Received: 5 August 2020 | Revised: 18 December 2020 | Accepted: 22 January 2021

DOI: 10.4218/etrij.2020-0300

ORIGINAL ARTICLE

ETRI Journal WILEY

A single-phase algorithm for mining high utility itemsets using compressed tree structures

Anup Bhat B | **Harish SV** | **Geetha M**

Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India

Correspondence
Harish SV, Department of Computer Science and Engineering, Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India.
Email: harish.sv@manipal.edu

Abstract

Mining high utility itemsets (HUIs) from transaction databases considers such factors as the unit profit and quantity of purchased items. Two-phase tree-based algorithms transform a database into compressed tree structures and generate candidate patterns through a recursive pattern-growth procedure. This procedure requires a lot of memory and time to construct conditional pattern trees. To address this issue, this study employs two compressed tree structures, namely, Utility Count Tree and String Utility Tree, to enumerate valid patterns and thus promote fast utility computation. Furthermore, the study presents an algorithm called single-phase utility computation (SPUC) that leverages these two tree structures to mine HUIs in a single phase by incorporating novel pruning strategies. Experiments conducted on both real and synthetic datasets demonstrate the superior performance of SPUC compared with IHUP, UP-Growth, and UP-Growth+ algorithms.

KEYWORDS

data mining, high utility itemsets, utility mining

1 | INTRODUCTION

Transactional databases of supermarket stores serve as a primary source for mining interesting patterns that indicate associations between different items purchased by customers. This mining task is termed as association rule mining (ARM) and used in many applications across various domains, including text mining [1], bioinformatics [2], and pharmacovigilance [3]. To obtain association rules from a huge number of customer transactions, items that co-occur frequently are enumerated to assert the validity and interestingness of associations. Hence, mining such frequent itemsets forms the core phase of the ARM task, which has been extensively researched [4].

More often than not, frequently purchased items do not necessarily contribute adequately to the revenue of a supermarket store. This is due to the inherent nature of the frequent itemset mining (FIM) model that relies only on the presence or absence of an item in a transaction when determining the frequency of the item. Formally, a measure called support is calculated as the ratio of the number of transactions, in which the items of an itemset co-occur, to the total number of transactions to decide whether the itemset is frequent or not based on a threshold set by the user. This measure does not rely on the quantity or unit profit of the items that are essential in determining the revenue incurred. Hence, a framework called the high utility itemset mining (HUIM) has evolved to consider

such factors, which can be seen as a generalized form of the FIM task.

The utility of an itemset is often measured as the product of the quantity and unit profit of the items that form the itemset. While the algorithms that mine frequent items exploit the downward closure property with respect to the support for effective search space exploration, the measure of utility is not downward closed. This void is filled by various upper bounds on the utility, with the transaction-weighted utility (TWU) being a prominent example. The algorithms for mining HUIs have evolved from two-phase to single-phase. The most efficient two-phase algorithms such as IHUP, UP-Growth, and UP-Growth+ construct tree structures and enumerate candidate patterns via a recursive pattern-growth procedure. This bottom-up procedure constructs conditional pattern trees that consume a lot of memory. This is a severe performance bottleneck. Furthermore, once candidates are outputted from such a procedure, the utility has to be determined from an additional database scan. This paper presents a single-phase algorithm called single-phase utility computation (SPUC) to efficiently utilize the concept of tree structures in mining HUIs. SPUC has the following advantages:

- It utilizes two new tree structures, namely, Utility Count Tree (UCT) and String Utility Tree (SUT). Both trees are constructed from a single database scan and are complete. While UCT guides the search space exploration for enumerating valid patterns, SUT is a compact tree that provides the utility of these patterns. This completely eliminates the need for rescanning the database to calculate the utility.
- It executes faster than IHUP, UP-Growth, and UP-Growth+ tree-based algorithms according to experiments on real and synthetic datasets.

The rest of this paper is organized as follows. Section 2 provides a formal introduction to the problem of mining HUIs, along with related work. The procedures of constructing the proposed trees, UCT and SUT, and proposed algorithm incorporating novel pruning strategies are outlined in Section 3. The results of performance evaluation are reported in Section 4. Section 5 concludes the paper.

2 | BACKGROUND

2.1 | Preliminaries

Given a transaction database D with n distinct items $I = \{i_1, i_2, \ldots, i_n\}$, each transaction T_d in D is identified by the transaction identifier *TID* and records a collection of items purchased, along with their quantities or internal utility (Table 1B). The ordered pair (i_x, q_x) in each transaction indicates that an item i_x was purchased in a quantity of q_x in that transaction. Each item is also associated with the unit profit or external utility (Table 1A).

TABLE 1 Sample database

(A) Profit Table

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

(B) Transaction Table

TID	Transaction
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) The utility $u(i, T_d)$ of an item i in a transaction T_d is measured as the product of the quantity $q(i, T_d)$ and unit profit $p(i)$.

Definition 2 (Utility of an itemset) The utility $u(X, T_d)$ of an itemset X in a transaction T_d is defined as $\sum_{i \in X \wedge X \subseteq T_d} u(i, T_d)$.

Definition 3 (Utility of an itemset in a database). The utility $u(X)$ of an itemset X in D is defined as

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

for example,

$$u(\{2\}, T_2) = 4 \times 2 = 8$$
$$u(\{2,3\}, T_2) = u(\{2\}, T_2) + u(\{3\}, T_2) = 8 + 3 = 11$$
$$u(\{2,3\}) = u(\{2,3\}, T_1) + u(\{2,3\}, T_2) + u(\{2,3\}, T_5) = 11 + 11 + 6 = 28$$

The utility measure is neither anti-monotone nor monotone. The utility of a few subsets of $\{5, 1, 2\}$ is compared in Table 2. While the support is strictly increasing across the subsets, the utility is neither increasing nor decreasing. Hence, while the support of an itemset is downward closed, the utility measure is not.

TABLE 2 Support vs Utility

Itemset	Support	Utility
{5, 1, 2}	1	18
{5, 1}	2	24
{1}	3	20

Definition 4 (Transaction Utility (TU)) The TU $TU\left(T_d\right)$ of a transaction T_d is defined as the sum of the utilities of all the items in that transaction, that is, $\sum_{i\subseteq T_d} u\left(i, T_d\right)$,

Definition 5 (High Utility Itemset (HUI)) An itemset X is called a HUI if $u(X) \geq min\_util$, where min_util is the minimum utility provided by the user.

For instance, if the threshold is set to 35%, then $min\_util = 0.35 \times 96 = 33.6$, where 96 is the sum of the TU of all the transactions in the sample database. Then, HUIs for this threshold are $\{2, 4, 5\}$, $\{2, 3, 5\}$, and $\{3, 5, 2, 4\}$, with utilities of 36, 37, and 40, respectively.

Definition 6 (Transaction Weighted Utility (TWU)). The TWU $TWU(X)$ of an itemset X is defined as the sum of the transaction utility of all the transactions in D that contain X, that is, $TWU(X) = \sum_{X\subseteq T_d \wedge T_d \in D} TU(T_d)$.

Definition 7 (High-transaction Weighted Utility Itemset (HTWUI)). An itemset X is a HTWUI, if $TWU(X) \geq min\_util$. 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 a HTWUI.

For instance, $TWU(\{3, 1, 2\}) = 30 < min\_util$. Hence, higher-order itemsets need not be enumerated from $\{3, 1, 2\}$ as this property ensures they are neither HTWUI nor HUI.

2.2 | Related work

Algorithms for mining frequent itemsets explore the combinatorial search space by employing the downward closure property with respect to the support of an itemset. However, the measure of utility is not downward closed (Definition 3). Yao et al. [5,6] performed a theoretical analysis of mining HUIs for the first time. In this study, the authors proposed two properties, namely, utility bound property and its extension, support bound property, as heuristic for pruning the search space. While this work formalized the problem of HUIM, the proposed heuristic could not discover a complete set of HUIs. The property of the TWU to be downward closed was proposed by Liu et al. [7–9]. The two-phase algorithm proposed by these authors employed the TWU to enumerate candidate patterns in a level-wise manner analogous to the Apriori algorithm [10]. The candidate generation phase of this algorithms outputs all the k-itemsets that are HTWUIs and whose utility calculation is performed by scanning the database again, which is the second phase of the algorithm. Overall, $(k + 1)$ scans are required to output HUIs of length k.

Another category of algorithms resulting in significant performance gains are tree-based algorithms [11–15]. These algorithms transform the database into a compressed tree structure with at most two database scans. Items that do not have the TWU of at least min_util are discarded, while the remaining items of the transactions are arranged in a predetermined order (for example, descending or ascending) of the item TWU or the lexicographic order. The tree construction is similar to the FP-tree construction procedure [16]. In particular, the TWU [12] or TU [13] is stored instead of storing the utility of items in the node of the tree. To promote access to the nodes of the tree that carry similar items in different branches, a header table is also constructed. This table is scanned from the bottom to obtain conditional pattern trees. Each item in the tree is then appended to the item whose conditional tree is constructed and output as a candidate. Pruning strategies play a vital role in reducing the number of candidates. In this regard, the UP-Growth algorithm [14] discards the items that are unpromising in local conditional trees (the DLU strategy) and decreases the utilities of the remaining items by the utility of the discarded items (the DLN strategy). These strategies are further tightened by using minimal node utilities in the UP-Growth+ algorithm [15].

List-based algorithms such as HUI-Miner [17] and FHM eliminate candidate generation entirely. They adopt the depth-first search strategy and its valid extensions in a single phase to mine patterns by transforming the database information into a utility list (UL) structure. Initially, a UL is constructed for every item whose TWU is at least min_util. Based on a predetermined order, items are extended recursively. Utility information remains intact in the list and is accumulated as the lists are combined during the extension of the item. Hence, unlike the tree-based algorithms, the second phase of rescanning the database for utility calculation is not required. The pruning strategy aims mainly at determining the valid extensions of an itemset. In this regard, U-Prune proposed for HUI-Miner was extended in HUP-Miner [18] by adopting two more pruning strategies, namely, PU-Prune and LA-Prune, that use the utility information stored in partitioned ULs. The FHM algorithm [19] uses the estimated utility co-occurrence structure to store the TWU of two itemsets for faster lookup.

Projection-based algorithms have proven to be more efficient than list-based algorithms [20]. EFIM [21] and d^{2}hup [22,23] are examples of projection-based algorithms.

EFIM uses local tree and sub-tree utility pruning strategies in conjunction with a database projection technique for faster exploration of the search space, while d^{2}hup explores search space via a reverse set enumeration tree and makes use of the chain of accurate utility lists for utility computation. Using real and synthetic datasets, the empirical study conducted by Zhang et al. [20] demonstrated the superior performance of EFIM and d^{2}hup on dense and sparse datasets, respectively. However, a recent study modeling the utility measure via subadditivity and monotonicity has revealed that neither list- nor projection-based algorithms perform the best at all times [24].

Pattern-growth tree-based algorithms for HUIM construct conditional pattern trees during recursive enumeration of itemsets. This procedure requires a significant amount of memory, especially for dense datasets. While list structures offer tighter upper bounds on the utility to facilitate effective pruning, the construction of ULs involves significant comparison overhead. Further, when an itemset X is extended with any of its valid extensions, it is possible that they do not co-occur in any transactions, thus waste CPU cycles for comparison operations. This can be avoided by employing a prefix tree with bottom-up traversal. Enumerated itemsets are guaranteed to be valid patterns. Nevertheless, candidates are outputted only toward the end of the first phase. Revisiting the transaction database for the utility calculation incurs a significant input/output cost. Several studies highlight the significance of the compact representation of transaction databases for FIM [25,26]. Hence, this study aims to compress the entire database on a per transaction basis and leverage the advantages of the prefix tree for mining HUIs efficiently. While the prefix tree provides with valid patterns, the compressed tree structure with utility information can be accessed to output the HUIs without rescanning the database, that is, ensuring the utility computation in a single phase.

3 | METHODOLOGY

A majority of tree-based algorithms eliminate unpromising items during the initial tree construction and require two scans of the database. This section presents two tree structures (UCT and SUT) that are constructed using a single scan by incorporating all items. While both structures ensure prefix sharing, UCT is constructed on per item basis, whereas SUT is constructed on per transaction basis. Subsection 3.1 introduces the tree structures and their respective construction procedures. Subsection 3.2 details the mining procedure using these trees, along with the pruning strategy and SPUC algorithm.

3.1 | Proposed data structures

3.1.1 | Utility count tree

A node in the UCT has the following fields:

1. *item* denoting the name of an item;
2. *count* denoting the count of an item in the given path of the tree;
3. *nodeUtility* denoting the accumulated utility of an *item* in the given path of the tree;
4. *parent* pointing to the parent of the node.

Utility Count Tree is constructed without discarding any items during the initial tree construction. The database is scanned, and a node N is constructed for every item in a transaction T_j. Algorithm 1 outlines the procedure of inserting transactions into UCT. Initially, N is set to the root node of the tree. Items in a transaction are arranged in ascending order and 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. Figure 1 shows UCT for the sample database presented in Table 1.

3.1.2 | String utility tree

Unlike UCT, SUT captures transaction-level information in a node, resulting in a more compact representation of a transaction database compared with FP-tree-like structures. A node in SUT has the following fields:

Algorithm 1 Inserting transactions into Utility Count Tree

Input: UCT, $T_j = \{(i_1, q_1), (i_2, q_2), ..., (i_p, q_p)\}$, $(i_k \epsilon I,\ 1 \leqslant k \leqslant n$ and $T_j \epsilon D)$

1: $N \leftarrow$ the root node of UCT
2: **for all** $(i_x, q_x) \in T_j$ **do**
3: **if** N has a child C such that $C.item = i_x$ **then**
4: $C.count \leftarrow C.count + 1$
5: $C.nodeUtility \leftarrow C.nodeUtility + u(i_x, T_j)$
6: **else**
7: Create a new child node C with:
8: $C.item \leftarrow i_x$, $C.count \leftarrow 1$ and $C.utility \leftarrow u(i_x, T_j)$
9: $C.parent \leftarrow N$
10: **end if**
11: $N \leftarrow C$
12: **end for**

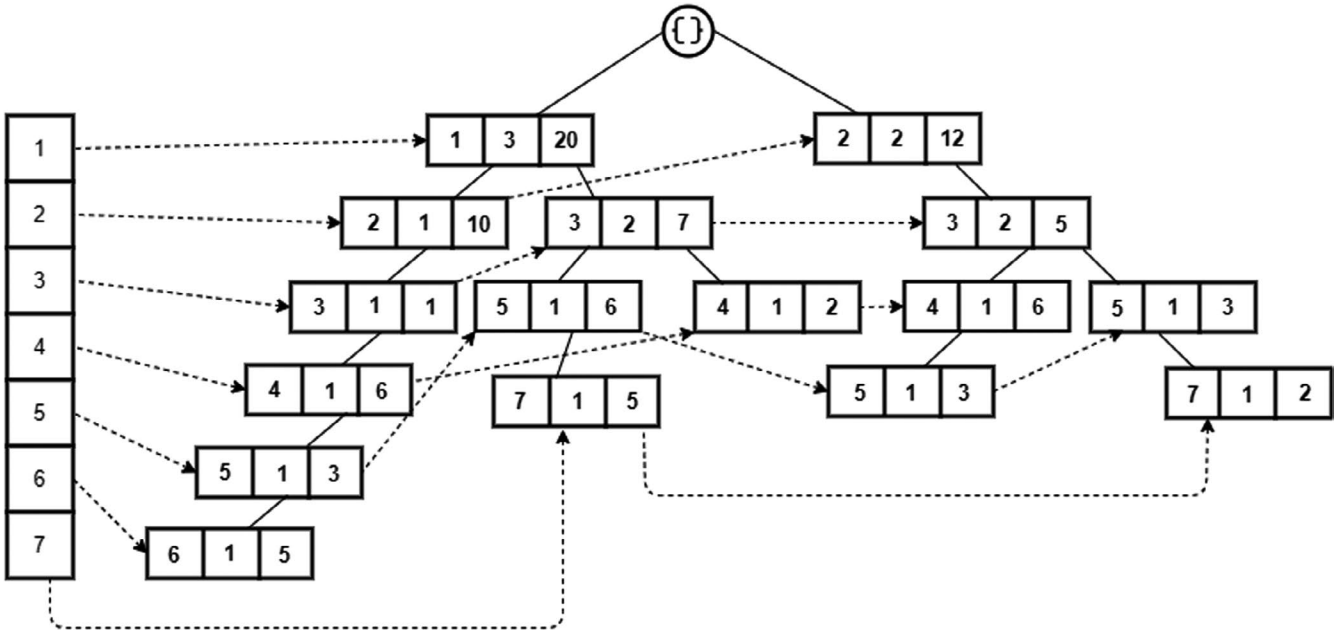

FIGURE 1 Utility Count Tree (UCT) for the database presented in Table 1

1. *stringItems* denoting the concatenation of items purchased in a transaction;
2. *TU* denoting the TU of a transaction;
3. *stringUtilities* denoting the concatenation of utilities of corresponding items;
4. *parent* pointing to the parent of the node.

Algorithm 2 outlines the construction procedure of SUT. Once the items of a transaction T_j are arranged in ascending order, they are concatenated using a delimiter such as *x* and stored in the *stringItems* field. The utilities are indexed per the order of the items and concatenated in a similar manner. As each node corresponds to a transaction, the tree offers a compact representation without eliminating any items. To ensure prefix sharing, substring comparison is performed to check whether the *stringItems* of a transaction T_j is present in existing nodes of the tree (line 11). If there is a match, then the new node is appended as the child of this existing node. Figure 2 shows SUT for the sample database presented in Table 1.

Algorithm 2 Inserting transactions into String Utility Tree

Input: $T_j = \{(i_1, q_1), (i_2, q_2), ..., (i_p, q_p)\}$,$(i_k \epsilon I,\ 1 \leqslant k \leqslant n$ and $T_j \epsilon D)$

1: **for** transaction $T_j \in D$ **do**
2: Calculate $TU(T_j)$
3: $stringItems(T_j) \leftarrow$ items concatenated as string
4: $stringUtilities(T_j) \leftarrow$ item utilities concatenated as string
5: **end for**
6: **procedure** INSERT_TRANSACTION_SUPT($root$, T_j)
7: Create a new child node S with:
8: $S.stringItems \leftarrow stringItems(T_j)$
9: $S.TU \leftarrow TU(T_j)$
10: $S.stringUtilities \leftarrow stringUtilities(T_j)$
11: **if** *root* has a child C such that $S.stringItems$ is a substring of $C.stringItems$) **then**
12: $S.parent \leftarrow C$
13: **else**
14: $S.parent \leftarrow root$
15: **end if**
16: **end procedure**

3.2 | Proposed SPUC algorithm

A path for a node in UCT is a list containing items from this node to the root. The *nodeUtility* field of a node stores the cumulated utility value of the item the node represents. This value is the sum of the utilities of the item in different transactions sharing a common prefix. From the header list, if the node link for an item is traversed, then the sum of the *nodeUtility* fields denotes the utility of the item in the transaction database. With this item as suffix, each path of the node is the prefix path for this item. To mine UCT, the header list is traversed from the bottom. The prefix path(s) for each item is (are) obtained, which forms the conditional pattern base (CPB) for the item. The sum of *nodeUtility* fields of all items, including the suffix, is also calculated for each prefix path, which denotes the path utility. All possible subsets from prefix paths containing the suffix item are then generated. The

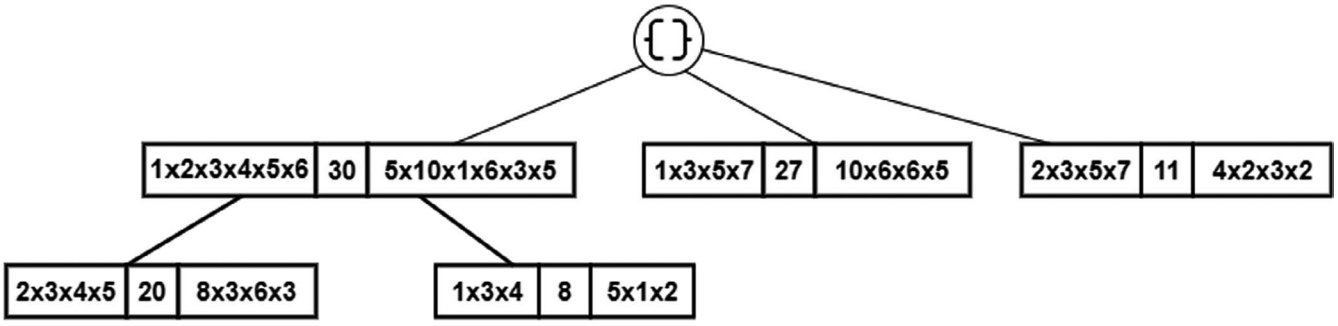

FIGURE 2 String Utility Tree (SUT) for the sample database presented in Table 1

rationale here is that these items represent the database conditioned on the suffix item. Hence, it is sufficient to mine only the *k*-itemsets generated from these items. The value of *k* ranges from one to the length of the considered prefix path.

Once the different subsets of a prefix path in the CPB of an item are obtained, the actual utility value of them cannot be obtained from UCT. Hence, SUT is employed in mining. The utility of each itemset is accumulated using level-order traversal. As each node in SUT represents a transaction, the accumulated utility value for an itemset post the traversal denotes its real utility value. The following strategy is adopted to enable efficient search of an itemset in SUT.

- First, a check for the presence of the suffix item in the current node is performed instead of the entire itemset. The node is examined for the presence of the entire itemset only if it contains the suffix item. Subsequently, the current node's children are also examined.
- If the node does not contain the suffix item, the next sibling is considered without examining the contents of the current node or any of its children.

This procedure exploits the characteristic of SUT and knowledge of the suffix item being considered. If a node does not contain the suffix item, then none of its children will contain this item since the children of a node in SUT are substrings completely contained in its parent. Furthermore, all subsets are formed conditioned on this suffix item. Hence, it is sufficient to check for the suffix before examining for the presence of the entire itemset.

The following pruning strategy has been proposed by us to improve the efficiency of the mining process:

Theorem 1 *If the sum of path utilities for an item is lower than min_util, then no itemset from the CPB will be a HUI, and hence, these itemsets need not be generated and evaluated.*

Case 1 (Isolated path): An isolated path is a non-prefix sharing path in a tree. Hence, *nodeUtility* for all items is the utility value. For any suffix item i_k in this path, the path utility is the utility of the largest *k*-itemset possible. For any subset of this *k*-itemset that contains i_k as the suffix, the utility cannot exceed the utility of the largest *k* -itemset, that is, the path utility. Hence, if this utility is lower than *min_util*, the utility of the subsets cannot exceed *min_util*. Therefore, the subsets for the paths in the CPB do not need to be generated with respect to the suffix.

Case 2 (Non-isolated path): In the case of prefix sharing, *nodeUtility* has the utility value greater than the real utility value, that is, it denotes the sum of the utilities of different transactions along the path for an item. Thus, any subset formed for the suffix item i_k will have a utility lower than the path utility. If two or more prefix paths are present for i_k and a common subset exists, the sum of the utilities of subsets will always be lower than the sum of the path utilities (since individually, every utility is lower than the path utility of the path they occur in).

An example of mining employing UCT, SUT, and the proposed pruning strategy is described below.

- Consider mining HUIs for itemsets with a suffix item 7. As the header list is traversed, two prefix paths are obtained, namely, P_1: ⟨1, 3, 5, 7⟩ and P_2: ⟨2, 3, 5, 7⟩, with path utilities $pu_1 = 38\,(5 + 6 + 7 + 20)$ and $pu_2 = 22\,(2 + 3 + 5 + 12)$, respectively (Figure 1). This forms the CPB for item 7 as shown in Table 3. If *min_util* is set to 30, then item 7 is not pruned as the sum of the path utilities $(38 + 22 = 60)$ exceeds *min_util*.
- Subsets with suffix 7 can be generated using these prefix paths. Since both paths have a length of 4, the largest possible subset is of length four. When the prefix path

TABLE 3 Conditional pattern base for item 7

Prefix Path	Items	Path Utility
P_1	1, 3, 5, 7	38
P_2	2, 3, 5, 7	22

P_1 is considered from the CPB, the generated subsets are $\{\{7\}, \{1,7\}, \{3,7\}, \{5,7\}, \{1,3,7\}, \{1,5,7\}, \{3,5,7\}, \{1,3,5,7\}\}$.

- Next, SUT is traversed to obtain the utilities of these itemsets. The traversal of SUT begins from its *root* node. The first match to the suffix occurs at node 1*x*3*x*5*x*7. The utilities of all itemsets are obtained from the *utility* field of this node and are recorded in the *hash_item_utilites* hash table. A snapshot of this table is shown in Table 4.
- The next sibling found during traversal, that is, 2*x*3*x*5*x*7, contains the suffix. However, only the itemsets $\{7\}, \{3,7\}, \{5,7\}$, and $\{3,5,7\}$ are examined as present in this node, and their utilities are updated in the hash table as shown in Table 5.
- Once the traversal is complete, the next prefix path, P_2, is selected from the CPB. The subsets generated from this path are $\{\{7\}, \{2,7\}, \{3,7\}, \{5,7\}, \{2,3,7\}, \{2,5,7\}, \{3,5,7\}, \{2,3,5,7\}\}$ Since the itemsets $\{\{7\}, \{3,7\}, \{5,7\}, \{3,5,7\}\}$ have already been generated in P_1 and examined, only the remaining itemsets, that is, $\{\{2,7\}, \{2,3,7\}, \{2,5,7\}, \{2,3,5,7\}\}$ are taken up for mining. The utilities of these itemsets are recorded in *hash_item_utilites* as shown in Table 6.

TABLE 4 Utilities of subsets of P_1 in *hash_item_utilites* after examining the node 1*x*3*x*5*x*7

Itemset	Utility
{7}	5
{1,7}	15
{3,7}	11
{5,7}	11
{1,3,7}	21
{1,5,7}	21
{3,5,7}	17
{1,3,5,7}	27

TABLE 5 Utilities of subsets of P_1 in *hash_item_utilites* after examining the node 2*x*3*x*5*x*7

Itemset	Utility
{7}	7
{1,7}	15
{3,7}	15
{5,7}	16
{1,3,7}	21
{1,5,7}	21
{3,5,7}	24
{1,3,5,7}	27

- The above procedure is repeated for all items of the header list in a bottom-up manner. Eventually, *hash_item_utilities* is filtered to retain only those itemsets whose utility is at least *min_util*.

3.2.1 | Enhancing the mining process

As demonstrated in the above example, there is a possibility that the prefix paths within a CPB can generate the same subsets. While the same subsets are not considered during SUT traversal for the utility calculation, they are still generated. To tackle this problem, the CPB is checked for the presence of a path that contains all the different items in the CPB. Such a path has to be the longest path, with the length equal to the suffix item being considered. Such a path will generate all possible subsets, including those generated by the remaining prefix paths in the CPB.

Consider the suffix item 5. The possible prefix paths are P_1: ⟨1, 2, 3, 4, 5⟩, P_2: ⟨1, 3, 5⟩, P_3: ⟨2, 3, 4, 5⟩, and P_4: ⟨2, 3, 5⟩. Here, the longest prefix path with a length equal to the suffix item is P_1. The subsets generated out of P_1 include those generated by the paths from P_2 to P_4. Hence, it is sufficient to generate subsets from P_1 for subsequent mining. The remaining paths can be ignored when generating subsets.

Once the subsets corresponding to a prefix path are generated, their utilities can be obtained by traversing SUT. The *nodeUtility* value stored in UCT is employed to reduce the number of the subsets being evaluated for the utility computation. The *nodeUtility* values corresponding to every item in the CPB are accumulated as the prefix paths are determined. This results in an overestimated utility value that can be employed to filter itemsets as explained below using the subsets of P_1 of *CPB* (7).

TABLE 6 Utilities of subsets of P_2 in *hash_item_utilites* after examining the node 2*x*3*x*5*x*7

Itemset	Utility
{7}	7
{1,7}	15
{3,7}	15
{5,7}	16
{1,3,7}	21
{1,5,7}	21
{3,5,7}	24
{1,3,5,7}	27
{2,7}	6
{2,3,7}	8
{2,5,7}	9
{2,3,5,7}	11

- An overestimated utility (OU) table for *CPB* (7) accumulates the *nodeUtility* values corresponding to every item in *CPB* (7). The OU values are accumulated as and when items are added to the prefix path of the CPB. Table 7 shows the OU values for items $\{1, 2, 3, 5, 7\}$ that form *CPB* (7). Items $\{1\}$, $\{2\}$ are present only in P_1 and P_2; their OUs hold the corresponding *nodeUtility* values of 20 and 12, respectively. However, the OUs of the other items in both P_1 and P_2 are also accumulated, that is, $OU(\{3\}) = P_1(\{3\}).nodeUtility + P_2(\{3\}).nodeUtility = 7 + 5 = 12$.
- Next, OUs of different subsets formed from P_1 are determined using the above-mentioned OU table (Table 8). It can be observed from Table 8 that only the itemsets $\{1, 3, 7\}$, $\{1, 5, 7\}$, and $\{1, 3, 5, 7\}$ have their OUs above the *min_util* of 30. Hence, only the utilities of these itemsets are computed using SUT (which is different from Tables 4 and 5).

This strategy further enhances the mining process in conjunction with the pruning strategy provided in Theorem 1.

Algorithm 3 lists the SPUC algorithm that incorporates the proposed pruning strategy. SPUC takes two trees (UCT and SUT) and the *min_util* values provided by the user as inputs. A global hash map acts as a table for storing the utilities of subsets generated from different prefix paths as shown in line 1. A bottom-up procedure is then initiated to obtain the CPB of items for subsequent mining (lines 2 to 18). After calculating the path utilities for each prefix path, the pruning strategy (Theorem 3.2) is applied (line 4). The mining procedure considers the next item in the header list if the sum of the path utilities is not at least *min_util*.

TABLE 7 Overestimated utility table corresponding to items of *CPB* (7)

Itemset	OU
{1}	20
{2}	12
{3}	12
{5}	9
{7}	7

TABLE 8 Overestimated utility table of subsets in P_1 of *CPB* (7)

Itemset	OU
{7}	7
{1, 7}	27
{3, 7}	19
{5, 7}	16
{1, 3, 7}	39
{1, 5, 7}	36
{3, 5, 7}	28
{1, 3, 5, 7}	48

Algorithm 3 Single-phase Utility Computation Algorithm: Mining using UCT and SUT

Input: $UCT, SUT, minUtil$

```
1: HashMap hash_item_utilities(List_Integer itemset, Integer utility)
2: for each item i from the bottom of H do
3:     CPB(i) ← Get all prefix paths of i and calculate the path utility
4:     if sum of the path utilities >= minUtil then
5:         if ∃ a path p ∋ p.length = i then
6:             discard all paths except p from CPB(i)
7:         end if
8:         for each prefix path, p ∈ CPB(i) do
9:             itemset_list ← generate subsets from the items in p that include i
10:            itemset_list ← itemset_list \ subset_List
11:            subset_List ← subset_List ⋃ itemset_list
12:            for each itemset X in itemset_list do
13:                if OU(X) < minUtil then
14:                    itemset_list ← itemset_list \ X
15:                end if
16:            end for
17:            Call MINE(itemset_list, SUT.root)
18:        end for
19:    end if
20: end for
21: procedure MINE(itemset_list, SUT.node)
22:    if SUT.node is not the root node then
23:        for each itemset of itemset_list do
24:            if itemset.suffix is present in SUT.node then
25:                if SUT.node contains itemset then
26:                    utility ← hash_item_utilities.Get(itemset)
27:                    if itemset is not in hash_item_utilities then
28:                        utility ← 0
29:                    end if
30:                    Calculate utility of itemset from SUT.node.utility
31:                    hash_item_utilities.Put(itemset, utility)
32:                end if
33:            else
34:                break
35:            end if
36:        end for
37:    end if
38:    for each child of SUT.node do
39:        Call MINE(itemset_list, child)
40:    end for
41: end procedure
```

For items that satisfy this condition, the longest path is searched to avoid generating subsets from each path in the CPB (lines 5 to 7). Itemsets that are subsets of each prefix path and contain only the suffix item are then generated (line 9). *subset_List* keeps track of different itemsets generated from the prefix paths considered till the current one and thus can be used to eliminate any subsets common to any of the previous paths (lines 10 and 11). As the OUs of the itemsets are determined, only those itemsets whose OU exceeds the threshold are retained in *itemset_list* (line 12 to 16). The procedure Mine takes the filtered *itemset_list* as input and starts searching for the presence of itemsets from the *root* of SUT (line 23 to 32). Line 24 is the summary of the search strategy explained before. If the suffix is absent in the current SUT node, then none of the itemsets in *itemset_list* will be present in the current node. Hence, the search proceeds with the next sibling after the break (line 34). The utility of the itemset is calculated and added to *hash_item_utilities* if not present already (lines 26 to 30). Traversal is continued as shown in lines 38 to 40 to mine itemsets from the child of the current node or move to siblings if the suffix item is not present.

3.3 | Complexity analysis

The longest path in the CPB of an item is identified using the assumption that its length matches the location of the item in the header list of UCT. If such a path exists, then the number of subsets with this item as prefix is 2^{n-1} (assuming that the sum of the path utilities of this item exceeds *min_util*). However, if such a path is absent, then in the worst case, all prefix paths in the CPB have to be examined for generating subsets. Hence, the computation load due to item i depends on the number of prefix paths in its CPB. Furthermore, the subsets of item i will visit each node in SUT and perform the utility calculation when a match is found. Subsequently, the subsets of i will visit all child nodes as well. The utility computation is a trivial retrieval operation, whereas the number of node visits during mining is a major component that adds to the complexity. Overall, the complexity can be computed as the total number of subsets generated for n items in the header list of UCT.

$$\begin{aligned}\text{Number of prefix paths} &= 2^{n-1}+2^{n-2}+\cdots+2+1\\ &= 2\left(2^{n-1}-1\right)\\ &\approx 2^n, \text{ for large values of } n\end{aligned}$$

Hence, the computational complexity of the proposed algorithm is $\mathcal{O}(2^n)$.

The tree structures are constructed using a single database scan without eliminating any items. To determine the amount of memory taken by the two tree structures, let $|D|$ denote the total number of transactions in the database and b denote the number of bytes taken up to allocate memory for a field of a node in a tree. In the worst case scenario, the sum of the numbers of nodes in each level of UCT can determine the memory upper bound.

$$\begin{gathered}\text{Number of item nodes at depth } 0 = 0 \text{ (the root node)}\\ \text{Number of item nodes at depth } 1 = n_1\\ \text{Number of item nodes at depth } 2 = n_2\\ \vdots\\ \text{Number of item nodes at depth } n = n_n\end{gathered}$$

As UCT has three fields in each node, the total space taken is upper-bounded by $3b*\left(0+\binom{n}{1}+\binom{n}{2}+\cdots+\binom{n}{n}\right)=3b*(2^n-1)$ bytes. Hence, in the worst case, UCT consumes $\mathcal{O}(2^n)$ bytes of memory. A tighter upper bound on the memory consumed by SUT can be provided using the average length of transactions, T_{avg}. Each of the fields *stringItems* and *stringUtilities* consumes $b * T_{\text{avg}}$ bytes, and an additional b bytes will be taken up by the *TU* field. Since the total number of nodes in SUT is $|D|$, the taken space is $b * \left(2 * |D| * T_{\text{avg}} + 1\right)$ bytes. Hence, the worst case memory space for SUT is $\mathcal{O}\left(|D| * T_{\text{avg}}\right)$.

4 | EXPERIMENTAL EVALUATION

To evaluate the proposed algorithm, it was compared with IHUP, UP-Growth, and UP-Growth+. The Java implementation of these algorithms is provided by SPMF [27]. IHUPTWU was used in the experiments as it has shown to be efficient [13]. *Foodmart* provided by SPMF [28] was used a real dataset. Furthermore, three synthetic datasets, *s*1, *s*2, and *s*3, were generated using the transaction database generator included in the SPMF toolbox. For these datasets, the quantities of items (integral values) were generated in the range of $[1, 10]$ using a uniform distribution, while the unit profit values followed a Gaussian distribution. Table 9 summarizes the characteristics of the

TABLE 9 Characteristics of Datasets

Dataset	$\|D\|$	$\|I\|$	T	*Density* (%)
Foodmart	4141	1559	4.4	0.28
*s*1	10 000	1000	5.5	0.054
*s*2	10 000	50 000	5.5	0.016
*s*3	10 000	100 000	5.4	0.013

datasets, where $|D|$ denotes the number of transactions in the database, $|I|$ denotes the number of distinct items, T denotes the average number of items per transaction, and *Density* indicates the extent to which each dataset is sparse or dense and is calculated as $T/|I|$. The experiments were conducted on a Windows 7 computer equipped with 8 GB RAM and Intel Core i5 processor working at 3.00 GHz.

Section 4.1 presents the performance evaluation of the proposed pruning strategies. The execution time of SPUC, IHUP, UP-Growth, and UP-Growth+ is compared in Section 4.2. The results of scalability tests using synthetic datasets are presented in Section 4.3.

4.1 | Evaluation of the pruning strategies

Foodmart and *s*2 datasets were used to evaluate the effectiveness of the proposed pruning strategies. The first pruning strategy that employs the path utility upper bound is denoted as *SPUC_Prune* (1) and the second pruning strategy that discards the itemsets based on *OU* is termed *SPUC_Prune* (2). Figure 3 compares the execution time when SPUC was executed with only *SPUC_Prune* (1) as against both, that is, *SPUC_Prune* (1 + 2). For both the datasets, across higher thresholds the difference in execution time was more evident. *SPUC_Prune* (1) effectively prunes the items that appear at the top of the header list due to their lower path utility. Hence, as the threshold increases, *SPUC_Prune* (1) avoids determination of CPB for a greater number of items. In addition to this, *SPUC_Prune* (2) ensures lesser itemsets to be evaluated for utility computation and thus completes the mining faster. Overall, an improvement of 0.8992% and 23.41% was observed for *Foodmart* and *s*2 datasets, respectively. Figure 4 compares the pruning strategies in terms of the explored number of itemsets. While the number of explored candidate itemsets for *Foodmart* remained same up to *min_util* of 1000 for the first pruning strategy, *SPUC_Prune* (2) further pruned given this low threshold, thus reducing the mining time at low thresholds. The effectiveness of *SPUC_Prune* (2) in conjunction with *SPUC_Prune* (1) is more evident in the case of *s*2 with a lower number of explored itemsets and a significant difference as the threshold increased. Overall, *SPUC_Prune* (1 + 2) improved the mining performance by reducing the number of candidates and hence was adopted for the remaining experiments.

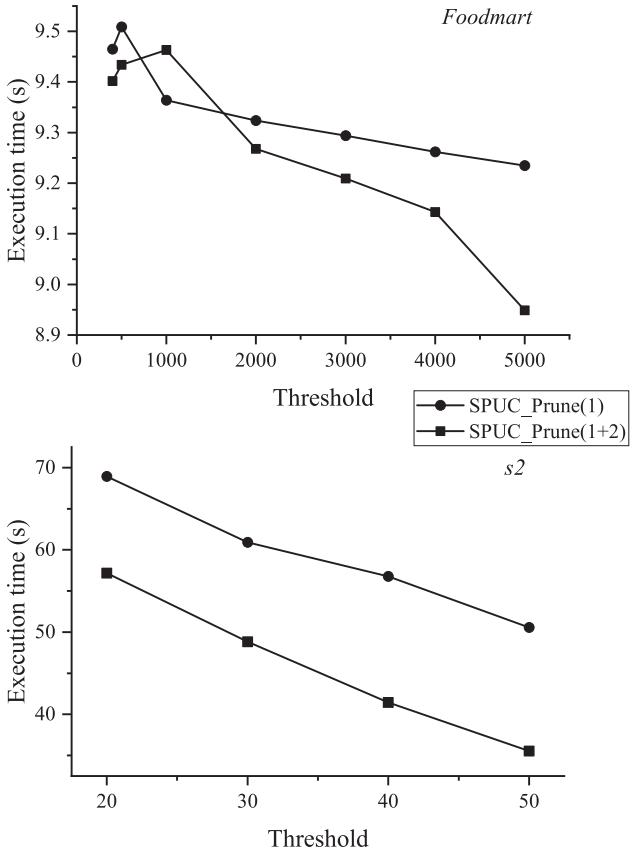

FIGURE 3 Comparison of the pruning strategy in terms of their execution time

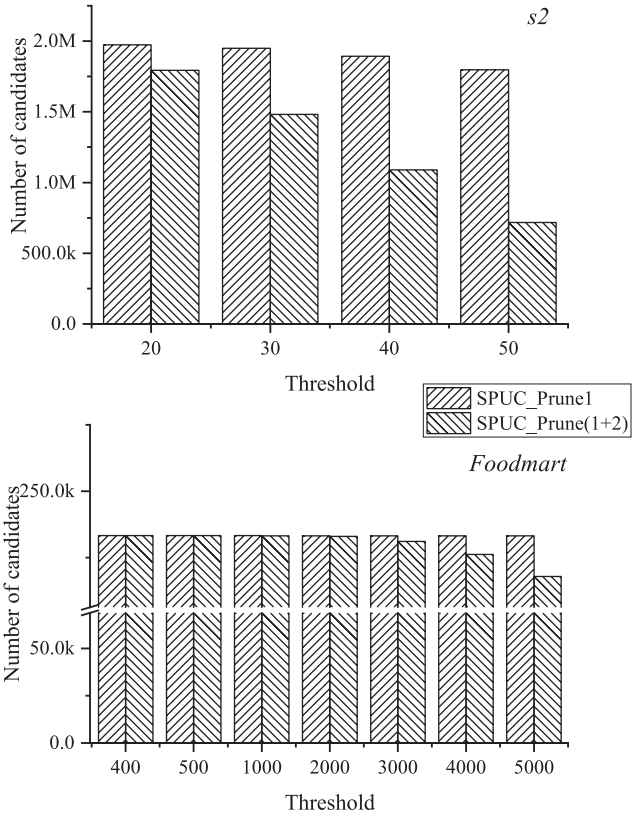

FIGURE 4 Comparison of the pruning strategy in terms of the number of candidates

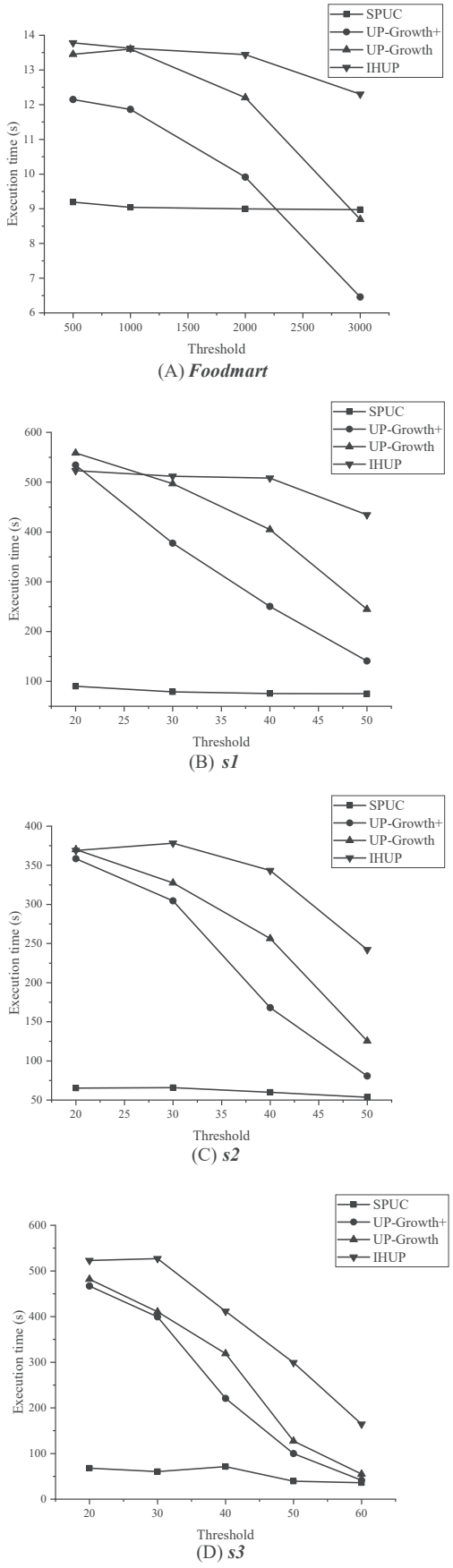

FIGURE 5 Execution time of the algorithms on different datasets

4.2 | Execution time comparison

Figure 5 shows the comparison of the execution times of the different algorithms. It can be noticed from the figure that the proposed SPUC algorithm clearly outperforms the other algorithms. The time taken for mining gradually reduces at higher *min_util* due to the lower number of candidates. However, the difference in time for any two consecutive thresholds is significantly higher for the benchmark algorithms (Table 10). In contrast, the execution time does not vary significantly in the case of SPUC. This can be attributed to the fact that SPUC relies on UCT and SUT that do not eliminate any items as unpromising; hence, the tree structure remains the same across all the thresholds. In contrast, the benchmark algorithms eliminate unpromising items and hence explore a smaller part of the search space for mining. In addition, the recursive mining procedure involves tree construction after eliminating local unpromising items. However, this is overcome in the case of SPUC, where itemsets are directly generated and filtered. Furthermore, the utilities are determined on the fly, without requiring an additional database scan. Table 11 lists the percentage improvement in the execution time of SPUC over IHUP, UP-Growth+, and UP-Growth.

4.3 | Scalability test

Scalability tests were conducted to determine the impact of the database size increase in terms of the number of transactions on the performance of SPUC. The three synthetic datasets were scaled in four steps by inserting 10 000 transactions at each step. Figure 6 shows the execution times of

TABLE 10 Average increase in the execution time (*s*)

Dataset	SPUC	UP-Growth+	UP-Growth	IHUP
Foodmart	0.073	0.493	1.890	2.450
*s*1	4.974	131.162	104.560	29.520
*s*2	3.860	92.580	81.550	48.450
*s*3	8.028	106.397	106.676	89.675

TABLE 11 Percentage improvement of the proposed algorithm compared with the benchmark algorithms across the datasets

Datasets vs. Algorithms	UP-Growth+	UP-Growth	IHUP
Foodmart	18.370	22.143	31.760
*s*1	69.640	79.640	83.780
*s*2	81.820	82.310	82.340
*s*3	62.460	70.576	84.650

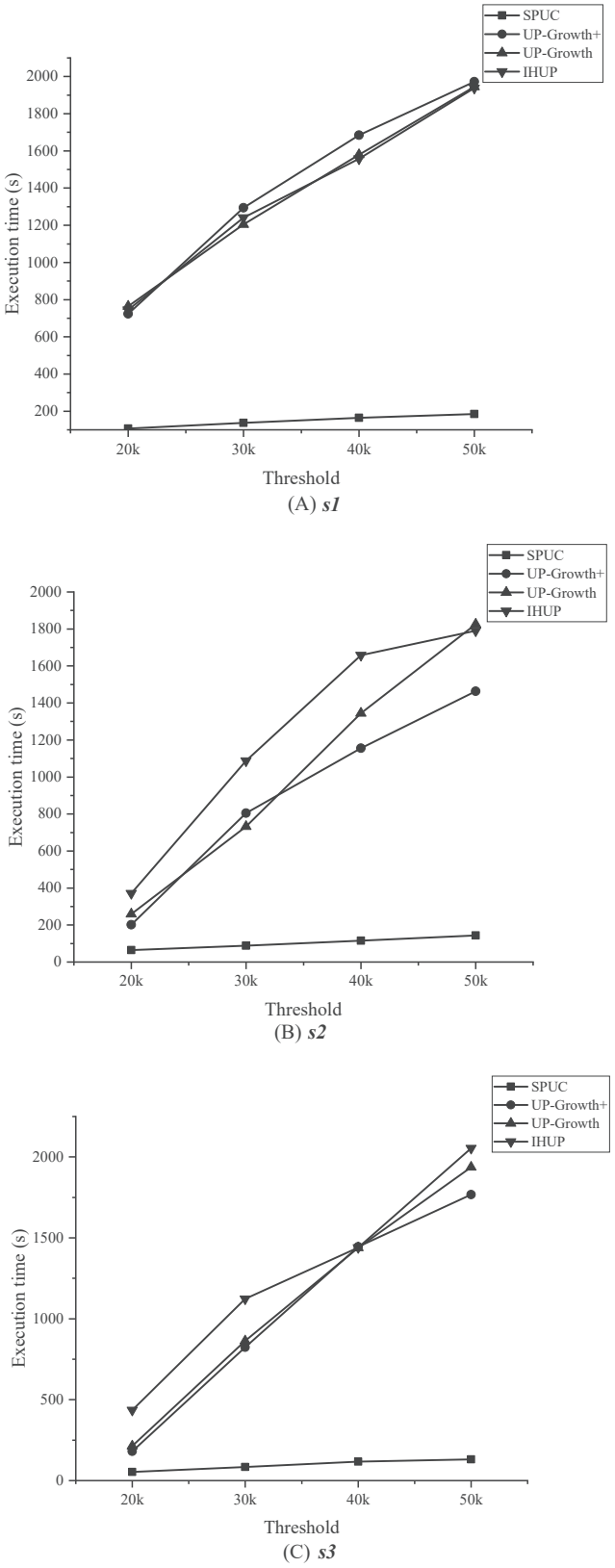

FIGURE 6 Execution time of the compared algorithms for varying $|D|$ on the synthetic datasets

SPUC and the benchmark algorithms. While all the algorithms showed an increase in the execution time with the increase in the number of transactions, the execution time of SPUC increased by a small margin. This is because the tree construction time required by the benchmark algorithms, which use two scans, builds up as the number of transactions to be processed increases. In contrast, SPUC constructs the two trees in a single scan. Furthermore, unlike the other algorithms, SPUC completely eliminates the evaluation of candidates for the utility computation, thus provides better scalability.

5 | CONCLUSIONS

Tree-based algorithms for mining HUIs require two phases: (1) constructing a tree structure and mining candidate patterns and (2) rescanning the database for calculating candidate utilities. This paper proposed two tree structures called UCT and SUT. While SUT stores transaction-level information in a node, UCT stores item-level information. Furthermore, the paper presented a mining algorithm called SPUC for mining HUIs in a single phase by employing new pruning strategies based on the path and overestimated utility, respectively. In SPUC, UCT guides the pattern generation process, while SUT helps in calculating candidate utilities. This enables SPUC to completely eliminate the second phase and thus outperform existing tree-based algorithms on both real and synthetic datasets.

With the profound improvement in storage technologies and explosion of data generation rate, mining itemsets is considered to be feasible through big data technologies such as MapReduce and Apache Spark. Accordingly, we plan to extend SPUC for mining HUIs in distributed environments and very large datasets.

ACKNOWLEDGMENTS

This work was supported by Manipal Academy of Higher Education Dr. T.M.A Pai Research Scholarship under Research Registration No. 170900117.

CONFLICT OF INTEREST

The authors declare no potential conflict of interest.

ORCID

Anup Bhat B https://orcid.org/0000-0002-9910-6758

REFERENCES

1. W. Zhang et al., *Text clustering using frequent itemsets*, Knowl-Based Syst. **23** (2010), no. 5, 379–388.

2. S. Naulaerts, et al., *A primer to frequent itemset mining for bioinformatics*, Brief Bioinform. **16** (2015), 216–231.
3. R. Harpaz, H. S. Chase, and C. Friedman, *Mining multi-item drug adverse effect associations in spontaneous reporting systems*, BMC Bioinform. **11** (2010), no. 9, S7.
4. J. Han et al., *Frequent pattern mining: Current status and future directions*, Data Min. Knowl. Disc. **15** (2007), no. 1, 55–86.
5. H. Yao, H. J. Hamilton, and C. J. Butz, *A foundational approach to mining itemset utilities from databases*, in Proc. SIAM Int. Conf. Data Min. (Lake Buena Vista, FL, USA), Apr. 2004, pp. 482–486.
6. H. Yao and H. J. Hamilton. *Mining itemset utilities from transaction databases*, Data Knowl. Eng. **59** (2006), no. 3, 603–626.
7. Y. Liu and W.-K. Liao, *A fast high utility itemsets mining algorithm*, in Proc. Int. Workshop Utility-Based Data Min. (New York, NY, USA), Aug. 2005, pp. 90–99.
8. Y. Liu, W.-K. Liao, and A. Choudhary, *A two-phase algorithm for fast discovery of high utility itemsets*, in Advances in Knowledge Discovery and Data Mining, Springer, Berlin, Heidelberg, Germany, 2005, pp. 689–695.
9. Y. Liu et al., *High utility itemsets mining*, Int. J. Inf. Tech. Decis. Making **9** (2010), no. 6, 905–934.
10. R. Agrawal and R. Srikant, *Fast algorithms for mining association rules*, in Proc. Int. Conf. Very Large Data Bases (Santiago, Chile), Sept. 1994, 487–499.
11. C. W. Lin, T. P. Hong, and W. H. Lu, *An effective tree structure for mining high utility itemsets*, Expert Syst. Appl. **38** (2011), no. 6, 7419–7424.
12. C. F. Ahmed et al., *HUC-Prune: An efficient candidate pruning technique to mine high utility patterns*, Appl. Intell. **34** (2011), no. 2, 181–198.
13. C. F. Ahmed et al., *Efficient tree structures for high utility pattern mining in incremental databases*, IEEE Trans. Knowl. Data Eng. **21** (2009), no. 12, 1708–1721.
14. V. S. Tseng et al., *UP-Growth: An efficient algorithm for high utility itemset mining*, Discov. Data Min. (New York, NY, USA), July (2010), 253–262.
15. V. S. Tseng et al., *Efficient algorithms for mining high utility itemsets from transactional databases*, IEEE Trans. Knowl. Data Eng. **28** (2016), no 1, 54–67.
16. J. Han et al., *Mining frequent patterns without candidate generation: A frequent-pattern tree approach*, Data Min. Knowl. Disc. **8** (2004), no. 1, 53–87.
17. M Liu and J Qu, *Mining high utility itemsets without candidate generation*, in Proc. ACM Int. Conf. Inform. Knowl. Manag. (New York, NY, USA), Oct. 2012, pp. 55–64.
18. S. Krishnamoorthy, *Pruning strategies for mining high utility itemsets*, Expert Syst. Appl. **42** (2015), no. 5, 2371–2381.
19. P. Fournier-Viger et al., *Fhm: Faster high-utility itemset mining using estimated utility co-occurrence pruning*, in International Symposium on Methodologies for Intelligent Systems, Springer, Berlin, Heidelberg, Germany, 2014, pp. 83–92.
20. C. Zhang et al., *An empirical evaluation of high utility itemset mining algorithms*, Expert Syst. Appl. **101** (2018), 91–115.
21. S. Zida et al., *Efim: A fast and memory efficient algorithm for high-utility itemset mining*, Knowl. Inf. Syst. **51** (2017), no. 2, 595–625.
22. J. Liu, K. E. Wang, and B. C. M. Fung, *Direct discovery of high utility itemsets without candidate generation*, in Proc. IEEE Int. Conf. Data Min. (Brussels, Belgium), Dec. 2012, pp. 984–989.
23. J. Liu, K. Wang, and B. C. M. Fung, *Mining high utility patterns in one phase without generating candidates*, IEEE Trans. Knowl. Data Eng. **28** (2016), no. 5, 1245–1257.
24. S. Dawar, D. Bera, and V. Goyal, *High-utility itemset mining for subadditive monotone utility functions*, arXiv preprint, CoRR, 2018, arXiv:1812.07208.
25. V. S. Ananthanarayana, D. K. Subramanian, and M. N. Murty, *Scalable, distributed and dynamic mining of association rules*, in High Performance Computing—HiPC 2000, vol. 1970, Springer, Berlin, Heidelberg, Germany, 2000, pp. 559–566.
26. M. Geetha and R. J. D'souza, *An efficient reduced pattern count tree method for discovering most accurate set of frequent itemsets*, Int. J. Comp. Sci. Netw. Sec. **8** (2008), no. 8, 121–126.
27. P. Fournier-Viger, SPMF An Open-Source Data Mining Library, Developer's Guide, 2020, available at https://www.philippe-fournier-viger.com/spmf/index.php?link=developers.php
28. P. Fournier-Viger, SPMF An Open-Source Data Mining Library, Datasets, 2020. available at https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php

AUTHOR BIOGRAPHIES

Anup Bhat B received his BE degree in Information and Communication Technology and MTech degree in Computer Science and Engineering from the Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India, in 2013 and 2017, respectively. He is currently pursuing a PhD degree in the same institute. His research interests include data mining and machine learning.

Harish SV received his PhD degree from the Department of Computer Science and Engineering, National Institute of Technology Karnataka, in 2011. He is currently serving as a Professor in the Department of Computer Science and Engineering, the Manipal Institute of Technology, Manipal Academy of Higher Education, Manipal, India. He published two book chapters and many papers in reputable international journals. His research interests include algorithms, machine learning, and data mining.

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