Appl Intell (2011) 34: 181–198
DOI 10.1007/s10489-009-0188-5

HUC-Prune: an efficient candidate pruning technique to mine high utility patterns

Chowdhury Farhan Ahmed · Syed Khairuzzaman Tanbeer · Byeong-Soo Jeong · Young-Koo Lee

Published online: 14 July 2009

Abstract Traditional frequent pattern mining methods consider an equal profit/weight for all items and only binary occurrences (0/1) of the items in transactions. High utility pattern mining becomes a very important research issue in data mining by considering the non-binary frequency values of items in transactions and different profit values for each item. However, most of the existing high utility pattern mining algorithms suffer in the level-wise candidate generation-and-test problem and generate too many candidate patterns. Moreover, they need several database scans which are directly dependent on the maximum candidate length. In this paper, we present a novel tree-based candidate pruning technique, called HUC-Prune (High Utility Candidates Prune), to solve these problems. Our technique uses a novel tree structure, called HUC-tree (High Utility Candidates tree), to capture important utility information of the candidate patterns. HUC-Prune avoids the level-wise candidate generation process by adopting a pattern growth approach. In contrast to the existing algorithms, its number of database scans is completely independent of the maximum candidate length. Extensive experimental results show that our algorithm is very efficient for high utility pattern mining and it outperforms the existing algorithms.

C.F. Ahmed · S.K. Tanbeer · B.-S. Jeong (✉) · Y.-K. Lee
Department of Computer Engineering, Kyung Hee University, 1 Seochun-dong, Kihung-gu, Youngin-si, Kyunggi-do, 446-701, Republic of Korea
e-mail: jeong@khu.ac.kr

C.F. Ahmed
e-mail: farhan@khu.ac.kr

S.K. Tanbeer
e-mail: tanbeer@khu.ac.kr

Y.-K. Lee
e-mail: yklee@khu.ac.kr

Keywords Data mining · Knowledge discovery · Frequent pattern mining · High utility pattern mining

1 Introduction

Data mining techniques can efficiently discover hidden knowledge from databases. Frequent pattern mining [1, 3, 4, 9, 12–14, 26] plays an essential role in many data mining tasks such as association rule mining, classification, clustering, time-series mining, graph mining, web mining, and so on. The initial solution of the frequent pattern mining, the *Apriori* algorithm [3, 4], is based on the candidate generation-and-test methodology and requires several database scans. In the first database scan, it finds all the 1-element frequent itemsets and based on that, it generates candidates for 2-element frequent itemsets. In the second database scan, it finds all the 2-element frequent itemsets and based on that, it generates the candidates for 3-element frequent itemsets and so on. This level-wise candidate generation process may create the problems of several database scans and huge candidate pattern generation. Han et al. [14] solved these problems by introducing a prefix tree (FP-tree)-based algorithm without candidate set generation and testing. This algorithm is called the frequent-pattern growth or FP-growth algorithm and needs two database scans.

Although frequent pattern mining plays an important role in data mining applications, it has two limitations. First, it treats all items with the same importance/weight/price and second, in one transaction each item appears in a binary (0/1) form, i.e. either present or absent. In the real world, however, each item in the supermarket has a different importance/price and one customer can buy multiple copies of an item. Therefore, finding only traditional frequent patterns in a database cannot fulfill the requirement of finding the most valuable customers/itemsets that contribute the most to the total profit in a retail business.

A high utility mining [36, 37] model was defined for retrieving more useful information from a database. We can measure how useful the itemset is by its utility. This allows us to handle a dataset with different price values for each item and it more accurately represents real world market data. By utility mining several important business area decisions like maximizing revenue, minimizing marketing and/or inventory costs can be considered and knowledge about itemsets/customers contributing to the majority of the profit can be discovered. In addition to the real world retail market, if we consider examples such as the biological gene database and web click streams, then the importance of each gene or website is different and their occurrences are not limited to binary values. Other application areas, such as stock tickers, network traffic measurements, web-server logs, data feeds from sensor networks, and telecom call records can have similar solutions.

To understand the necessity of high utility patterns in today's market, we can consider a small example in the real world market basket databases where different items have different profit values and where different items in a transaction have different selling quantities. Consider an example where customer C_1 has bought 3 pens, 4 pencils and 1 eraser; customer C_2 has bought 1 gold ring; customer C_3 has bought 3 loaves of bread and 5 cartons of milk; and customer C_4 has bought 2 shirts and 1 pair of shoes. According to our real world profit values, we can assume that the profit value of a gold ring is much larger than the profit values of other items mentioned in this example. Therefore, the businessman gets more profit from customer C_2 although the selling frequency is only one. This example demonstrates that selling quantity of an itemset is much less important than its total profit. For this reason, finding high utility patterns is more useful than finding only frequent patterns.

Most of the existing high utility pattern mining algorithms [7, 17, 19, 20, 36, 37] suffer from the level-wise candidate generation-and-test problem and they need several database scans depending on the length of the candidate high utility patterns. Therefore, the most challenging tasks are to efficiently reduce the number of candidates and number of database scans. In this paper, we propose a novel candidate pruning technique, called HUC-Prune (High Utility Candidates Prune), which avoids the level-wise candidate generation-and-test problem by exploiting a pattern growth mining technique. Our technique first finds the length-one candidates in one database scan. In the second database scan, it uses a novel tree structure, called HUC-tree (High Utility Candidates tree), to capture important utility information of the transactions. After that, it discovers all the candidate high utility patterns by using a pattern growth mining approach. Finally, a third database scan is needed to determine the actual high utility patterns from the candidate patterns. Therefore, our technique needs a maximum of three database scans to determine the complete set of high utility patterns, i.e. its number of database scans is not dependent on the length of the candidate high utility patterns. We demonstrate that our technique works efficiently for synthetic and real datasets, which can be dense or sparse, and it outperforms the existing algorithms.

The remainder of this paper is organized as follows. In Sect. 2, we describe related work. In Sect. 3, we describe the high utility pattern mining problem. In Sect. 4, we describe the proposed HUC-Prune technique for high utility pattern mining. In Sect. 5, experimental results are presented and analyzed. Finally, conclusions are presented in Sect. 6.

2 Related work

Research about frequent pattern mining problem will be discussed first, followed by a discussion of the research on frequent pattern mining with weight constraints or weighted frequent pattern mining and their main challenges. After that, research on high utility pattern mining and their major problems will be discussed.

2.1 Frequent pattern mining

The support/frequency of a pattern is the number of transactions containing the pattern in the transaction database. The problem of frequent pattern mining is to find the complete set of patterns satisfying a minimum support in the transaction database. The *downward closure* property [3, 4] is used to prune the infrequent patterns. This property says that if a pattern is infrequent, then all of its super patterns must be infrequent. The *Apriori* [3, 4] algorithm is the initial solution of frequent pattern mining problem and very useful in association rule mining [3, 4, 18, 25, 28, 30, 32]. However, it suffers from the level-wise candidate generation-and-test problem and needs several database scans. The FP-growth [14] algorithm solved this problem by using the FP-tree-based solution without any candidate generation and using only two database scans. The FP-array [12] technique was proposed to reduce the FP-tree traversals and it works efficiently, especially in sparse datasets. One interesting measure h-confidence [15, 35] was proposed to identify the strong support affinity frequent patterns. CP-tree [26] has been proposed for efficient single-pass frequent pattern mining. Some other research [9, 13, 23, 34] has been done to determine frequent patterns. This traditional frequent pattern mining considers equal profit/weight for all items and only binary occurrences (0/1) of the items in one transaction.

2.2 Weighted frequent pattern mining

Research has been done on weighted frequent pattern mining [2, 27, 33, 39–41] in binary databases where frequency of an item in each transaction can be either 1 or 0. Weighted frequent itemset mining (WFIM) [41] and weighted interesting pattern mining (WIP) [39] showed that the main challenge in this area is that itemset weighted frequency does not have the *downward closure* property. The weight of a pattern P is the ratio of the sum of all its items' weight value to the length of P. Consider item "*a*" has a weight of 0.6 and a frequency of 4, item "*b*" has a weight of 0.2 and a frequency of 5, itemset "*ab*" has a frequency of 3. Then the weight of itemset "*ab*" will be $(0.6 + 0.2)/2 = 0.4$ and its weighted frequency will be $0.4 \times 3 = 1.2$. The weighted frequency of "*a*" is $0.6 \times 4 = 2.4$ and "*b*" is $0.2 \times 5 = 1.0$. If the minimum weighted frequency threshold is 1.2, then pattern "*b*" is weighted infrequent but pattern "*ab*" is weighted frequent. As a result, the *downward closure* property is not satisfied here. The WFIM and WIP algorithms maintain the *downward closure* property by multiplying each itemset's frequency by the maximum weight. In the above example, if "*a*" has the maximum weight of 0.6, then by multiplying it with the frequency of item "*b*", 3.0 can be obtained. So, pattern "*b*" is not pruned at the early stage and pattern "*ab*" will not be missed. However, pattern "*b*" is overestimated and will be pruned later by using its actual weighted frequency. The WCloset [40] algorithm calculates weighted closed frequent patterns. The DWFPM algorithm [2] is proposed to handle dynamic weights in weighted frequent pattern mining. However, maintaining the *downward closure* property in high utility pattern mining is more challenging as it considers non-binary frequency values of an item in transactions.

2.3 High utility pattern mining

The Itemset Share approach [5] considers non-binary frequency values of an item in each transaction. Share is the percentage of a numerical total that is contributed by the items in an itemset. These authors defined the problem of finding share frequent itemsets and compared the share and support measures to illustrate that share measure can provide useful information about numerical values that are associated with transaction items, which is not possible by using only the support measure. This method cannot rely on the *downward closure* property. Heuristic methods to find itemsets with share values above the minimum share threshold were developed. However, share-frequent pattern mining considers equal profit/weight for all items. Chan et al. [7] developed a method to discover top-K objective-directed high utility closed patterns. The definitions used by these researchers are different from those used in our work. They assume the same medical treatment for different patients (different transactions) will have different levels of effectiveness. They cannot maintain the *downward closure* property but they develop a pruning strategy to prune low utility itemsets based on a weaker *anti-monotonic* condition.

The theoretical model and definitions of high utility pattern mining were given in [37]. This approach, called MEU (mining with expected utility), cannot maintain the *downward closure* property. They used heuristics to determine whether an itemset should be considered as a candidate itemset. This approach usually overestimates, especially at the beginning stages, where the number of candidates approaches the number of all the combinations of items. This is impractical whenever the number of distinct items is large and the utility threshold is low. Later, the same authors proposed two new algorithms, UMining and UMining_H, to calculate high utility patterns [36]. In UMining, a pruning strategy based on the utility upper bound property is used. UMining_H was designed with another pruning strategy based on a heuristic method. However, some high utility itemsets may be erroneously pruned by this heuristic method. Furthermore, these methods do not satisfy the *downward closure* property and may overestimate too many patterns. They also suffer candidate generation-and-test methodology problem.

Based on the definitions of [37], the Two-Phase [19, 20] algorithm was developed to find high utility itemsets. The authors have defined the *transaction weighted utilization* (*twu*) and using it they proved that it is possible to maintain the *downward closure* property. For the first database scan, the algorithm finds all the one-element transaction weighted utilization itemsets and based on that result it generates the candidates for two-element transaction weighted utilization itemsets. In the second database scan, it finds all the two-element transaction weighted utilization itemset, and based on that result it generates the candidates for three-element transaction weighted utilization itemsets and so on. At the last scan, the Two-Phase algorithm determines the actual high utility itemsets from the high transaction weighted utilization itemsets. This algorithm suffers from the same problem of the level-wise candidate generation-and-test methodology. CTU-Mine [10] proposed an algorithm that is more efficient than the Two-Phase method only in dense databases when the minimum utility threshold is very low.

The isolated items discarding strategy (IIDS) [17] for discovering high utility itemsets was proposed to reduce the number of candidates in every database scan. IIDS shows that itemset share mining [5] problem can be directly converted to the utility mining problem by replacing the frequency value of each item in a transaction by its total profit, i.e. multiplying the frequency value by its unit profit. Applying IIDS, the authors developed efficient high utility itemset mining algorithms called FUM and DCG+ and showed that their technique is better than all previous high utility pattern mining techniques. Nevertheless, their algorithms still suffer from the candidate set generation-and-test problem of *Apriori*, and need multiple database scans. In this paper, we propose a novel tree-based candidate pruning technique to address the limitations of the existing algorithms.

3 Problem definition

We adopted similar definitions to those presented in the previous works [19, 20, 36, 37]. Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of items and D be a transaction database $\{T_1, T_2, \ldots, T_n\}$ where each transaction $T_i \in D$ is a subset of I. A pattern or itemset is defined by the set $X = \{x_1, x_2, \ldots, x_k\}$, where $X \subseteq I$ and $k \in [1, m]$. However, an itemset is called k-itemset when it contains k distinct items. For example, "*ab*" is a 2-itemset and "*abde*" is a 4-itemset in Fig. 1.

Definition 1 The internal utility or local transaction utility value $l(i_p, T_q)$, represents the quantity of item i_p in transaction T_q. For example, in Fig. 1(a), $l(\text{“}b\text{”}, T_2) = 4$.

Definition 2 The external utility $p(i_p)$ represents the unit profit value of item i_p. For example, in Fig. 1(b), $p(\text{“}c\text{”}) = 3$.

Definition 3 Utility $u(i_p, T_q)$, is the quantitative measure of utility for item i_p in transaction T_q, defined by

$$u(i_p, T_q) = l(i_p, T_q) \times p(i_p) \tag{1}$$

For example, $u(\text{“}b\text{”}, T_1) = 2 \times 6 = 12$ in Fig. 1.

Definition 4 The utility of an itemset X in transaction T_q, $u(X, T_q)$ is defined by,

$$u(X, T_q) = \sum_{i_p \in X} u(i_p, T_q) \tag{2}$$

where $X = \{i_1, i_2, \ldots, i_k\}$ is a k-itemset, $X \subseteq T_q$ and $1 \leq k \leq m$. For example, $u(\text{“}bc\text{”}, T_1) = 2 \times 6 + 8 \times 3 = 36$ in Fig. 1.

Definition 5 The utility of an itemset X is defined by,

$$u(X) = \sum_{T_q \in D} \sum_{i_p \in X \subseteq T_q} u(i_p, T_q) \tag{3}$$

For example, $u(\text{“}ab\text{”}) = u(\text{“}ab\text{”}, T_2) + u(\text{“}ab\text{”}, T_4) + u(\text{“}ab\text{”}, T_5) + u(\text{“}ab\text{”}, T_6) = 32 + 16 + 42 + 44 = 134$ in Fig. 1.

Definition 6 The transaction utility of transaction T_q denoted as $tu(T_q)$ describes the total profit of that transaction and it is defined by,

$$tu(T_q) = \sum_{i_p \in T_q} u(i_p, T_q) \tag{4}$$

For example, $tu(T_1) = u(\text{“}b\text{”}, T_1) + u(\text{“}c\text{”}, T_1) + u(\text{“}d\text{”}, T_1) = 12 + 24 + 16 = 52$ in Fig. 1.

However, in the high utility pattern mining problem, we need to find those patterns with a remarkable contribution to the total profit. Therefore, we have to quantify the remarkable amount by using a measure. For this purpose, a measure, called minimum utility threshold (δ), is described in the next definition. By using this measure, businessmen or other users can express their remarkable contribution to the total profit in a percentage form according to their requirements.

Definition 7 The minimum utility threshold δ, is given by the percentage of total transaction utility values of the database. In Fig. 1, the summation of all the transaction utility values is 427. If δ is 25% or we can also express it as 0.25, then the minimum utility value can be defined as

$$minutil = \delta \times \sum_{T_q \in D} tu(T_q) \tag{5}$$

Therefore, in this example $minutil = 0.25 \times 427 = 106.75$ in Fig. 1.

Definition 8 An itemset X is a high utility itemset, if $u(X) \geq minutil$. Finding high utility itemsets means determining all the itemsets X having criteria $u(X) \geq minutil$.

Fig. 1 Example of a transaction database and utility table

ITEM / TID	a	b	c	d	e	Trans. Utility($)
T_1	0	2	8	2	0	52
T_2	4	4	0	5	3	102
T_3	0	3	0	0	2	38
T_4	2	2	7	0	0	37
T_5	6	5	0	4	0	74
T_6	4	6	0	0	0	44
T_7	0	0	0	3	4	64
T_8	0	0	0	2	0	16

(a) Transaction database

ITEM	PROFIT($) (per unit)
a	2
b	6
c	3
d	8
e	10

(b) Utility table

For *minutil* $= 106.75$ pattern "*ab*" is a high utility pattern, as $u(\text{"}ab\text{"}) = 134$ (calculated in the example of Definition 5)

The greatest challenge in high utility pattern mining is that the itemset utility does not have the *downward closure* property. An item could not derive remarkable profits by itself, but if it is sold together with a highly profitable item, then such pair of items can be also profitable. For example, if *minutil* $= 106.75$ in Fig. 1, then "*e*" is a low utility item as $u(\text{"}e\text{"}) = 90$, but "*de*" is a high utility itemset as $u(\text{"}de\text{"}) = 134$ according to (3) and Definition 8. According to Definition 5 and (3), the definition of itemset utility is based on the actual sum of product calculation similar to real world business calculations. Therefore, maintaining the *downward closure* property is very challenging compared to weighted frequent itemset mining for two reasons. First, weighted frequent itemset mining considers the binary appearance of an item inside a transaction. Second, the weight of an itemset is calculated using the average of all the item weights inside it. In high utility pattern mining, we can maintain the *downward closure* property by transaction weighted utilization.

Definition 9 Transaction weighted utilization of an itemset X, denoted by $twu(X)$, is the sum of the transaction utilities of all transactions containing X.

$$twu(X) = \sum_{X \subseteq T_q \in D} tu(T_q) \tag{6}$$

For example, $twu(\text{"}bc\text{"}) = tu(T_1) + tu(T_4) = 52 + 37 = 89$ in Fig. 1. The *downward closure* property can be maintained using transaction weighted utilization. Here, for $minutil = 106.75$ in Fig. 1 as $twu(\text{"}bc\text{"}) < minutil$, any super pattern of "*bc*" cannot be a high *twu* itemset (candidate itemset) and obviously cannot be a high utility itemset.

Lemma 1 *Transaction weighted utilization value of an itemset X maintains the downward closure property.*

Proof Let X be a *transaction weighted utilization* itemset and DB_X is the set of transactions containing itemset X. Let Y be a superset of itemset X, then Y cannot be present in any transaction where X is absent. Therefore, according to Definition 9, the maximum *transaction weighted utilization* value of Y is $twu(X)$. So, if $twu(X)$ is less than *minutil*, Y cannot be a *transaction weighted utilization* itemset. □

Definition 10 X is a high transaction weighted utilization itemset (i.e. a candidate itemset) if $twu(X) \geq minutil$.

Lemma 2 *For a database DB and minimum utility threshold, the set of high utility itemsets*(S) *is a subset of the set of transaction weighted utilization itemset* (TS).

Proof Let X be a high utility itemset. According to Definitions 5 and 9, $utility(X)$ must be less than or equal to $twu(X)$. So, if X is a high utility itemset then it must be a *transaction weighted utilization* itemset. As a result, X is a member of the set TS and $S \subseteq TS$. □

In our technique, after finding all the high *twu* patterns maintaining the *downward closure* property, we calculate all the high utility patterns from high *twu* patterns by performing the original utility calculation according to (3) for all high *twu* patterns.

4 HUC-Prune: our proposed technique

In this section, we develop our proposed technique HUC-Prune (High Utility Candidates Prune) for high utility pattern mining. For a user-given minimum threshold, HUC-Prune detects the single-element candidate patterns (high *twu* items) in the first database scan. In the second database scan, it uses a novel tree structure, called HUC-tree (High Utility Candidates tree), to capture the *twu* information of items in transactions. After creating HUC-tree, it mines all the candidate patterns by using a pattern growth approach. Finally, a third database scan is used to discover all the high utility patterns from the candidate patterns.

4.1 Preliminaries

A prefix tree is an ordered tree with any predefined item order such as lexicographic order, frequency ascending order, frequency descending order, and so on. We can read transactions one at a time from a transaction database, arrange them in a predefined order, and insert each transaction into a path of the prefix tree. In this way prefix tree can represent a transaction database in a very compressed form when different transactions have many items in common. This type of path overlapping is called prefix sharing. The more the prefix sharing the more compression we can achieve from the prefix tree structure.

Definition 11 An HUC-tree (High Utility Candidates tree) is a prefix tree which stores the candidate items in the *twu* value descending order. Each node in an HUC-tree consists of five fields: *item-name*, *twu value*, *next node-link, parent-link*, a *list of links to its children*. The first field represents the name of the item whose information is represented by this node. The second field represents the *twu* value of the item in the sub-tree rooted at that node. If there is a next node containing the same item, the third field points to that next node, otherwise it stores a *NULL* value. A node may have zero, one or more than one child. In the fourth field, it stores a link to its parent node. In the fifth field it stores a *list*

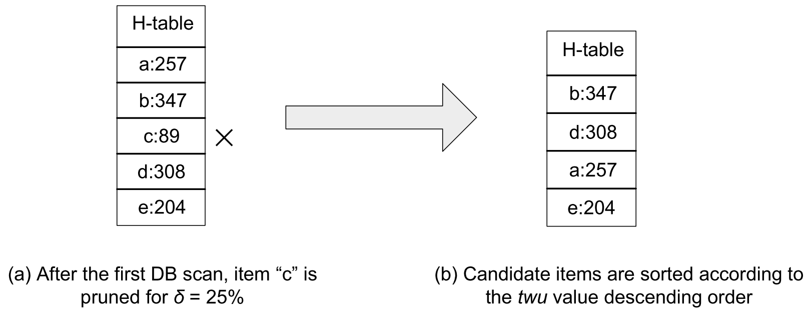

H-table
a:257
b:347
c:89
d:308
e:204
H-table
b:347
d:308
a:257
e:204
(a) After the first DB scan, item "c" is pruned for δ = 25%
(b) Candidate items are sorted according to the *twu* value descending order

Fig. 2 Obtaining candidate items sort order

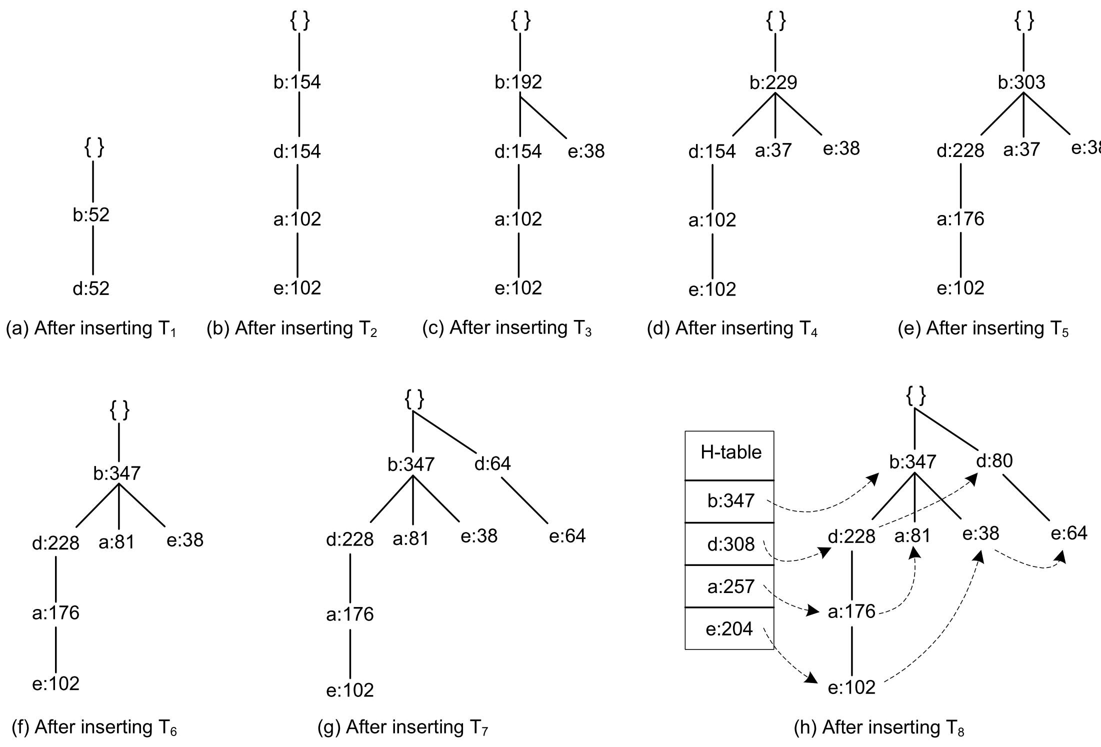


Fig. 3 Construction process of HUC-tree

of links to its children or a *NULL* value if it does not have any child. However, the root node keeps information in its list of links to the children and stores a *NULL* value for its other fields. An HUC-tree maintains a header table to keep the candidate items in the *twu* value descending order. Each entry in the header table consists of three fields: *item-name*, *twu* value (total *twu* value of that item in the *DB*), *next node-link* (points to the first node in the HUC-tree). Figure 3(h) shows the HUC-tree with header table constructed for the example *DB* ($\delta = 25\%$) presented in Fig. 1.

4.2 Construction process of HUC-tree

In the first database scan, HUC-Prune scans each transaction $Trans_i$ and calculates its transaction utility value according to (4). After that, it adds this value to the *twu* value of each item presented in $Trans_i$. For example, after scanning T_1 in Fig. 1, $tu(T_1) = 52$ is calculated and it is added to the *twu* value of items "*b*", "*c*" and "*d*". Consider the database shown in Fig. 1. After the first database scan, the *twu* values of the items are displayed in Fig. 2(a). If $minutil = 106.75$, then "*c*" is a low *twu* item. According to the *downward closure* property, any super pattern of "*c*" cannot be a high *twu* itemset. Therefore, neither "*c*" nor any of its super patterns can be a high utility itemset according to Lemma 2. Therefore, we prune the item "*c*" at this stage without further consideration. Next, we sort the header table in descending order according to the *twu* values of the items. Figure 2(b) shows the resultant sorted header table.

In the second database scan, we take only the high *twu* items from each transaction and sort them in the header table sort order presented in Fig. 2(b) and then insert into the tree. For the first transaction T_1, which contains items "*b*", "*c*", and "*d*", we first discard the low *twu* item "*c*" and then arrange it according to the header table order. Initially the root of HUC-tree contains a *NULL* value. One new node is created as a child of root and initialized with *item-name* "*b*" and *twu* value of 52. In a recursive way, this node now becomes the parent and another new node is created as its child to store the item "*d*". This type of operation where a new node must be created is defined as *insert node operation* in an HUC-tree. Figure 3(a) shows the resultant HUC-tree after inserting T_1.

Subsequently, transaction T_2 is inserted in HUC-tree as shown in Fig. 3(b). Before insertion, the items of T_2 are arranged ("*b*", "*d*", "*a*", "*e*") in the descending *twu* value order presented in the header table. Item "*b*" gets the prefix sharing with the existing node (child node of the root) containing item "*b*". The *twu* value of this node becomes $52 + 102 = 154$. This type of operation where an existing node can be found for the item to be inserted and the corresponding *twu* value must be only updated is defined as *update node operation* in an HUC-tree. Now the current node containing item "*b*" becomes the root for the insertion of the next item. Hence, we have to insert the next item "*d*" as its child. As it already has one child node containing item "*d*", we can perform an *update node operation* to insert item "*d*". After that, this "*d*" node becomes the new root with *twu* value of 154 (52 + 102) and we have to perform an *insert node operation* for item "*a*" (with a *twu* value of 102) as its child. Same operation continues for the last item "*e*" as shown in Fig. 3(b).

In summary, the strategy to visit the tree according to the current transaction is (1) sort the items in the *twu* value descending order, (2) discard the non-candidate items, (3) perform an *update node operation* if the current root node contains a child with the item to be inserted, otherwise perform an *insert node operation*, (4) perform the third step for the remaining items until the transaction becomes *NULL*.

By using the above strategy the remaining transactions (T_3 to T_8) are also inserted into the HUC-tree as shown in Figs. 3(c) to 3(h). Figure 3(h) shows the final tree with the header table for the full database presented in Fig. 1. Figure 3(h) also shows that the first node of a particular item in the HUC-tree is pointed by its entry in the header table, the second node is pointed by the first node, and so on. However, header table is not shown in every figure for simplicity. The following property is true for HUC-tree.

Property 1 *The twu value of any node in HUC-tree is greater than or equal to the sum of the twu values of its children.*

4.3 Mining process of HUC-Prune

In this section, we describe the mining process of HUC-Prune technique using a pattern growth approach. Prefix tree is first created for a particular item. To create the prefix tree, all the branches prefixing that item are taken along with the *twu* value of that item. After that, conditional tree for that item is created from the prefix tree by eliminating the non-candidate items with that particular item.

Consider the database of Fig. 1. If we take $\delta = 25\%$ in that database, then $minutil = 106.75$ according to (5). The final HUC-tree created for this database is shown in Fig. 3(h). We start from the bottom-most item "*e*" and create its prefix tree. The prefix tree for item "*e*" is shown in Fig. 4(a). By following the node-link of the header table, we can reach the first "*e*" node of the HUC-tree. This node contains a *twu* value of 102. Hence, a path {"*b*": 102, "*d*": 102, "*a*": 102} prefixing item "*e*" is taken from the global HUC-tree and inserted into the prefix tree of item "*e*" according to the strategy of HUC-tree construction process. After that, we can reach to the next node containing item "*e*" by following the node link of the current node. A path {"*b*": 38} is taken and inserted into the prefix tree of item "*e*". In a similar fashion, a path {"*d*": 64} is taken for the last node containing item "*e*" and inserted into the prefix tree of item "*e*".

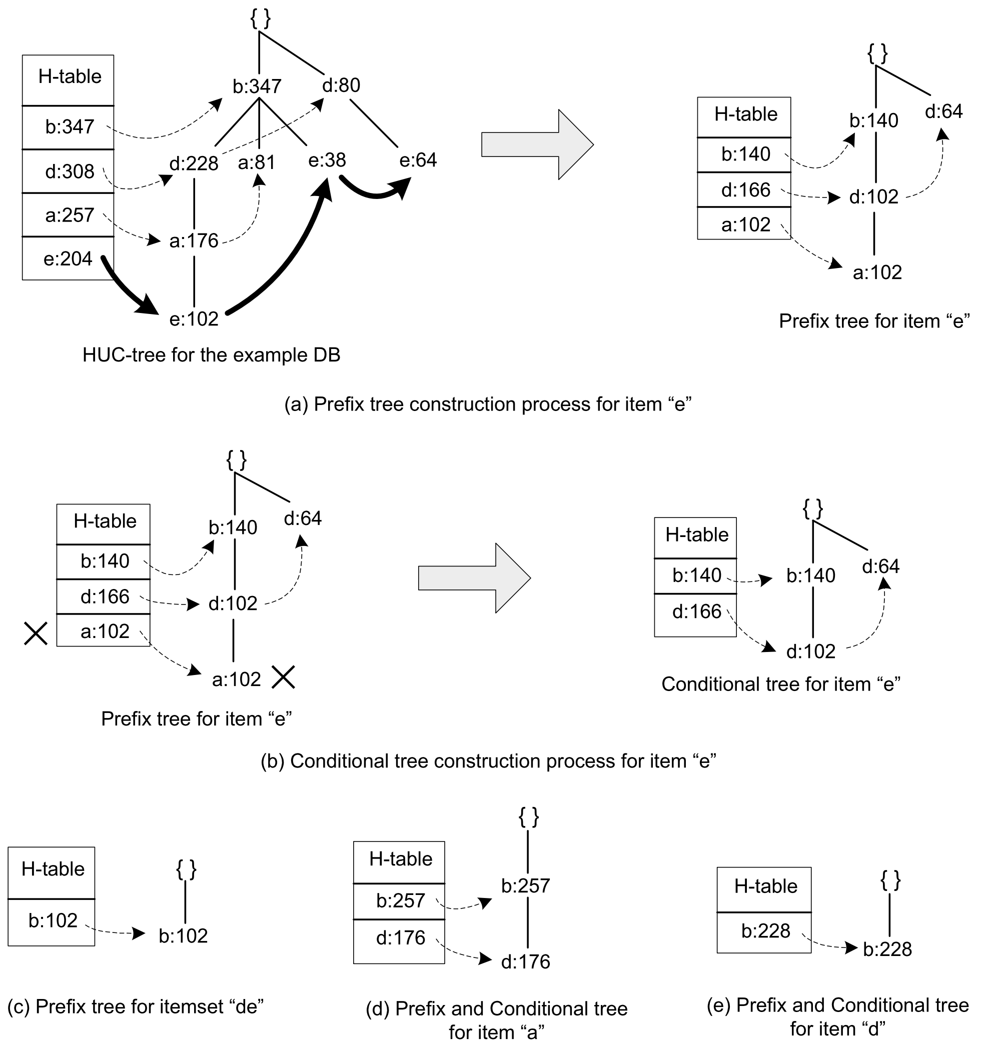
Subsequently, we create the conditional tree for item "*e*" from its prefix tree. The prefix tree for item "*e*" shows that item "*a*" cannot be a candidate itemset along with item "*e*". Therefore, the conditional tree of item "*e*" (shown in Fig. 4(b)) is derived by deleting all the nodes containing item "*a*" from the prefix tree of "*e*". Candidate patterns (1) {"*be*": 140}, (2) {"*de*": 166}, (3) {"*e*": 204} are generated here. The prefix tree for itemset "*de*" is created in Fig. 4(c) from the conditional tree in Fig. 4(b). It contains one item "*b*" and it has a low *twu* value with "*de*", so no conditional tree is created for itemset "*de*".

The prefix tree for item "*a*" is created in Fig. 4(d). All the items in the header table show that they have high *twu* value with item "*a*". Therefore, it is also the conditional tree for item "*a*" and candidate patterns (4) {"*ab*": 257}, (5) {"*ad*": 176}, (6) {"*abd*": 176}, (7) {"*a*": 257} are generated. Similarly, the prefix tree (also conditional-tree) for item "*d*" is created in Fig. 4(e) and candidate patterns (8) {"*bd*": 228}, (9) {"*d*": 308} are generated. The last candidate pattern (10) {"*b*": 347} is generated for the top-most item "*b*". A third database scan is required to find high utility itemsets from these ten candidate itemsets. The six high utility itemsets are {"*ab*": 134}, {"*abd*": 146}, {"*b*": 132}, {"*bd*": 154}, {"*d*": 128} and {"*de*": 134}.

4.4 Algorithm description

Figure 5 shows the tree construction and mining algorithm of HUC-Prune. The main algorithm is described in line 1 to line 31. It calls the tree construction procedure, called InsertHUC-tree (described in lines 32 to 47), to create the HUC-tree. Moreover, it calls the mining procedure, called

Fig. 4 Mining process of HUC-Prune


H-table
b:347
d:308
a:257
e:204
{ }
b:347
d:80
d:228
a:81
e:38
e:64
a:176
e:102
HUC-tree for the example DB
H-table
b:140
d:166
a:102
b:140
d:64
d:102
a:102
Prefix tree for item "e"
(a) Prefix tree construction process for item "e"
Prefix tree for item "e"
Conditional tree for item "e"
(b) Conditional tree construction process for item "e"
b:102
(c) Prefix tree for itemset "de"
b:257
d:176
(d) Prefix and Conditional tree for item "a"
b:228
(e) Prefix and Conditional tree for item "d"

Mining (described in lines 48 to 62), to mine the candidates for high utility patterns.

In line 2, a variable named *TU* is declared and initialized to zero in order to store the total transaction utility value of the *DB*. In line 3, a list named *L* is created for storing all the candidate patterns. A header table named *H* is created in line 4 to keep the candidate items in the *twu* value descending order. The first *DB* scan is demonstrated in the "*for loop*" described in lines 5 to 11. In line 6, transaction utility value of a transaction is calculated and added with the *twu* value of each item appeared in that transaction by using the nested "*for loop*" described in lines 7 to 9. This transaction utility value is also added with the *TU* variable in line 10. Subsequently, in line 12, the *minutil* value is calculated by multiplying the *TU* variable and minimum utility threshold δ (according to (5)). The "*for loop*" described in lines 13 to 17 eliminates all the non-candidate items from H. In line 18, H is sorted in the *twu* value descending order. An HUC-tree T is initialized with the root R, in line 19 and line 20. The second *DB* scan is performed in the "*for loop*" described in lines 21 to 25. Each iteration of the loop deletes the non-candidate items from a transaction, sorts the candidate items in the order of H and calls the InsertHUC-tree procedure to insert that transaction into the HUC-tree. The "*for loop*" described in lines 26 to 29 creates the prefix tree of an item and calls the Mining procedure to generate candidate patterns prefixing that item. As discussed in Sect. 4.3, the prefix tree of an item is created by traversing all the paths prefixing that item with the *node-link*, taking the paths with the *twu* value of that item and inserting into the prefix-tree. After generating all the candidate patterns prefixing all the candidate items, HUC-Prune performs the third *DB* scan to discover the high utility patterns from the candidate patterns (line 30).

The InsertHUC-tree procedure described in lines 32 to 47 recursively inserts the elements of a transaction in HUC-tree. It receives one transaction and the current root of a sub-tree where the front element of the transaction has to be inserted as a child. The "*if condition*" in lines 34 to 36 tests whether the transaction is empty or not. The procedure

Fig. 5 Tree construction and mining algorithm of HUC-Prune

```
   Input: $DB$, minimum utility threshold $\delta$
   Output: High Utility Patterns
 1 begin
 2     Let total transaction utility value $TU = 0$
 3     Let $L$ be the list of all candidate patterns
 4     Create a header table $H$ to keep the candidate items in the $twu$ value descending order
 5     foreach transaction $Trans_i$ in $DB$ do
 6         Calculate the $tu$ value $t$ of $Trans_i$
 7         foreach item $i$ in $Trans_i$ do
 8             Add $t$ with the $twu$ of $i$ in $H$
 9         end
10         $TU = TU + t$
11     end
12     $minutil = TU \times \delta$
13     foreach item $i$ in the Header table $H$ do
14         if $twu(i) < minutil$ then
15             Delete $i$ from $H$
16         end
17     end
18     Sort $H$ according to the $twu$ values of items in descending order
19     Let $T$ be an $HUC$-$tree$
20     Create the root $R$ of $T$, and label it as $NULL$
21     foreach transaction $Trans_i$ in $DB$ do
22         Delete the items of $Trans_i$ not present in $H$
23         Sort the remaining items of $Trans_i$ in the order of $H$
24         Call InsertHUC-tree ($Trans_i$, $R$)
25     end
26     foreach item $\alpha$ from the bottom of $H$ do
27         Create Prefix-tree $PT_\alpha$ with its header table $HT_\alpha$ for item $\alpha$
28         Call Mining ($PT_\alpha$, $HT_\alpha$, $\alpha$)
29     end
30     Scan $DB$ to find out actual high utility patterns from list $L$
31 end
32 Procedure InsertHUC-tree($Trans_i$, $R$)
33 begin
34     if $Trans_i$ is $NULL$ then
35         return
36     end
37     Let divide $Trans_i$ as $[x|X]$, where $x$ is the first element and $X$ is the remaining list
38     if $R$ has a child $C$ such that $C.item\text{-}name = x.item\text{-}name$ then
39         $C.twu = C.twu + x.twu$
40     end
41     else
42         create a new node $C$ as child of $R$
43         $C.item\text{-}name = x.item\text{-}name$
44         $C.twu = x.twu$
45     end
46     Call InsertHUC-tree($X$,$C$)
47 end
48 Procedure Mining($T, H, \alpha$)
49 begin
50     foreach item $\beta$ of $H$ do
51         if $twu(\beta) < minutil$ then
52             Delete $\beta$ from $T$ and $H$ to create conditional tree and header table
53         end
54     end
55     Let $CT$ be the Conditional tree of $\alpha$ created from $T$
56     Let $HC$ be the Header table of Conditional tree $CT$ created from $H$
57     foreach item $\beta$ in $HC$ do
58         Add pattern $\alpha\beta$ in the candidate pattern list $L$
59         Create Prefix-tree $PT_{\alpha\beta}$ and Header table $HT_{\alpha\beta}$ for pattern $\alpha\beta$
60         Call Mining ($PT_{\alpha\beta}$ , $HT_{\alpha\beta}$ , $\alpha\beta$)
61     end
62 end
```

returns when the received transaction is empty. In line 37, the front element, x, of the received transaction is taken to insert as a child of the current root R and remaining elements are taken form the transaction for the next recursion. If the *item-name* of x is matched with the *item-name* of any child node of R, then we can perform the *update node oper-*

ation (described in Sect. 4.2). Here, we only need to update the *twu* value of the existing node. The *twu* value of the existing node is incremented by the *twu* value of the item to be inserted. However, it creates a new child node of the current root if it fails to find any match (*insert node operation*) in line 42. The new node is initialized with the *item-name* and *twu* value of x (line 43 and line 44). In line 46, the procedure recursively calls itself with the remaining items of the transaction and current child node as the root.

The Mining procedure is described in lines 48 to 62. It receives a pattern "α", prefix tree T of "α" and header table H of prefix tree T. It recursively mines all the candidate patterns prefixing "α". The "*for loop*" described in lines 50 to 54 creates the conditional tree of item "α". As discussed in Sect. 4.3, a conditional tree for an item is created from its prefix tree by deleting the nodes containing non-candidate items. Accordingly, this "*for loop*" deletes the items having *twu* value less than the *minutil* value from the prefix tree and header table of the prefix tree in order to obtain the conditional tree (*CT*) and header table (*HC*) of the conditional tree. Subsequently, for each item "β" in *HC*, the "*for loop*" described in lines 57 to 61 creates a new candidate pattern "$\alpha\beta$" by joining item "β" in *HC* with pattern "α" and inserts it into the candidate pattern list L (line 58). Finally, this "*for loop*" creates the prefix tree $PT_{\alpha\beta}$ of pattern "$\alpha\beta$", header table $HT_{\alpha\beta}$ for $PT_{\alpha\beta}$ (line 59) and makes a recursive call to the Mining procedure (line 60).

4.5 Time complexity analysis

The existing algorithms are based on the level-wise candidate generation-and-test methodology of the *Apriori* algorithm. They first scan the database to find the single high *twu* items. They then generate all the candidates for 2-itemsets. The example shown in the mining section has four single high *twu* items ("*b*", "*d*", "*a*", and "*e*"). For 2-itemsets, they generate all combinations, taking two at a time $\binom{4}{2}$ without knowledge of whether those patterns appear in the database or not. If the number of single high *twu* items is 1000, then these algorithms will generate $\binom{1000}{2}$ candidate 2-itemsets. After scanning the database again, they can find the actual high *twu* 2-itemsets. For 3-itemsets, a pattern is considered as a candidate pattern if all of its sub-patterns are candidate patterns. For example, pattern "*abc*" is considered as a candidate pattern if patterns "*ab*", "*bc*" and "*ca*" are candidate patterns. In this case, pattern "*abc*" may not appear in the dataset or it could have very small utility value. Higher order candidates are generated similarly. Therefore, the database must be scanned several times with a huge number of candidates.

Lemma 3 *If N_1 is the number of candidate itemsets generated by HUC-Prune technique and N_2 is the number of candidate itemsets generated by Apriori-based high utility pattern mining algorithms, then $N_1 \leq N_2$.*

Proof A pattern $X\{x_1, x_2, \ldots, x_n\}$ is a high *twu* itemset (candidate itemset) iff all of its subsets of length $n-1$ are high *twu* itemsets in *Apriori*-based high utility pattern mining algorithms. So, X may not be present in the database or it could have too low utility value to become a candidate. In HUC-Prune technique, if X is not present in the database then it cannot appear in any branch of the tree and therefore it cannot appear as a candidate. Moreover, after determining X is a low *twu* itemset, it is pruned. Therefore, the candidate set of HUC-Prune contains only the true high *twu* itemsets; hence, N_1 cannot be greater than N_2. □

Observation 1 When the maximum length of the candidate patterns increases, *Apriori* based high utility mining algorithms have to scan database repeatedly. In the example shown in the mining section, the maximum length of a candidate pattern is $3(abd)$. Existing recent FUM and DCG+ algorithms scan the database at least three times to find all of the candidate and high utility itemsets. For the Two-Phase algorithm, one extra scan is required at the last for finding out the high utility itemsets from the candidate itemsets. Therefore, a total of four database scans are required for this example. If, the maximum length of candidate patterns is N, a total of N database scans could be required for the existing recent FUM and DCG+ algorithms and a total of $N+1$ database scans are required for the Two-Phase algorithm. On the other hand, number of database scans required for our technique is totally independent of the maximum length of candidate patterns. Always maximum three database scans are required. As the minimum utility threshold decreases, the number of candidate patterns and their maximum length also increases. Therefore, as the minimum utility threshold decreases, running time increases very sharply in *Apriori* based algorithms.

5 Experimental results

To evaluate the performance of our proposed technique, we performed several experiments on synthetic datasets (*T10I4D100K, 20K and 60K*) and real life datasets (*chess*, *mushroom*, *retail*, *connect* and *kosarak*) from frequent itemset mining dataset repository [11] and the UCI Machine Learning Repository [29]. These datasets do not provide profit values or the quantity of each item for each transaction. As for the performance evaluation of the previous utility based pattern mining [17, 19, 20], we generated random numbers for the profit values of each item and quantity of each item in each transaction, ranging from 1.0 to 10.0 and 1 to 10, respectively. Based on our observation in real world databases that most items carry low profit, we generated the profit values using a log-normal distribution. Some other high utility pattern mining research [17, 19, 20] has adopted the same technique. Figure 6 shows the external utility distribution of 2000 distinct items using a log-normal distribution. Finally, we present our result by using a real-life dataset (*Chain-store*) with real utility values [22]. The performance of our technique was compared with the existing algorithms Two-Phase [19, 20], FUM and DCG+ [17]. Our programs were written in Microsoft Visual C++ 6.0 and run with the Windows XP operating system on a Pentium dual core 2.13 GHz CPU with 2 GB main memory.

5.1 Dataset characteristics

Table 1 shows different important characteristics of the synthetic and real life datasets. The dense or sparse nature of a dataset is a very useful property [21, 24, 31, 38]. Many pattern mining algorithms [12–14, 21, 26, 31, 38] have evaluated their performances by using this characteristic of datasets. A dataset becomes dense when it has many items per transaction and the number of distinct items is small. Consider the chess dataset in Table 1. It has 75 distinct items and its average transaction length is 37. Therefore, 49.33% of items are present in every transaction. If we take the measure $R = A/D$ (in Table 1), we know the probability that an item appears in a transaction. We can tell that a dataset is dense if $R > 10\%$ and sparse if $R \leq 10\%$.

5.2 Performance analysis in dense datasets

Dense datasets have many long frequent as well as high utility patterns. Because the probability of an item's occurrence is very high in every transaction, for comparatively higher thresholds, dense datasets have too many candidate patterns. It is stated in Observation 1 in Sect. 4.5 that long patterns need several database scans. When the dataset becomes more dense, the number of candidates and total running time sharply increase in *Apriori*-based existing algorithms.

The *mushroom* dataset is a moderately dense dataset with $R = 19.327\%$. For this dataset, we first compare the number of candidates. Figure 7 shows the number of candidates for comparison. The minimum utility threshold range of 10% to 30% is used here. The number of candidates increases rapidly below the utility threshold of 20%. For utility thresholds of 10% and 15%, the numbers of candidate patterns are

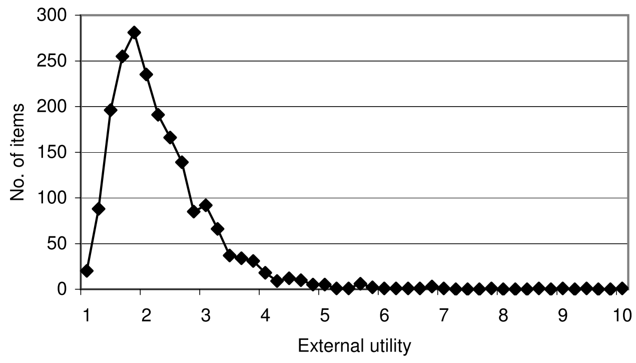


Fig. 6 External utility distribution for 2000 distinct items using log-normal distribution

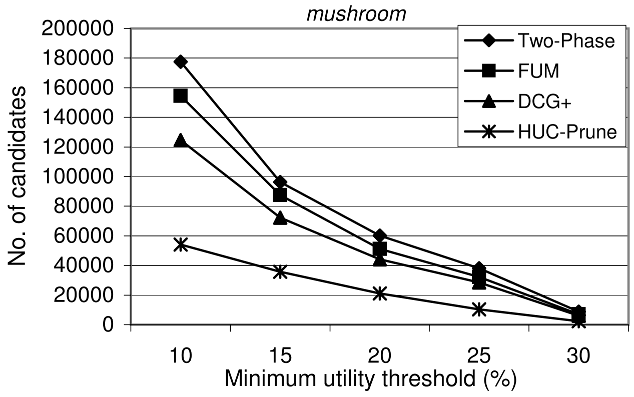


Fig. 7 Number of candidates comparison on the *mushroom* dataset

Table 1 Dataset characteristics

Dataset	No. of Trans.	No. of distinct items (D)	Avg. Trans. length (A)	Dense/Sparse characteristic ratio R (%) $R = (A/D) \times 100$
mushroom	8124	119	23	19.327
chess	3196	75	37	49.33
connect	67557	129	43	33.33
T10I4D100K	100000	870	10.1	1.16
20K	20000	7902	7	0.0885
60K	60000	5949	7	0.1176
retail	88162	16470	10.3	0.0625
kosarak	990002	41270	8.1	0.0196
Chain-store	1112949	46086	7.2	0.0156

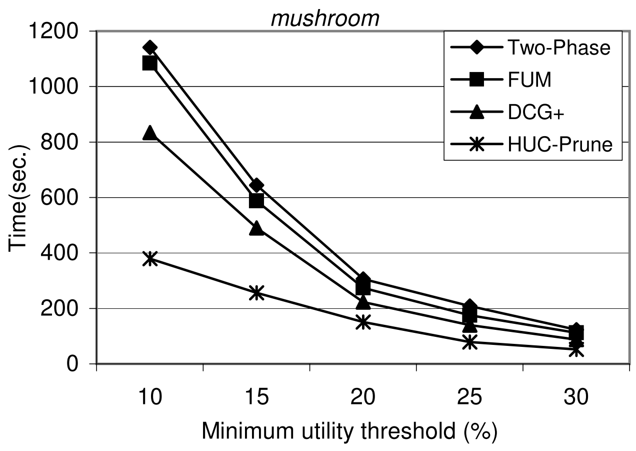


Fig. 8 Execution time comparison on the *mushroom* dataset

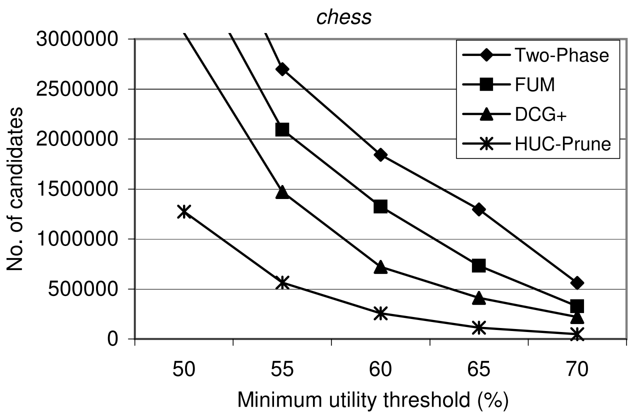


Fig. 9 Number of candidates comparison on the *chess* dataset

too large for the existing algorithms. According to Lemma 3, our candidates are only the actual high *twu* patterns, but candidates of the existing algorithms are not only the actual high *twu* patterns but also many overestimated *twu* patterns that seem to be high *twu* patterns because their subsets are high *twu* patterns. That is why there are a large number of candidates that increase sharply as the minimum utility threshold decreases. Figure 8 shows the running time comparison in the *mushroom* dataset. Since the lower threshold has too many long candidate patterns and several database scans are needed for the very large number of long candidate patterns, the time difference between existing algorithms and our technique becomes larger as δ decreases. It is obvious that existing algorithms are inefficient for moderately dense datasets when the utility threshold is low.

The *chess* dataset is an extremely dense dataset with $R = 49.33\%$. We first compare the number of candidates. Figure 9 shows the number of candidates comparison on the *chess* dataset. The minimum utility threshold range of 50% to 70% is used here. It is remarkable that since *chess* is a huge dense dataset, a large number of candidate patterns occur at a comparatively higher threshold (above 50%). The number of candidates increases rapidly below the utility threshold of 60%. For utility thresholds of 50% and 55%, the numbers of candidate patterns are too large for the existing algorithms as shown in Fig. 9. Figure 10 shows the running time comparison on the *chess* dataset. Due to several database scans with a large candidate set, the total time needed for the existing algorithms is also very large.

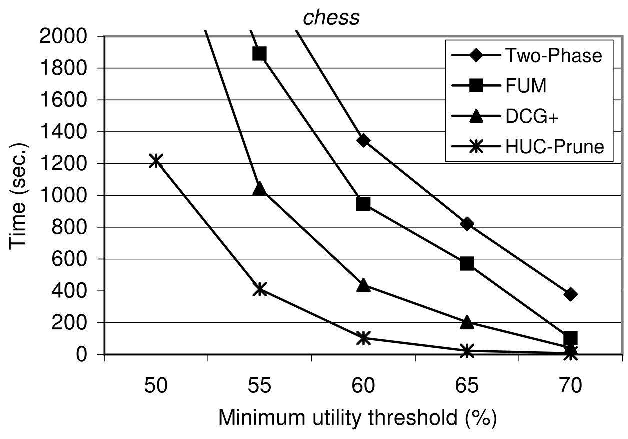


Fig. 10 Execution time comparison on the *chess* dataset

The *connect* dataset is an extremely dense dataset with $R = 33.33\%$. However, it is much larger than the *chess* and *mushroom* dataset with respect to number of transactions, number of distinct items and average transaction length (Table 1). Figure 11 shows that existing algorithms generate too many candidate patterns for this dataset. The minimum utility threshold range of 80% to 95% is used here. In contrast to the *mushroom* and *chess* datasets, the number of candidates increases rapidly below the utility threshold of 90%. For utility thresholds of 80% and 85%, the numbers of candidate patterns are too large for the existing algorithms as shown in Fig. 11. Figure 12 shows the execution time comparison on this dataset. The existing algorithms take too much time for scanning this large dense dataset with a huge number of candidates. Therefore, existing algorithms are very inefficient for an extremely dense dataset. On the other hand, our technique can efficiently handle this huge dense dataset by avoiding the level-wise candidate generation-and-test methodology.

5.3 Performance analysis in sparse datasets

Sparse datasets normally have too many distinct items. Although in the average case their transaction length is small, they normally have many transactions. According to our time complexity analysis in Sect. 4.5, handling many distinct items is a severe problem in *Apriori*-based algorithms. We will show here that handling a large number of distinct items and several database scans over long sparse datasets also make existing algorithms inefficient for sparse datasets.

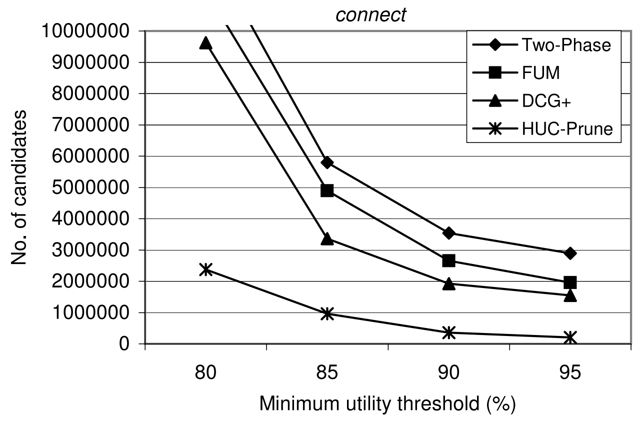


Fig. 11 Number of candidates comparison on the *connect* dataset

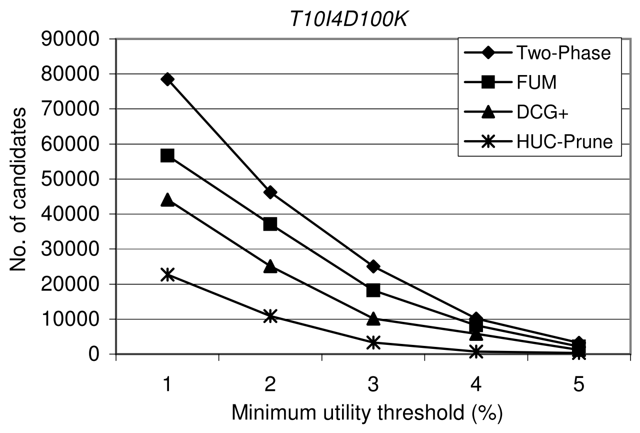


Fig. 13 Number of candidates comparison on the *T10I4D100K* dataset

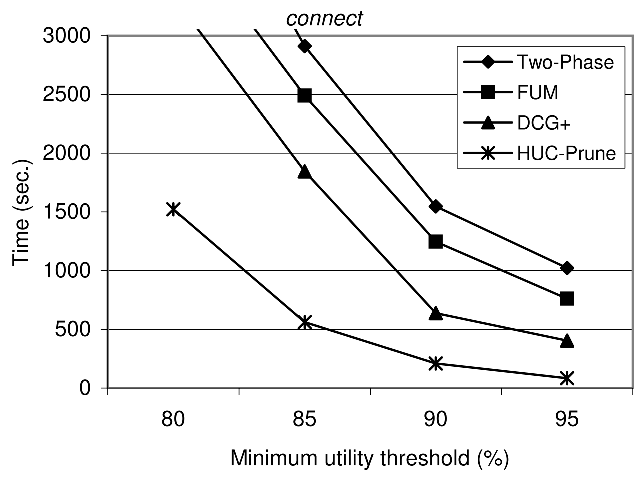


Fig. 12 Execution time comparison on the *connect* dataset

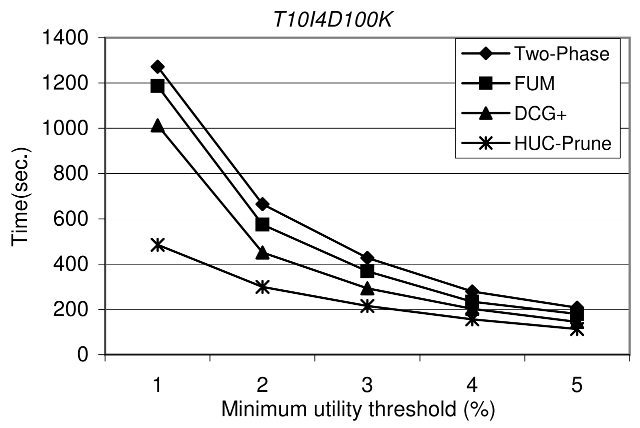


Fig. 14 Execution time comparison on the *T10I4D100K* dataset

The IBM synthetic dataset *T10I4D100K* was developed by the IBM Almaden Quest research group [4, 16] and obtained from the frequent itemset mining dataset repository [11]. Here, T represents average size of the transactions, I represents average size of the maximal potentially large itemsets and D represents the number of transactions. *T10I4D100K* is a moderately sparse dataset with $R = 1.16\%$. Its other characteristics (number of distinct items, total number of transactions etc.) are also in a moderate range. Although existing algorithms can handle these types of datasets, the performance of our technique is better than their performance as shown in Figs. 13 and 14. The minimum utility threshold range of 1% to 5% is used here. As expected, the differences in candidate patterns and running time of the existing algorithms and our technique become larger when the minimum utility threshold becomes low.

Recently, Cooper and Zito [8] have proposed an alternative model for generating synthetic data. Their dataset generator programs have been downloaded from http://www.csc.liv.ac.uk/~michele/soft.html. These programs (written in Java) can be used to generate a synthetic dataset with h transactions (h is the number of transactions). The transaction sizes are given by the absolute value of a normal distribution with mean μ and deviation σ. There are two types of transactions, old and new. A new transaction (chosen with probability α) consists of a mix of new items and items occurring in previous transactions. An old transaction consists of only items occurring in previous transactions. Moreover, If transactions of type old (new) are chosen in a step we assume that each of them is selected using preferential attachment with probability P_O (P_N) and randomly otherwise [8]. Number of distinct items is represented by n in a dataset. We have generated two datasets *20K* ($h = 20{,}000$, $\alpha = 40\%$, $n = 7{,}902$) and *60K* ($h = 60{,}000$, $\alpha = 10\%$, $n = 5{,}949$) by using the generator programs with $\mu = 7$, $\sigma = 3$ and $P = 50\%$ ($P = P_O = P_N$). The datasets, *20K* ($R = 0.08858\%$) and *60K* ($R = 0.1176\%$), are also sparse datasets. The performance of our technique in these datasets compared with the existing algorithms is shown in Figs. 15 to 18. The minimum utility threshold range of 1% to 5% is

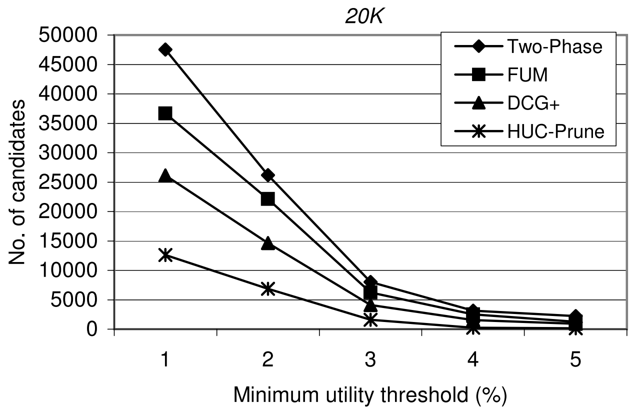


Fig. 15 Number of candidates comparison on the *20K* dataset

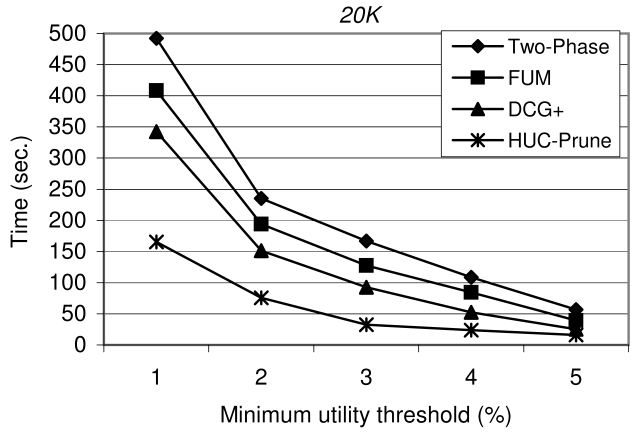


Fig. 16 Execution time comparison on the *20K* dataset

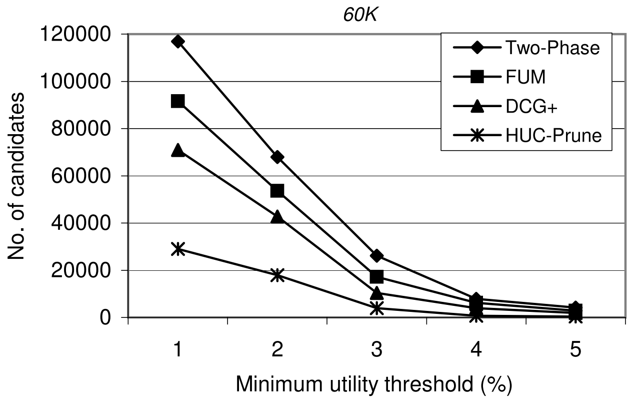


Fig. 17 Number of candidates comparison on the *60K* dataset

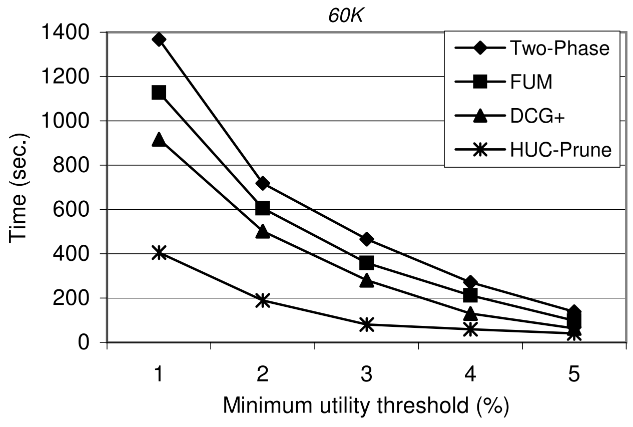


Fig. 18 Execution time comparison on the *60K* dataset

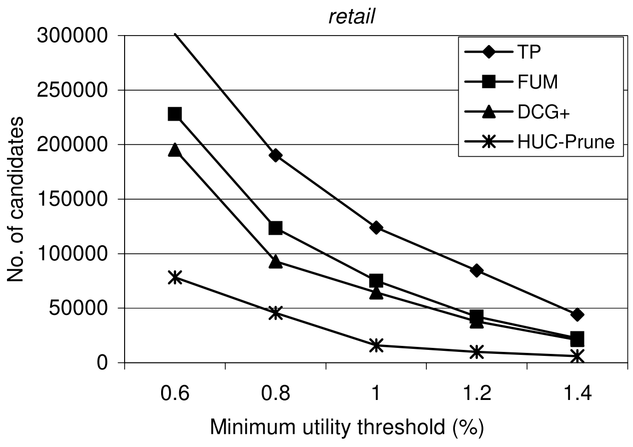


Fig. 19 Number of candidates comparison on the *retail* dataset

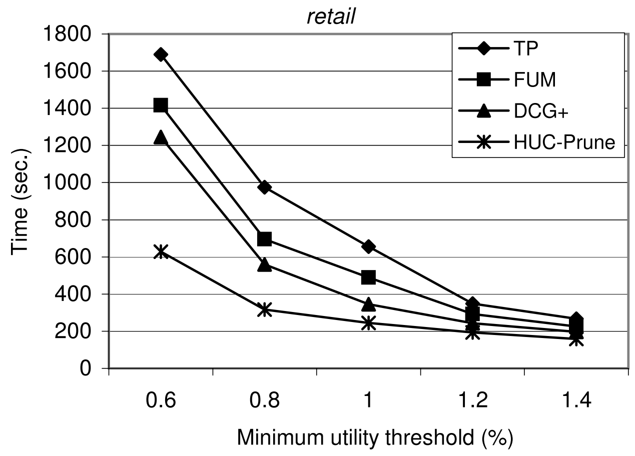


Fig. 20 Execution time comparison on the *retail* dataset

used here. These experimental results show that our technique outperforms the existing algorithms in these datasets. Moreover, it also shows the effect of many distinct items.

The dataset *retail* is provided by Tom Brijs, and contains the retail market basket data from an anonymous Belgian retail store [6, 11]. It is an extreme sparse dataset ($R =$ 0.0625%) with too many distinct items (16,470). Figure 19 shows the comparison of candidate number and Fig. 20 shows the runtime comparison on the *retail* dataset. The minimum utility threshold range of 0.6% to 1.4% is used

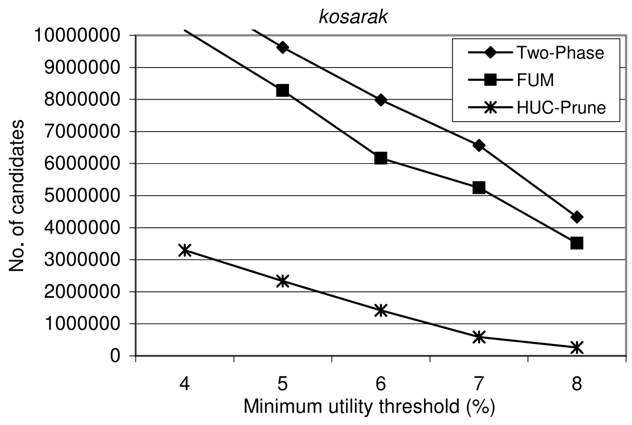


Fig. 21 Number of candidates comparison on the *kosarak* dataset

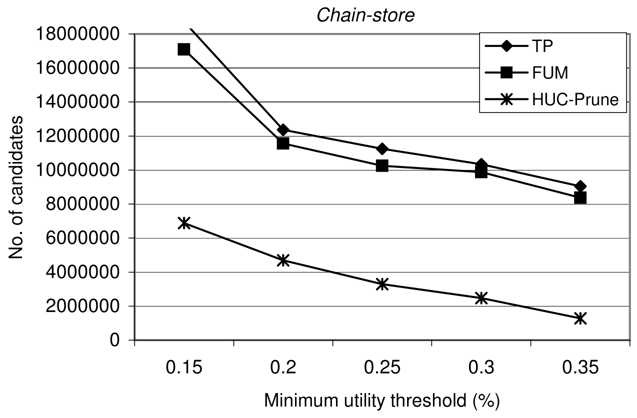


Fig. 23 Number of candidates comparison on the *Chain-store* dataset

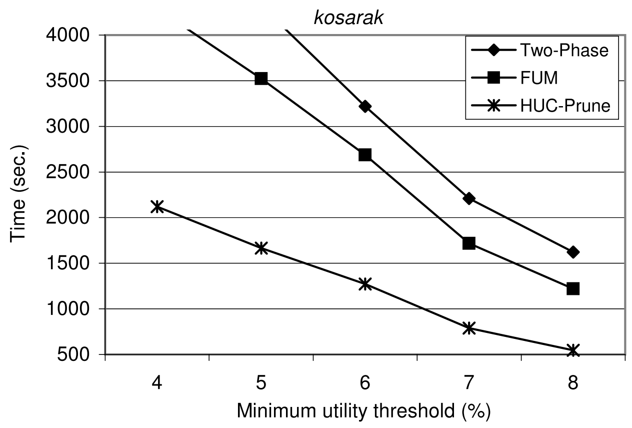


Fig. 22 Execution time comparison on the *kosarak* dataset

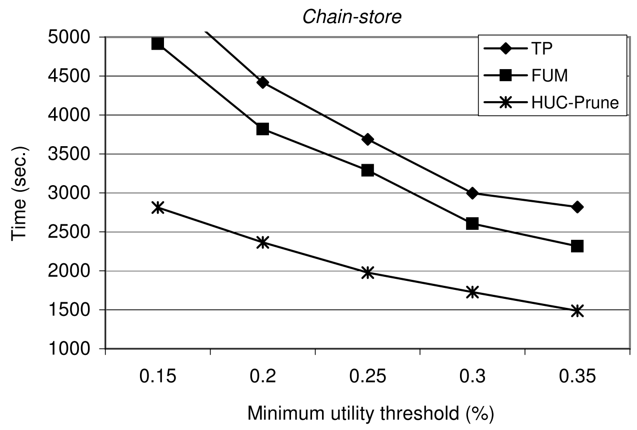


Fig. 24 Execution time comparison on the *Chain-store* dataset

here. As stated in Observation 1 in Sect. 4.5, Fig. 19 and Fig. 20 show the effect of too many distinct items in *retail* dataset over the existing algorithms.

The dataset *kosarak* was provided by Ferenc Bodon and contains click-stream data of a Hungarian on-line news portal [11]. It is a huge sparse dataset with $R = 0.0196\%$. It has a very large number (41,270) of distinct items and around 1 million transactions. According to our time complexity analysis in Sect. 4.5, too many candidate patterns are generated for this large number of distinct items as shown in Fig. 21. The minimum utility threshold range of 4% to 8% is used here. Obviously, too much time is needed to handle these candidates and scan this long dataset with them. A time comparison is shown in Fig. 22. We have compared the performance of our technique with the existing Two-Phase and FUM algorithms. As DCG+ maintains an extra array for each candidate [17], we could not keep its all candidates in each pass in the main memory. These experimental results demonstrate that the existing algorithms are very inefficient for sparse datasets with many distinct items and number of transactions.

5.4 Performance analysis in real life dataset with real utility values

In this section, we use a real-life dataset adopted from NU-MineBench 2.0, a powerful benchmark suite consisting of multiple data mining applications and datasets [22]. This dataset, called *Chain-store*, was taken from a major chain in California and contains 1,112,949 transactions and 46,086 distinct items [17, 22]. The dataset's utility table stores the profit for each item. The total profit of the dataset is $26,388,499.80.

The *Chain-store* is a large sparse dataset with $R = 0.0156\%$. Figure 23 and Fig. 24 show the comparison of candidate number and the runtime respectively on this dataset. The minimum utility threshold range of 0.15% to 0.35% is used here. We have compared the performance of our technique with the existing Two-Phase and FUM algorithms. As DCG+ maintains an extra array for each candidate [17], we could not keep all its candidates in each pass in the main memory. These figures demonstrate that our technique outperforms the existing algorithms by means of using an efficient candidate pruning method. Moreover,

Table 2 Memory comparison (MB)

Dataset	Min. utility threshold (%)	HUC-Prune	FUM	Two-Phase
mushroom	20	0.283	2.194	2.739
chess	60	0.381	2.872	3.91
connect	85	7.218	45.481	49.71
T10I4D100K	3	5.27	28.69	30.43
20K	2	0.835	6.739	7.682
60K	3	2.74	23.094	25.613
retail	1	6.52	34.892	37.28
kosarak	6	105.372	442.51	451.725
Chain-store	0.25	154.318	513.94	524.126

our technique has easily handled the 46,086 distinct items and more than 1 million transactions in this real-life dataset. Therefore, these experimental results also demonstrate the scalability of our technique to handle large number of distinct items and transactions.

5.5 Memory usage

Prefix-tree-based frequent pattern mining techniques [2, 12–14, 26, 31, 39–41] have shown that the memory requirement for the prefix trees is low enough to use the gigabyte-range memory now available. We have also handled our tree structure very efficiently and kept it within this memory range. Our prefix-tree structure can represent the useful information in a very compressed form because transactions have many items in common. By utilizing this type of path overlapping (prefix sharing), our tree structure can save memory space. Moreover, by using a pattern growth approach we generate much less number of candidates compared to the existing algorithms. Table 2 shows that our technique outperforms the existing algorithms in memory usage.

6 Conclusions

This paper provides an efficient method for high utility pattern mining in the area of data mining and knowledge discovery. A new candidate pruning technique, HUC-Prune, and a novel tree structure, HUC-tree, are proposed. The number of database scans with HUC-Prune is totally independent of the high utility candidate pattern length. It requires a maximum of three database scans to calculate the resultant set of high utility patterns for a user-given minimum utility threshold. Moreover, it prunes a large number of unnecessary candidate patterns during tree creation time by eliminating the non-candidate single-element patterns and also during mining time by using a pattern growth approach. These are the main reasons for its success over the existing algorithms where too many candidate patterns are generated and several database scans are required. HUC-Prune also outperforms the existing algorithms in memory requirements by using an efficient tree-based approach. Extensive performance analyses show that our technique is very efficient and it outperforms the existing algorithms for both dense and sparse datasets.

Acknowledgements We would like to express our deep gratitude to the anonymous reviewers of this paper. Their useful comments have played a significant role in improving the quality of this work.

References

1. Adnan M, Alhajj R (2009) DRFP-tree: disk resident frequent pattern tree. Appl Intell 30:84–97
2. Ahmed CF, Tanbeer SK, Jeong B-S, Lee Y-K (2008) Handling dynamic weights in weighted frequent pattern mining. IEICE Trans Inf Syst E91-D(11):2578–2588
3. Agrawal R, Imielinski T, Swami A (1993) Mining association rules between sets of items in large databases. In: Proceedings of the 12th ACM SIGMOD international conference on management of data, 1993, pp 207–216
4. Agrawal R, Srikant R (1994) Fast algorithms for mining association rules. In: Proceedings of the 20th international conference on very large data bases (VLDB), 1994, pp 487–499
5. Barber B, Hamilton HJ (2003) Extracting share frequent itemsets with infrequent subsets. Data Min Knowl Discov 7:153–185
6. Brijs T, Swinnen G, Vanhoof K, Wets G (1999) Using association rules for product assortment decisions: a case study. In: Proceedings of the 5th ACM SIGKDD international conference on knowledge discovery and data mining, 1999, pp 254–260
7. Chan R, Yang Q, Shen YD (2003) Mining high utility itemsets. In: Proceedings of the 3rd IEEE international conference on data mining, 2003, pp 19–26
8. Cooper C, Zito M (2007) Realistic synthetic data for testing association rule mining algorithms for market basket databases. In: Proceedings of the 11th international conference on principles and practice of knowledge discovery in databases (PKDD), 2007, pp 398–405
9. Dong J, Han M (2007) BitTableFI: An efficient mining frequent itemsets algorithm. Knowl-Based Syst 20:329–335

10. Erwin A, Gopalan RP, Achuthan NR (2007) CTU-Mine: an efficient high utility itemset mining algorithm using the pattern growth approach. In: Proceedings of the 7th IEEE international conference on computer and information technology (CIT), 2007, pp 71–76
11. Frequent itemset mining dataset repository. Available from: http://fimi.cs.helsinki.fi/data/
12. Grahne G, Zhu J (2005) Fast algorithms for frequent itemset mining using FP-Trees. IEEE Trans Knowl Data Eng 17(10):1347–1362
13. Han J, Cheng H, Xin D, Yan X (2007) Frequent pattern mining: current status and future directions. Data Min Knowl Discov 15:55–86
14. Han J, Pei J, Yin Y, Mao R (2004) Mining frequent patterns without candidate generation: a frequent-pattern tree approach. Data Min Knowl Discov 8:53–87
15. Huang Y, Xiong H, Wu W, Deng P, Zhang Z (2007) Mining maximal hyperclique pattern: a hybrid search strategy. Inf Sci 177:703–721
16. IBM (2009) QUEST Data Mining Project. Available from: http://www.almaden.ibm.com/cs/disciplines/iis/
17. Li Y-C, Yeh J-S, Chang C-C (2008) Isolated items discarding strategy for discovering high utility itemsets. Data Knowl Eng 64:198–217
18. Liu B, Ma Y, Wong CK (2003) Scoring the data using association rules. Appl Intell 18:119–135
19. Liu Y, Liao W-K, Choudhary A (2005) A fast high utility itemsets mining algorithm. In: Proceedings of the 1st international conference on utility-based data mining, 2005, pp 90–99
20. Liu Y, Liao W-K, Choudhary A (2005) A two phase algorithm for fast discovery of high utility of itemsets. In: Proceedings of the 9th Pacific-Asia conference on knowledge discovery and data mining (PAKDD), 2005, pp 689–695
21. Pei J, Han J (2000) CLOSET: An efficient algorithm for mining frequent closed itemsets. In: Proceedings of ACM SIGMOD workshop on research issues in data mining and knowledge discovery, 2000, pp 21–30
22. Pisharath J, Liu Y, Parhi J, Liao W-K, Choudhary A, Memik G (2006) NU-MineBench version 2.0 source code and datasets. Available from: http://cucis.ece.northwestern.edu/projects/DMS/MineBench.html
23. Song M, Rajasekaran S (2006) A transaction mapping algorithm for frequent itemsets mining. IEEE Trans Knowl Data Eng 18(4):472–481
24. Sucahyo YG, Gopalan RP, Rudra A (2003) Efficient mining frequent patterns from dense datasets using a cluster of computers. In: AI 2003. LNAI, vol 2903. Springer, Berlin, pp 233–244
25. Tan PN, Kumar V, Srivastava J (2002) Selecting the right interestingness measure for association patterns. In: Proceedings of the 8th ACM SIGKDD international conference on knowledge discovery and data mining, 2002, pp 32–41
26. Tanbeer SK, Ahmed CF, Jeong B-S, Lee Y-K (2009) Efficient single-pass frequent pattern mining using a prefix-tree. Inf Sci 179(5):559–583
27. Tao F (2003) Weighted association rule mining using weighted support and significant framework. In: Proceedings of the 9th ACM SIGKDD international conference on knowledge discovery and data mining, 2003, pp 661–666
28. Tseng M-C, Lin W-Y, Jeng R (2008) Updating generalized association rules with evolving taxonomies. Appl Intell 29:306–320
29. UCI machine learning repository. Available from: http://archive.ics.uci.edu/ml/
30. Verma K, Vyas OP (2005) Efficient calendar based temporal association rule. SIGMOD Rec 34(3):63–70
31. Wang J, Han J, Pei J (2003) CLOSET+: searching for the best strategies for mining frequent closed itemsets. In: Proceedings of the 9th ACM SIGKDD international conference on knowledge discovery and data mining, 2003, pp 236–245
32. Wang CY, Tseng SS, Hong TP (2006) Flexible online association rule mining based on multidimensional pattern relations. Inf Sci 176:1752–1780
33. Wang W, Yang J, Yu PS (2004) WAR: weighted association rules for item intensities. Knowl Inf Syst 6:203–229
34. Wu F, Chiang S-W, Lin J-R (2007) A new approach to mine frequent patterns using item-transformation method. Inf Syst 32:1056–1072
35. Xiong H, Tan P-N, Kumar V (2006) Hyperclique Pattern Discovery. Data Min Knowl Discov 13:219–242
36. Yao H, Hamilton HJ (2006) Mining itemset utilities from transaction databases. Data Knowl Eng 59:603–626
37. Yao H, Hamilton HJ, Butz CJ (2004) A foundational approach to mining itemset utilities from databases. In: Proceedings of the 4th SIAM international conference on data mining, 2004, pp 482–486
38. Ye F-Y, Wang J-D, Shao B-L (2005) New algorithm for mining frequent itemsets in sparse database. In: Proceeding of the 4th international conference on machine learning and cybernetics, 2005, pp 1554–1558
39. Yun U (2007) Efficient mining of weighted interesting patterns with a strong weight and/or support affinity. Inf Sci 177:3477–3499
40. Yun U (2007) Mining lossless closed frequent patterns with weight constraints. Knowl-Based Syst 20:86–97
41. Yun U, Leggett JJ (2005) WFIM: weighted frequent itemset mining with a weight range and a minimum weight. In: Proceedings of the 5th SIAM international conference on data mining, 2005, pp 636–640

Chowdhury Farhan Ahmed received his B.S. and M.S. degrees in Computer Science from the University of Dhaka, Bangladesh in 2000 and 2002 respectively. From 2003–2004 he worked as a faculty member at the Institute of Information Technology, University of Dhaka, Bangladesh. In 2004, he became a faculty member in the Department of Computer Science and Engineering, University of Dhaka, Bangladesh. Currently, he is pursuing his Ph.D. degree in the Department of Computer Engineering at Kyung Hee University, South Korea. His research interests are in the areas of data mining and knowledge discovery.

Syed Khairuzzaman Tanbeer received his B.S. degree in Applied Physics and Electronics, and his M.S. degree in Computer Science from the University of Dhaka, Bangladesh in 1996 and 1998 respectively. Since 1999, he has been working as a faculty member in the Department of Computer Science and Information Technology at the Islamic University of Technology, Dhaka, Bangladesh. Currently, he is pursuing his Ph.D. in the Department of Computer Engineering at Kyung Hee University, South Korea. His research interests include data mining and knowledge engineering.

Byeong-Soo Jeong received his B.S. degree in Computer Engineering from Seoul National University, Korea in 1983, his M.S. degree in Computer Science from the Korea Advanced Institute of Science and Technology, Korea in 1985, and his Ph.D. in Computer Science from the Georgia Institute of Technology, Atlanta, USA in 1995. In 1996, he joined the faculty at Kyung Hee University, Korea where he is now an associate professor at the College of Electronics & Information. From 1985 to 1989, he was on the research staff at Data Communications Corp., Korea. From 2003 to 2004, he was a visiting scholar at the Georgia Institute of Technology, Atlanta. His research interests include database systems, data mining, and mobile computing.

Young-Koo Lee received his B.S., M.S., and Ph.D. degrees in Computer Science from the Korea Advanced Institute of Science and Technology, Korea. He is a professor in the Department of Computer Engineering at Kyung Hee University, Korea. His research interests include ubiquitous data management, data mining, and databases. He is a member of IEEE, the IEEE Computer Society, and the ACM.