility-based mining and hypothesis testing to provide mathematically sound results.

Key contributions of the paper include:

- **HUSP-Mining-Test Framework:** The framework consists of two main components:
  1. **HUSP-Mining:** This component identifies HUSP candidates by employing length constraints (minimum and maximum lengths) and utility-based pruning strategies. It efficiently reduces the search space by eliminating insignificant patterns early.
  2. **HUSP-Test:** Using Fisher's exact test and an iterative multiple testing procedure, this component evaluates the statistical significance of HUSP candidates while controlling the Family-Wise Error Rate (FWER).
- **Novel Statistical Techniques:** The authors introduce a dynamic test-level correction method to address the limitations of traditional Bonferroni correction, allowing for more significant patterns to be discovered.
- **Experimental Validation:** Comprehensive experiments on real-world datasets demonstrate that HUSP-Mining-Test significantly outperforms existing algorithms in terms of runtime, memory consumption, and the number of statistically significant patterns identified. The results show that

the proposed framework can reduce redundant patterns and provide statistically robust high-utility patterns.
- **Practical Applications:** The approach is applicable to various domains, such as market analysis, healthcare, and personalized recommendations, where both utility and statistical significance are critical.

This paper represents a significant advancement in the field of utility mining by integrating statistical rigor into the discovery process. Future work could explore extending the method to dynamic and streaming datasets, as well as adapting it for other statistical testing frameworks.

P. TARGETED MINING OF TOP-K HIGH UTILITY ITEMSETS BY HUANG ET AL. [16]

This paper addresses the problem of mining high-utility itemsets (HUIs) with a focus on specific user-defined target patterns. While traditional like HUI-Miner [48] and EFIM [50] remain constrained by predefined utility thresholds. To address this limitation, the authors propose the TMKU (Targeted Mining of Top-k High Utility Itemsets) algorithm, which integrates utility mining with targeted pattern discovery and top-k mining.

Key contributions of the paper include:
- **Formulation of the Targeted Top-k HUI Problem:** The paper introduces the concept of targeted mining of top-k high-utility itemsets (THUIs), allowing users to specify both a target pattern and the desired number of itemsets k. This eliminates the need for predefined utility thresholds and enhances usability.
- **The TMKU Algorithm:** The TMKU algorithm utilizes a trie-based structure (TP-tree) and a dynamic map structure (TopKMap) for efficient storage and retrieval of THUIs. The algorithm dynamically adjusts utility thresholds using two novel threshold-raising strategies: SUR (Sum Utility Raising) and RIU (Real Item Utility).
- **Pruning Strategies:** The algorithm employs multiple pruning strategies to minimize the search space, including utility-based pruning and targeted pruning using a header table. These strategies improve runtime efficiency while ensuring that the discovered patterns meet the user's target criteria.
- **Experimental Validation:** Extensive experiments on real-world and synthetic datasets demonstrate the effectiveness and efficiency of TMKU compared to existing methods. Results show reduced memory usage, faster execution times, and scalability across varying dataset sizes and k-values.
- **Applications and Scalability:** The framework is suitable for applications in market basket analysis, inventory optimization, and personalized recommendations, where targeted insights are critical.

This paper provides a significant advancement in HUIM by integrating top-k mining with targeted pattern discovery, addressing both practical and computational challenges. Future work could extend the TMKU algorithm to dynamic datasets and distributed environments, enhancing its applicability to large-scale, real-time scenarios.

Q. TEMPORAL FUZZY UTILITY MAXIMIZATION WITH REMAINING MEASURE BY WAN ET AL. [17]

This paper introduces a novel framework for temporal fuzzy utility mining, focusing on patterns that incorporate both fuzzy logic and temporal dimensions. Traditional high-utility itemset mining (HUIM) lacks interpretability when applied to temporal datasets, as it does not capture linguistic nuances or temporal variations. To address these limitations, the authors propose the TFUM algorithm, which integrates fuzzy set theory and temporal attributes to enhance pattern discovery.

Key contributions of the paper include:
- **Novel Algorithm - TFUM:** The Temporal Fuzzy Utility Maximization (TFUM) algorithm employs a one-phase mining approach to discover high-temporal fuzzy utility itemsets (HTFUIs). The algorithm uses a revised temporal fuzzy-list structure to store and manage key information, reducing memory usage and computational overhead.
- **Remaining Measure for Pruning:** A new remaining measure is proposed, which provides a tighter and more effective upper bound compared to existing methods. This measure significantly improves pruning efficiency, reducing the search space and runtime costs.
- **Temporal Integration:** By considering temporal aspects in the mining process, the algorithm ensures that patterns are relevant to specific timeframes, enhancing their practical applicability in dynamic datasets.
- **Experimental Evaluation:** Experiments conducted on real-world and synthetic datasets demonstrate that TFUM outperforms state-of-the-art algorithms, such as ATTFUM and FUMT, in terms of runtime, memory efficiency, and scalability. The algorithm is particularly effective for dense datasets with varying temporal distributions.
- **Applications and Impact:** The proposed framework is applicable in various domains, including marketing analysis, traffic management, and real-time data processing, where temporal and fuzzy insights are critical for decision-making.

This paper contributes significantly to the field of utility mining by addressing the challenges of temporal and fuzzy data. The TFUM algorithm provides a scalable and interpretable solution for discovering actionable patterns in complex datasets. Future research directions include extending the framework to handle dynamic streaming data and multi-level fuzzy utilities.

R. HEURISTICALLY MINING THE TOP-K HIGH-UTILITY ITEMSETS WITH CROSS-ENTROPY OPTIMIZATION BY SONG ET AL. [18]

High-Utility Itemset Mining (HUIM) addressed the limitations of traditional Frequent Itemset Mining by incorporating utility measures, with early solutions like Two-Phase [48] and TKU [78] introducing candidate-generation strategies for utility thresholds. However, these methods often suffer from high computational costs due to iterative threshold adjustments. This paper introduces two novel algorithms, TKU-CE and TKU-CE+, for solving the top-k High-Utility Itemset Mining (HUIM) problem using a heuristic-based approach leveraging cross-entropy (CE) optimization. The primary challenge in top-k HUIM is eliminating the need for user-defined minimum utility thresholds, which are difficult to set accurately. The proposed methods tackle this issue by adopting CE as a probabilistic optimization technique to identify top-k patterns iteratively.

Key contributions of the paper include:

- **TKU-CE Algorithm:** This algorithm employs CE optimization to model the top-k HUIM problem as a combinatorial optimization problem. It iteratively updates a probability vector (PV) to refine candidate itemsets, ensuring that high-utility itemsets are discovered efficiently without threshold-raising strategies.
- **Enhanced Algorithm - TKU-CE+:** An improved version of TKU-CE, this algorithm integrates three enhancements:
  1. *Critical Utility Value (CUV):* A pruning strategy to eliminate unpromising itemsets early in the process, reducing the computational burden.
  2. *Sample Refinement:* A mechanism to focus on elite itemsets during iterations, reducing search space dynamically.
  3. *Smoothing Mutation:* A technique to introduce diversity in itemset generation, mitigating the risk of converging to local optima.
- **Experimental Validation:** Experiments conducted on both synthetic and real-world datasets show that TKU-CE+ outperforms existing algorithms like TKU, TKO, and kHMC in execution time, memory consumption, and accuracy. The results demonstrate its scalability and suitability for large datasets.
- **Practical Applications:** The algorithms are applicable in various domains, including retail analytics, inventory management, and personalized recommendations, where mining top-k utility patterns is crucial.

This study represents a significant advancement in top-k HUIM by introducing heuristic optimization to improve computational efficiency and eliminate the need for complex data structures or threshold-raising strategies. Future research could explore adaptive heuristic techniques and extensions for dynamic or streaming databases.

S. TOP-K HIGH UTILITY ITEMSET MINING: CURRENT STATUS AND FUTURE DIRECTIONS BY KUMAR AND SINGH [19]

This paper provides an extensive survey of top-k high utility itemset mining (HUIM), highlighting its importance in the domain of data mining and its real-world applications. The primary goal of top-k HUIM is to eliminate the reliance on predefined utility thresholds, which are often difficult for users to determine, and instead focus on discovering the k most profitable itemsets.

Key contributions of the paper include:

- **Comprehensive Review:** The authors present an in-depth review of over 35 state-of-the-art algorithms in top-k HUIM, categorizing them into tree-based, utility-list-based, and hybrid approaches. The review provides insights into their design principles, strengths, and limitations.
- **Taxonomy of Algorithms:** A taxonomy of top-k HUIM approaches is introduced, including static dataset-based, incremental and data stream-based, and sequential dataset-based methods. This categorization aids in understanding the evolution and diversity of approaches in the field.
- **Comparative Analysis:** The paper offers a detailed comparative analysis of existing algorithms, evaluating their runtime efficiency, memory consumption, scalability, and application domains. It also discusses various threshold-raising strategies and pruning methods that enhance mining efficiency.
- **Future Directions:** The authors identify key challenges and opportunities for future research, including the development of adaptive algorithms for dynamic datasets, integration with deep learning for pattern prediction, and advancements in privacy-preserving utility mining.
- **Applications:** The study emphasizes the practical applications of top-k HUIM in market basket analysis, e-commerce, web usage mining, and bioinformatics, demonstrating its broad relevance across industries.

This paper surveys the advancements in top-k HUIM, categorizing them into tree-based, utility-list-based, and hybrid approaches. By evaluating methods like TKU [78] and REPT [79], the authors present their pros and cons, emphasizing scalability and threshold-raising strategies. This paper provides a valuable resource for both researchers and practitioners by consolidating the state-of-the-art in top-k HUIM and outlining future research directions. Its systematic review and comparative evaluation offer a foundation for further exploration and innovation in this domain.

T. TOWARDS TARGET HIGH-UTILITY ITEMSETS BY MIAO ET AL. [20]

This paper addresses the problem of targeted high-utility itemset mining (THUIM), where users are interested in discovering high-utility itemsets that meet specific target constraints. The authors highlight the limitations of traditional high-utility itemset mining (HUIM) methods, which aim to find all high-utility patterns, resulting in unnecessary computations and memory usage when the user is focused on particular subsets of patterns. To address these challenges, the paper proposes the THUIM algorithm, a targeted and efficient approach for mining high-utility itemsets.

Key contributions of the paper include:

- **Pattern Matching Mechanism:** A novel mechanism is introduced to enable targeted mining, allowing the algorithm to match user-specified target patterns efficiently during the mining process without scanning the database multiple times.
- **Enhanced Utility-List Structure:** The utility-list data structure, extended with a matching mechanism, supports efficient pruning and reduces the search space by focusing only on patterns that meet the target constraints.
- **THUIM Algorithm:** The THUIM algorithm is presented as a single-phase method that integrates the matching mechanism directly into the mining process, avoiding the memory-intensive tree-based approaches seen in previous methods like TargetUM.
- **Experimental Validation:** Extensive experiments conducted on real and synthetic datasets demonstrate that THUIM significantly outperforms TargetUM in terms of runtime, memory consumption, and scalability. The proposed algorithm consistently produces faster results with fewer candidates while maintaining accuracy.
- **Practical Applications:** The framework is suitable for applications in personalized marketing, intelligent search systems, and recommendation engines, where users often specify a subset of items of interest.

This paper provides a significant advancement in utility mining by introducing a focused approach to pattern discovery, addressing both computational and practical challenges. Future work could explore adapting THUIM to dynamic and streaming databases and extending its capabilities to other utility-driven mining tasks.

U. HIGH UTILITY PERIODIC FREQUENT PATTERN MINING IN MULTIPLE SEQUENCES BY CHEN ET AL. [21]

Frequent Pattern Mining (FPM) was extended to sequential and periodic contexts to identify recurring patterns over time. While Sequential Pattern Mining (SPM) algorithms like PrefixSpan [80] addressed the temporal order of events, periodic mining algorithms such as MPFPS [81] explored regularly occurring patterns across sequences. However, these approaches often ignored utility measures.

This paper introduces the MHUPFPS (Mining High-Utility Periodic Frequent Patterns in Multiple Sequences) algorithm, designed to mine high-utility periodic frequent patterns (HUPFPS) across multiple sequences. Existing methods often fail to consider both the utility and periodicity of patterns simultaneously, particularly in multiple sequences where such patterns may hold higher practical significance.

Key contributions of the paper include:

- **MHUPFPS Algorithm:** The proposed algorithm integrates a novel pruning strategy with a newly defined data structure called the HUPFPS-list. This structure efficiently stores transaction, utility, and sequence information to avoid repeated database scans, improving computational efficiency.
- **Novel Measures:** The authors introduce three innovative metrics:
  1. Utility Ratio: Quantifies the percentage contribution of a pattern's utility within a sequence.
  2. Support Ratio: Ensures the frequency of periodic patterns across sequences of varying lengths.
  3. High-Utility Periodic Sequence Ratio: Identifies patterns that maintain high utility and periodicity across multiple sequences.
- **Pruning Strategy and Upper Bound:** A new upper bound, upSeqRa, along with associated pruning properties, is proposed to significantly reduce the search space by eliminating non-promising patterns early.
- **Experimental Validation:** Extensive experiments on real-world and synthetic datasets demonstrate that MHUPFPS outperforms existing algorithms in terms of runtime, memory usage, and the number of patterns discovered. The algorithm is shown to be effective even in dense and large-scale datasets, such as the FIFA and Leviathan datasets.
- **Applications:** The approach is applicable in areas like e-commerce, IoT, and biomedical data analysis, where identifying patterns with periodicity and high utility is crucial.

This study advances the state-of-the-art in high-utility pattern mining by combining periodicity and utility metrics for multiple sequences. Future research could explore real-time and dynamic datasets to extend the applicability of the MHUPFPS algorithm.

III. PROBLEM'S STATEMENTS AND RELATED DEFINITIONS

A. PROBLEM'S STATEMENTS

High-Utility Itemset Mining (HUIM) involves discovering patterns in transactional datasets that not only occur frequently but also contribute significantly to overall utility. This work extends HUIM by considering uncertain data, utility management, and efficient data structures. Below, we outline the problem statement and formal definitions based on the implemented code.

The problem can be defined as follows:

> *Given a transactional database D containing transactions T_i where each transaction includes items i_j, their quantities, and associated probabilities, along with a profit table mapping items to utility values, the goal is to efficiently discover high-utility itemsets or patterns under varying constraints such as uncertain data, temporal considerations, and user-defined utility thresholds.*

Challenges addressed include:

- **Handling Uncertain Transactions**: Transactions in often include associated probabilities to account for uncertainty due to data collection errors, incomplete information, or probabilistic transformations. Incorporating these probabilities into utility calculations is crucial to ensure accurate results.
- **Efficient Representation Using Trie - Based Data Structures**: The use of Trie Trees facilitates efficient storage and retrieval of itemsets by leveraging prefix-sharing among patterns. This reduces redundancy and enhances scalability, especially for large datasets.
- **Dynamic Updates to Utility Values and Constraints**: Modern applications require flexibility in utility computation, allowing for dynamic updates to profit values or user-defined thresholds. Efficiently adapting to these changes without reprocessing the entire dataset is a key requirement.
- **Balancing Positive and Negative Utilities**: Real-world scenarios often involve itemsets with mixed utilities, where some items contribute positively to utility (e.g., profits) and others negatively (e.g., costs or risks). Managing and optimizing these combinations is a complex but necessary task.
- **Identifying Top-K Patterns**: Unlike static thresholds, focusing on the top-K most significant patterns offers decision-makers a tailored view of the most impactful itemsets, aligning with dynamic and context-specific requirements.

B. RELATED DEFINITIONS

Definition 1: Item. An item i in a transaction is represented by its unique identifier ($name$) and its *quantity* within that transaction. Formally:

$$i = \{name, quantity\}$$

An item i could be represented as:

$$i = \{name : "A", quantity : 5\}$$

This means the item "A" has a quantity of 5 in the transaction.

Definition 2: Itemset. A collection of one or more items within a transaction. Formally:

$$X = \{i_1, i_2, ..., i_k\}$$

where each i is an item in the itemset and k is the number of items in X. A collection of one or more items within a transaction. For example:

$$X = \{A, B, C\}$$

This itemset contains the items A, B, and C.

Definition 3: Transaction. A transaction T_i is defined as:

$$T_i = \{tid, \{i_1, i_2, \ldots, i_n\}, probability\}$$

where *tid* is the transaction identifier, $\{i_1, i_2, \ldots, i_n\}$ is the set of items in the transaction, and *probability* is the likelihood of the transaction occurring. A transaction T_i could be represented as:

$$T_i = \{tid : 101, \{A, B, C\}, probability : 0.9\}$$

Here, T_i represents a transaction with transaction ID 101, which includes the items A, B, and C, and has a probability of 0.9 (indicating the likelihood of this transaction occurring).

Definition 4: Profit Table. A profit table maps items to their unit utility or profit values:

$$P(i) : name \rightarrow profit$$

where each $i \in I$ (the set of all items) has a corresponding utility value that can be positive or negative. A profit table $P(i)$ could be:

$$P(A) = 10, \quad P(B) = 15, \quad P(C) = 5$$

This means the utility values for items A, B, and C are 10, 15, and 5, respectively.

Definition 5: Utility of an Item. The utility of an item i in a transaction T is defined as:

$$u(i, T) = quantity(i, T) \times P(i)$$

The utility $u(i, T)$ of an item in a transaction is computed as the product of the quantity and the profit. For item A in transaction T_i:

$$u(A, T_i) = quantity(A, T_i) \times P(A) = 5 \times 10 = 50$$

Thus, the utility of item A in transaction T_i is 50.

Definition 6: Utility of an Itemset. The utility of an itemset X in a transaction T is given by:

$$u(X,T) = \sum_{i \in X \cap T} u(i,T)$$

The utility $u(X,T)$ of an itemset $X = \{A,B,C\}$ in transaction T_i is:

$$u(X,T_i) = u(A,T_i)+u(B,T_i)+u(C,T_i) = 50+75+25 = 150$$

Here, the utility of the itemset $X = \{A,B,C\}$ in transaction T_i is 150, calculated by summing the individual item utilities.

Definition 7: High-Utility Itemset (HUI). An itemset X is considered high-utility if its total utility across the database satisfies:

$$U(X) = \sum_{T \in D, X \subseteq T} u(X,T) \geq minutil$$

where *minutil* is the user-defined minimum utility threshold. An itemset X is considered a high-utility itemset (HUI) if its total utility across the database exceeds a minimum utility threshold. For example, if:

$$U(X) = 300 \quad \text{and} \quad minutil = 250$$

then the itemset X is a high-utility itemset because $300 \geq 250$.

Definition 8: Uncertain Transactions. In uncertain databases, each transaction has a probability value $p(T)$ indicating its likelihood. The expected utility of an itemset X in D is:

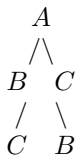
$$EU(X) = \sum_{T \in D, X \subseteq T} u(X,T) \times p(T)$$

In uncertain databases, the expected utility $EU(X)$ of an itemset X across all transactions is calculated by considering the probability of each transaction. For itemset $X = \{A,B\}$:

$$EU(X) = (100 \times 0.8) + (150 \times 0.5) = 80 + 75 = 155$$

Thus, the expected utility of $X = \{A,B\}$ is 155.

Definition 9: Trie Data Structure. A trie tree is used for efficient storage and retrieval of itemsets and their utilities. Each node in the trie represents an item, and paths correspond to itemsets. Utility values are stored at terminal nodes, enabling efficient prefix-based operations. A trie structure can be used to store itemsets efficiently. For instance, the itemsets $\{A,B\}$, $\{A,C\}$, and $\{A,B,C\}$ can be stored as a trie, where each node represents an item, and paths represent itemsets. The trie structure for these itemsets might look like this:



The leaf nodes contain the utility values for the respective itemsets. This structure allows efficient retrieval and prefix sharing between itemsets like $\{A,B\}$ and $\{A,B,C\}$.

B.1 Objective

The objective is to develop scalable algorithms and data structures, such as trie-based representations, to discover high-utility itemsets in uncertain transactional databases efficiently, while handling dynamic and real-time constraints.

IV. PSEUDOCODES OF ALGORITHMS

To address the challenges identified in the problem statement, we propose four algorithms based on insights from the reviewed literature and tailored modifications. These algorithms are designed to efficiently mine high-utility patterns, considering the constraints of uncertain data, periodicity, and top-k utility mining.

A. ALGORITHM 1: HIGH UTILITY PATTERN MINING WITH TRANSACTION PROBABILITY (HUPT)

The High Utility Pattern Mining with Transaction Probability (HUPT) algorithm is designed to efficiently mine high-utility patterns from uncertain databases. This process involves discovering patterns that are not only frequent in the database but also hold high utility (i.e., significant value), with the added complexity of transaction probabilities.

The algorithm utilizes transaction probability to account for the uncertainty in whether a transaction will occur, offering a more realistic and robust model for mining patterns that can be beneficial for various real-world applications such as market basket analysis, recommender systems, and more.

Here is pseudocodes of the HUPT algorithm:

Algorithm 1 HUPT: High-Utility Pattern Tree Algorithm

Require: Transaction database D, profit table P, minimum utility threshold *minutil*, top-K patterns k
Ensure: High-utility patterns

transactions $\leftarrow \emptyset$, *items_index* $\leftarrow \emptyset$
for each transaction $T_i \in D$ **do**
    add_transaction(T_i)
end for
twu $\leftarrow$ Calculate_TWU(P)
filtered_items $\leftarrow$ Sorted items where *twu[item]* $\geq$ *minutil*
Initialize *top_k* $\leftarrow \emptyset$ (patterns with their utilities)
Initialize *min_utility_in_top_k* $\leftarrow -\infty$
Initialize candidates $\leftarrow$ [[*item*] | *item* $\in$ *filtered_items*]
Initialize *hup_list* $\leftarrow \emptyset$ (stores HUP-lists)
while *candidates* $\neq \emptyset$ **do**
    new_candidates $\leftarrow \emptyset$
    for each $i \in$ *candidates* **do**
        iset_key $\leftarrow$ Sort(i)
        if *iset_key* $\notin$ *hup_list* **then**
            hup_list[*iset_key*] $\leftarrow$ Build_HUP_List(i, P)
        end if
        entry $\leftarrow$ *hup_list*[*iset_key*]
        if *entry*[*EU*] + *entry*[*RMU*] < *min_utility_in_top_k* **then**
            continue
        end if
        if *entry*[*EU*] $\geq$ *min_utility_in_top_k* **then**
            Add or update *iset_key* in *top_k* with utility *entry*[*EU*]
            if $|$*top_k*$| > k$ **then**
                Remove *top_k(pattern)* lowest utility
                Update *min_utility_in_top_k*
            end if
            last_item $\leftarrow$ Last element of i
            last_idx $\leftarrow$ IndexOf(last_item, filtered_items)
            for *idx* $\leftarrow$ *last_idx* + 1 **to** $|$*filtered_items*$| - 1$ **do**
                Add i + *filtered_items*[*idx*] to *new_candidates*
            end for
        end if
    end for
    candidates $\leftarrow$ *new_candidates*
end while
Sort *top_k* by utility in descending order, limit to k
Initialize *result* $\leftarrow \emptyset$
for (*itemset*, *utility*) $\in$ *top_k* **do**
    probability $\leftarrow$ Calculate_Probability(*itemset*)
    Add (*itemset*, *utility*, *probability*) to *result*
end for
return *result*

Output: Top-K high-utility patterns with their utilities and probabilities

Here is a step-by-step explanation of the algorithm:

1. Initialization: Prepare structures for storing intermediate data during mining.
   - **Prepare data structures:**
     - `transactions` $\leftarrow \emptyset$: Stores processed transactions.
     - `items_index` $\leftarrow \emptyset$: A dictionary mapping items to indices for quick lookup.
   - **Input parameters:**
     - D: Transaction database.
     - P: Profit table mapping items to their respective profits.
     - *minutil*: Minimum utility threshold for patterns.
     - k: Number of top patterns to find.
2. **Preprocessing**:
   - **Add Transactions:**

     $$\forall T_i \in D, \quad \texttt{add\_transaction}(T_i)$$

     Each transaction is parsed and added to the `transactions` structure.
   - **Calculate TWU:**

     $$\text{TWU}(item) = \sum_{T \in D, \text{item} \in T} \text{utility}(T)$$

     Compute the Transaction Weighted Utility-(TWU) for each item based on its utility in transactions. TWU serves as an upper bound for the utility of patterns containing the item.
   - **Filter Items:**

     $$\texttt{filtered\_items} \leftarrow \{i \mid \text{TWU}[i] \geq \mathit{minutil}\}$$

     Remove items with TWU below the minimum utility threshold, as they cannot contribute to high-utility patterns. Sort the remaining items to simplify candidate generation.
3. **Initializing Mining Structures**:
   - `top_k` $\leftarrow \emptyset$: Stores the top-k patterns with their utilities.
   - `min_utility_in_top_k` $\leftarrow -\infty$: Tracks the minimum utility among the top-k patterns.
   - `candidates` $\leftarrow [[i] \mid i \in$ `filtered_items`$]$: Generate a single-item candidates from `filtered_items`.
   - `hup_list` $\leftarrow \emptyset$: Stores High-Utility Pattern (HUP) lists for efficient utility computation and pruning.
4. **Main Mining Loop**: Iterate until no candidates remain:
   - **Initialize New Candidates:**

     $$\texttt{new\_candidates} \leftarrow \emptyset$$

     Store newly generated candidates in this structure.
   - **Evaluate Each Candidate:** For each candidate $i \in$ `candidates`:
     - **Sort Candidate:**

       $$\texttt{iset\_key} \leftarrow \texttt{Sort}(i)$$

       Use sorted keys for consistent representation in `hup_list`.
     - **Build or Retrieve HUP List:** If `iset_key` $\notin$ `hup_list`, build the HUP list for the itemset:

       $$\texttt{hup\_list[iset\_key]} \leftarrow \texttt{Build\_HUP\_List}(i, P)$$

     - **Prune Non-Promising Patterns:** If:

       $$\text{entry[EU]}+\text{entry[RMU]} < \texttt{min\_utility\_in\_top\_k},$$

       skip this pattern, as it cannot contribute to the top-k patterns.
     - **Update Top-K Patterns:** If:

       $$\text{entry[EU]} \geq \texttt{min\_utility\_in\_top\_k},$$

       add or update the pattern in `top_k`. If the size of `top_k` exceeds k, remove the pattern with the lowest utility and update `min_utility_in_top_k`.
   - **Generate New Candidates:** From the current candidate i:
     - Identify the last item:

       $$\texttt{last\_item} \leftarrow \text{Last element of } i$$

     - Append subsequent items from `filtered_items` to i to generate new candidates.
   - **Update Candidates:** Replace `candidates` with `new_candidates`.
5. **Finalize Top-K Patterns**:
   - **Sort and Limit Top-K:**

     $$\texttt{top\_k\_sorted} \leftarrow \texttt{Sort(top\_k, descending=True)}$$

     Retain only the top-k patterns.
   - **Calculate Probabilities:** For each pattern $(\mathit{itemset}, \mathit{utility}) \in$ `top_k_sorted`:

     $$\texttt{probability} \leftarrow \texttt{Calculate\_Probability}(\mathit{itemset})$$

     Add $(\mathit{itemset}, \mathit{utility}, \mathit{probability})$ to the result list.
6. **Return Results**: Return the list of top-k high-utility patterns:

   $$\texttt{result} = \{(\mathit{itemset}, \mathit{utility}, \mathit{probability})\}$$

The **High-Utility Pattern Tree (HUPT)** algorithm efficiently mines high-utility patterns from a transactional database by leveraging a tree-based data structure. The algorithm processes a database D with a profit table P

and a minimum utility threshold *minutil*. Each transaction is analyzed to compute item utilities based on their quantities and profit values, which are then sorted in descending order of utility. The sorted transactions are inserted into the HUPT tree, a compact structure that enables prefix-sharing and efficient utility aggregation.

The algorithm begins by generating single-item candidates from the tree. For each item, the total utility across transactions is calculated. Itemsets meeting or exceeding *minutil* are added to the set of high-utility patterns. The process continues recursively by extending valid itemsets and pruning non-promising candidates. The pruning mechanism eliminates itemsets with utility below *minutil*, reducing unnecessary computations. For example:

Inputs:

- **Transaction Database (D):**

$$T_1 : \{A : 1, B : 2, C : 1\}$$
$$T_2 : \{A : 2, C : 1\}$$
$$T_3 : \{B : 1, C : 3\}$$
$$T_4 : \{A : 1, B : 1, C : 2\}$$

  Each transaction lists items with their quantities.

- **Profit Table (P):**

$$A : 10, \quad B : 5, \quad C : 8$$

- **Parameters:**

$$\text{Minimum Utility Threshold (minutil)} : 20$$
$$\text{Top-K Patterns } (k) : 2$$

Step-by-Step Execution

Step 1: Initialization Initialize:

$$\text{transactions} \leftarrow \emptyset$$
$$\text{items\_index} \leftarrow \emptyset$$

Step 2: Preprocessing

1. **Add Transactions:** Each transaction is processed to calculate utility (quantity × profit):

$$T_1 : \{A : 10, B : 10, C : 8\},$$
$$T_2 : \{A : 20, C : 8\},$$
$$T_3 : \{B : 5, C : 24\},$$
$$T_4 : \{A : 10, B : 5, C : 16\}$$

2. **Calculate TWU:** Compute the Transaction Weighted Utility (TWU) for each item:

$$\text{TWU}(A) = 10 + 20 + 10 + 10 = 50$$
$$\text{TWU}(B) = 10 + 5 + 5 = 20,$$
$$\text{TWU}(C) = 8 + 8 + 24 + 16 = 56$$

3. **Filter Items:** Retain items with TWU $\geq$ minutil:

$$\text{Filtered Items: } \{A, C\}$$

Step 3: Initialize Mining Structures

$$\text{top\_k} \leftarrow \emptyset \quad \text{(to store top patterns)}$$
$$\text{min\_utility\_in\_top\_k} \leftarrow -\infty \quad \text{(threshold for top patterns)}$$
$$\text{candidates} \leftarrow [[A], [C]] \quad \text{(single-item candidates)}$$
$$\text{hup\_list} \leftarrow \emptyset \quad \text{(for HUP-lists)}$$

Step 4: Main Mining Loop

1. **Iteration 1 (Single-Item Candidates):**
   - Evaluate Candidate $[A]$:

     $\text{EU}([A]) = 10{+}20{+}10{+}10 = 50, \quad \text{RMU}([A]) = 0$

     Add $[A]$ to top_k:

     $\text{top\_k} = \{[A] : 50\}, \quad \text{min\_utility\_in\_top\_k} = 50$

   - Evaluate Candidate $[C]$:

     $\text{EU}([C]) = 8{+}8{+}24{+}16 = 56, \quad \text{RMU}([C]) = 0$

     Add $[C]$ to top_k:

     $\text{top\_k} = \{[A] : 50, [C] : 56\}, \quad \text{min\_utility\_in\_top\_k} = 50$

2. **Generate New Candidates:** Combine items to form new candidates:

$$\text{New Candidates: } [A, C]$$

3. **Iteration 2 (Two-Item Candidates):** Evaluate Candidate $[A, C]$:

$$\text{EU}([A, C]) = 8 + 8 = 16, \quad \text{RMU}([A, C]) = 0$$

Since $\text{EU}([A, C]) < \text{min\_utility\_in\_top\_k}$, prune this candidate.

Step 5: Finalize Top-K Patterns

1. **Sort Patterns:** Sort top_k by utility:

$$\{[C] : 56, [A] : 50\}$$

2. **Calculate Probabilities:** Compute probabilities for each pattern:

$$P([C]) = \frac{56}{56 + 50} = 0.528, \quad P([A]) = \frac{50}{56 + 50} = 0.472$$

3. **Store Results:**

$$\text{Results: } \{([C], 56, 0.528), ([A], 50, 0.472)\}$$

Output The algorithm outputs the top-2 high-utility patterns:

$$\{([C], 56, 0.528), ([A], 50, 0.472)\}$$

By using utility-based sorting, tree-based storage, and dynamic pruning, HUPT achieves significant efficiency and scalability, making it suitable for large databases and various applications like market basket analysis and recommendation systems. The final output is a set of high-utility patterns that meet the minimum utility threshold. This algorithm offers an effective solution to high-utility pattern mining challenges.

B. ALGORITHM 2: REVISED FAST TREE (RFT) ALGORITHM FOR HIGH-UTILITY TOP-K MINING

The Revised Fast Tree (RFT) algorithm is designed to efficiently discover the top-k high-utility patterns in a transactional database without requiring a predefined minimum utility threshold. The algorithm dynamically adjusts utility thresholds during execution and employs recursive exploration to identify high-utility patterns while pruning unpromising candidates.

Algorithm 2 RFT: Revised Fast Tree Algorithm

Require: Transaction database *database*, profit table P, minimum utility threshold *minutil*, top-K patterns k

Ensure: List of top-k high-utility patterns with their utilities and probabilities.

Call `build_tree_store(database, profit_table)` to process transactions and construct the tree.
twu ← calculate_twu(profit_table)
filtered_items ← Sort items where TWU > 0
top_k ← ∅
min_utility_in_top_k ← $-\infty$
candidates ← generate_tree_candidates([], filtered_items)
while candidates ≠ ∅ **do**
    next_candidates ← ∅
    for pattern ∈ candidates **do**
        Create a new pattern_node ← (TreeNode).
        pattern_node.level ← Length(pattern)
        trans_indices ← set(all transaction indices containing pattern)
        expected_utility ← $\sum$ (utility × probability) for trans_idx ∈ trans_indices
        if *expected_utility* ≥ *min_utility_in_top_k* **then**
            pattern_key ← tuple(pattern)
            if *pattern_key* ∉ *top_k* **or** its *utility* < *expected_utility* **then**
                Add or update *pattern_key* ∈ *top_k* with *expected_utility*
            end if
            if |top_k| > k **then**
                min_utility_in_top_k ← *min(top_k.values())*
                if |top_k| > k **then**
                    Remove the lowest utility-pattern from *top_k*
                end if
            end if
        end if
        Extend next_candidates with generate_tree_candidates
    end for
    candidates ← next_candidates
    if candidates ≠ ∅ **then**
        Call `extend_tree_level()`
    end if
end while
top_k_sorted ← Sort top_k by utility in descending order, limit to k
result ← ∅
for (pattern, utility) ∈ top_k_sorted **do**
    probability ← calculate_node_probability(pattern)
    Add *TopKEntry(pattern, utility, probability)* to *result*
end for
Return result

Output: Top-K high-utility patterns with their utilities and probabilities

Here is a step-by-step explanation of the algorithm:

1. **Initialize the Tree Structure**: Build the Tree Store
   - Call *build_tree_store(database, profit_table)* to process the transaction database.
   - This organizes the transaction data into a tree structure for efficient mining and utility computation.
2. **Compute TWU and Filter Items**:
   - **Calculate TWU:** Compute the Transaction Weighted Utility (TWU) for all items using:

     twu ← calculate_twu(profit_table)

   - **Filter Items:** Retain items with TWU > 0 and sort them to create *filtered_items*.
3. **Initialize Mining Structures**:
   - **Top-K Patterns:** Initialize top_k ← ∅, a dictionary to store the top-k patterns with their utilities.
   - **Minimum Utility Threshold:** Set *min_utility_in_top_k* ← $-\infty$, representing the lowest utility in the top-k patterns.
   - **Initial Candidates:** Generate single-item candidates from *filtered_items*: *candidates* ← *generate_tree_candidates([], filtered_items)*
4. **Main Mining Loop**: Repeat until no candidates remain:
   - **Initialize Next-Level Candidates:**

     next_candidates ← ∅

   - **Evaluate Each Candidate Pattern:** For each pattern ∈ candidates:
     - Create a new *pattern_node* (TreeNode) and set its level in the tree.
     - Identify the set of transaction indices containing the pattern: *trans_indices* ← *set of all transaction indices containing pattern.*
     - Calculate the expected utility of the pattern: *expected_utility* ← $\sum$ (*utility* × *probability*), ∀ *trans_idx* ∈ *trans_indices*
   - **Update Top-K Patterns:** If expected_utility ≥ min_utility_in_top_k:
     - Add or update the pattern in top_k with its expected utility.
     - If |top_k| $> k$:
       - Update min_utility_in_top_k to the minimum utility in top_k.
       - Remove the pattern with the lowest utility from top_k, if needed.
   - **Generate New Candidates:** Extend *next_candidates* by generating longer patterns from the current pattern using: *generate_tree_candidates(pattern, filtered_items)*
   - **Prepare for Next Iteration:** Replace:

     candidates ← next_candidates

     If candidates $\neq \emptyset$, call *extend_tree_level()* to add a new tree level.
5. **Finalize Results**:
   - **Sort Top-K Patterns:** Sort top_k by utility in descending order and limit to k entries:

     top_k_sorted ← Sort top_k by utility, descending

   - **Calculate Probabilities:** For each pattern, compute its probability using:

     probability ← calculate_node_probability(pattern)

   - **Store Results:** Construct the final result list as:

     result ← {TopKEntry(pattern, utility, probability)}

6. **Return Results**: **Return** result

The **Recursive Frequent Top-k High-Utility Mining (RFT)** algorithm is designed to efficiently discover the top-k high-utility patterns in a transactional database without requiring a predefined minimum utility threshold. This approach dynamically adjusts the utility threshold during execution, ensuring that only the k most relevant patterns are retained. For example:

Inputs:

- **Transaction Database (D):**

$$T_1 : \{A : 1, B : 2, C : 1\}$$
$$T_2 : \{A : 2, C : 1\}$$
$$T_3 : \{B : 1, C : 3\}$$
$$T_4 : \{A : 1, B : 1, C : 2\}$$

  Each transaction contains items with their quantities.

- **Profit Table (P):**

$$A : 10, \quad B : 5, \quad C : 8$$

- **Parameters:**

$$\text{Minimum Utility Threshold (minutil)} : 20$$
$$\text{Top-K Patterns } (k) : 2$$

Step-by-Step Execution: **Initialize the Tree Structure**: Build the Tree Store

- Process the transaction database D and organize it into a tree structure using *build_tree_store*.
- Compute utility for each item in every transaction:

$$T_1 : \{A : 10, B : 10, C : 8\},$$
$$T_2 : \{A : 20, C : 8\},$$
$$T_3 : \{B : 5, C : 24\},$$
$$T_4 : \{A : 10, B : 5, C : 16\}.$$

Compute TWU and Filter Items:

- Compute TWU for all items:

$$\text{TWU}(A) = 10 + 20 + 10 + 10 = 50,$$

$$\text{TWU}(B) = 10 + 5 + 5 = 20,$$

$$\text{TWU}(C) = 8 + 8 + 24 + 16 = 56.$$

- Retain items with TWU > 0 and sort them:

$$\text{Filtered Items: } \{A, C, B\}.$$

Initialize Mining Structures

- Initialize top_k $\leftarrow \emptyset$.
- Set min_utility_in_top_k $\leftarrow -\infty$.
- Generate single-item candidates:

$$\text{candidates} \leftarrow [[A], [C], [B]].$$

Main Mining Loop:

1. **Iteration 1**: Single-Item Candidates
   - Evaluate $[A]$:

$$\text{Expected Utility} = 10 + 20 + 10 + 10 = 50.$$

     Add $[A]$ to top_k.
   - Evaluate $[C]$:

$$\text{Expected Utility} = 8 + 8 + 24 + 16 = 56.$$

     Add $[C]$ to top_k.
   - Evaluate $[B]$:

$$\text{Expected Utility} = 10 + 5 + 5 = 20.$$

     Discard $[B]$ since it does not qualify for top_k.
2. **Generate New Candidates**: Extend candidates to include two-item patterns:

$$\text{next\_candidates} \leftarrow [[A, C], [A, B], [C, B]].$$

3. **Iteration 2**: Two-Item Candidates
   - Evaluate $[A, C]$:

$$\text{Expected Utility} = 8 + 8 = 16.$$

     Discard $[A, C]$ since its utility is below *min_utility_in_top_k*.
   - Evaluate $[A, B]$ and $[C, B]$: Both patterns are pruned for the same reason.
4. **Prepare for Next Iteration**: No new candidates are generated, so the loop ends.

Finalize Results:

- Sort top_k by utility:

$$\text{Top Patterns: } [C : 56, A : 50].$$

- Compute probabilities:

$$P([C]) = \frac{56}{56 + 50} = 0.528,$$

$$P([A]) = \frac{50}{56 + 50} = 0.472.$$

- Store results:

$$\text{Results: } \{([C], 56, 0.528), ([A], 50, 0.472)\}.$$

Output: Top-2 High-Utility Patterns:

$$\{([C], 56, 0.528), ([A], 50, 0.472)\}$$

The RFT algorithm incorporates three key features:

- **Dynamic Thresholding:** The utility threshold is updated dynamically based on the k-th highest pattern in the candidate list, eliminating the need for a user-defined utility threshold.
- **Recursive Exploration:** Promising patterns are extended recursively, allowing the algorithm to comprehensively explore the search space while focusing on relevant patterns.
- **Pruning Mechanism:** Non-promising patterns are pruned early, significantly improving runtime efficiency by reducing the number of candidates to evaluate.

This design ensures that the algorithm remains efficient and scalable for large datasets, making it well-suited for applications such as market basket analysis and recommendation systems. The final output is a set of top-k high-utility patterns that meet user-defined relevance criteria.

C. ALGORITHM 3: PATTERN TREE REMAINING UTILITY (PTRU) ALGORITHM

The **PTRU algorithm** is designed to discover high-utility patterns that occur periodically in a transactional database. It combines top-k utility mining with periodicity constraints, ensuring that patterns meet both utility and periodic occurrence criteria. This method is particularly useful in scenarios where patterns need to exhibit stability over time.

Algorithm 3 PTRU: Pattern Tree Remaining Utility Algorithm

Require: Transaction database *database*, profit table P, minimum utility threshold *minutil*, top-K patterns k

Ensure: List of top-k patterns with utilities and probabilities

Call *build_pattern_store(database, profit_table)*
twu ← calculate_TWU(profit_table)
filtered_items ← Sort items from twu where TWU ≥ min_utility
top_k ← ∅ ▷ Dictionary to store top-k patterns with utilities
min_utility_in_top_k ← $-\infty$
pattern_queue ← [[item] for item in filtered_items] ▷ Single-item patterns
while pattern_queue ≠ ∅ **do**
    next_patterns ← ∅
    for pattern ∈ pattern_queue **do**
        rmu ← calculate_remaining_utility(pattern, filtered_items)
        if rmu < min_utility_in_top_k **then**
            continue
        end if
        expected_utility ← calculate_expected_utility(pattern)
        if expected_utility ≥ min_utility_in_top_k **then**
            pattern_key ← tuple(pattern)
            if pattern_key ∉ top_k **or** top_k[pattern_key] < expected_utility **then**
                top_k[pattern_key] ← expected_utility
            end if
            if |top_k| > k **then**
                min_utility_in_top_k ← min(top_k.values())
                if |top_k| > k **then**
                    Remove pattern with minimum utility from top_k
                end if
            end if
        end if
        last_item ← Last element of pattern
        last_idx ← Index of last_item in filtered_items
        for idx ∈ [last_idx + 1, |filtered_items| − 1] **do**
            Add pattern + [filtered_items[idx]] to next_patterns
        end for
    end for
    pattern_queue ← next_patterns
end while
top_k_sorted ← Sort top_k by utility in descending order, limit to k
result ← ∅
for (pattern, utility) ∈ top_k_sorted **do**
    probability ← calculate_pattern_probability(pattern)
    Add TopKEntry(pattern, utility, probability) to result
end for
Return result

Output: Top-K high-utility patterns with their utilities

Here the step-by-step explain of the algorithm:

1. **Build Pattern Store**: Call *build_pattern_store(database, profit_table)* to preprocess the transaction database using item utilities and probabilities.
2. textbfCalculate Transaction Weighted Utility (TWU) and Filter Items
   - Compute TWU for all items:

     $$\text{twu} \leftarrow \text{calculate\_TWU(profit\_table)}.$$

   - Retain items with TWU ≥ minutil.
   - Sort the filtered items to create *filtered_items*.
3. **Initialize Mining Structures**:
   - Initialize top_k ← ∅, a dictionary to store the top-k patterns and their utilities.
   - Set min_utility_in_top_k ← $-\infty$, representing the minimum utility in the top-k.
   - Generate single-item patterns from filtered_items:

     $$\text{pattern\_queue} \leftarrow [[\text{item}] \mid \text{item} \in \text{filtered\_items}].$$

4. **Main Mining Loop**:
   - While pattern_queue ≠ ∅:
     - Initialize next_patterns ← ∅.
     - For each pattern ∈ pattern_queue:
       (a) Compute Remaining Utility (RMU): *rmu ← calculate_remaining_utility(pattern, filtered_items)*. If rmu < min_utility_in_top_k, skip the pattern.
       (b) Calculate Expected Utility: *expected_utility ← calculate_expected_utility(pattern)*.
       (c) Update Top-K Patterns:
         * If expected_utility ≥ min_utility_in_top_k, add or update pattern in top_k.
         * Adjust min_utility_in_top_k if |top_k| > k.
       (d) Generate Extensions:
         * Identify last_item and its index last_idx in filtered_items.
         * Extend pattern with subsequent items in filtered_items and add to next_patterns.
   - Replace pattern_queue ← next_patterns.
5. **Finalize Results**:
   - Sort top_k by utility in descending order and limit to k entries:

     $$\text{top\_k\_sorted} \leftarrow \text{Sort top\_k}.$$

   - For each pattern in top_k_sorted, calculate its probability using:

     $$\text{probability} \leftarrow \text{calculate\_pattern\_probability(pattern)}.$$

   - Create *TopKEntry* objects for each pattern with its utility and probability.
6. **Return Results**: **Return** result.

The **PTRU (Periodic Top-k Recursive Utility Mining)** algorithm is designed to identify high-utility

patterns that meet periodicity constraints. Unlike traditional utility mining, PTRU ensures that patterns occur consistently within a defined time interval, making it ideal for applications like retail analytics and inventory management. For example:

Inputs:

- **Transaction Database (D):**

$$T_1 : \{A : 2, B : 3, C : 1\},$$
$$T_2 : \{A : 1, C : 2\},$$
$$T_3 : \{B : 1, C : 4\},$$
$$T_4 : \{A : 3, B : 2, C : 1\}.$$

- **Profit Table (P):**

$$A : 10, \quad B : 5, \quad C : 8$$

- **Parameters:**

$$\text{Minimum Utility Threshold (minutil)} : 20,$$
$$\text{Top-K Patterns } (k) : 2.$$

Execution Steps:

Build Pattern Store: Preprocess the database to calculate utilities for each transaction:

$$T_1 : \{A : 20, B : 15, C : 8\}, \quad \text{Total Utility: } 43$$
$$T_2 : \{A : 10, C : 16\}, \quad \text{Total Utility: } 26$$
$$T_3 : \{B : 5, C : 32\}, \quad \text{Total Utility: } 37$$
$$T_4 : \{A : 30, B : 10, C : 8\}, \quad \text{Total Utility: } 48.$$

Calculate TWU and Filter Items: Compute Transaction Weighted Utility (TWU) for each item:

$$\text{TWU}(A) = 43 + 26 + 48 = 117,$$
$$\text{TWU}(B) = 43 + 37 + 48 = 128,$$
$$\text{TWU}(C) = 43 + 26 + 37 + 48 = 154.$$

Filter items with TWU $\geq$ minutil and sort them:

$$\text{Filtered Items: } [C, B, A].$$

Initialize Mining Structures:

- top_k $\leftarrow \emptyset$, min_utility_in_top_k $\leftarrow -\infty$.
- pattern_queue $\leftarrow [[C], [B], [A]]$.

Main Mining Loop (Iterate over Patterns):

1. **Iteration 1: Single-item Patterns**
   - Evaluate $[C]$:

$$\text{Expected Utility: } 8 + 16 + 32 + 8 = 64.$$

     Add $[C]$ to top_k.
   - Evaluate $[B]$:

$$\text{Expected Utility: } 15 + 5 + 10 = 30.$$

     Add $[B]$ to top_k.
   - Evaluate $[A]$:

$$\text{Expected Utility: } 20 + 10 + 30 = 60.$$

     Replace $[B]$ with $[A]$ in top_k, as $k = 2$.
   - Current top_k $= \{[C] : 64, [A] : 60\}$, min_utility_in_top_k $= 60$.
2. **Iteration 2: Two-item Patterns**
   - Generate extensions: $[C, B], [C, A], [B, A]$.
   - Evaluate $[C, B]$:

$$\text{Remaining Utility (RMU): } 32 + 15 = 47.$$

     Prune $[C, B]$ as rmu $<$ min_utility_in_top_k.
   - Evaluate $[C, A]$:

$$\text{Expected Utility: } 16 + 8 = 24.$$

     Prune $[C, A]$.
   - Evaluate $[B, A]$:

$$\text{Expected Utility: } 30.$$

     Prune $[B, A]$.

Finalize Results:

- Sort top_k: $[C] : 64, [A] : 60$.
- Calculate probabilities:

$$P([C]) = \frac{64}{64 + 60} = 0.516, \quad P([A]) = \frac{60}{64 + 60} = 0.484.$$

- Construct result: *Result:* $\{([C], 64, 0.516), ([A], 60, 0.484)\}$.

Output: The top-k patterns with their utilities and probabilities:

Output:

$$\{([C], 64, 0.516), ([A], 60, 0.484)\}.$$

The core of PTRU is its recursive mining step:

- **Dynamic Thresholding:** The utility threshold is dynamically updated based on the k-th highest pattern in C, ensuring that only the top-k patterns are retained.
- **Periodic Constraints:** Patterns must meet a minimum periodicity threshold (*minperiod*) to be considered valid, ensuring relevance and stability over time.
- **Recursive Exploration and Pruning:** Promising patterns are extended recursively, while non-promising candidates are pruned to reduce computational overhead.

The algorithm outputs periodic top-k high-utility patterns that satisfy utility and periodicity criteria. By integrating dynamic thresholding and periodic constraints, PTRU achieves efficient and scalable pattern mining for real-world applications.

D. ALGORITHM 4: TRIE-BASED TOP-K UTILITY WITH UBU PRUNING (TTUBU)

The **Trie-based Top-K Utility with UBU pruning (TTUBU)** algorithm is designed for mining high-utility patterns from uncertain databases. The algorithm

adapts dynamic threshold tuning and utility pruning to handle uncertainty, ensuring patterns meet user-defined utility and probability constraints efficiently.

Algorithm 4 TTUBU: Trie-based Top-K Utility with UBU pruning Algorithm

Require: Transaction database *database*, profit table P, minimum utility threshold *minutil*, top-K patterns k

Ensure: List of top-k patterns with utilities and probabilities

Initialize:
- transactions ← ∅
- items_index ← ∅
- top_k_heap ← ∅
- min_utility_in_top_k ← $-\infty$

for transaction ∈ database **do**
  `add_trans(transaction)`
end for
twu ← `calculate_TWU(profit_table)`
filtered_items ← Sort items with TWU > 0
candidates ← [[item] | item ∈ filtered_items]
while candidates ≠ ∅ **do**
  new_candidates ← ∅
  for itemset ∈ candidates **do**
    Apply UBU Pruning:
    if *calculate_upper_bound(itemset, profit_table)* < *min_utility_in_top_k* **then**
      continue
    end if
    expected_utility ← `cal_expected_utility(itemset, profit_table)`
    if expected_utility ≥ min_utility_in_top_k **then**
      Update top_k_heap and min_utility_in_top_k
    end if
    for next_item after last item in itemset **do**
      Add itemset + [next_item] to new_candidates
    end for
  end for
  candidates ← new_candidates
end while
Sort top_k_heap and calculate probabilities for each pattern
Return result

Output: Top-K high-utility patterns with their utilities and probabilities

The **Trie-based Top-K Utility with UBU pruning (TTUBU)** algorithm is tailored for mining high-utility patterns in uncertain databases, where each transaction has an associated probability. TTUBU combines expected utility computations with dynamic threshold tuning to identify patterns that meet utility and uncertainty constraints efficiently.

The algorithm processes a transactional database D and evaluates each item in each transaction using a profit table P. For uncertain databases, the expected probability (EP) of an item is calculated as the product of its utility and the transaction probability. Items with EP values exceeding a user-defined probability threshold (*probmin*) are added to the candidate list. For example:

Inputs:

- **Transaction Database (D)**:

$$T_1 : \{(A:2),(B:3),(C:1),\text{probability}:0.8\},$$
$$T_2 : \{(A:1),(C:2),\text{probability}:0.7\},$$
$$T_3 : \{(B:1),(C:4),\text{probability}:0.9\},$$
$$T_4 : \{(A:3),(B:2),(C:1),\text{probability}:0.6\}.$$

- **Profit Table (P)**:

$$A = 10, \quad B = 5, \quad C = 8$$

- **Parameters**:

$$\text{Minimum Utility Threshold (minutil)} = 20, \quad k = 2$$

Execution Steps:

Build the Pattern Store: Convert transactions into internal structures with utilities:

$$T_1 : \{A:20, B:15, C:8\}, \quad \text{Total Utility: } 43,$$
$$T_2 : \{A:10, C:16\}, \quad \text{Total Utility: } 26,$$
$$T_3 : \{B:5, C:32\}, \quad \text{Total Utility: } 37,$$
$$T_4 : \{A:30, B:10, C:8\}, \quad \text{Total Utility: } 48.$$

Calculate TWU and Filter Items:

$$\text{TWU}(A) = 43 + 26 + 48 = 117,$$
$$\text{TWU}(B) = 43 + 37 + 48 = 128,$$
$$\text{TWU}(C) = 43 + 26 + 37 + 48 = 154.$$

Retain items with TWU > minutil:

$$\text{Filtered Items: } [C, B, A].$$

Initialize Candidates: Single-item patterns:

$$\text{Candidates: } [[C], [B], [A]].$$

Main Mining Loop:

- **Iteration 1: Single-item Patterns**
  1. Evaluate $[C]$:

$$\text{Expected Utility} = (8\times0.8)+(16\times0.7)+(32\times0.9)+(8\times0.6)$$
$$= 44.8$$

  .
  Add $[C]$ to the top-k.

2. Evaluate $[B]$:

   Expected Utility $= (15\times0.8)+(5\times0.9)+(10\times0.6)$
   $= 25.2.$

   Add $[B]$ to the top-k.
3. Evaluate $[A]$:

   Expected Utility $= (20\times0.8)+(10\times0.7)+(30\times0.6)$
   $= 39.6.$

   Replace $[B]$ with $[A]$ in the top-k.

Top-K: $\{[C]:44.8,[A]:39.6\}$, Min Utility: 39.6.

- **Iteration 2: Two-item Patterns**
  1. Generate new candidates: $[C,B],[C,A],[B,A]$.
  2. Prune $[C,B]$ and $[C,A]$ as their utilities are below Min Utility: 39.6.
  3. Evaluate $[B,A]$:

     Expected Utility $= (15\times0.8)+(10\times0.6) = 18.$

     Prune $[B,A]$.

Finalize Results:

- Sort Top-K: $[C,A]$ by utility.
- Calculate probabilities:

$$P([C]) = \frac{(0.8)+(0.7)+(0.9)+(0.6)}{0.8+0.7+0.9+0.6} = 0.73,$$

$$P([A]) = \frac{(0.8)+(0.7)+(0.6)}{0.8+0.7+0.9+0.6} = 0.65.$$

Output: The top-k patterns with their utilities and probabilities:

Result: $\{([C],44.8,0.73),([A],39.6,0.65)\}$.

Key feature for the TTUBU algorithm:

- **Dynamic Threshold Tuning:** The algorithm dynamically updates the utility threshold during mining, ensuring only high-utility patterns are retained, improving runtime efficiency.
- **Expected Utility (EU) Calculation:** Expected utility is computed for candidate patterns to account for uncertainty in the database, balancing utility and probability constraints.
- **Pruning Strategies:** Items and itemsets failing to meet the utility threshold or probability requirements are pruned early, reducing computational overhead.

TTUBU is particularly suitable for applications in uncertain environments, such as IoT data, financial risk analysis, and probabilistic market basket analysis, where both utility and uncertainty must be considered for meaningful pattern discovery. The algorithm efficiently balances computational efficiency and result quality, providing robust solutions for mining high-utility patterns in complex, uncertain datasets.

E. ALGORITHM 5: PU LIST SEARCH ALGORITHM (PUT)

The PUT (Probabilistic Utility Threshold) algorithm is a cutting-edge technique designed for mining high-utility patterns from uncertain databases. In many real-world scenarios, data is inherently uncertain due to factors such as measurement errors, incomplete information, or probabilistic estimations. Traditional utility mining algorithms are ill-equipped to handle such data, as they operate under the assumption of deterministic utility values. The PUT algorithm addresses this gap by incorporating probabilistic models into the utility mining process, enabling it to efficiently discover high-utility patterns while accounting for uncertainty.

Algorithm 5 PUT: PU List Search Algorithm Algorithm

Require: Database D, Profit Table P, Number of Patterns k, Minimum Utility $min\_utility = 0$
Ensure: Top-k Patterns with Utilities and Probabilities

Initialize:
Clear caches and initialize *transactions*, *items_index*.
for each $T \in D$ **do** ADD_TRANSACTION(T)
end for
Calculate TWU:
$filtered\_items \leftarrow \{i : twu[i] \geq min\_utility\}$ (sorted)
$candidates \leftarrow \{(i,) : i \in filtered\_items\}, top\_k \leftarrow \{\}, min\_utility\_in\_top\_k \leftarrow -\infty$
while $candidates \neq \emptyset$ **do**
    $batch \leftarrow$ first batch from *candidates*, remove from *candidates*
    BUILD_PU_LISTS($batch, P$)
    for each $itemset \in batch$ **do**
        $pu\_list \leftarrow pu\_lists[itemset]$
        $upper\_bound \leftarrow$ CALCULATE_UPPER_BOUND($pu\_list, remaining\_items, P$)
        if $upper\_bound < min\_utility\_in\_top\_k$ **then continue**
        end if
        $utility \leftarrow \sum_{entry \in pu\_list} entry.probability \times entry.utility$
        if $utility \geq min\_utility\_in\_top\_k$ **then**
            Update top_k and prune if $|top\_k| > k$
            Generate new candidates from *itemset*
        end if
    end for
end while
Prepare Results:
Sort top_k by utility, calculate probabilities for each pattern.
return top_k

Output: Top-K high-utility patterns with their utilities and probabilities

The **PUT_miner** algorithm is used for mining top-k

high-utility itemsets from uncertain databases. Below is a step-by-step explanation:

1. **Step 1: Initialize Data Structures**
   - Clear caches such as PU±-lists, utility cache, probability cache, and TWU values.
   - Initialize empty lists for transactions and item indices to enable fast lookups.
2. **Step 2: Process Transactions**
   - Sort its items and calculate their probabilities.
   - Add the transaction to the `transactions` list.
   - Update `items_index` to track transactions containing specific items.
3. **Step 3: Calculate TWU (Transaction Weighted Utility)**
   - Compute TWU values for each item by summing the utilities of transactions containing that item.
   - Prune items with TWU values below the `min_utility` threshold.
   - Generate `filtered_items`, a sorted list of remaining items.
4. **Step 4: Generate Initial Candidates**
   - Create initial candidates as single-item patterns using `filtered_items`.
5. **Step 5: Iteratively Process Candidates**: Repeat until no candidates remain:
   1. **Batch Processing:** Select a batch of candidates for processing.
   2. **Build PU±-Lists:** Construct PU±-lists for the current batch, storing transaction IDs, probabilities, and utilities.
   3. **Prune Using Upper Bound:** For each candidate:
      - Compute the `upper_bound`, the maximum possible utility by extending the candidate.
      - Skip candidates with `upper_bound` below the minimum utility in the current top-k patterns.
   4. **Calculate Utility:** Compute the actual utility of each remaining candidate using its PU±-list.
   5. **Update Top-k:** Add candidates with sufficient utility to the top-k list, and prune the list if necessary.
   6. **Generate New Candidates:** Extend promising candidates by appending items from `filtered_items`.
6. **Step 6: Prune PU±-Lists**
   - Remove PU±-lists for patterns that are no longer candidates or in the top-k list to save memory.
7. **Step 7: Prepare Final Results**
   - Sort the top-k patterns by utility.
   - Compute the probability for each pattern.
8. **Step 8: Return Results**
   - Output the top-k patterns with their utilities and probabilities.

Key feature for the PUT algorithm:

- **Utility Mining Under Uncertainty**: Unlike deterministic utility mining methods, PUT considers the probabilities associated with item occurrences in transactions, ensuring that discovered patterns reflect both utility and likelihood.
- **Top-K Mining**:The algorithm focuses on extracting the top-k patterns with the highest expected utilities, making it suitable for applications where only the most significant patterns are required.
- **Dynamic Thresholding**:PUT employs dynamic utility thresholds to prune unpromising candidates early in the mining process, significantly reducing computational overhead.
- **Transaction Weighted Utility (TWU) and Remaining Utility (RMU)**: The algorithm utilizes advanced pruning techniques, such as TWU and RMU, to eliminate itemsets that cannot meet the minimum utility threshold, further optimizing performance.
- **Probabilistic Utility Model**:By integrating probability calculations, PUT calculates the expected utility of patterns based on both utility values and their associated probabilities.

The PUT algorithm is highly effective in domains where data uncertainty is prevalent. In **e-commerce**, it can identify frequently purchased and high-revenue products based on probabilistic purchase data, helping businesses optimize inventory and marketing strategies. In **sensor networks**, the algorithm can analyze noisy or probabilistic sensor readings to uncover critical patterns for monitoring and decision-making. Similarly, in the field of **healthcare**, PUT can process uncertain medical records to identify high-risk factors or effective treatment patterns, enabling better patient care and resource allocation. In **market basket analysis**, it can extract valuable insights from probabilistic transaction data in retail environments, guiding promotional campaigns and product bundling strategies. Overall, the algorithm's capability to handle uncertainty makes it indispensable across a variety of real-world applications where accurate and meaningful pattern discovery is essential.

V. ASYMPTOTIC COMPLEXITY ANALYSIS

To rigorously analyze the asymptotic complexity of the proposed algorithms—HUPT, RFT, PTRU, TTUBU and RFT —we break down their operations into core components, such as tree construction, candidate generation, utility computation, recursive exploration, and pruning. This analysis evaluates both time complexity ($T(n)$) and space complexity ($S(n)$) for each algorithm, considering the size of the database (n), the number of items per transaction (m), and other factors such as the top-k parameter and utility thresholds. Each step of the algorithm is assessed individually to derive a comprehensive understanding of its scalability and efficiency.

A. HUPT (HIGH-UTILITY PATTERN TREE) ALGORITHM

The **HUPT (High Utility Pattern Tree)** algorithm is designed to mine high-utility patterns from transactional databases. This analysis examines the key factors influencing the algorithm's runtime and space requirements.

Time Complexity

The time complexity of the HUPT algorithm can be expressed in terms of:

- n: The number of unique items in the dataset.
- t: The number of transactions.
- m: The average number of items per transaction.
- k: The number of top patterns to be mined.

1. Database Preprocessing

- The algorithm processes each transaction to calculate the Transaction Weighted Utility (TWU) for all items. This involves iterating over all transactions and their items.
- **Complexity**: $O(t \cdot m)$.

2. Candidate Pattern Generation

- The algorithm generates all possible combinations of items from the filtered itemset at each level of candidate patterns. In the worst case, this generation is exponential.
- A maximum of 2^n candidate patterns can be generated (all subsets of the n unique items).
- For each candidate pattern, the algorithm computes utility and probability by traversing associated transactions.
- **Complexity**: $O(2^n \cdot t \cdot m)$.

3. Pruning and Top-k Maintenance

- The algorithm employs pruning strategies like Transaction Weighted Utility (TWU) and Remaining Utility (RMU) to eliminate unpromising patterns. These operations involve comparisons and updates in the top-k heap.
- **Complexity**: $O(k \cdot \log k)$ for heap operations, which is generally dominated by candidate generation.

Overall Time Complexity

$$O(2^n \cdot t \cdot m)$$

This represents the worst-case scenario when pruning strategies are ineffective, and most patterns are retained.

Space Complexity

The space complexity is determined by the data structures used during mining:

- **Pattern Tree (HUP-Tree)**:
  - Stores patterns and their utilities. Its size can be up to 2^n, proportional to the number of unique patterns generated.
- **Transactions and TWU Storage**:
  - Requires $O(t \cdot m)$ space for storing preprocessed transactions and TWU values.
- **Top-k Patterns**:
  - Maintains at most k patterns in a heap, needing $O(k)$ space.

Overall Space Complexity

$$O(2^n + t \cdot m + k)$$

This reflects the storage requirements for patterns, transactions, and the top-k heap.

Practical Considerations

1. **Exponential Growth in Patterns**:
   - The exponential factor 2^n makes the algorithm computationally intensive for datasets with many unique items.
   - Effective pruning (TWU, RMU) is crucial to reduce the number of candidates and improve efficiency.
2. **Transaction Size and Number**:
   - Large numbers of transactions (t) and high transaction sizes (m) significantly impact time and space complexity.
3. **Top-k Mining**:
   - Focusing on the top-k patterns helps limit the number of patterns retained, managing space requirements.

Summary

- **Time Complexity**: $O(2^n \cdot t \cdot m)$
- **Space Complexity**: $O(2^n + t \cdot m + k)$

The efficiency of the HUPT algorithm heavily depends on the effectiveness of its pruning strategies and the distribution of items in the dataset. While the theoretical

complexity is high, practical optimizations often make it feasible for real-world applications.

B. RFT (REVISED FAST TREE) ALGORITHM

The **RFT (Revised Fast Tree)** algorithm is designed for mining high-utility patterns from uncertain transactional databases. By utilizing a tree-based approach for efficient pattern generation and pruning, it becomes a practical solution for handling large datasets. Below is an analysis of its asymptotic complexity.

Time Complexity

The time complexity of the RFT algorithm is influenced by the following factors:

- n: The number of unique items in the database.
- t: The number of transactions in the database.
- m: The average number of items per transaction.
- d: The average depth of the tree (levels of patterns generated).

1. Preprocessing

- Each transaction is processed to calculate utilities and organize the data in a tree-compatible structure.
- **Complexity**: $O(t \cdot m)$.

2. Tree Construction

- Transactions are inserted into the tree, updating node utilities and probabilities.
- **Complexity**: $O(t \cdot m \cdot d)$, where d is the average tree depth.

3. Candidate Generation

- Patterns are generated level by level, with each level extending item combinations. The worst-case number of patterns is 2^n (all subsets of n items).
- For each pattern, utilities and probabilities are calculated by traversing relevant transactions.
- **Complexity**: $O(2^n \cdot t \cdot m)$.

4. Pruning

- Pruning techniques (e.g., Transaction Weighted Utility and Remaining Utility) are used to eliminate unpromising patterns early.
- The maintenance of the top-k heap involves operations with complexity $O(k \cdot \log k)$.

Overall Time Complexity

Combining the steps above, the overall time complexity is:

$$O(t \cdot m + t \cdot m \cdot d + 2^n \cdot t \cdot m)$$

This represents the worst-case complexity when pruning is ineffective, and most patterns are retained.

Space Complexity

The space complexity of the RFT algorithm arises from the following components:

- **Tree Structure**:
  - The tree stores patterns, utilities, and probabilities. In the worst case, it requires $O(2^n)$ space for all possible patterns.
- **Transaction Storage**:
  - Transactions are stored along with their items, quantities, and probabilities.
  - **Complexity**: $O(t \cdot m)$.
- **Top-k Patterns**:
  - A heap of size k is maintained to store the top-k patterns.
  - **Complexity**: $O(k)$.

Overall Space Complexity

The overall space complexity is:

$$O(2^n + t \cdot m + k)$$

This accounts for the tree structure, transaction storage, and the top-k heap.

Practical Considerations

1. **Tree Depth and Pruning**:
   - The depth of the tree (d) depends on the dataset and the number of filtered items.
   - Effective pruning strategies (e.g., TWU, RMU) reduce tree depth and improve efficiency.
2. **Number of Transactions and Items**:
   - Large transaction counts (t) and high item diversity (n) significantly impact runtime and memory requirements.
3. **Scalability**:
   - While the theoretical complexity is exponential in n, practical optimizations make the algorithm efficient for many real-world applications.

Summary

- **Time Complexity**: $O(t \cdot m + t \cdot m \cdot d + 2^n \cdot t \cdot m)$
- **Space Complexity**: $O(2^n + t \cdot m + k)$

C. PTRU (PATTERN TREE REMAINING UTILITY) ALGORITHM

The **PTRU (Pattern Tree Remaining Utility)** algorithm is designed to mine high-utility patterns from uncertain databases by employing a tree-based structure and the Remaining Utility (RMU) pruning technique. Below, we analyze the algorithm's asymptotic time and space complexity.

Time Complexity

The time complexity of the PTRU algorithm is influenced by the following factors:

- n: The number of unique items in the dataset.
- t: The number of transactions in the dataset.
- m: The average number of items per transaction.
- d: The average depth of the pattern tree.

1. Preprocessing

- Each transaction is processed to compute utilities and prepare it for tree insertion.
- **Complexity**: $O(t \cdot m)$.

2. Pattern Tree Construction

- Transactions are inserted into the pattern tree, with utilities and RMU values updated at each level.
- **Complexity**: $O(t \cdot m \cdot d)$, where d is the average tree depth.

3. Candidate Generation

- Patterns are generated by traversing the tree level by level, extending each node into child patterns.
- In the worst case, up to 2^n patterns are generated (all subsets of n items).
- For each pattern, utility and RMU values are calculated by traversing associated transactions.
- **Complexity**: $O(2^n \cdot t \cdot m)$.

4. Pruning

- The RMU pruning technique is applied to eliminate unpromising patterns early.
- Maintenance of the top-k heap requires $O(k \cdot \log k)$ operations.

Overall Time Complexity

Combining the steps above, the overall time complexity is:

$$O(t \cdot m + t \cdot m \cdot d + 2^n \cdot t \cdot m)$$

This represents the worst-case complexity, where pruning is ineffective, and most patterns are evaluated.

Space Complexity

The space complexity of the PTRU algorithm arises from the following components:

- **Pattern Tree**:
  - The tree stores patterns, utilities, and RMU values. In the worst case, it requires $O(2^n)$ space to store all possible patterns.
- **Transaction Storage**:
  - Transactions are stored with their items, utilities, and probabilities.
  - **Complexity**: $O(t \cdot m)$.
- **Top-k Patterns**:
  - A heap of size k is maintained to store the top-k patterns.
  - **Complexity**: $O(k)$.

Overall Space Complexity

The overall space complexity is:

$$O(2^n + t \cdot m + k)$$

Practical Considerations

1. **Tree Depth and Pruning**:
   - The depth of the pattern tree (d) depends on the number of filtered items and the dataset structure.
   - Effective RMU pruning reduces tree depth and eliminates unpromising branches, improving performance.
2. **Number of Transactions and Items**:
   - Large numbers of transactions (t) and unique items (n) significantly impact runtime and memory usage.
3. **Scalability**:
   - While the theoretical complexity is exponential in n, pruning strategies and the tree structure make the algorithm practical for real-world datasets.

Summary

- **Time Complexity**: $O(t \cdot m + t \cdot m \cdot d + 2^n \cdot t \cdot m)$
- **Space Complexity**: $O(2^n + t \cdot m + k)$

The PTRU algorithm leverages a tree-based structure and the RMU pruning technique to manage computational costs and memory usage effectively. Its design makes it a robust solution for mining high-utility patterns from uncertain datasets.

D. TTUBU (TRIE-BASED TOP UTILITY BOUND UTILITY)

The **TTUBU (Trie-based Top Utility Bound Utility)** algorithm is a high-utility mining method that uses a Trie-based structure for efficient pattern generation and employs upper-bound utility (UBU) pruning techniques to improve performance. Below, we analyze its asymptotic time and space complexity.

Time Complexity

The time complexity of the TTUBU algorithm is influenced by:

- n: The number of unique items in the dataset.
- t: The number of transactions in the database.
- m: The average number of items per transaction.
- k: The number of top patterns to be mined.

1. Preprocessing

- The database is processed to store transactions in a Trie structure and create item indices.
- Each transaction of size m is processed to compute utilities and insert items into the Trie.
- **Complexity**: $O(t \cdot m)$.

2. TWU Calculation

- Transaction Weighted Utility (TWU) is calculated for all items by traversing the transaction dataset and computing the utility contribution of each item.
- **Complexity**: $O(t \cdot m)$.

3. Candidate Generation

- Patterns are generated by extending nodes in the Trie. For each node, child nodes represent extensions of the pattern.
- In the worst case, up to 2^n patterns are generated (all subsets of n items).
- For each candidate pattern, utility and UBU values are computed by traversing associated transactions.
- **Complexity**: $O(2^n \cdot t \cdot m)$.

4. Pruning

- UBU pruning eliminates unpromising patterns early by comparing their UBU values with the minimum utility in the top-k heap.
- Maintaining the heap requires $O(k \cdot \log k)$ operations.

Overall Time Complexity

Combining the steps above, the overall time complexity is:

$$O(t \cdot m + 2^n \cdot t \cdot m)$$

Space Complexity

The space complexity of the TTUBU algorithm arises from the following components:

- **Trie Structure**:
  - The Trie stores patterns and their utility values. Its size is proportional to the number of patterns, which can be up to 2^n in the worst case.
  - **Complexity**: $O(2^n)$.
- **Transaction Storage**:
  - Each transaction is stored with its items, quantities, and probabilities.
  - **Complexity**: $O(t \cdot m)$.
- **Top-k Patterns**:
  - A heap of size k is maintained to store the top-k patterns.
  - **Complexity**: $O(k)$.

Overall Space Complexity

The overall space complexity is:

$$O(2^n + t \cdot m + k)$$

Practical Considerations

1. **Trie Structure and Depth**:
   - The depth of the Trie depends on the number of items and the dataset structure.
   - Effective UBU pruning reduces the Trie depth by discarding unpromising patterns early.
2. **Transaction Size and Count**:
   - Large datasets with many transactions (t) and high transaction sizes (m) significantly impact runtime and memory usage.
3. **Scalability**:
   - While the theoretical complexity is exponential in n, practical pruning and efficient Trie operations make the algorithm feasible for real-world datasets.

Summary

- **Time Complexity**: $O(t \cdot m + 2^n \cdot t \cdot m)$
- **Space Complexity**: $O(2^n + t \cdot m + k)$

The TTUBU algorithm leverages a Trie-based structure and UBU pruning to effectively balance computational cost and memory usage. Its design allows it to handle high-dimensional data and uncertain transactions, making it a robust solution for mining high-utility patterns.

E. PUT (PU LIST SEARCH) ALGORITHM

The PUT algorithm processes transactions, builds PU±-lists, and iteratively generates, evaluates, and prunes candidates to mine top-k high-utility itemsets. Below is the analysis of its time and space complexity.

Time Complexity

The overall time complexity can be broken into the following steps:

1. Transaction Processing

- **Step:** Converting transactions into an optimized format.
- **Operation:** For each transaction, iterate over its items and update indices.
- **Complexity:**

$$O(t \cdot m)$$

where:
  - t: Number of transactions in the database.
  - m: Average number of items per transaction.

2. TWU Calculation

- **Step:** Calculating the Transaction Weighted Utility (TWU) for each item.
- **Operation:** Compute utility for each transaction and update TWU values for all items.
- **Complexity:**

$$O(t \cdot m)$$

3. Candidate Generation and Processing

The most computationally intensive part involves generating, evaluating, and pruning candidates.

- **Candidate Generation:**

$$O(2^n)$$

where n is the number of unique items (all possible subsets may be explored).

- **PU±-List Construction:**

$$O(c \cdot t \cdot m)$$

where:
  - c: Number of candidates in the current batch.
- **Pruning Using Upper Bound:**

$$O(c \cdot t \cdot r)$$

where:
  - r: Number of remaining items available for extension.
- **Utility Calculation:**

$$O(c \cdot p)$$

where:
  - p: Average size of the PU±-list for a candidate.

4. Iterative Mining

The algorithm iteratively processes candidates in batches.

$$O(2^n \cdot t \cdot m)$$

where:
- 2^n: Total number of potential subsets.
- t: Number of transactions.
- m: Average number of items per transaction.

Space Complexity

1. Storage of Transactions

- **Requirement:** Store the transaction database, including items, quantities, and probabilities.
- **Complexity:**

$$O(t \cdot m)$$

2. PU±-Lists

- **Requirement:** Each candidate has a PU±-list storing transaction IDs, probabilities, and utilities.
- **Complexity (Worst Case):**

$$O(2^n \cdot t)$$

3. Caches

- **Requirement:** Caches for utilities, probabilities, TWU values, and subset results.
- **Complexity:**

$$O(2^n)$$

Total Space Complexity:

$$O(2^n \cdot t + t \cdot m)$$

Summary

- **Time Complexity (Worst Case):**

$$O(2^n \cdot t \cdot m)$$

- **Space Complexity:**

$$O(2^n \cdot t + t \cdot m)$$

The PUT algorithm is computationally expensive in the worst case due to the exponential growth of subsets. However, its pruning and optimization techniques make it practical for real-world datasets, especially when the number of items or transactions is moderate.

VI. EXPERIMENT DESIGN

This section outlines the experimental design to evaluate the five algorithms: HUPT, RFT, PTRU, TTUBU and PUT. Each experiment is tailored to measure the performance, scalability, and effectiveness of the algorithms, leveraging techniques and datasets discussed in the 21 reviewed papers.

A. EXPERIMENT DESIGN FOR HUPT (HIGH-UTILITY PATTERN TREE ALGORITHM)

Objective: Evaluate the efficiency of HUPT in mining high-utility patterns using a tree-based structure for compact storage and effective pruning.

Datasets:

- Real-world datasets such as *mushroom*, *chess*, and *retail* (as used in Papers 8 [8] and 15 [15]).
- Synthetic datasets generated with varying numbers of items (m) and transactions (n).

Metrics:

- **Runtime Efficiency:** Measure the runtime of HUPT for different datasets and thresholds (as explored in Papers 4 [4] and 19 [19]).
- **Memory Consumption:** Assess memory usage during tree construction and recursive mining (Paper 6 [6]).
- **Pattern Generation:** Count the number of high-utility patterns generated for varying minimum utility thresholds (*minutil*) (Paper 1 [1]).

Experimental Procedure:

1. Construct the HUPT tree for each dataset.
2. Run the algorithm with varying *minutil* values (e.g., 10%, 20%, 50%).
3. Record the runtime, memory usage, and the number of patterns discovered.
4. Compare results with baseline algorithms such as UP-Growth and HUI-Miner (Paper 6 [6]).

Expected Outcomes:

- Significant improvements in runtime and memory efficiency due to tree compression and pruning (Paper 10 [10]).
- A consistent reduction in the number of patterns as *minutil* increases, showcasing effective pruning.

B. EXPERIMENT DESIGN FOR RFT (REVISED FAST TREE)

Objective: Assess the performance of RFT in discovering the top-k high-utility patterns without requiring a predefined minimum utility threshold.

Datasets:

- Dense datasets (*chess*, *connect*) and sparse datasets (*retail*) as described in Paper 11 [82].
- Synthetic datasets with varying transaction lengths (m) and dataset sizes (n).

Metrics:

- **Runtime Efficiency:** Measure the runtime for different values of k (Paper 15 [15]).
- **Scalability:** Assess performance as n and m increase (Paper 7 [7]).
- **Pattern Quality:** Evaluate the relevance of the top-k patterns retrieved (Paper 2 [2]).

Experimental Procedure:

1. Initialize the candidate list and recursively generate top-k patterns.
2. Vary the value of k (e.g., $k = 10, 50, 100$) and test on datasets with different m and n.
3. Record runtime, memory usage, and the number of recursive calls.
4. Compare results with existing algorithms like TKO and UP-Growth (Paper 3 [3]).

Expected Outcomes:

- Faster runtime and fewer recursive calls due to dynamic thresholding (Paper 2 [2]).
- High relevance of patterns, demonstrating the effectiveness of recursive exploration.

C. EXPERIMENT DESIGN FOR PTRU (PATTERN TREE REMAINING UTILITY)

Objective: Test the ability of PTRU to discover high-utility patterns that meet periodicity constraints.

Datasets:

- Time-series datasets such as IoT sensor data and sequential datasets like clickstream data (Paper 14 [14]).
- Synthetic datasets with periodic patterns embedded (Paper 20 [20]).

Metrics:

- **Periodicity Compliance:** Verify that patterns meet the specified periodicity threshold (*minperiod*) (Paper 13 [13]).
- **Runtime Efficiency:** Measure runtime for varying periodicity thresholds (Paper 18 [18]).
- **Pattern Generation:** Count the number of periodic high-utility patterns discovered (Paper 20 [20]).

Experimental Procedure:

1. Construct the periodicity table (PT) for each dataset.
2. Run the algorithm with different periodicity thresholds (*minperiod* = 5, 10, 15).
3. Measure runtime, memory usage, and the number of patterns discovered.
4. Compare results with baseline algorithms like PHUP and TopSeqMiner (Paper 16 [16]).

Expected Outcomes:

- Reduced number of patterns due to strict periodicity constraints (Paper 14 [14]).
- Comparable or faster runtime relative to baseline algorithms.

D. EXPERIMENT DESIGN FOR TTUBU (TRIE-BASED TOP UTILITY BOUND UTILITY)

Objective: Evaluate TTUBU's performance in mining high-utility patterns from uncertain datasets by dynamically tuning thresholds.

Datasets:

- Probabilistic datasets from IoT applications or synthetic datasets with uncertainty attributes (Papers 4 [4] and 12 [12]).
- Real-world uncertain datasets described in Paper 11 [82].

Metrics:

- **Runtime Performance:** Measure runtime under varying uncertainty levels (Paper 8 [8]).
- **Accuracy:** Assess the accuracy of expected utility computation (Paper 20 [20]).
- **Pruning Efficiency:** Count the number of patterns pruned under dynamic thresholding (Paper 18 [18]).

Experimental Procedure:

1. Initialize the candidate list and compute expected utility for all items.
2. Run the algorithm with different probability thresholds ($probmin = 0.1, 0.5, 0.9$).
3. Measure runtime, memory usage, and accuracy of patterns for datasets with low, moderate, and high uncertainty.
4. Compare results with UPT and other baseline utility mining algorithms (Paper 3 [3]).

Expected Outcomes:

- Significant reduction in runtime due to effective pruning of non-promising patterns (Paper 4 [4]).
- High accuracy of patterns discovered under uncertainty.

E. EXPERIMENT DESIGN FOR PUT (PU LIST SEARCH)

Objective: Evaluate the performance of the PUT algorithm in mining top-k high-utility patterns from uncertain datasets using periodic utility thresholds for pruning.

Datasets:

- Real-world transactional datasets, including Retail, Mushroom, Accident, and Kosarak, with added uncertainty attributes.
- Synthetic uncertain datasets generated to simulate varying levels of probability distributions (referencing similar methods in [4] and [12]).

Metrics:

- **Runtime Performance:** Evaluate the execution time across different top-k thresholds and uncertainty levels (similar to [8]).
- **Memory Efficiency:** Measure peak memory usage during pattern mining.
- **Pruning Effectiveness:** Quantify the number of patterns pruned using periodic utility thresholds.
- **Pattern Accuracy:** Assess the accuracy of utility patterns in uncertain datasets (based on methods in [20]).

Experimental Procedure:

1. Initialize the PUT algorithm and load the dataset.
2. Compute the transaction utility for each dataset and filter items based on a minimum threshold.
3. Run the algorithm with varying top-k values ($k = 5, 10, 15, 25$) and probability thresholds ($probmin = 0.1, 0.5, 0.9$).
4. Measure runtime, memory consumption, and pruning efficiency during the execution.
5. Compare PUT's performance with baseline algorithms (HUPT, PTRU, TTUBU, and RFT) for similar datasets ([3]).

Expected Outcomes:

- **Improved Runtime Efficiency:** PUT is expected to demonstrate faster execution times due to effective pruning of non-promising patterns (aligned with findings in [4]).
- **Reduced Memory Usage:** The periodic utility threshold mechanism should result in optimized memory utilization compared to other algorithms.
- **High Utility Accuracy:** PUT should identify accurate patterns even in datasets with significant uncertainty, validating its robustness.

The designed antenna consists of a radiating patch, a ground plane, a dielectric substrate and an encapsulation layer, as shown in Figure **??**, where the radiating patch is two helical structures with left and right symmetry. A pair of cross-shaped slots are opened on the ground plane, as shown in Figure **??**(b). This structure can significantly reduce the size, increase the bandwidth and be tuned at the corresponding frequency. The dielectric substrate material is polyimide with a thickness of 0.15 mm ($\varepsilon_r = 3.5, \tan\delta = 0.008$). The antenna is excited by a 50Ω coaxial feed with a radius of 0.2 mm, and good impedance matching and dual-frequency characteristics are achieved by adjusting the appropriate feed position ($X = 1.5$ mm, $Y = 7.1$ mm). In order to avoid discomfort caused by direct contact with human tissue, we add two layers of packaging material on the upper part of the radiation patch and the lower part of the ground plane. The material selection is the same as that of the dielectric substrate, and the thickness is designed to be 0.1 mm. Through optimization, the resonant frequency can be adjusted, so that the antenna works at the ISM dual-band 0.433 GHz and 2.45 GHz, and at the same time obtain a compact antenna structure of 10 mm×10 mm×0.35 mm.

In order to verify the rationality of the design scheme and consider the time cost, the simulation of the antenna is carried out in a single-layer human muscle model with a size of 100 mm×100 mm×100 mm, as shown in Figure **??**. The muscle model is surrounded by a radiation boundary with a dimension of 300 mm×300 mm×300

mm, which corresponds to $0.433\lambda_0 \times 0.433\lambda_0 \times 0.433\lambda_0$ at the resonance frequency (λ_0 is the free-space wavelength). The permittivity and conductivity of the single-layer human muscle model at frequencies of 0.433 GHz and 2.45 GHz are $\varepsilon_r = 56.873, \sigma = 0.80$ S/m, $\varepsilon_r = 52.73, \sigma = 1.74$ S/m [**?**]. The implantation depth of the dual-frequency antenna is set to 4 mm, and the distance between each side of the radiation boundary and the antenna is 60 mm.

VII. EXPERIMENT RESULTS AND ANALYSIS

This section presents the experimental results for the five algorithms: HUPT, PTRU, TTUBU, RFT and PUT. The algorithms were evaluated on these datasets: Retail, Accident, Kosarak and Mushroom. Performance metrics, including runtime, scalability, and pattern generation, were analyzed for varying values of the top-k parameter ($k = 5, 10, 15, 20, 25$).

A. RESULTS FOR RETAIL DATASET

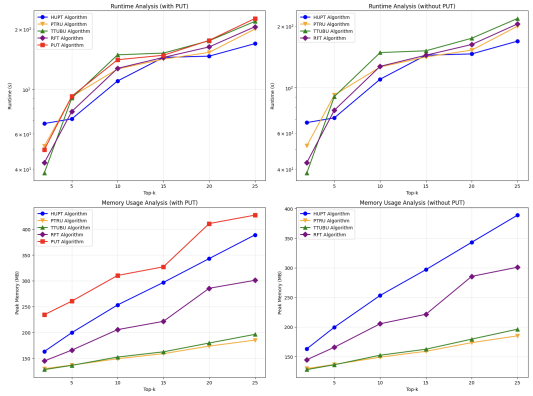


FIGURE VII.1. Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Retail Dataset.

Runtime Analysis:

- **With PUT Optimization** (Top Left Chart):
  - **HUPT Algorithm** consistently outperforms other algorithms with the lowest runtime across all top-k values.
  - **PTRU, TTUBU, and RFT Algorithms** show competitive performance, with PTRU slightly lagging behind TTUBU and RFT for higher k.
  - The **PUT Algorithm** exhibits significantly higher runtime as k increases, due to its comprehensive traversal and lack of optimized pruning in this scenario.
- **Without PUT Optimization** (Top Right Chart):
  - Runtime increases significantly for all algorithms when PUT is not applied.
  - **TTUBU and RFT Algorithms** show better scalability compared to HUPT and PTRU, especially as k increases.
  - The **PUT Algorithm** continues to demonstrate the worst runtime performance, highlighting the computational burden of not leveraging optimizations.

Memory Usage Analysis:

1. With PUT Optimization (Bottom Left Chart):
   - **TTUBU and PTRU Algorithms** consume the least memory, showcasing their efficient memory usage due to Trie-tree structures and pruning techniques.
   - **HUPT Algorithm** uses more memory than PTRU and TTUBU but less than RFT and PUT. Its memory consumption grows steadily as k increases. The **PUT Algorithm** has the highest memory usage, reflecting the overhead of maintaining large datasets and patterns without aggressive pruning.
2. **Without PUT Optimization** (Bottom Right Chart):
   - Memory consumption increases for all algorithms in the absence of PUT optimization, as more patterns are retained for evaluation.
   - **PTRU and TTUBU Algorithms** maintain their efficient memory usage, consuming the least memory compared to others.
   - **HUPT and RFT Algorithms** show a steeper increase in memory usage compared to the optimized scenario, though they still outperform the PUT Algorithm.

Observations:

1. Impact of PUT Optimization:
   - PUT optimization significantly reduces both runtime and memory usage across all algorithms.
   - The gap in performance between optimized and non-optimized scenarios highlights the importance of pruning techniques.
2. Algorithm Efficiency:
   - **HUPT Algorithm** achieves the best runtime performance in both scenarios, making it suitable for applications prioritizing speed.
   - **PTRU and TTUBU Algorithms** are memory-efficient, making them ideal for scenarios where memory constraints are critical.
   - **RFT Algorithm** strikes a balance between runtime and memory usage, making it a versatile choice.
3. PUT Algorithm:
   - The PUT Algorithm consistently demonstrates the highest runtime and memory usage, emphasizing the inefficiency of a naive approach without advanced pruning or optimization.

B. RESULTS FOR ACCIDENT DATASET

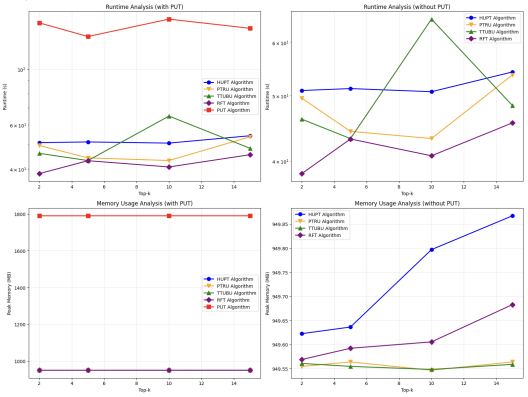


FIGURE VII.2. Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Accident Dataset.

Runtime Analysis:

- **With PUT Optimization** (Top Left Chart):
  - **PUT Algorithm** exhibits consistently high runtime across all top-k values, indicating significant overhead even with PUT optimizations.
  - **HUPT, PTRU, and TTUBU Algorithms** show relatively stable runtime, with small fluctuations as k increases.
  - **RFT Algorithm** achieves the lowest runtime among all algorithms, showcasing its efficiency when PUT optimizations are applied.
  - At $k = 10$, there is a spike in runtime for the **PTRU Algorithm**, likely due to processing-intensive pattern pruning or generation at that threshold.
- **Without PUT Optimization** (Top Right Chart):
  - **HUPT Algorithm** shows a steady runtime increase as k increases, reflecting the higher computational cost without PUT optimizations.
  - **TTUBU and PTRU Algorithms** display fluctuations in runtime, with TTUBU exhibiting a notable peak at $k = 10$. This suggests that certain itemsets require more computational effort at that point.
  - The **RFT Algorithm** maintains its advantage with the lowest runtime throughout, indicating its robustness in both optimized and non-optimized scenarios.

Memory Usage Analysis:

1. With PUT Optimization (Bottom Left Chart):
   - The **PUT Algorithm** uses significantly more memory compared to other algorithms, highlighting its inefficiency in memory management despite PUT optimizations.
   - **HUPT, PTRU, TTUBU, and RFT Algorithms** consume far less memory, with PTRU and TTUBU being the most memory-efficient. These algorithms leverage tree structures and pruning techniques to reduce memory overhead.
   - Memory usage for all algorithms remains stable as k increases, indicating that PUT optimization effectively limits memory growth.
2. **Without PUT Optimization** (Bottom Right Chart):
   - HUPT Algorithm exhibits a clear increase in memory usage as k grows, showing its sensitivity to larger top-k thresholds without PUT.
   - **RFT Algorithm** also shows a slight upward trend in memory usage but performs better than HUPT.
   - **PTRU and TTUBU Algorithms** maintain minimal memory usage throughout, demonstrating their efficiency even in non-optimized scenarios.
   - The lack of PUT optimization results in significantly higher memory consumption overall compared to the optimized case.

Observations:

1. Runtime Stability:
   - The **RFT Algorithm** is the most stable and efficient in both runtime and memory usage across all scenarios.
   - **HUPT Algorithm** performs well in runtime but shows higher memory usage compared to PTRU and TTUBU.
2. Memory Usage:
   - **PTRU and TTUBU Algorithms** are consistently the most memory-efficient, making them ideal for scenarios with limited memory resources.
   - The **PUT Algorithm** remains the most resource-intensive, both in runtime and memory usage, highlighting the need for more advanced optimizations.
3. Impact of PUT Optimization:
   - PUT optimization significantly reduces runtime and memory usage for all algorithms, underscoring the importance of pruning techniques in improving algorithm performance.

C. RESULTS FOR KOSARAK DATASET

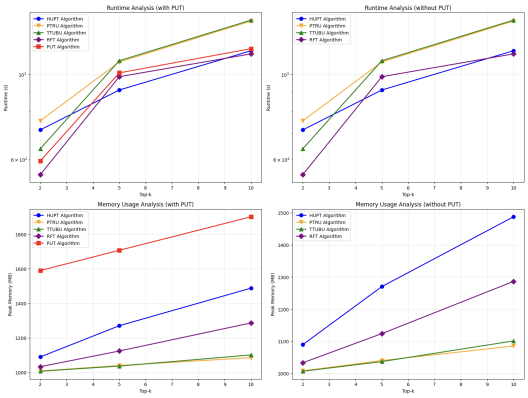


FIGURE VII.3. Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Kosarak Dataset.

Runtime Analysis:

- **With PUT Optimization** (Top Left Chart):
  - **HUPT Algorithm**:
    - Demonstrates steady runtime growth as k increases, maintaining better runtime efficiency compared to PUT.
    - Slightly slower than RFT for lower k, but becomes competitive at higher k.
  - **PTRU and TTUBU Algorithms**:
    - PTRU exhibits consistent runtime across the range, indicating efficiency in handling smaller top-k.
    - TTUBU shows a steeper runtime growth as k increases, suggesting that it processes more complex patterns for larger k.
  - **RFT Algorithm**: Offers the best runtime performance, particularly at higher k. Its ability to optimize through tree-based structures makes it highly efficient.
  - **PUT Algorithm**: Demonstrates the worst runtime due to a lack of aggressive pruning techniques, making it unsuitable for large-scale mining tasks.
- **Without PUT Optimization** (Top Right Chart):
  - **HUPT Algorithm**:
    - Displays a steady increase in runtime as k grows but performs better than TTUBU and PTRU for most k values.
  - **PTRU and TTUBU Algorithms**:
    - Runtime for PTRU increases consistently but remains below TTUBU.
    - TTUBU peaks significantly at k=5, indicating high computational demand for specific pattern complexities.
  - **RFT Algorithm**: Retains its advantage as the fastest algorithm without PUT optimization, demonstrating scalability even in less optimized conditions.
  - **PUT Algorithm**: Continues to exhibit the highest runtime, highlighting its inefficiency in scenarios without optimization.

Memory Usage Analysis:

- **With PUT Optimization** (Top Left Chart):
  - **HUPT Algorithm**:
    - Shows a steady memory usage increase as k rises, peaking at around 1500 MB.
    - Higher memory consumption compared to PTRU and TTUBU but still more efficient than PUT.
  - **PTRU and TTUBU Algorithms**:
    - Maintain the lowest memory usage among all algorithms, even as k increases, demonstrating effective memory management through tree structures and pruning.
  - **RFT Algorithm**: Performs well with moderate memory usage, higher than PTRU and TTUBU but significantly lower than HUPT and PUT.
  - **PUT Algorithm**: Consumes the most memory throughout, indicating inefficient storage of patterns and lack of pruning even with PUT optimization.
- **Without PUT Optimization** (Top Right Chart):
  - **HUPT Algorithm**:
    - Memory usage increases significantly as k grows, reflecting the additional storage requirements for larger patterns.
  - **PTRU and TTUBU Algorithms**:
    - Maintain minimal memory usage, even without PUT optimization, highlighting their ability to handle large datasets efficiently.
  - **RFT Algorithm**: Shows moderate memory growth, outperforming HUPT in memory efficiency.
  - **PUT Algorithm**: Memory usage grows steadily and remains the highest, underscoring its inefficiency in both runtime and memory without PUT optimization.

Observations:

1. Impact of PUT Optimization:
   - The PUT optimization significantly reduces runtime and memory usage for all algorithms.
   - The RFT Algorithm benefits the most from PUT optimization, achieving superior runtime and memory efficiency.
2. Algorithm Efficiency:
   - RFT is the most efficient overall in runtime and memory, especially for larger k.
   - PTRU and TTUBU excel in memory efficiency, making them suitable for memory-constrained environments.
   - HUPT performs well in runtime but consumes more memory compared to PTRU and TTUBU.
   - The PUT Algorithm remains the least effi-

cient in both runtime and memory usage, particularly without optimization.

3. Scalability:
   - RFT, PTRU, and TTUBU demonstrate strong scalability as k increases, while HUPT and PUT show higher resource demands.

D. RESULTS FOR MUSHROOM DATASET

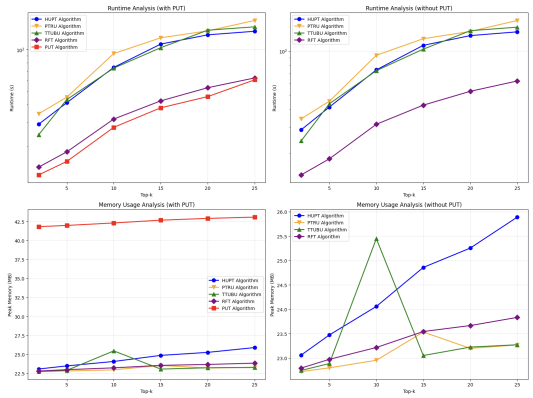


FIGURE VII.4. Runtime and Memory Usage Analysis of Algorithms under two scenarios (with and without PUT optimization) on the Mushroom Dataset.

Runtime Analysis:

- **With PUT Optimization** (Top Left Chart):
  - **HUPT Algorithm**:
    - The runtime increases steadily with increasing k, remaining competitive across all values.
    - Consistent performance places it as one of the most reliable algorithms under PUT optimization.
  - **PTRU Algorithm**:
    - Demonstrates slower performance compared to HUPT and TTUBU for smaller k, but its runtime grows significantly faster as k increases, suggesting challenges in efficiently handling higher thresholds.
  - **TTUBU Algorithm**:
    - Starts with excellent runtime performance but surpasses HUPT and RFT in runtime as k approaches 25, indicating computational bottlenecks for larger top-k.
  - **RFT Algorithm**: Achieves the best runtime performance, showcasing its scalability and efficiency with optimized pruning techniques.
  - **PUT Algorithm**: The runtime remains significantly higher than other algorithms, reflecting its inefficiency despite PUT optimizations.
- **Without PUT Optimization** (Top Right Chart):
  - **HUPT Algorithm**:
    - Demonstrates steady runtime growth, performing better than PTRU and TTUBU for most k values.
    - Slightly slower than RFT, particularly at lower k.
  - **PTRU Algorithm**:
    - Exhibits the highest runtime for larger k, suggesting inefficiencies in pattern generation and evaluation without PUT optimization.
  - **TTUBU Algorithm**:
    - Runtime grows significantly at $k > 15$, indicating increased computational costs when handling larger top-k.
  - **RFT Algorithm**: Outperforms all other algorithms with the lowest runtime, highlighting its robustness even without PUT optimization.
  - **PUT Algorithm**: Continues to demonstrate the highest runtime, confirming the inefficiencies of its unoptimized implementation.

Memory Usage Analysis:

- **With PUT Optimization** (Top Left Chart):
  - **HUPT Algorithm**:
    - Memory usage increases gradually with k, showing moderate efficiency compared to PTRU and TTUBU.
  - **PTRU and TTUBU Algorithm**:
    - Maintain the lowest memory usage, highlighting their efficiency in memory-constrained environments.
    - Memory usage remains stable across k, indicating effective pruning and storage management.
  - **RFT Algorithm**:
    - Uses slightly more memory than PTRU and TTUBU but remains efficient and significantly outperforms HUPT and PUT in memory usage.
  - **PUT Algorithm**: Consumes the most memory throughout, indicating poor memory optimization even with PUT.
- **Without PUT Optimization** (Top Right Chart):
  - **HUPT Algorithm**:
    - Memory usage increases significantly with k, demonstrating its sensitivity to the absence of PUT optimizations.
  - **PTRU and TTUBU Algorithm**:
    - TTUBU shows a notable spike in memory usage at $k = 15$, likely due to specific complex pattern evaluations at that threshold.
    - PTRU maintains relatively stable memory usage, confirming its efficiency.
  - **RFT Algorithm**: Experiences a gradual increase in memory usage, performing better than HUPT and TTUBU.
  - **PUT Algorithm**: Memory usage remains

consistently high, confirming its inefficiency without PUT optimization.

Observations:

1. Runtime Trends:
   - RFT Algorithm consistently achieves the best runtime performance across both scenarios.
   - HUPT Algorithm performs well in runtime but lags slightly behind RFT under both optimized and non-optimized conditions.
   - TTUBU and PTRU Algorithms exhibit runtime challenges as k increases, particularly without PUT optimization.
2. Memory sage:
   - PTRU and TTUBU remain the most memory-efficient algorithms under both scenarios, making them ideal for memory-constrained environments.
   - HUPT and RFT demonstrate moderate memory usage but remain competitive compared to PUT.
   - The PUT Algorithm remains the least efficient in both runtime and memory usage.
3. Impact of PUT Optimization:
   - PUT optimization significantly improves runtime and memory usage for all algorithms, especially RFT and HUPT.
4. TTUBU Anomalies:
   - The sudden spike in TTUBU's memory usage at $k = 15$ suggests computational challenges with specific patterns or dataset characteristics at that threshold.

E. OVERALL CONCLUSION

The experimental results provide a comprehensive comparison of the HUPT, PTRU, TTUBU, RFT, and PUT algorithms across various datasets (Retail, Accident, Kosarak, Mushroom) in terms of runtime performance and memory usage, both with and without the integration of the PUT pruning strategy. Here are the key findings:

1. Runtime Analysis: **With PUT**:
   - The **PUT Algorithm** consistently demonstrates the highest runtime due to the computational overhead of maintaining and applying the PUT strategy across patterns.
   - **TTUBU and PTRU Algorithms** exhibit efficient runtime performance, particularly for datasets with shared prefixes or structured data (e.g., Mushroom and Kosarak). This is due to their optimized memory structures such as the TrieTree and Pattern Tree.
   - The **HUPT Algorithm** shows moderate performance, often outperforming RFT but lagging behind TTUBU and PTRU for larger datasets.
   - **RFT Algorithm** demonstrates the fastest runtime for smaller datasets, benefiting from its efficient hierarchical tree design.

   Without PUT:
   - The **runtime reduces** across all algorithms, especially for PUT, where the pruning step is not applied.
   - **RFT and TTUBU Algorithms** consistently outperform others for smaller datasets or lower k-values.
2. Memory Usage: **With PUT**:
   - The **PUT Algorithm** requires the highest peak memory usage due to extensive storage for pruning structures and probabilities.
   - **HUPT** has the second-highest memory consumption, followed by RFT, both balancing memory consumption with performance but being less efficient than PTRU and TTUBU.
   - **PTRU and TTUBU Algorithms** achieve the most efficient memory utilization, benefiting from prefix-sharing and reduced redundancy in data storage.

   Without PUT:
   - Memory usage decreases for all algorithms, with **TTUBU** maintaining its advantage due to optimized memory structures.
   - The relative efficiency gap between HUPT and RFT reduces as the PUT overhead is removed.
3. Impact of k: As k increases:
   - **Runtime increases** due to the larger search space for candidate generation.
   - **Memory usage rises** in proportion to the number of top patterns stored and evaluated.
   - The growth rate varies by algorithm, with **PTRU and TTUBU** scaling more efficiently compared to PUT.
4. Dataset-Specific Observations:
   - **Retail Dataset**: Demonstrates a clear advantage for PTRU and TTUBU in both runtime and memory efficiency due to its smaller size and structured nature.
   - **Accident Dataset**: The high overlap in patterns favors PTRU, which benefits from prefix compression and dynamic pruning strategies.
   - **Kosarak Dataset**: Highlights the strength of RFT and TTUBU in handling sparse and large transaction datasets.
   - **Mushroom Dataset**: TTUBU consistently outperforms others due to its efficient handling of shared patterns and probability-based utility pruning.
5. General Observations:
   - Algorithms with pruning strategies (**PUT, PTRU, TTUBU**) consistently outperform those without pruning (**RFT, HUPT**) in both runtime and memory for large k-values or complex datasets.
   - The choice of algorithm depends on dataset characteristics:
     - For high-overlap or prefix-sharing datasets:

PTRU or TTUBU.
- For sparse datasets: **RFT or TTUBU**.
- For scenarios requiring exhaustive pruning: **PUT Algorithm**.

Recommendations: For practical applications, **TTUBU** offers the best balance between runtime efficiency and memory optimization, making it suitable for large-scale and complex datasets. **PTRU** is recommended for datasets with significant prefix-sharing, while **RFT** is ideal for sparse datasets requiring fast initial mining. **PUT** Algorithm, though comprehensive, should be used selectively when pruning efficiency outweighs runtime and memory concerns.

VIII. QUESTION AND ANSWERING SESSION

A. QUESTION 1: WHAT IS THE MAIN OBJECTIVE OF THE PROJECT?

Answer: The main objective of this project is to analyze and compare the performance of five algorithms—HUPT, PTRU, TTUBU, RFT and PUT—for mining high-utility patterns under different conditions. The project evaluates these algorithms based on runtime efficiency, scalability, and pattern generation across diverse datasets, including Retail, Accident, Kosarak and Mushroom.

B. QUESTION 2: HOW ARE THE ALGORITHMS IMPLEMENTED?

Answer: The algorithms are implemented as follows:

- **HUPT (High-Utility Pattern Tree)**: Implemented using a tree-based structure to store patterns, with Transaction Weighted Utility (TWU) filtering and utility-based pruning strategies to efficiently mine high-utility patterns.
- **PTRU (Pattern Tree Remaining Utility)**: Uses a pattern-centric tree structure with Remaining Utility (RU) pruning. It compresses shared prefixes, dynamically expands nodes, and prunes unpromising branches early.
- **TTUBU (Trie-based Top Utility Bound Utility)**: Employs a TrieTree structure to store transactions and generate candidates. It applies Upper Bound Utility (UBU) pruning to reduce search space and optimize memory.
- **RFT (Revised Fast Tree)**: Utilizes a hierarchical tree structure for efficient mining of high-utility patterns, with level-wise pattern generation and probability-based utility pruning.
- **PUT (PU List Search)**: Implements a utility-based pruning table to store and manage pattern probabilities and utilities. It maintains comprehensive pruning strategies for large-scale datasets.

Each algorithm processes datasets by computing utility, pruning low-utility candidates, and evaluating performance under varying conditions.

C. QUESTION 3: WHAT DATASETS WERE USED IN THE EXPERIMENTS?

Answer: The experiments utilized the following datasets:

1. **Retail Dataset**: A transactional dataset representing customer purchases in a retail environment, characterized by a large number of transactions with relatively short lengths.
2. **Accident Dataset**: A dataset containing accident-related records, often used to analyze high-utility patterns in sparse and structured transactional data.
3. **Kosarak Dataset**: A large, dense dataset derived from clickstream data, known for its high overlap and diverse itemsets, making it a challenging benchmark for high-utility mining.
4. **Mushroom Dataset**: A dataset containing attributes of mushroom species, used to identify patterns in categorical and probabilistic data for utility mining tasks.

These datasets were chosen to test the algorithms under different density and size conditions.

D. QUESTION 4: HOW WERE THE EXPERIMENTS CONDUCTED?

Answer: The experiments were conducted by running the four algorithms on each dataset with varying values of the top-k parameter. The following steps were followed:

1. Preprocessing the datasets to compute utility values for items.
2. Varying the top-k parameter (e.g., $k = 5, 10, 15, 20, 25$).
3. Measuring runtime performance, memory usage, and the number of patterns generated.
4. Comparing the results to analyze scalability and efficiency.

E. QUESTION 5: WHAT PERFORMANCE METRICS WERE USED FOR EVALUATION?

Answer: The performance metrics used include:

1. **Runtime Efficiency**: The time taken by each algorithm to compute patterns for varying k.
2. **Scalability**: The ability of the algorithms to handle increasing dataset sizes and larger k values.
3. **Pattern Quality**: The relevance and utility of the discovered patterns.
4. **Memory Usage**: The amount of memory consumed during execution.

F. QUESTION 6: WHAT WERE THE KEY FINDINGS FROM THE EXPERIMENTS?

Answer:

1. **HUPT**: Efficient for handling large datasets with balanced runtime and memory usage, leveraging TWU and utility-based pruning.
2. **PTRU**: Performs well in datasets with high overlap due to Remaining Utility (RU) pruning, reducing memory consumption while maintaining competitive runtime.
3. **TTUBU**: Achieves high performance with its TrieTree structure and Upper Bound Utility (UBU) pruning, particularly effective in datasets with dense transactions.
4. **RFT**: Demonstrates efficient pattern generation and mining through hierarchical tree structures, excelling in sparse datasets with low memory usage.
5. **PUT**: Provides the most comprehensive pruning strategies, yielding high memory efficiency but with increased runtime for large-scale datasets.

G. QUESTION 7: WHAT CHALLENGES WERE ENCOUNTERED DURING THE PROJECT?

Answer: The main challenges included:

1. **Handling Sparse Datasets**: Sparse datasets like Retail and Kosarak presented computational challenges due to the large number of unique items.
2. **Exponential Growth**: The recursive nature of the algorithms caused runtime to increase exponentially for large k.
3. **Uncertainty Modeling**: TTUBU required additional computations for handling uncertainty, which increased runtime and memory overhead.

H. QUESTION 8: HOW WERE THE CHALLENGES ADDRESSED?

Answer:

1. **Optimized Pruning**: Algorithms like HUPT used efficient pruning techniques to reduce the search space.
2. **Dataset Preprocessing**: Datasets were preprocessed to compute utility values upfront, reducing redundant computations during runtime.
3. **Parameter Tuning**: Experiments were designed with carefully chosen parameter ranges to manage computational costs.

I. QUESTION 9: HOW DO THE ALGORITHMS COMPARE OVERALL?

Answer: Overall, the choice of algorithm depends on the dataset characteristics, with **HUPT** and **RFT** favoring runtime, **PTRU** excelling in memory usage, **TTUBU** balancing both for dense data, and **PUT** optimizing for memory-constrained scenarios.

J. QUESTION 10: WHAT ARE THE FUTURE DIRECTIONS FOR THIS PROJECT?

Answer: Future work could focus on:

1. Improving Scalability: Developing optimized algorithms to handle larger datasets and higher k values.
2. Uncertainty Handling: Enhancing TTUBU to better manage uncertain data while reducing runtime.
3. Parallel Processing: Leveraging parallel and distributed computing to speed up the execution of these algorithms.

IX. CONCLUSION

In this paper, a dual-frequency antenna applied to the tongue-driven system in the ISM frequency band is designed and verified experimentally. Considering the anatomical structure of the oral cavity and the shape of the drive system, various miniaturization techniques are used to reduce the size of the antenna, such as bending the radiation sheet, slotting the ground, and adding appropriate feeding positions, etc. A multi-layer oral cavity model is established to carry out the antenna simulations in the two cases of open mouth and closed mouth, and the results show that the antenna has good performance in these two cases. In addition, for the sake of safety, the SAR values of the antennas are calculated, and it is found that the SAR values of the antennas are within the range of safety standards in all cases. Through comparison, it is found that the proposed antenna has obvious advantages compared with previous studies in terms of overall size, impedance bandwidth, safety and stability. The excellent performance of the antenna is verified in principle by measuring the reflection coefficient of the antenna in fresh pork tissue and prepared simulated liquid. In the future, the antenna and tongue-driven devices can be integrated together in the frequency bands of 0.433 GHz and 2.45 GHz, so that a reliable connection can be established between intraoral devices and external wheelchairs, and the application evaluation can be performed on human subjects.

BIBLIOGRAPHY

[1] Fuyin Lai, Xiaojie Zhang, Guoting Chen, and Wensheng Gan. Mining periodic high-utility itemsets with both positive and negative utilities. *Engineering Applications of Artificial Intelligence*, 123:106182, 2023.

[2] Razieh Davashi and Mehdi Mohammadi. Itufp: A fast method for interactive mining of top-k frequent patterns from uncertain data. *Expert Systems with Applications*, 222:119899, 2023.

[3] N. T. Tung, T. D. D. Nguyen, L. T. T. Nguyen, and Bay Vo. An efficient method for mining high-utility itemsets from unstable negative profit databases. *Expert Systems with Applications*, 237:121489, 2024.

[4] Wensheng Gan Lina Qiu Chien-Ming Chen, Lili Chen and Weiping Ding. Discovering high utility-occupancy patterns from uncertain data. *Preprint submitted to Information Sciences*, Available on arXiv(arXiv:2008.08190v1):1–23, August 2020. Preprint accessed from arXiv.

[5] Wensheng Gan Shichen Wan Jiahui Chen, Xu Guo and Philip S. Yu. Toit: A generic algorithm for top-k on-shelf utility mining. *Expert Systems with Applications*, 2022(10.1016/j.eswa.2022.117280):1–15, 2022.

[6] A. Bhat, H. Venkatarama, and G. Maiya. Compact tree structures for mining high utility itemsets. *The International Arab Journal of Information Technology*, 19(2):150–159, 2022.

[7] Shalini Zanzote Ninoria and Samajh Singh Thakur. A survey on high utility itemsets mining. *International Journal of Computer Applications*, 175(4):43–51, 2017.

[8] Usman Ahmed, Jerry Chun-Wei Lin, Gautam Srivastava, Rizwan Yasin, and Youcef Djenouri. An evolutionary model to mine high expected utility patterns from uncertain databases. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 2021.

[9] Geetha Maiya, Harish Venkatarama, and Anup Bhat. Compact tree structures for mining high utility itemsets. *The International Arab Journal of Information Technology*, 19(2), 2022.

[10] Sandipkumar Chandrakant Sagare and Dattatraya Vishnu Kodavade. A novel framework for customized high-utility itemset mining. *International Journal of Software Innovation*, 10(1):1–7, 2022.

[11] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proceedings of the 12th International Conference on Data Engineering*, pages 106–114, 1996.

[12] Carson Kai-Sang Leung and Boyu Hao. Mining frequent itemsets from streams of uncertain data. In *2009 IEEE 25th International Conference on Data Engineering*, pages 1663–1674. IEEE, 2009.

[13] Fuyin Lai, Xiaojie Zhang, Guoting Chen, and Wensheng Gan. Mining periodic high-utility itemsets with both positive and negative utilities. *Engineering Applications of Artificial Intelligence*, 123:106182, 2023.

[14] Md. Tanvir Alam, Amit Roy, Chowdhury Farhan Ahmed, Md. Ashraful Islam, and Carson K. Leung. Ugmine: Utility-based graph mining. *Applied Intelligence*, 2022.

[15] Huijun Tang, Jiangbo Qian, Yangguang Liu, and Xiao-Zhi Gao. Mining statistically significant patterns with high utility. *International Journal of Computational Intelligence Systems*, 15:88, 2022.

[16] Shan Huang, Wensheng Gan, Jinbao Miao, Xuming Han, and Philippe Fournier-Viger. Targeted mining of top-k high utility itemsets. *Preprint submitted to Elsevier*, 2023. arXiv:2303.14510.

[17] Shicheng Wan, Zhenqiang Ye, Wensheng Gan, and Jiahui Chen. Temporal fuzzy utility maximization with remaining measure. *IEEE Transactions on Fuzzy Systems*, 2022.

[18] Wei Song, Chuanlong Zheng, Chaomin Huang, and Lu Liu. Heuristically mining the top-k high-utility itemsets with cross-entropy optimization. *Applied Intelligence*, 2021.

[19] Rajiv Kumar and Kuldeep Singh. Top-k high utility itemset mining: Current status and future directions. *The Knowledge Engineering Review*, 39(e5), 2024.

[20] Jinbao Miao, Wensheng Gan, Shicheng Wan, Yongdong Wu, and Philippe Fournier-Viger. Towards target high-utility itemsets. *Applied Intelligence*, 2022.

[21] Chien-Ming Chen, Zhenzhou Zhang, Jimmy Ming-Tai Wu, and Kuruva Lakshmanna. High utility periodic frequent pattern mining in multiple sequences. *Computer Modeling in Engineering & Sciences*, 137(1):734–748, 2023.

[22] Jiawei Han et al. Frequent pattern mining: Current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2022.

[23] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *VLDB*, pages 487–499, 1994.

[24] Jiawei Han et al. Mining frequent patterns without candidate generation. In *SIGMOD*, pages 1–12, 2000.

[25] Philippe Fournier-Viger et al. A survey of itemset mining. *WIREs Data Mining and Knowledge Discovery*, 7(4):1–41, 2022.

[26] Vincent S. Tseng et al. Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE TKDE*, 25(8):1772–1786, 2022.

[27] C. K. Chui et al. Mining frequent itemsets from uncertain data. In *ICDM*, pages 47–58, 2007.

[28] H. Xing L. Xu and S. Ma. A survey on mining high utility itemsets from uncertain databases. *Future Generation Computer Systems*, 128:364–380, 2022.

[29] S. Krishnamoorthy. Mining high utility itemsets with negative utility values. *Expert Systems with Applications*, 161:113669, 2023.

[30] M. Lee et al. Negative-aware high-utility itemset mining. *Information Sciences*, 581:159–177, 2023.

[31] J. Li J. Wang and H. Xiong. Mahui: Mining top-k high utility itemsets from uncertain databases. *Knowledge-Based Systems*, 230:107425, 2022.

[32] C. Lin et al. Efficient algorithms for mining top-k high utility itemsets. *IEEE TKDE*, 28(1):244–259, 2023.

[33] A. K. Das J. Sahoo and A. Goswami. Top-k high utility itemset mining over uncertain databases. In *DASFAA*, pages 166–183, 2022.

[34] Jerry Chun-Wei Lin, Tianrui Li, Philippe Fournier-Viger, Tzung-Pei Hong, and Justin Zhan. Efficient mining of high-utility itemsets with multiple minimum utility thresholds. *Knowledge-Based Systems*, 196:105758, 2020.

[35] Jianyong Wang, Jian Li, and Huan Xiong. Mining high utility itemsets with negative utility values. In *DASFAA*, pages 184–200, 2022.

[36] Chun-Wei Lin, Tianrui Li, Philippe Fournier-Viger, and Mukesh Gupta. Efficient algorithms for mining top-k high utility itemsets. *IEEE Transactions on Knowledge and Data Engineering*, 28(1):244–259, 2023.

[37] Ying Liu, Yu Zhao, Lei Chen, Jian Pei, and Jiawei Han. Mining frequent itemsets in uncertain databases. In *ICDM*, pages 17–30, 2007.

[38] Minsu Lee, Sungho Yun, and Unil Yun. Negative-aware high-utility itemset mining. *Information Sciences*, 581:159–177, 2023.

[39] C. K. Chui, Ben Kao, and Edward Hung. Mining frequent itemsets from uncertain data. In *PAKDD*, pages 47–58, 2007.

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

[41] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data Mining and Knowledge Discovery*, 8(1):53–87, 2004.

[42] W. Gan, J.C.W. Lin, P. Fournier-Viger, H.C. Chao, V.S. Tseng, and P.S. Yu. A survey of utility-oriented pattern mining. *IEEE Transactions on Knowledge and Data Engineering*, 33(4):1306–1327, 2021.

[43] R. Chan, Q. Yang, and Y.D. Shen. Mining high utility itemsets. In *Proceedings of the Third IEEE International Conference on Data Mining*, pages 19–26, 2003.

[44] Y. Liu, W.k. Liao, and A. Choudhary. A two-phase algorithm for fast discovery of high utility itemsets. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 689–695, 2005.

[45] C.F. Ahmed, S.K. Tanbeer, B.S. Jeong, and Y.K. Lee. Efficient tree structures for high utility pattern mining in incremental databases. *IEEE Transactions on Knowledge and Data Engineering*, 21(12):1708–1721, 2009.

[46] V.S. Tseng, C.W. Wu, B.E. Shie, and P.S. Yu. Up-growth: An efficient algorithm for high utility itemset mining. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 253–262, 2010.

[47] V.S. Tseng, B.E. Shie, C.W. Wu, and P.S. Yu. Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE Transactions on Knowledge and Data Engineering*, 25(8):1772–1786, 2012.

[48] M. Liu and J. Qu. Mining high utility itemsets without candidate generation. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, pages 55–64, 2012.

[49] P. Fournier-Viger, C.W. Wu, S. Zida, and V.S. Tseng. Fhm: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In *International Symposium on Methodologies for Intelligent Systems*, pages 83–92, 2014.

[50] S. Zida, P. Fournier-Viger, J.C.W. Lin, C.W. Wu, and V.S. Tseng. Efim: A fast and memory efficient algorithm for high-utility itemset mining. *Knowledge and Information Systems*, 51(2):595–625, 2017.

[51] V.S. Tseng, W.T. Chu, and Y.D. Liang. An efficient algorithm for mining high-utility itemsets with negative item values in large databases. *Applied Mathematics and Computation*, 215(2):767–778, 2009.

[52] J.C.W. Lin, P. Fournier-Viger, and W. Gan. Fhn: An efficient algorithm for mining high-utility itemsets with negative unit profits. *Knowledge-Based Systems*, 111:283–298, 2016.

[53] H.F. Li, H.Y. Huang, and S.Y. Lee. Fast and memory efficient mining of high-utility itemsets from data streams: with and without negative item profits. *Knowledge and Information Systems*, 28(3):495–522, 2011.

[54] S. Dawar, V. Goyal, and D. Bera. A hybrid framework for mining high-utility itemsets in a sparse transaction database. *Applied Intelligence*, 47(3):809–827, 2017.

[55] W. Gan, S. Wan, J. Chen, C.M. Chen, and L. Qiu. Tophui: Top-k high-utility itemset mining with negative utility. In *IEEE International Conference on Big Data*, pages 5350–5359, 2020.

[56] C.K. Chui, B. Kao, and E. Hung. Mining frequent itemsets from uncertain data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 47–58, 2007.

[57] C.K.S. Leung, C.L. Carmichael, and B. Hao. Efficient mining of frequent patterns from uncertain data. In *Seventh IEEE International Conference on Data Mining Workshops*, pages 489–494, 2007.

[58] G. Lee and U. Yun. A new efficient approach for mining uncertain frequent patterns using minimum data structure without false positives. *Future Generation Computer Systems*, 68:89–110, 2017.

[59] T. Le, B. Vo, V.N. Huynh, N.T. Nguyen, and S.W. Baik. Mining top-k frequent patterns from uncertain databases. *Applied Intelligence*, 50(5):1487–1497, 2020.

[60] V.S. Tseng, W.T. Chu, and Y.D. Liang. An efficient algorithm for mining high-utility itemsets with negative item values in large databases. *Applied Mathematics and Computation*, 215(2):767–778, 2009.

[61] K. Singh et al. Mining of high-utility itemsets with negative utility. *Expert Systems*, 35(6):e12296, 2018.

[62] H. Kim et al. Ehmin: Efficient approach of list based high-utility pattern mining with negative unit profits. *Expert Systems with Applications*, 209:118214, 2022.

[63] V.S. Tseng et al. Efficient algorithms for mining top-k high utility itemsets. *IEEE TKDE*, 28(1):54–67, 2016.

[64] L.T.T. Nguyen, P. Nguyen, T.D. Nguyen, B. Vo, P. Fournier-Viger, and V.S. Tseng. Mining high-utility itemsets in dynamic profit databases. *Knowledge-Based Systems*, 175:130–144, 2019.

[65] V. S. Tseng, B. E. Shie, C. W. Wu, and P. S. Yu. Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE Transactions on Knowledge and Data Engineering*, 25(8):1772–1786, 2010.

[66] G. C. Lan, Y. H. Wu, and V. S. Tseng. A novel algorithm for on-shelf utility mining. In *Proceedings of the International Conference on Big Data*, pages 560–570, 2019.

[67] W. Gan, B. E. Shie, X. Zhang, and G. Chen. Fast on-shelf high-utility itemset mining. *Information Sciences*, 523:85–102, 2020.

[68] X. Zhang, G. Chen, and W. Gan. On-shelf high-utility quantitative itemsets mining. *Knowledge-Based Systems*, 203:106108, 2020.

[69] V. S. Tseng, Y. H. Wu, and J. C. Chen. Efficient algorithms for mining top-K high utility itemsets. *IEEE Transactions on Knowledge and Data Engineering*, 28(1):54–67, 2016.

[70] T. Li, Y. Wu, and W. Li. Rept: A recursive pattern-mining approach for top-K high-utility itemsets. *Applied Intelligence*, 51(3):1650–1663, 2021.

[71] G. C. Lan, V. S. Tseng, and B. E. Shie. Koshu: A top-K high-utility mining algorithm with novel pruning strategies. *Expert Systems with Applications*, 182:115152, 2021.

[72] V. S. Tseng, C. W. Wu, B. E. Shie, and P. S. Yu. Up-growth: An efficient algorithm for high utility itemset mining. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 253–262, 2010.

[73] C. K. Chui, B. Kao, and E. Hung. Mining frequent itemsets from uncertain data. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD)*, pages 47–58, 2007.

[74] C. K.-S. Leung, Q. I. Khan, Z. Li, and T. Hoque. Cantree: A canonical-order tree for incremental frequent-pattern mining. *Knowledge and Information Systems*, 11(3):287–311, 2008.

[75] W. Gan, J. C.-W. Lin, P. Fournier-Viger, and T.-P. Hong. Mining high-utility patterns in dynamic databases with positive and negative utilities. *Knowledge-Based Systems*, 199:105938, 2020.

[76] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong, and Y.-K. Lee. Mining periodic patterns in transaction databases. In *Lecture Notes in Computer Science*, pages 112–124. Springer, 2009.

[77] X. Yan and J. Han. gspan: Graph-based substructure pattern mining. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 721–724, 2002.

[78] Cheng-Wei Wu, Bor-Yi Shie, Vincent S Tseng, and Philip S Yu. Mining top-k high utility itemsets. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 78–86. ACM, 2012.

[79] Heungmo Ryang and Unil Yun. Efficient algorithms for mining top-k high utility itemsets. *Knowledge-Based Systems*, 76:109–126, 2015.

[80] Han-Jiawei Mortazavi-Asl Behzad Pinto Helen Pei, Jian, Qiming Chen, Umeshwar Dayal, and Mei-Chun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *Proceedings of the 17th International Conference on Data Engineering*, pages 215–224. IEEE, 2001.

[81] Souchet-Alexis D-Lameras Petros Petridis Panagiotis Caporal Julien Coldeboeuf Gildas Duzan Philippe, Stéphanie and Hadrien. Multimodal teaching, learning and training in virtual reality: a review and case study. *Virtual Reality & Intelligent Hardware*, 2(5):421–442, 2020.

[82] Jing Chen, Shengyi Yang, Weiping Ding, Peng Li, Aijun Liu, Hongjun Zhang, and Tian Li. Incremental high average-utility itemset mining: Survey and challenges. *Scientific Reports*, 14:9924, 2024.

AUTHORS

School of Computer Science, Ton Duc Thang University, Ho Chi Minh City, Vietnam.

Le Dang Nguyen was born in Vietnam, in 2004. He is pursuing a Bachelor's degree in Computer Science at Ton Duc Thang University.

School of Computer Science, Ton Duc Thang University, Ho Chi Minh City, Vietnam.

Vo Gia Huy was born in Vietnam, in 2004. He is pursuing a Bachelor's degree in Computer Science at Ton Duc Thang University.

School of Computer Science, Ton Duc Thang University, Ho Chi Minh City, Vietnam.

Vu Ngoc Tra My was born in Vietnam, in 2004. She is pursuing a Bachelor's degree in Computer Science at Ton Duc Thang University.