IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 17, Issue 5, Ver. V (Sep. – Oct. 2015), PP 08-14
www.iosrjournals.org

Mining High Utility Itemsets from its Concise and Lossless Representations

Nasreen Ali A.[1], Arunkumar M[2]
[1](Computer Science and Engineering, Ilahia College of Engineering and Technology/ Mahatma Gandhi University, India)
[2](Computer Science and Engineering, Ilahia College of Engineering and Technology/ Mahatma Gandhi University, India)

Abstract: *Mining high utility items from databases using the utility of items is an emerging technology.Recent algorithms have a drawback in the performance level considering memory and time.Novel strategy proposed here is the Miner Algorithm.A vertical data structure is used to store the elements with the utility values. A matrix representation is generated to identify the element co-occurrences and reduce the join operation for the patterns generated. An extensive experimental study with the datasets shows that the resulting algorithm reduces the join operation upto 95% compared with the UP Growth state of the art algorithm.*
Keywords: *Utility, utility list, co-occurences, pruning*

I. Introduction

Data mining is concerned with large volumes of data to analyze and automatically discover interesting regularities or relationships. The primary goal is to discover hidden patterns, unexpected trends in the data. Data mining activities uses combination of techniques from database technologies, statistics, artificial intelligence and machine learning. Real world applications include bioinformatics, genetics, medicine, clinical research, education, retail and marketing research.

Frequency Mining [1] is a popular data mining task with a wide range of applications. Given a transaction database, it consists of discovering frequent itemsets. i.e. groups of items (itemsets) appearing frequently in transactions [1]. However, an important limitation of frequency mining is that all items have the same importance (weight, unit profit or value). These assumptions often do not hold in real applications. For example, consider a database of customer transactions containing unit profit for each item and different quantities of each item. Frequency mining algorithms would discard this information and may thus discover many frequent itemsets generating a low profit and fail to discover less frequent itemsets that generate a high profit.

Utility mining [5], [6], [7], [8], emerges as an important topic in data mining. In utility mining, each item has a weight (e.g. unit profit) and can appear more than once in each transaction (e.g. purchase quantity). The utility of an itemset represents its importance, which can be measured in terms of weight, profit, cost, quantity or other information depending on the user preference. Utility is a measure of how useful or profitable an itemset X is .The utility of items in a transaction database consists of two aspects: (1) the importance of distinct items, which is called external utility, and (2) the importance of the items in the transaction, which is called internal utility. The utility of an item is defined as the external utility multiplied by the internal utility. The utility of an itemset X, i.e., u(X), is the sum of the utilities of itemset X in all the transactions containing X. An itemset X is called a high utility itemset if and only if u(X) > min_utility, where min_utility is a user-defined minimum utility threshold. However, mining high utility itemsets from databases is not an easy task since downward closure property in frequent itemset mining does not hold. In other words, pruning search space for high utility itemset mining is difficult because a superset of a low-utility itemset may be a high utility itemset. A naı¨ve method to address this problem is to enumerate all itemsets from databases by the principle of exhaustion. Obviously, this method suffers from the problems of a large search space, especially when databases contain lots of long transactions or a low minimum utility threshold is set. Hence, how to effectively prune the search space and efficiently capture all high utility itemsets with no miss is a crucial challenge in utility mining.

To identify high utility itemsets, most existing algorithms first generate candidate itemsets by overestimating their utilities, and subsequently compute the exact utilities of these candidates. These algorithms incur the problem that a very large number of candidates are generated, but most of the candidates are found out to b e not high utility after their exact utilities are computed. In this paper, we propose an algorithm, called Miner, for high utility itemset mining .Miner uses a novel structure, called utility-list, to store both the utility information about an itemset and the heuristic information for pruning the search space of Miner. By avoiding the costly generation and utility computation of numerous candidate itemsets, Miner can efficiently mine high utility itemsets from the utility lists constructed from a mined database. To reduce the number of costly joins

that are performed, we propose a novel pruning strategy named EUCP (Estimated Utility Cooccurrence Pruning) that can prune itemsets without having to perform joins. This strategy is easy to implement and very effective. We compare the performance of Miner and UP Growth on real-life datasets. Results show that Miner performs upto 95% less join operations than UP Growth and is up to six times faster than UP Growth. Experimental results show that Miner outperforms this algorithm in terms of both running time and memory consumption.

II. Review of literature

Fast Algorithms for mining Association Rule by R.Agrawal and R.Srikant in 1994 proposed Apriori algorithm.Apriori is more efficient during the candidate generation process for two reasons; Apriori employs a different candidate's generation method and a new pruning technique. There are two processes to finds out all the large itemsets from the database in Apriori algorithm. First the candidate itemsets are generated, and then the database is scanned to check the actual support count of the corresponding itemsets. During the first scanning of the database the support count of each item is calculated and the large 1 -itemsets are generated by pruning those itemsets whose supports are below the predefined threshold. In each pass only those candidate itemsets that include the same specified number of items are generated and checked.

Advantages are 1] Uses large itemset property. 2] Easily parallelized. 3] Easy to implement. 4] It doesn't need to generate conditional pattern bases. Disadvantages: 1] It requires multiple database scans. 2] Assumes transaction database is memory resident. 3] Generating candidate itemsets.

Mining Frequent Patterns without Candidate Generation by J. Han, J. Pei, and Y. Yin in 2000 proposed a novel frequent pattern tree (FP-tree),which is an extended prefix tree structure for storing compressed, crucial information about frequent patterns, and develop an efficient FP-tree based mining method, FP-growth, for mining the complete set of frequent patterns by pattern fragment growth. Efficiency of mining is achieved with three techniques: (1) a large database is compressed into a highly condensed, much smaller data structure, which avoids costly, repeated database scans, (2) FP-tree-based mining adopts a pattern fragment growth method to avoid the costly generation of a large number of candidate sets, and (3)a partitioning-based, divide-and-conquer method is used to decompose the mining task into a set of smaller tasks for mining confined patterns in conditional databases, which dramatically reduces the search space. Performance study shows that the FP-growth method is efficient and scalable for mining both long and short frequent patterns, and is about an order of magnitude faster than the Apriori algorithm and also faster than some recently reported new frequent pattern mining methods. Here the item priority is not taken into consideration. Advantages are 1] It finds frequent itemsets without generating any candidate itemset 2] Scans database just twice. 3] Does not generate candidate itemsets. Disadvantages are 1] It treats all items with the same importance/weight/price. 2] Consumes more memory and performs badly with long pattern data sets.

A Fast High Utility Itemsets Mining Algorithm by Y. Liu, W.-K. Liao and A. Choudhary in 2005 proposed Two Phase algorithm. Utility mining focuses on identifying the itemsets with high utilities. As "downward closure property" doesn't apply to utility mining, the generation of candidate itemsets is the most costly in terms of time and memory space. In this paper, a Two-Phase algorithm is presented to efficiently prune down the number of candidates and can precisely obtain the complete set of high utility itemsets. In the first phase, a model is proposed that applies the "transaction-weighted downward closure property" on the search space to expedite the identification of candidates. In the second phase, one extra database scan is performed to identify the high utility itemsets. It performs very efficiently in terms of speed and memory cost, and shows good scalability on multiple processors, even on large databases that are difficult for existing algorithms to handle. Advantages are 1] It performs very efficiently in terms of speed and memory cost .Disadvantages are 1] Generate too many candidates to obtain HTWUI require multiple database scan.

UP-Growth: An Efficient Algorithm for High Utility Itemsets Mining by Vincent S. Tseng, Cheng-Wei Wu, Bai-En Shie, and Philip S. Yu in 2010, proposed an efficient algorithm, namely UP-Growth (Utility Pattern Growth), for mining high utility itemsets with a set of techniques for pruning candidate itemsets. The information of high utility itemsets is maintained in a special data structure named UP-Tree (Utility Pattern Tree) such that the candidate itemsets can be generated efficiently with only two scans of the database. The experimental results show that UP-Growth not only reduces the number of candidates effectively but also outperforms other algorithms substantially in terms of execution time, especially when the database contains lots of long transactions.

Mining High utility Itemsets without Candidate Generation by Mengchi Liu Wuhan proposed the algorithm HUI-Miner. High utility itemsets refer to the sets of items with high utility like profit in a database, and efficient mining of high utility itemsets plays a crucial role in many real-life applications and is an important research issue in data mining area. To identify high utility itemsets, most existing algorithms first generate candidate itemsets by overestimating their utilities, and subsequently compute the exact utilities of these candidates. These algorithms incur the problem that a very large number of candidates are generated, but most of the candidates are found out to be not high utility after their exact utilities are computed. HUI-Miner

uses a novel structure, called utility-list, to store both the utility information about an itemset and the heuristic information for pruning the search space of HUI-Miner. By avoiding the costly generation and utility computation of numerous candidate itemsets, HUI-Miner can efficiently mine high utility itemsets from the utility-lists constructed from a mined database. We compared HUI-Miner with the state-of-the-art algorithms on various databases, and experimental results show that HUI-Miner outperforms these algorithms in terms of both running time and memory consumption.

III. current work

We first introduce important preliminary definitions.

A transaction database is a set of transactions D = {T1, T2, Tn} such that for each transaction Tc has a unique identifier c called its Tid. Each item i ∈ I where I is the set of items is associated with a positive number p(i), called its external importance or utility (e.g. unit profit). For each transaction Tc such that i ∈ Tc, a positive number q (i, Tc) is called the internal importance or utility of i (e.g. purchase quantity).

Example 1. Consider the database of Fig. 1 (left), which will be used as our running example. This database contains five transactions (T1, T2...T5). Transaction T2 indicates that items a, c, e and g appear in this transaction with an internal utility of respectively 2, 6, 2 and 5. FIG. 1 (right) indicates that the external utility of these items are respectively 5, 1, 3 and 1.

The importance or utility of an item i in a transaction Tc is denoted as u (i, Tc) and defined as p (i) × q (i, Tc).The importance or utility of an itemset X (a group of items X ⊆ I) in a transaction Tc is denoted as u(X, Tc) and defined as u(X, Tc) = Pi ∈X u (i, Tc).

Example 2. The utility of item a in T2 is u(a,T2) = 5 × 2 = 10. The utility of the itemset {a, c} in T2 is u ({a, c}, T2) = u (a, T2) + u(c, T2) = 5 × 2 + 1 × 6 = 16.

The importance or utility of an itemset X is denoted as u(X) and defined as u(X) = PTc ∈g (X) u(X, Tc), where g(X) is the set of transactions containing X.

Example 3. The importance or utility of the itemset {a,c} is u({a,c}) = u(a)+u(c) = u(a,T1)+u(a,T2) + u(a,T3) + u(c,T1) + u(c,T2) + u(c,T3) = 5 + 10 + 5 + 1 + 6 + 1 = 28.

The purpose of high-utility itemset mining is to discover all high-utility itemsets. An itemset X is a high-utility itemset if its utility u(X) is no less than a user-specified minimum utility threshold min_util given by the user. Otherwise, X is a low-utility itemset.

Example 4. If min_util = 30, the high-utility itemsets in the database of our running example are {b,d}, {a,c,e}, {b,c,d}, {b,c,e}, {b,d,e}, {b,c,d,e} with respectively a utility of 30, 31, 34, 31, 36, 40 and 30.

The utility of the transaction (TU) Tc is the sum of the utility of the items from Tc in Tc. i.e. TU (Tc) =Px ∈Tc u(x, Tc).

Example 5. FIG. 2 (left) shows the TU of transactions T1, T2, T3, T4, and T5 from our running example.

The transaction utility of transactions containing the item gives the Transaction Weighted Utility of the item X, i.e. TWU(X) = PT c ∈g (X) TU (Tc).

Example 6. Fig. 2 (center) shows the TWU of single items a, b,c,d, d, e, f, g.Consider item a. TWU(A) = TU(T1) + TU(T2) + TU(T3) = 8 + 27 + 30 = 65.

The TWU measure has three important properties that are used to prune the search space. Property (overestimation). The TWU of an itemset X is higher than or equal to its utility, i.e. TWU(X) ≥ u(X) [8].

Property 2 (antimonotonicity). The TWU measure is anti-monotonic. Let X and Y be two itemsets. If X ⊂ Y, then TWU(X) ≥TWU(Y) [8].

Property 3 (pruning). Let X be an itemset. If TWU(X) < min_util, then the itemset X is a low-utility itemset as well as all its supersets. Proof. This directly follows from Property 1 and Property 2.

The set of items in the utility-list of an itemset X in a database D is a set of tuples such that there is a tuple (tid, iutil, rutil) for each transaction T tid containing X. The iutil element of a tuple is the utility of X in Ttid. i.e. u(X, Ttid). The rutil element of a tuple is defined as Pi ∈Ttid ∧i 6∈X U (i, Ttid).

Example 7. The utility-list of {a} is {(T1, 5, 3) (T2, 10, 17) (T3, 5, 25)}. The utility list of {e} is {(T2, 6, 5) (T3, 3, 5) (T4, 3, 0)}. The utility-listof {a, e} is {(T2, 16, 5), (T3, 8, 5)}.

To obtain the high-utility itemsets, Miner perform only one database scan to create utility-lists of patterns containing single items. Then, bigger patterns are obtained by performing the join operation of utility-lists of smaller patterns. Pruning the search space is done using the two following properties.

Property 4 (sum of iutils). Let X be an itemset. If the sum of iutil values in the utility-list of x is higher than or equal to min_util, then X is a high-utility itemset. Otherwise, it is a low-utility itemset [7].

Property 5 (sum of iutils and rutils). Let X be an itemset. Let the extensions of X be the itemsets that can be obtained by appending an item y to X such that y i for all item i in X. If the sum of iutil and rutil values in the utility-list of x is less than min_util, all extensions of X and their transitive extensions are low-utility itemsets [7].

In the next section, we introduce our novel algorithm, which improves upon existing algorithms by being able to eliminate low-utility itemsets without performing join operations.

Table. 1. A transaction database (left) and external utility values (right)

id	**Transactions**
T1	(a,1)(c,1)(d,1)
T2	(a,2)(c,6)(e,2)(g,5)
T3	(a,1)(b,2)(c,1)(d,6),(e,1),(f,5)
T4	(b,4)(c,3)(d,3)(e,1)
T5	(b,2)(c,2)(e,1)(g,2)

Item	a	b	c	d	e	f	g
Profit	5	2	1	2	3	1	1

Table. 2. Transaction utilities (left), TWU values (center) and EUCS (right)

TID	TU
T_1	8
T_2	27
T_3	30
T_4	20
T_5	11

Items	TWU
a	61
b	65
c	96
d	58
e	88
f	30
g	38

Item	a	b	c	d	e	f
b	30					
c	65	61				
d	38	50	58			
e	57	61	77	50		
f	30	30	30	30	30	
g	27	38	38	0	38	0

IV. Algorithm

In this section, we present our proposal, the Miner algorithm. The main procedure (Algorithm 1) takes as input a transaction database with utility values and the min_util threshold. The algorithm first scans the database to calculate the TWU of each item. Then, the algorithm identifies the set I∗ of all items having a TWU greater than min_util .TWU values can be used to arrange the items in the ascending order. Items in transactions are reordered according to the total order during the second database scan and the utility-list of each item i ∈ I∗ is built and the novel structure named EUCS (Estimated Utility Co-Occurrence Structure) is built. This structure is defined as a set of triples of the form (a, b, c) ∈ I∗ × I∗ × R. A triple (a, b, c) indicates that TWU ({a, b}) = c.

The EUCS can be implemented as a triangular matrix as shown in TABLE. 2 (right) where only tuple of the form (a, b, c) such that c ≠ 0 are kept. EUCS structure is more memory efficient because only few items co-occur with other items. Building the EUCS is very fast (it is performed with a single database scan) and occupies a small amount of memory, bounded by |I∗|×|I∗|, although in practice the size is much smaller because a limited number of pairs of items co-occur in transactions. After the construction of the EUCS, the depth-first search exploration of itemsets starts by calling the recursive procedure Search with the empty itemset ∅, the set of single items I∗, min_util and the EUCS structure.

The Search procedure (Algorithm 2) has input (1) an itemset P, (2) extensions of P having the form Pz meaning that Pz was previously obtained by appending an item z to P, (3) min_util and (4) the EUCS. The search procedure operates as follows. The sum of iutil values of each extension of P, i.e Px, is taken and if it is greater than min_util then Px is a high-utility itemset and it is output. The sums of iutil and rutil values in the utility list of Px are greater than min_util then the extensions of Px should be explored. This is performed by merging Px with all extensions Py of P such that order of y greater than x to form extensions of the form Pxy containing |Px| + 1 items. The utility-list of Pxy is then constructed by calling the Construct procedure (cf. Algorithm 3) to join the utility-lists of P, Px and Py. Then, a recursive call to the Search procedure with Pxy is

done to calculate its utility and explore its extension(s). Search procedure is recursive and starts from single items, and appends single items and it only prunes the search space based on Property 5. It can be easily seen based on Property 4 and 5 that this procedure is correct and complete to discover all high-utility itemsets.

The Construct procedure considers the revised transactions where the transactions are arranged in the ascending order of the TWU.X/T is the set of all the items in T after X.ru(X/T) is the remaining utility of itemset X in T which is the sum of all the items in X/T.Here there is no need of database scan. First we identify the common one item transaction and combine it to form the two item utility list. To construct the k item utility list we have to combine the k-1 item utility list and k item utility list

Co-occurrence based Pruning. The main novelty in Miner is a pruning mechanism named EUCP (Estimated Utility Co-occurrence Pruning), which uses the EUCS. EUCP performs the construction of the utility list by eliminating the low-utility extension of Pxy and all its transitive extensions. This is done on line 8 of the Search procedure. The pruning condition do not explore Pxy and its supersets if any tuple (x, y, c) in EUCS such that $c \leq$ min_util. This strategy is correct (only prune low-utility itemsets). The proof is that by Property 3, if an itemset X contains another itemset Y such that TWU(Y) <min_util, then X and its supersets are low-utility itemsets. Search procedure is recursive and will check all other pairs of items in Pxy in previous recursions of the Search procedure leading to Pxy. For example, consider an itemset Z = {a1, a2, a3, a4}. To generate this itemset, the search procedure had to combine {a1, a2 a3} and {a1, a2, a4}, obtained by combining {a1, a2} and {a1, a3}, and {a1, a2} and {a1, a4}, obtained by combining single items. It can be easily observed that when generating Z all pairs of items in Z have been checked by EUCP except {a3, a4}.

Algorithm 1: Miner Algorithm
input: D: a transaction database, min_util: a user-specified threshold
output: the set of high-utility itemsets

1 Scan D to calculate the TWU of single items;
2 I ∗ ← each item i such that TWU (i) < min_util;
3 Let be the total order of TWU ascending values on I ∗;
4 Scan D to built the utility-list of each item i ∈ I ∗ and build the EUCS structure;
5 Search (∅, I ∗, min_util, EUCS);

Algorithm 2: Search Algorithm
input: P: an itemset, ExtensionsOfP: a set of extensions of P, the min_util threshold, the EUCS structure
output: the set of high-utility itemsets

```
1 foreach itemset P x ∈ ExtensionsOfP do
2       if SUM (Px.utilitylist.iutils) ≥ min_util then
3               output Px;
4        end
5       if SUM (Px.utilitylist.iutils) +SUM (Px.utilitylist.rutils) ≥ min_util then
6               ExtensionsOfPx ← ∅;
7               foreach itemset Py ∈ ExtensionsOfP such that y  x do
8                       if ∃(x, y, c) ∈ EUCS such that c ≥ min_util) then
9                               Pxy ← Px ∪ Py;
10                              Pxy.utilitylist ← Construct (P, Px, Py);
11                              ExtensionsOfPx ← ExtensionsOfPx ∪ Pxy;
12                      end
13              end
14              Search (Px, ExtensionsOfPx, min_util);
15      end
16 end
```

Algorithm 3: Construct Algorithm
input : P: an itemset, P x: the extension of P with an item x, P y: the extension of P with an item y
output: the utility-list of P xy

```
1 UtilityListOfPxy ← ∅;
2 foreach tuple ex ∈ P x.utilitylist do
3       if ∃ey ∈ P y.utilitylist and ex.tid = exy.tid then
4                if P.utilitylist ≠ ∅ then
```

```
5                   Search element e ∈ P.utilitylist such that e.tid = ex.tid.;
6               exy ← (ex.tid, ex.iutil + ey.iutil − e.iutil, ey.rutil);
7               end
8               else
9                   exy ← (ex.tid, ex.iutil + ey.iutil, ey.rutil);
10              end
11              UtilityListOfP xy ← UtilityListOfP xy ∪ {exy};
12      end
13 end
14 return UtilityListPxy;
```

V. Experimental study

We performed experiments to assess the performance of the proposed algorithm. Experiments were performed on a computer with a 64 bit Corei5 processor running Windows 7 and 5 GB of free RAM. We compared the performance of Miner with the state-of-the-art algorithm UP Growth for high-utility itemset mining. All memory measurements were done using the Java API. Experiments were carried on real-life dataset having varied characteristics. The dataset contains 1,112,949 transactions with 46,086 distinct items and an average transaction length of 7.26 items. External utilities for items are generated between 1 and 1,000 by using a log-normal distribution and quantities of items are generated randomly between 1 and 5, as the settings of [2, 7, 10].

Execution time: We first ran the Miner and UP growth algorithms on the dataset while decreasing the min_util threshold until algorithms became too long to execute, ran out of memory or a clear winner was observed. We recorded the execution time, the percentage of candidate pruned by the Miner algorithm and the total size of the EUCS. The comparison of execution time against min-utility threshold is shown in Fig 1. Miner was faster up to 6 times than UP growth.

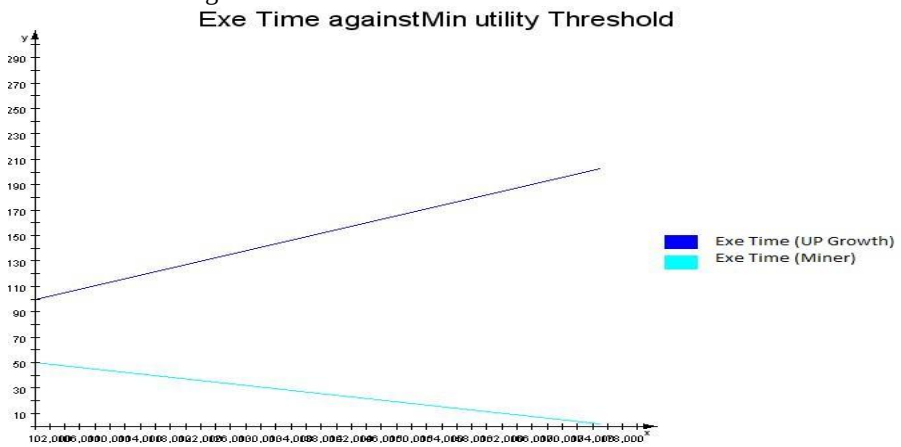

Fig: 1 Performance of Execution Time against Min utility Threshold

Pruning effectiveness: These results show that candidate pruning can be very effective by pruning up to 95% of candidates. As expected, when more pruning was done, the performance gap between Miner and UP growth became larger.

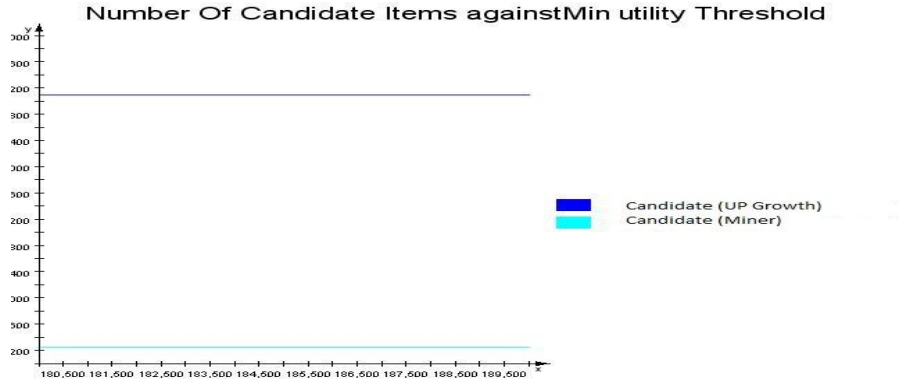

Fig: 2 Performance of Candidate Items against Min utility threshold

Memory overhead: We also studied the memory overhead of using the EUCS structure. We found that the memory footprint of EUCS was 6 times less than UP Growth. We therefore conclude that the cost of using the EUCP strategy in terms of memory is low.

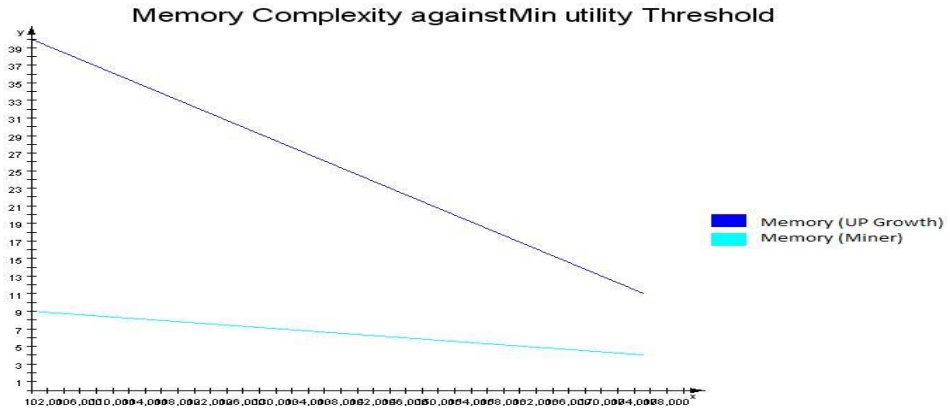

Fig: 3 Performance of Memory Complexity against Threshold.

VI. Conclusion

In this paper, we have presented a novel algorithm for high-utility itemset mining named Miner. This algorithm integrates a novel strategy named EUCP (Estimated Utility Cooccurrence Pruning) to reduce the number of joins operations when mining high-utility itemsets using the utilitylist data structure. We have performed an extensive experimental study on real-life datasets to compare the performance of Miner with the state-of-the-art algorithm UP Growth. Results show that the pruning strategy reduces the search space by upto 95 % and that Miner is up to 8 times faster than UP Growth.

Acknowledgements

The authors wish to thank the Management, the Principal and Head of the Department (CSE) of ICET for the support and help in completing the work.

References

[1]. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: Proc. Int. Conf. Very Large Databases, pp. 487–499, (1994)
[2]. V. S. Tseng, C.-W. Wu, B.-E. Shie, and P. S. Yu, "UP-Growth: An efficient algorithm for high utility itemset mining," in Proc. ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining, 2010, pp. 253–262.
[3]. Y. Liu, W. Liao, and A. Choudhary, "A fast high utility itemsets mining algorithm," in Proc. Utility-Based Data Mining Workshop,2005, pp. 90–99.
[4]. Vincent S. Tseng, Bai-En Shie, Cheng-Wei Wu, and Philip S. Yu,Fellow, IEEE, "Efficient Algorithms for Mining High Utility Itemsets from Transactional Databases", IEEE Transactions On Knowledge And Data Engineering, Vol. 25, No. 8, August 2013.
[5]. C. Lucchese, S. Orlando, and R. Perego, "Fast and memory efficient mining of frequent closed itemsets," IEEE Trans. Knowl. Data Eng., vol. 18, no. 1, pp. 21–36, Jan. 2006.
[6]. K. Chuang, J. Huang, and M. Chen, "Mining top-k frequent patterns in the presence of the memory constraint," VLDB J., vol. 17, pp. 1321–1344, 2008.
[7]. R. Chan, Q. Yang, and Y. Shen, "Mining high utility itemsets," in Proc. IEEE Int. Conf. Data Min., 2003, pp. 19–26.
[8]. Erwin, R. P. Gopalan, and N. R. Achuthan, "Efficient mining of high utility itemsets from large datasets," in Proc. Int. Conf. Pacific- Asia Conf. Knowl. Discovery Data Mining, 2008, pp. 554–561.
[9]. K. Gouda and M. J. Zaki, "Efficiently mining maximal frequent itemsets," in Proc. IEEE Int. Conf. Data Mining, 2001, pp. 163–170.
[10]. Mengchi Liu, Junfeng Qu, "Mining High Utility Itemsets without Candidate Generation",in Proceeding CIKM '12 Proceedings of the 21st ACM international conference on Information and knowledge management,Pages 55-64
[11]. C. W. Wu, P. Fournier-Viger, P. S. Yu, and V. S. Tseng," Efficient mining of a concise and lossless representation of high utility itemsets", In Proc. IEEE Int'l Conf. Data Mining, pages 824 –833, 2011.
[12]. C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong and Y.-K. Lee, "Efficient tree structures for high utility pattern mining in incremental databases," IEEE Trans. Knowl. Data Eng., vol. 21, no. 12, pp. 1708– 1721, Dec. 2009.