Compact Tree Structures for Mining High Utility Itemsets

Anup Bhat
Department of Computer Science and Engineering, Manipal Academy of Higher Education, India
bhatanupb@gmail.com

Harish Venkatarama
Department of Computer Science and Engineering, Manipal Academy of Higher Education, India
harish.sv@manipal.edu

Geetha Maiya
Department of Computer Science and Engineering, Manipal Academy of Higher Education, India
geetha.maiya@manipal.edu

Abstract: *High Utility Item set Mining (HUIM) from large transaction databases has garnered significant attention as it accounts for the revenue of the items purchased in a transaction. Existing tree-based HUIM algorithms discard unpromising items and require at most two database scans for their construction. Hence, whenever utility threshold is changed, the trees have to be reconstructed from scratch. In this regard, the current study proposes to not only incorporate all the items in the tree structure but compactly represent transaction information. The proposed trees namely- Utility Prime Tree (UPT), Prime Cantor Function Tree (PCFT), and String based Utility Prime Tree (SUPT) store transaction-level information in a node unlike item-based prefix trees. Experiments conducted on both real and synthetic datasets compare the execution time and memory of these tree structures with a proposed Utility Count Tree (UCT) and existing IHUP, UP-Growth trees. Due to transaction-level encoding, these structures consume significantly less memory when compared to the tree structures in the literature.*

Keywords: *High utility itemset mining, tree based algorithms.*

Received February 11, 2020; accepted July 13, 2021
https://doi.org/10.34028/iajit/19/2/2

1. Introduction

Mining frequent itemsets from a large transaction database is an indispensable step in obtaining patterns that indicate associations among items. Such a task addresses the market basket analysis problem of identifying frequently purchased items by a customer on his visit to a supermarket store [2]. Since its inception, Frequent Itemset Mining (FIM) has been applied in diverse areas such as text mining [13], Bioinformatics [24], Pharmacovigilance [10] and so on and has delivered novel insights. Amongst plethora of application areas-Recommendation Systems [23], intrusion detection systems [6], detection of fraudulent transactions [25], Web-click analysis [12] are noteworthy. Due to its versatility, FIM is viewed as a general and popular data mining task [1].

FIM algorithms are designed to consider only the frequency of occurrence of an itemset in a transaction database. However, the factors such as purchase quantities of items and their unit profit are not handled. Consequently, the patterns obtained are devoid of the revenue information. High Utility Itemset Mining (HUIM) provides for a model where the aforementioned factors can be accommodated during mining. Hence, this area has played a pivotal role since the past decade as a more generalised form of FIM. Most of the algorithms employed for FIM work on the downward closure property of the support or frequency of an itemset in order to enumerate the patterns. However, the revenue or utility measured as (unit profit×purchase quantities) is neither monotone nor anti monotone. For e.g., if the database shown in Table 2 is considered, the utilities of itemsets {1}, {1, 2} and {1, 2, 4} are respectively 20, 15 and 21. If the user defined threshold of minimum utility to extract the high utility patterns were set to 18, then the high utility itemset {1, 2, 4} contains both a high utility-1} and low utility-{1, 2} subsets. Hence, the utility measure for an itemset does not satisfy the downward closure property.

HUIM has evolved from two-phase to single phase algorithms. Although there is an absence of downward closure property, studies have explored various measures to prune the search space and efficiently mine High Utility Itemsets (HUIs). Generally, the algorithms discard the items that are deemed to be unpromising during the initial phase when a data structure such as a tree or a list is constructed. If the user requests for mining with a lower threshold, certain unpromising items may turn out to be promising. In such a scenario, the data structures have to be reconstructed from scratch.

In order to alleviate these shortcomings, the current work proposes trees that compactly represent the transaction database. Most of the existing trees encode the transaction information on a per item basis in the nodes of the tree. In contrast to this, the proposed tree structures compactly encode information in the nodes of the tree at the transaction level thus providing a

higher abstraction. Also, the memory efficiency of these structures namely Utility Prime Tree (UPT), Prime Cantor Function Tree (PCFT), and String based Utility Prime Tree (SUPT) have been compared with a conventional prefix-sharing tree called the Utility Count Tree (UCT). The significance and advantages of the proposed tree structures are enumerated below:

- All these trees are complete i.e., these are constructed using a single database scan without discarding any items. This ensures faster construction as multiple database scans (a costly I/O operation) in discarding the unpromising items is overcome.
- The trees are compact due to the encoding of information with respect to a transaction rather than per item basis (except UCT) in the nodes of the trees. Also, experimental evaluations prove that they are memory efficient.

The rest of this paper is organized as follows. In subsections 2.1 and 2.2 of section 2 formal concepts and related work in the area of HUIM have been described. The proposed data structures have been detailed in section 3. The paper concludes with experimental evaluation and conclusion provided in sections 4 and 5 respectively.

2. Background

2.1. Preliminary

Given a transaction database *D*, each transaction T_d in *D* is identified by *TID*, the transaction identifier and records a collection of items purchased along with its quantity or internal utility. Formally, $T \subseteq I$ where $I=\{i_1, i_2, i_3, \ldots, i_n\}$ denotes the collection of items. A typical transaction database is as shown in Table 2. An ordered pair (i_x,q_x) in each transaction indicates that the item i_x was purchased in q_x quantities in that transaction. Each item is also associated with unit profit or external utility as shown in Table 1.

Table 1. Profit table.

Item	1	2	3	4	5	6	7
Profit	5	2	1	2	3	5	1

Table 2. Transaction table.

TID	Transactions
T_1	{(3,1)(5,1)(1,1)(2,5)(4,3)(6,1)}
T_2	{(3,3)(5,1)(2,4)(4,3)}
T_3	{(3,1)(1,1)(4,1)}
T_4	{(3,6)(5,2)(1,2)(7,5)}
T_5	{(3,2)(5,1)(2,2)(7,2)}

- *Definition* 1: Utility of an item *i* in transaction T_d, denoted by $u(i,T_d)$ is measured as the product of quantity $q(i, T_d)$ and unit profit $p(i)$.
- *Definition* 2: Utility of an itemset *X* in transaction T_d, denoted by $u(X, T_d)$ is defined as $\sum_{i \in X \wedge X \subseteq T_d} u(i, T_d)$
- *Definition* 3: Utility of an itemset *X* in *D* denoted by *u(X)*, is defined as

$$u(X) = \sum_{X \subseteq T_d \wedge T_d \in D} u(X, Td) \quad (1)$$

For e.g.,: $u(\{2\}, T_2)$= 4×2=8
$u(\{2,3\},T_2) = u(\{2\}, T_2)+u(\{3\}, T_2)$=8+3=11
$u(\{2,3\}) = u(\{2,3\}, T_1)+u(\{2,3\}, T_2)+u(\{2, 3\}, T_5)$=11 +11+6=28

- *Definition* 4: An itemset *X* is called a High Utility Itemset (HUI) if $u(X) \geq min\_util$, where *min_util* is the minimum utility provided by the user.
- Definition 5: Transaction Utility of a transaction T_d, denoted by $TU(T_d)$ is defined as the sum of the utilities of all the items in that transaction i.e., $\sum_{i \subseteq T_d} u(i, T_d)$
- *Definition* 6: Transaction Weighted Utility of an itemset *X*, denoted by *TWU(X)*, is defined as the sum of the transaction utility of all the transactions in *D* that contain *X*, i.e., $\sum_{X \subseteq T_d \wedge T_d \in D} TU(T_d)$
- *Definition* 7: An itemset *X* is a high-transaction weighted utility itemset (HTWUI), if $TWU(X) \geq min\_util$. Also, if an itemset *X* is not a HTWUI, then it cannot be a HUI.
- *Property* 1 (TWU Downward Closure Property) If an itemset *X* is a HTWUI, then all its subsets are HTWUIs or if an itemset *X* is not a HTWUI, then none of its supersets can be HTWUIs.

2.2. Related Work

For an itemset *X*, $u(X)$ is upper bounded by $TWU(X)$. Also, $TWU(X)$ is downward closed. Hence, most of the algorithms that generate and test candidates in a level-wise manner [15, 16, 21] employ this measure to prune the search space. However, repeated database scans are required to determine the TWU of candidate itemsets at every level. Further, the join operation adds to the complexity of enumerating higher order itemsets. To minimise database scans, several algorithms have been proposed that transform the information in the database to a data structure. The subsequent mining involves operating on this data structure for mining; thus eliminating repeated database scans.

In this regard, the role of HUP-Growth [17], HUC-Prune [3] based on FP- Growth [9] is significant. While these algorithms transform the database into a tree with a node for every item of a transaction, the former captures the quantity information in one of the elements of the node. Due to the absence of a mechanism to enumerate itemsets from the HUP-tree efficiently, a lot of candidate itemsets have to be examined. In contrast, HUC-Tree makes use of Property 1in conjunction with pattern-growth to enumerate candidate HUIs. Although both the algorithms generate significantly lesser number of candidates in comparison to the Two-Phase algorithm,

the tree structure needs to be reconstructed once the user defined threshold for minimum utility, i.e., *min_util* is modified.

UP-Growth [28] and UP-Growth+ [27] are the efficient state of the art tree based algorithms. Here, the authors proposed several strategies to prune the search space during construction and recursive mining of the UP-Tree. At the very outset, during the first database scan, those 1-items that were not HTWUI are discarded. Further, the utilities of each item is calculated without considering its descendants. In a similar manner, during the mining operation local UP-trees are constructed after discarding local unpromising items and decreasing minimum utilities of descendant nodes. The bounds are further strengthened in UP-Growth+based on path utilities and estimated utilities of descendant nodes.

In order to facilitate efficient incremental and interactive mining of HUIs, IHUP algorithm that constructs tree without ignoring any items based on TWU was proposed [4]. Authors proposed three different approaches of tree construction- based on lexicographic order of items called $IHUP_L$-Tree, based on descending order of Transaction Frequency called $IHUP_{TF}$-Tree and based on descending order of TWU called $IHUP_{TWU}$-tree. After *N* transactions are read, $IHUP_{TF}$-Tree and $IHUP_{TWU}$-Tree has to be reordered that increases the data structure construction time.

Apart from the tree based algorithms, several list based algorithms are available in the literature such as HUI-Miner [20], HUP-Miner [14], d^2HUP [18, 19]. These algorithms are mostly single-phase and employ pruning strategies similar to the tree-based algorithms. However, the memory consumed in storing the Utility Lists and the costly comparison and join operation is a severe performance bottleneck. Further, a recent study has indicated that trees can outperform list based and projection based algorithms [5]. However, the recursive mining operation on trees necessitate the creation of large number of conditional pattern trees that consume the memory space.

Not many studies for mining HUIs incorporate all the items in the tree structure. Besides, these trees are prefix-shared on items in the transaction. To address the shortcomings, a prefix-based tree structure called Utility Count Tree is proposed. Further, to achieve better compaction, the items and their utilities are encoded such that each node in the tree represents a transaction. In addition to this, the proposed compact trees are prefix-sharing with respect to the transaction information stored in the node. Also, these trees ensure completeness and memory efficiency.

3. Methodology

In this section four different tree structures are proposed to represent the transaction database namely:

- Utility Count Tree (UCT)
- Utility Prime Tree (UPT)
- Prime Cantor Function Tree (PCFT)
- String based Utility Prime Tree (SUPT)

The following subsections are dedicated to each of these tree structures where the node structure and brief procedure of representing the information in the database is described.

3.1. Utility Count Tree

A node in the Utility Count Tree has the following fields:

1. Item which denotes the name of the item
2. Count that indicates the count of the item in the given path of the tree
3. Utility that accumulates the utility of the item in the given path of the tree
4. Parent pointer that points to the parent of the node

UCT is constructed without discarding any items during the initial tree construction unlike HUP-Growth or HUC-Prune. The database is scanned and a node, *N* is constructed for every item in a transaction T_j. The brief procedure for inserting transactions into UCT is provided in Algorithm (1). Initially *N* is set to the root node of the tree. The items in the transaction are inserted as child nodes of one another. Hence, each path of the tree corresponds to a particular transaction. If a transaction contains a node that is already present in the tree, the procedure updates the count and utility instead of creating a new node in the given path. This ensures prefix sharing. The UCT for sample database in Table 1 is displayed in Figure 1.

Algorithm 1. Inserting into Utility Count Tree

Step 0. Input: UCT, T_j={(i_1, q_1), (i_2, q_2), (i_3, q_3),. . . , (i_p,q_p) }, (i_k ∈ I, $1 \le k \le n$ and $T_j \in D$)

Step 1. N ← the root node of UCT

Step 2. For each(i_x, q_x) ∈ T_j

Step 3. If N has a child C such that C. item = i_x then

Step 4. C. count ← C. count + 1

Step 5. C. nodeUtility ← C. nodeUtility + u(i_x, T_j)

Step 6. Else

Step 7. Create a new child node C with:

Step 8. C. item← i_x, C.count← 1 and C. utility ← u(i_x, T_j)

Step 9. C.parent ← N

Step 10. End If

Step 11. N ← C

Step 12.End For

With a single scan of the database, UCT captures relevant information in its tree structure. Although the count field indicates the number of transactions a given item is participating, reconstructing the database from this information is not possible. For example, consider the paths <3, 5, 2, 4> and < 3, 5, 2, 7 > for

UCT shown in Figure 1. The count field for item 2 has the value 2 as it is participating in two transactions, T_2 and T_5. However, the utility value is cumulative of utilities in these two transactions and does not enable in resolving the utility individually across the two transactions.

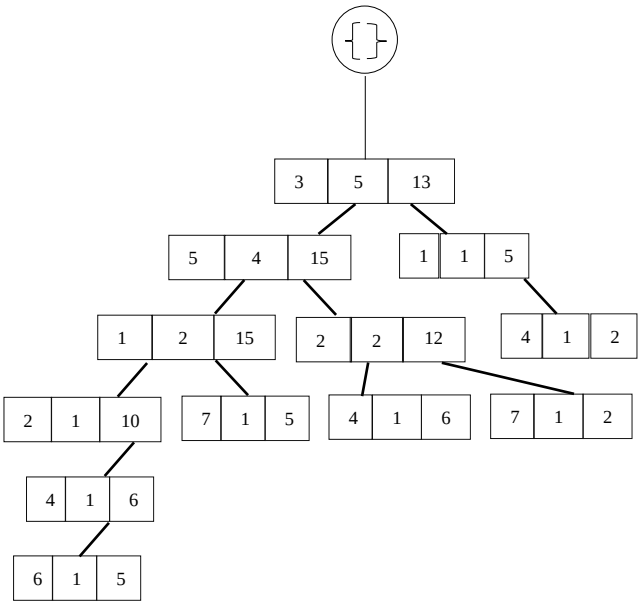

Figure 1. UCT for database in Table 2.

3.2. Utility Prime Tree

Utility Prime Tree captures the transaction level information in a single node unlike the UCT. The items and utilities are mapped to corresponding prime numbers by employing them as indices [22]. The assigned prime numbers are then stored compactly in a node of the UPT. Every node of the UPT contains the following fields:

- *Prime Items*- This field stores the product obtained after multiplying the prime numbers assigned to every item of a transaction
- *TU*-This field stores the Transaction Utility of a transaction
- *Prime Utility (T_j)*- This field stores the product obtained after multiplying the prime numbers assigned to utility of every item of a transaction
- *Paren*- pointer that points to the parent of the node

A node in the UPT captures compressed transaction level information, i.e., *prime Items(T_j)* and *prime Utility(T_j)* in the first and the third fields respectively. e.g., consider the transaction T_2: the items <3, 5, 2, 4> are assigned their corresponding prime numbers as <5, 11, 3, 7> and utilities <3, 3, 8, 6> get assigned to <5, 5, 19, 13>. Now, the node corresponding to T_2 has *prime Items(T_2)*=1155 and *prime Utility(T_2)*=6175. The second field of the node stores the TU of this transaction.

Every node is inserted from the root of the UPT. In order to ensure prefix sharing, transactions with common items share the same path. This is ensured by checking if any node in UPT has its *prime Item* divisible by *prime Item(T_j)*. For example, transaction T_2 is inserted as child of T_1 because *prime Item (T_1)*= 30030 is divisible by *prime Item(T_2)*=1155. Hence, transactions that are sub-transactions of one another (with respect to the items participating) share a common parent.

The database reconstruction involves factorising the *prime Item* and *prime Utility* of every node to retrieve the transaction level information. For example, if the node corresponding to T_2 is considered from the tree, upon factorising *prime Item* and *prime Utility* we get, <3, 5, 7, 11> and <5, 5, 13, 19>. These factors correspond to <2, 3, 4, 5> th and <3, 3, 6, 8> th prime numbers. However, this incorrectly maps the utilities of items 2 and 5 to 3 and 8 instead of 8 and 3. Although the items and utilities are intact, it is not possible to resolve the utilities of every item correctly. Hence, utility values are encoded to $((prime(i_x))^{u(ix,\ Tj)})$ th prime number and multiplied to get *primeUtility*. The procedure is highlighted in Algorithm (2). For T_2, $primeUtility(T_2)=5^{(3)}.11^{(3)}.3^{(8)}.7^{(6)}$ instead of previously obtained value of 6175. During database reconstruction, it is easier to map the utilities which are present as exponents of the corresponding prime encoded items. Figure 2 shows transactions encoded using this procedure.

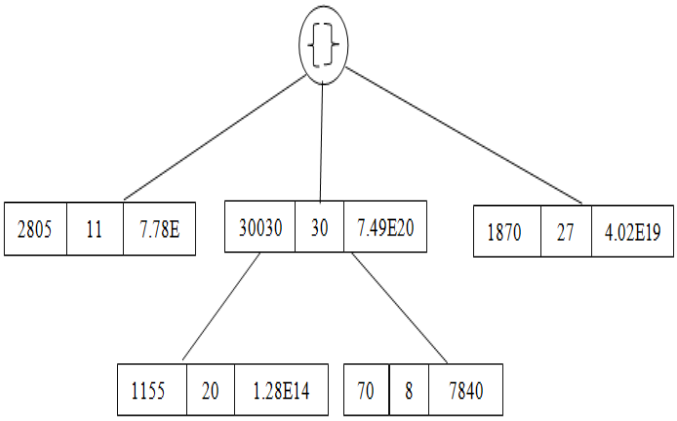

Figure 2. UPT for database in Table 2.

Algorithm 2. Inserting into Utility Prime Tree
Step 0. Input:T_j={(i_1, q_1), (i_2, q_2), (i_3, q_3),. . . , (i_p,q_p) }, ($i_k \in I$, $1 \le k \le n$ and $T_j \in D$)
Step 1.For transaction $T_j \in D$
Step 2. Calculate $TU(T_j)$
Step 3. Replace item i_x with i_xth prime number, $prime(i_x)$
Step 4.Replace $u(i_x, T_j)$ with $prime(i_x)^{u(ix,\ Tj)}$
Step 5. primeItems(T_j) ← product of prime encoded items
Step 6. primeUtility(T_j) ← product of prime encoded item utilities
Step 7. End For
Step 8.procedure INSERT_TRANSACTION_UPT(root, T_j)
Step 9. Create a new child node S with:
Step 10. S.primeItems ←primeItems(T_j)
Step 11. S.TU ← TU(T_j)
Step 12. S.primeUtility← primeUtility(T_j)
Step 13. I froot has child C such that (

(C.primeItems % primeItems(T_j) = 0 OR (primeItems(T_j) % C.primeItems) = 0)
Step 14. S.parent← C
Step 15. Else
Step 16. S.parent ← root
Step 17. End If
Step 18. End procedure

UPT compactly stores the database information by capturing the information at the transaction level instead of creating node for every item in a transaction. However, as the number of items and transactions increase, the space required to store the product, *prime Utility (T_j)* overwhelms the allocated node space. Hence, factorising and subsequent resolution of utilities of items is not facilitated.

3.3. Prime Cantor Function Tree

In order to resolve the utility corresponding to the items in a transaction T_j and satisfy the completeness constraint, Cantor Function (CF) that reversibly maps the pair of non-negative integers *(i_x, $u(i_x,T_j)$)* onto another non-negative integer is explored [11]. For an ordered pair *(a, b)* CF is defined as:

$$CF(a,b) = \frac{(a+b)(a+b+1)}{2} + b$$

Inverse of *CF(a, b)* is calculated as follows:

Let *CF(a, b)=z*

- *Step* 1: $w = \lfloor \frac{\sqrt{8z+1}-1}{2} \rfloor$
- *Step* 2: $t = \frac{w^2+w}{2}$
- *Step* 3: *b=z–t*
- *Step* 4: *a=w–b*

The details of the different fields in the node of PCFT are provided below:

- *PrimeCF*- This field stores the product obtained after multiplying the prime numbers assigned to *CF(i_x, $u(i_x,T_j)$)* for every item i_x of a transaction
- *TU*- This field stores the Transaction Utility of a transaction
- *Parent* pointer that points to the parent of the node

Brief procedure to construct PCFT is provided in Algorithm (3). Figure 3 displays the PCFT of Table 1. Although the tree is complete and ensures database reconstruction, the main drawback is absence of path sharing as evident from Figure 3. For example, while T_2 and T_3 appear as child of T_1 in UPT as *items(T_2) ⊂ items(T_1)* and *items(T_3) ⊂ items(T_1)* as shown in the Figure 2, due to the uniqueness in mapping of *(i_x, u(i_x, T_j))* through CF prior to prime encoding, the sharing is absent in PCFT. With PCFT, if items are purchased in similar quantities across two transactions T_i and T_j such that either *items(T_i) ⊂ items(T_j)* or *items(T_j) ⊂ items(T_i)* sharing can be ensured.

Utility values are used along with items in the Cantor Function prior to assigning them with the corresponding prime numbers. If an item is present in two different transactions, CF maps the item-utility pair to a unique number and hence the same item may get assigned to different prime numbers if the utility values are different. This limits the prefix sharing in the PCFT owing to the inherent feature of the CF. The major implementation drawback is due to the large value obtained to store *primeCF(T_j)* for every transaction. This is bound to increase overwhelmingly with growing number of items in the database.

Algorithm 3. Inserting into Prime Cantor Function Tree
Step 0. Input: Tj={(i1, q1), (i2, q2), (i3, q3),. . . , (ip,qp) }, (ik ∈ I, 1≤ k ≤ n and Tj ∈ D)
Step 1. For transaction Tj∈ D
Step 2. Calculate TU(Tj)
Step 3. Calculate CF(ix, u(ix,Tj))
Step 4.Replace (ix, u(ix,Tj)) in Tjwith CF(ix, u(ix,Tj))th prime number
Step 5. prime CF(Tj) ←product of prime encoded items and their utilities after applying CF
Step 6. End For
Step 7. procedure INSERT_TRANSACTION_PCFT(root, Tj)
Step 8.Create a new child node S with:
Step 9. S.primeCF ←prime CF(Tj)
Step 10. S.TU ← TU(Tj)
Step 11.S.primeUtility← primeUtility(Tj)
Step 12.I froot has child C such that
((C. prime CF % prime CF(Tj)) = 0 OR
(prime CF(Tj) % C. prime CF) = 0)
Step 13. S.parent← C
Step 14. Else
Step 15. S.parent ← root
Step 16. End If
Step 17. End procedure

3.4. String based Utility Prime Tree

This tree is similar to UPT. In order to overcome the problem of storing large number that arises from computing *primeItem(T_j)*, the prime numbers assigned to items and utilities are concatenated by a delimiter. As this information is stored in textual format, substring comparison is performed while inserting transactions into the tree structure to identify the transactions containing common set of items. This ensures prefix-sharing. Also, it is possible to reconstruct the entire database due to the string representation of the primeItems and primeUtility. The procedure and SUPT for sample database is displayed in Algorithm (4) and Figure 4 respectively.

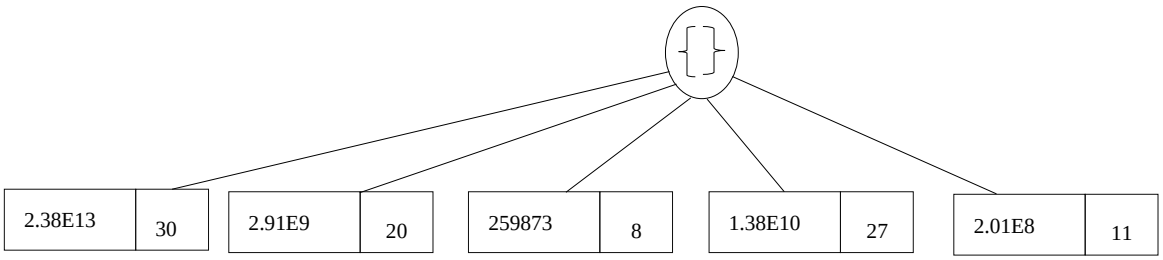

Figure 3. PCFT for database in Table 2.

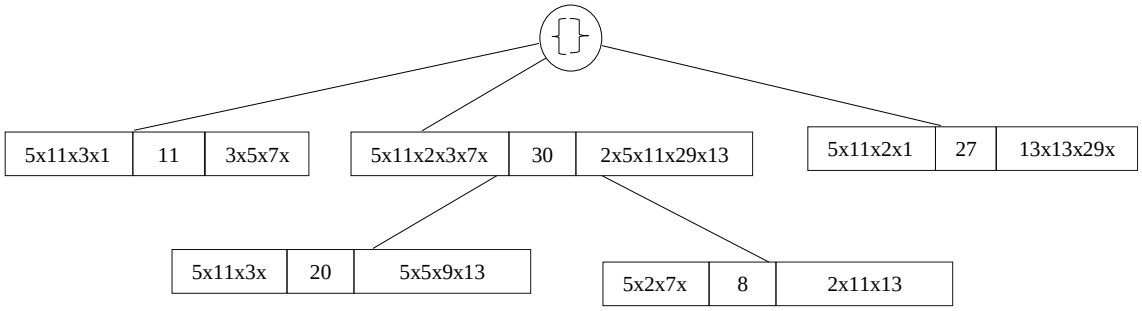

Figure 4. SUPT for database in Table 2.

Algorithm 4. Inserting into String based Utility Prime Tree

Step 0. Input: Tj={(i1, q1), (i2, q2), (i3, q3),. . . , (ip,qp) }, (ik ∈ I, 1≤ k ≤ n and Tj ∈ D)
Step 1. For transaction Tj∈ D
Step 2. Calculate TU(Tj)
Step 3. Replace item ix with ixth prime number, prime(ix)
Step 4.Replace u(ix, Tj) with u(ix, Tj)th prime number
Step 5. primeItems(Tj) ← prime encoded items concatenated as string
Step 6.primeUtility(Tj) ← prime encoded item utilities concatenated as string
Step 7.End For
Step 8.procedure INSERT_TRANSACTION_SUPT(root, Tj)
Step 9. Create a new child node S with:
Step 10.S.primeItems ←primeItems(Tj)
Step 11. S.TU ← TU(Tj)
Step 12.S.primeUtility← prime Utility(Tj)
Step 13.I froot has child C such that
((C. prime Itemsis a substring of prime Items(Tj)) OR (prime Items(Tj) is a substring of C. prime Items))
Step 14. S. parent← C
Step 15.Else
Step 16. S. parent ← root
Step 17.End If
Step 18. End procedure

4. Experimental Evaluation

The source code implementation in Java provided by SPMF Data Mining Library was extended to implement the proposed tree structures [8]. Experiments were conducted on both real and synthetic datasets to compare the execution time and memory consumed. The characteristics of the datasets used is provided in Table 3 [7]. $|D|$ denotes the number of transactions, $|I|$ denotes the number of items, T denotes average transaction length. *Density* calculated as $T/|I|$ indicates how sparse or dense the dataset is. The dataset *Foodmart* contains real utility values. For the remaining datasets, the internal utility values have been generated using a uniform distribution in [1, 10], and the profit values follow a Gaussian distribution.

Datasets are chosen such that different ranges of $|D|$, $|I|$, T and density are taken up for evaluation. The proposed trees are compared to the two popular tree structures in the literature namely, IHUP and UP-Growth trees. For the experiments, a system with 8GB RAM, Windows 7 OS with Intel Core i5 processor at 3.00 GHz was used.

Table 3. Characteristics of datasets.

Dataset	IHUP	UP-Growth
Chess	76.47	91.56
Mushroom	64.26	91.38
Foodmart	72.10	92.91
Retail	64.76	99.82
Connect	50.83	99.48

4.1. Performance Analysis on Real Datasets

Figure 5 depicts the execution time of the algorithms. Across all the datasets, UCT executed faster than the remaining algorithms. The percentage improvement obtained due to UCT in comparison to IHUP and UP-Growth trees is recorded in Table 4. Among the prime-based trees, UPT performed better, especially when the dense datasets were considered. As shown in the figure, it executed faster than UP-Growth by 66%, 36% and 65% on *Chess*, *Mushroom* and *Connect* datasets respectively. Also, PCFT performed 48.8% faster than UP-Growth on *Connect* dataset. However, its performance was poor on large and sparse datasets such as, *Food mart* and *Retail*. Owing to the longer execution time, PCFT was executed on only 100 and 500 transactions of these two datasets. The larger values obtained after applying CF to $(i_x, u(i_x, T_j))$ pair increased the prime encoding time that subsequently

affected the overall execution time. Although SUPT took longer time for construction, it performed significantly faster on Food mart, one of the sparse datasets where PCFT failed. The low value of *T* for this dataset ensured the presence of common set of items across different transactions leading to lesser string comparisons during the tree construction.

Table 4. Percentage improvement in execution time of UCT in comparison to IHUP and UP-Growth.

Dataset	|*D*|	|*I*|	*T*	*Density (%)*
Chess	3196	75	37	49.33
Mushroom	8124	119	23	19.32
Food mart	4141	1559	4.4	0.28
Retail	88162	16470	10.3	0.06
Connect	67557	129	43	33.33

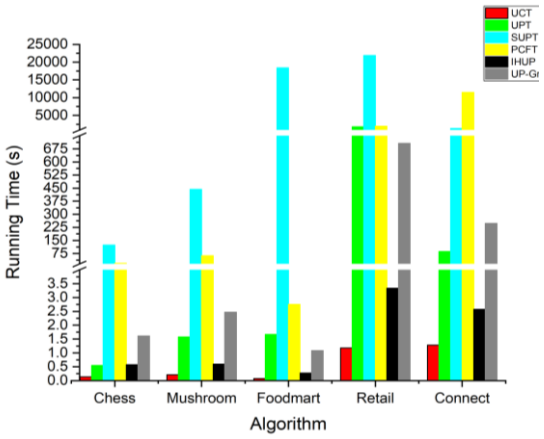

Figure 5. Execution time of the algorithms for real datasets.

Figure 6 denotes the memory consumed by the proposed structures in comparison with IHUP and UP-Growth trees. The get Object Size (Object) method of Instrumentation interface implemented and provided in size of package was used to calculate the amount of memory consumed [26]. Due to PCFT's longer execution time only 100 and 500 transactions of Foodmart and Retail was considered. The prime-based tree structures clearly consumed significantly lesser space in comparison to the remaining trees. The transaction level encoding of database information ascertains the lower memory consumption. Factor-wise reduction in the space consumed by Prime Trees is tabulated in Table 5.

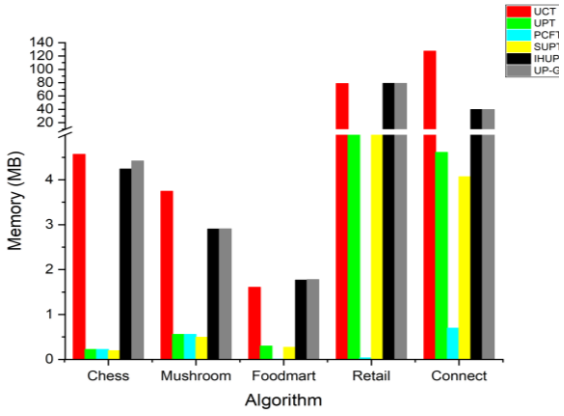

Figure 6. Memory consumed by the tree structures for real datasets.

On an average across Chess dataset, UPT, PCFT, and SUPT consume 19.5, 19.5 and 22.1 times lesser memory than IHUP and UP-Growth trees. Across Mushroom, in the same order the reduction in memory was 5.2, 5.2 and 5.8 times. Due to partial database considered when running PCFT implementation of Foodmart, UPT and SUPT consumed 5.9 and 6.5 times lesser space. This reduction for the two trees was about 12.6 and 13.3 in the case of Retail, another sparse dataset. A reduction of about 8.6 and 9.7 times was observed when Connect dataset was considered. Although PCFT has only two fields, the prefix sharing is easier across SUPT than in PCFT. Hence SUPT turned out to be memory efficient among the proposed trees.

Table 5. Space reduction (ratio) of prime-based trees.

Dataset	UPT		PCFT		SUPT	
	IHUP	UP-Growth	IHUP	UP-Growth	IHUP	UP-Growth
Chess	19.97	19.16	19.97	19.16	22.57	21.66
Mushroom	5.21	5.20	5.21	5.20	5.89	5.89
Foodmart	5.95	5.91	255.40	253.54	6.56	6.51
Retail	12.64	12.64	2226.0	2226.04	13.31	13.31
Connect	8.61	8.61	57.01	57.01	9.75	9.75

4.2. Performance Analysis on Synthetic Datasets

In order to further evaluate the performance of the proposed structures, synthetic datasets were generated using the SPMF tool. First set of datasets were mostly dense and their characteristics are provided in Table 6 where the parameter T_{max} denotes the maximum transaction length.

Table 6. Characteristics of synthetic datasets.

Dataset	|*D*|	|*I*|	Tmax	Density(%)
d01	5000	100	10	5.54
d02	5000	100	50	25.53
d03	5000	500	10	1.09
d04	5000	500	50	5.04
d05	10000	100	10	5.43
d06	10000	100	50	25.61
d07	10000	500	10	1.10
d08	10000	500	50	5.10

The execution time of different algorithms is compared in Figure 7. As in the case of real datasets, UCT clearly outperformed all the algorithms. Table 7 records the percentage improvement obtained in execution time when UCT was compared with IHUP and UP-Growthtrees. On an average an improvement of 82.82% on IHUP and 52.49% on UP-Growth was observed.

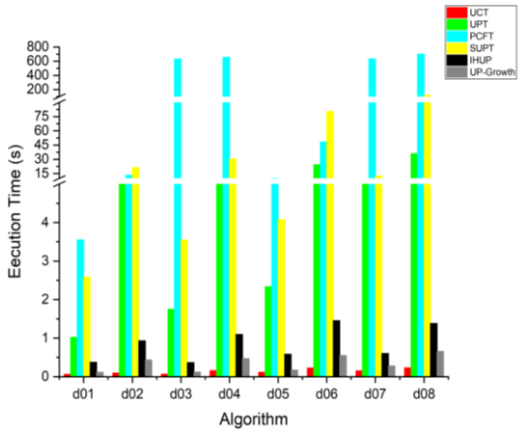

Figure 7. Execution Time of algorithms on synthetic dense datasets.

Among the prime-based trees, UPT and SUPT showed promising results. Further, on very dense datasets like *d02* and *d06*, PCFT executed faster than SUPT, although not considerably. However, as the datasets became relatively sparse its performance degraded, especially in the case of *d03* and *d07* where SUPT and UPT performed significantly better. This indicates that PCFT is more sensitive to sparseness. In the case of UPT and SUPT the increase in execution time with the increase in density for a constant database size was significant in contrast to PCFT. Especially in the case of d03 and d04 where the change in density was around4 units, execution time of PCFT was almost the same while there was sharp increase in execution time of both UPT and SUPT. This indicates that UPT and SUPT are more sensitive to density changes for agiven size of the database than PCFT.

Table 7. Percentage improvement in execution time of UCT on synthetic dense datasets.

Dataset	IHUP	UP-Growth
d01	83.24	44.24
d02	89.83	78.16
d03	81.96	45.45
d04	85.59	66.09
d05	80.34	32.74
d06	84.42	58.65
d07	74.08	44.48
d08	83.14	64.53

Further, the memory consumed by the various structures was compared as shown in Figure 8. SUPT turned out to be the memory efficient one. For a given database size, although the memory taken up by tree structures seemed to be mostly independent of the changing density, SUPT showed slight variations when compared to other prime-based trees. This difference was evident with growing database size. Table 8 records the factor-wise memory consumption of Prime Trees in comparison to IHUP and UP-Growth trees. Comparison on real and synthetic dense datasets indicated that UCT is more time efficient whereas SUPT is more memory efficient.

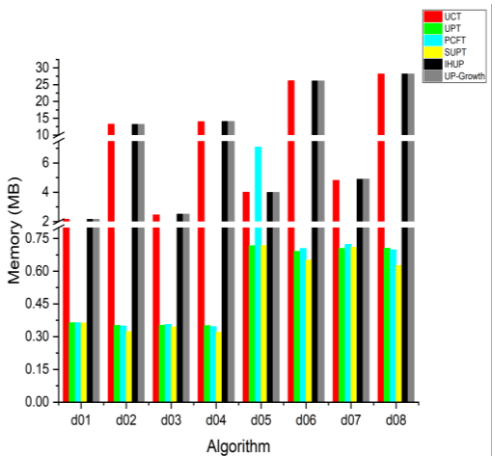

Figure 8. Memory consumption by the trees on synthetic dense datasets.

Table 8. Reduction in memory consumption across synthetic dense datasets.

Dataset	UPT		PCFT		SUPT	
	IHUP	UP-Growth	IHUP	UP-Growth	IHUP	UP-Growth
d01	5.89	5.89	5.90	5.90	5.92	5.92
d02	37.50	37.50	37.78	37.78	40.82	40.82
d03	7.08	7.08	7.01	7.01	7.24	7.25
d04	39.95	39.96	40.44	40.44	43.72	43.72
d05	5.56	5.56	0.562	0.562	5.55	5.55
d06	37.77	37.78	37.06	37.06	40.11	40.12
d07	6.95	6.95	6.77	6.77	6.89	6.90
d08	39.94	39.95	40.35	40.35	44.99	44.99

In order to further explore the characteristics, experiments were conducted to compare these two structures on sparse datasets. The characteristics of the datasets is described in Table 9. Figures 9 and 10 depict the execution time and space consumed respectively. UCT performed graciously even with sparsest of the datasets. However, SUPT clearly outperformed UCT in terms of memory requirements.

Table 9. Characteristics of synthetic sparse datasets.

Dataset	$\|D\|$	$\|I\|$	*Tmax*	Density(%)
s01	10000	10000	10	0.055
s02	10000	10000	50	0.257
s03	10000	50000	10	0.017
s04	10000	50000	50	0.051
s05	100000	10000	10	0.055
s06	100000	10000	50	0.255
s07	100000	50000	10	0.011
s08	100000	50000	50	0.051

4.3. Inferences

In the previous section, the proposed tree structures were compared with IHUP and UP Growth which are item-based prefix trees. As IHUP involves reordering the tree after *N* transactions and UP-Growth involves two database scans for complete tree construction, such overheads were eliminated in UCT leading to faster execution. In terms of memory requirements, the prime trees were more efficient due to the transaction level encoding of information. Among these, SUPT was more efficient across real and synthetic dataset sowing to better prefix-sharing. As

the datasets became sparser, PCFT performed poorly in terms of execution time. However, for database of shorter transactions, PCFT can be selected as it displayed faster execution. Overall, UCT and SUPT are promising choices for tree constructions.

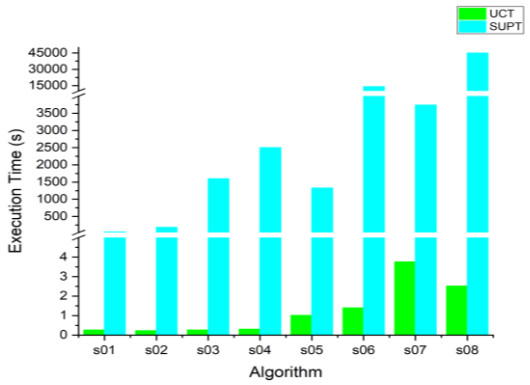

Figure 9. Execution time of UCT and SUPT on synthetic sparse datasets.

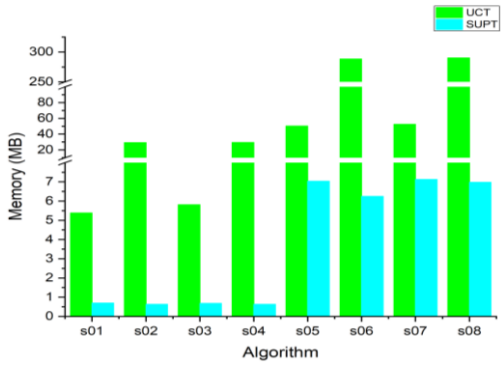

Figure 10. Space consumed by UCT and SUPT on synthetic sparse datasets.

5. Conclusions

With ever increasing database sizes the need for accommodating the essential utility information from is of prime importance. In this regard, the current work proposes tree structures that are constructed via a single database scan without neglecting any items. Especially the proposed prime-based tree structures namely, Utility Prime Tree, Prime Cantor Function Tree and String based Prime Utility Tree have been promising ways of storing the database information in a compact manner in the memory. Apart from this, the proposed Utility Count Tree is not only time efficient on real datasets but also on large sparse and dense databases. This work can be extended further to mine high utility itemsets from very large databases in a distributed environment.

References

[1] Aggarwal C., *Frequent Pattern Mining*, Springer International Publishing, 2014.

[2] Agrawal R. and Srikant R., "Fast Algorithms for Mining Association Rules," *in Proceedings of the 20th VLDB Conference*, Santiago, pp. 487-499, 1994.

[3] Ahmed C., Tanbeer S., and Jeong B., and Lee Y., "HUC-Prune: An Efficient Candidate Pruning Technique to Mine High Utility Patterns," *Applied Intelligence*, vol. 34, no. 2, pp. 181-198, 2011.

[4] Ahmed C., Tanbeer S., Jeong B., and Lee Y., "Efficient Tree Structures for High Utility Pattern Mining in Incremental Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 12, pp. 1708-1721, 2009.

[5] Dawar S., Bera D., and Goyal V., "High-Utility Itemset Mining for Subadditive Monotone Utility Functions," CoRR abs/1812.07208, 2018.

[6] Duan Y., Fu X., Luo B., Wang Z., Shi J., and Du X., "Detective: Automatically Identify and Analyze Malware Processes in Forensic Scenarios Via Dlls," *in Proceedings of IEEE International Conference on Communications*, London, pp. 5691-5696, 2015.

[7] Fournier-Viger P., SPMF An Open-Source Data Mining Library, Datasets, https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php, Last Visited, 2019.

[8] Fournier-Viger P., SPMF An Open-Source Data Mining Library, Developer's Guide, https://www.philippe-fournier-viger.com/spmf/index.php?link=developers.php, Last Visited, 2019.

[9] Han J., Pei J., Yin Y., and Mao R., "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach," *Data Mining and Knowledge Discovery*, vol. 8, no. 1, pp. 53-87, 2004.

[10] Harpaz R., Chase H., and Friedman C., "Mining Multi-Item Drug Adverse Effect Associations in Spontaneous Reporting Systems," *BMC Bioinformatics*, vol. 11, no. 9, pp. 1-8, 2010.

[11] Hopcroft J., Motwani R., and Ullman J., *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, 2006.

[12] Ivancsy R. and Vajk I., "Frequent Pattern Mining in Web Log Data," *Acta Polytechnica Hungarica*, vol. 3, no. 1, pp. 77-90, 2006.

[13] Krishna S. and Bhavani S., "An Efficient Approach for Text Clustering Based on Frequent Itemsets," *European Journal of Scientific Research*, vol. 42, no. 3, pp. 385-396, 2010.

[14] Krishnamoorthy S., "Pruning Strategies for Mining High Utility Itemsets," *Expert Systems with Applications*, vol. 42, no. 5, pp. 2371-2381, 2015.

[15] Lan G., Hong T., and Tseng V., "An Efficient Gradual Pruning Technique for Utility Mining," *International Journal of Innovative Computing*

Information and Control, vol. 8, no. 7B, pp. 5165-5178, 2012.
[16] Li Y., Yeh J., and Chang C., "Isolated Items Discarding Strategy for Discovering Highutility Itemsets," *Data and Knowledge Engineering*, vol. 64, no. 1, pp. 198-217, 2008.
[17] Lin C., Hong T., and Lu W., "An Effective Tree Structure for Mining High Utility Itemsets," *Expert Systems with Applications*, vol. 38, no. 6, pp. 7419-7424, 2011.
[18] Liu J., Wang K., and Fung B., "Direct Discovery of High Utility Itemsets without Candidate Generation," *in Proceedings of IEEE 12th International Conference on Data Mining*, Brussels, pp. 984-989, 2012.
[19] Liu J., Wang K., and Fung B., "Mining High Utility Patterns in one Phase Without Generating Candidates," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 5, pp. 1245-1257, 2016.
[20] Liu M. and Qu J., "Mining High Utility Itemsets without Candidate Generation," *in Proceedings of the 21st ACM International Conference on Information and Knowledge Management*, Maui, Hawaii, pp. 55-64, 2012.
[21] Liu Y., Liao W., and Choudhary A., "A Two-Phase Algorithm for Fast Discovery of High Utility Itemsets," *in Proceedings of Pacific-Asia Conference on Knowledge Discovery and Data Mining*, Hanoi, pp. 689-695, 2005.
[22] Maiya G. and D'Souza R., "An Efficient Reduced Pattern Count Tree Method for Discovering Most Accurate Set of Frequent Itemsets," *IJCSNS International Journal of Computer Science and Network Security*, vol. 8, no. 8, pp. 121-126, 2008.
[23] Mobasher B., Cooley R., and Srivastava J., "Automatic Personalization Based on Web Usage Mining," *Communications of the ACM*, vol. 43, no. 8, pp. 142-151, 2000.
[24] Naulaerts S., Meysman P., Bittremieux W., Vu T., VandenBerghe W., Goethals B., and Laukens K., "A Primer to Frequent Itemset Mining for Bioinformatics," *Briefings in Bioinformatics*, vol. 16, no. 2, pp. 216-231, 2013.
[25] Poongodi K. and Kumar D., "Support Vector Machine with Information Gain Based Classification for Credit Card Fraud Detection System," *The International Arab Journal Information Technology*, vol. 18, no. 2, pp. 199-207, 2021.
[26] Slashdot Media: java. size of. https://sourceforge.net/projects/sizeof/_les/, Last Visited, 2019.
[27] Tseng V., Shie B., Wu C., and Yu P., "Efficient Algorithms for Mining High Utility Itemsets from Transactional Databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 1, pp. 54-67, 2016.
[28] Tseng V., Wu C., Shie B., and Yu P., "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*, Washington, pp. 253-262, 2010.

Anup Bhat was born in Manipal, Udupi, Karnataka, India in 1991. He received his BE degree in information and communication technology (2013) and MTech degree in computer science and engineering (2017) from Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India in 2013 and 2017, respectively. He is currently pursuing PhD in the same institute. His research interests include Data Mining and Machine Learning.

Harish Venkatarama received his PhD degree from National Institute of Technology Karnataka in 2011 from the Department of Computer Science and Engineering. He is currently serving as Professor in the Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India. His work has been published in journals of international repute. He also has two book chapters to his credit. His research interests include Algorithms, Machine Learning and Data Mining.

Geetha Maiya received her PhD degree from the Department of Mathematical and Computational Sciences, National Institute of Technology Karnataka in 2010.She is currently a Professor with the Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India. She has presented several articles in national and international conferences. Her work has also been published in several international journals. Her current research interests include Algorithms, Data Mining, Text mining in healthcare, and financial sectors.