Expert Systems With Applications 101 (2018) 91–115

Contents lists available at ScienceDirect

Expert Systems With Applications

journal homepage: www.elsevier.com/locate/eswa

An empirical evaluation of high utility itemset mining algorithms

Chongsheng Zhang, George Almpanidis*, Wanwan Wang, Changchang Liu

School of Computer and Information Engineering, Henan University, KaiFeng, 475001, China

ARTICLE INFO

Article history:
Received 21 August 2017
Revised 10 January 2018
Accepted 3 February 2018
Available online 8 February 2018

Keywords:
Itemset mining
High utility itemsets
State-of-the-art high utility itemset mining

ABSTRACT

High utility itemset mining (HUIM) has emerged as an important research topic in data mining, with applications to retail-market data analysis, stock market prediction, and recommender systems, etc. However, there are very few empirical studies that systematically compare the performance of state-of-the-art HUIM algorithms. In this paper, we present an experimental evaluation on 10 major HUIM algorithms, using 9 real world and 27 synthetic datasets to evaluate their performance. Our experiments show that EFIM and d2HUP are generally the top two performers in running time, while EFIM also consumes the least memory in most cases. In order to compare these two algorithms in depth, we use another 45 synthetic datasets with varying parameters so as to study the influence of the related parameters, in particular the number of transactions, the number of distinct items and average transaction length, on the running time and memory consumption of EFIM and d2HUP. In this work, we demonstrate that, d2HUP is more efficient than EFIM under low minimum utility values and with large sparse datasets, in terms of running time; although EFIM is the fastest in dense real datasets, it is among the slowest algorithms in sparse datasets. We suggest that, when a dataset is very sparse or the average transaction length is large, and running time is favoured over memory consumption, d2HUP should be chosen. Finally, we compare d2HUP and EFIM with two newest algorithms, mHUIMiner and ULB-Miner, and find these two algorithms have moderate performance. This work has reference value for researchers and practitioners when choosing the most appropriate HUIM algorithm for their specific applications.

1. Introduction

High utility itemset (HUI) mining (HUIM), which discovers sets of items occurring together with high combined profits in sales, is an important research topic in data mining that has received the focus of many researchers. This is reflected in a variety of business applications and scientific domains, where HUIM techniques are exploited for finding profitable patterns in transactional databases. In the online shopping scenario, besides recommending the correlated items, the sales managers are also interested in suggestions of HUIs that can be placed together for attracting customers' attention and obtain higher profits. HUIM has also found place in other application areas, such as web click stream analysis, mobile commerce environment planning (Shie, Hsiao, Tseng, & Philip, 2011), and biological gene data analysis (Ahmed, Tanbeer, Jeong, & Lee, 2009).

HUIM is directly related to Frequent Itemset Mining (FIM), which chronologically precedes the former for almost a decade. HUIM can be considered a generalisation of FIM, since it associates weights (utilities) with the items in transactions - if these weights have unit values it then degenerates to FIM. Naturally, many algorithms and techniques in HUIM have been adapted from FIM. Since the early studies in HUIM, it became evident that it is a harder problem than FIM, because the utility of an itemset is neither monotonic nor anti-monotonic, i.e., a HUI itemset does not imply that its subsets will be HUIs as well (Liu, Liao, & Choudhary, 2005a). Thus, techniques that prune the search space developed in FIM, based on the anti-monotonicity of the support property, cannot be directly applied to HUIM.

HUIM algorithms can be chronologically divided into three categories. Algorithms in the first category, such as the Two-Phase algorithm (Liu, Liao, & Choudhary, 2005b), IHUP (Ahmed et al., 2009), and UP-Growth (Tseng, Wu, Shie, & Yu, 2010), first generate potential HUIs (candidates), then compute the utilities of each candidate in the second phase, by recursively exploring the tree structure that keeps the transactions. But this type of approach leads to memory and running time overhead, due to the overly large number of candidates generated in the first phase. Algorithms in the second category try to avoid the above problem by utilising various techniques, data structures, and properties, which limit the number of candidates and prune further the unpromising candi-

* Corresponding author.
E-mail addresses: chongsheng.zhang@yahoo.com (C. Zhang), almpanidis@gmail.com (G. Almpanidis), 1241094554@qq.com (W. Wang), 25475510@qq.com (C. Liu).

https://doi.org/10.1016/j.eswa.2018.02.008
0957-4174/

dates. More recent algorithms, such as HUI-Miner (Liu & Qu, 2012), HUP-Miner (Krishnamoorthy, 2015), d2HUP (Liu, Wang, & Fung, 2012), FHM (Fournier-Viger, Wu, Zida, & Tseng, 2014), and EFIM (Zida, Fournier-Viger, Lin, Wu, & Tseng, 2015) belong to this category. The third category consists of the newest HUIM algorithms, including mHUIMiner (Peng, Koh, & Riddle, 2017) and ULB-Miner (Duong, Fournier-Viger, Ramampiaro, Nørvåg, & Dam, 2017). These algorithms are based upon algorithms of the second category; they introduce effective structures to further reduce unnecessary utility-list constructions or speed up the join operation of the utility-lists.

Up to now, there is no theoretical work that addresses the time and space complexity of existing HUIM algorithms. Furthermore, there is little benchmarking work that comprehensively compares and summarises the performance of all the well-known HUIM algorithms. In this work, we conduct extensive experiments on both synthetic datasets (with 27 + 45 = 72 databases) and 9 real-world datasets, to benchmark the performance of 10 major HUIM algorithms from all the three categories. This work is important for researchers, domain experts, engineers, and developers, who need to apply a HUIM algorithm in their specific applications. Since it is very time-demanding to try out all the different algorithms, a comprehensive guideline or a benchmark reference to promptly pick the best performing HUIM algorithm for their datasets is beneficial.

This study gives a review of current techniques in HUIM and tries to answer the following question: is there a clear winner out of all HUIM algorithms, regarding running time and memory consumption? How do different parameters influence the performance of certain algorithms? In specific, we investigate the impact of the minimum utility threshold on the scalability of the top algorithms and check the performance hit when the dataset is very sparse or very dense, and when the database size varies.

Through extensive experiments on a large repository containing both real-world and synthetic datasets, we find out that, EFIM is a memory-efficient algorithm; it is the fastest method in dense real datasets and small synthetic datasets, but at large minimum utility thresholds. However, in all the other cases, especially when the dataset is very sparse or when minimum utility threshold is low, d2HUP is attested to be the fastest algorithm, with EFIM being the second. The running time performance of the newest HUIM algorithms, i.e. mHUIMiner and ULB-Miner, falls between EFIM and d2HUP.

The rest of the paper is structured as follows: in Section 2, we introduce the mathematical preliminaries associated with the HUIM concepts, and review the literature. Next, we briefly describe the HUIM algorithms to be compared in Section 3. In Section 4, we give details of the experimental setup, then in Section 5 we present and analyse the results. Finally, we conclude the work with directions for future research in Section 6.

2. Background and preliminaries

2.1. Background

Transactional databases, such as collections of items bought by customers recorded by point-of-sale (POS) systems in supermarkets or online Customer Relationship Management (CRM) systems for online stores, have important reference value to marketers for discovering hidden customer purchase patterns. This can serve as a basis for decisions about marketing activities, such as targeted marketing campaigns, promotional pricing, and products placement, to maximize the revenue or improve the customers' shopping experience.

While Frequent Itemset Mining (FIM) aims at finding the items that frequently co-occur in the same transactions, the goal of HUIM is to identify the items that appear together but also bring large profits to the merchants. Although the managers of large supermarkets and online shops know each product's profit per unit and the corresponding purchase quantities, it is not advisable to just focus on selling or promoting the most profitable products, because, in real situations, there may not be enough sales of these products. Likewise, it is inadequate to emphasize just popular products that are selling in large quantities, as these might have lower unit profits and would require an order of magnitude increase in purchase quantity to obtain significant total profit gain and will also occupy more space.

Instead, managers should investigate the historical transactions and extract the *set of items* (itemsets) that have maximum combined profits, by first multiplying the individual profit with the number of sales of the same item (hereafter referred to as the *utility* of the item), then summing up all the utilities of the items that have appeared together in the actual transactions (hereafter referred to as the *utility* of an *itemset*). The itemsets having top utility values should be preferred.

However, as it is computationally infeasible to enumerate all the possible itemsets and compute their corresponding utilities, it is necessary to prune the search space. To this end, a great many HUIM algorithms have been proposed.

2.2. Motivating applications: online advertising

In the domain of online advertising, the revenue for advertising agencies (such as Google and Bing) can increase by identifying and displaying the advertisements (ads) that are both relevant to the interests of the viewers who browse a Web site, but also profitable for the Web site owner who is an Ad agency customer.

It is clear that an Ad agency should display ads which the viewers may find interesting. This can be done by checking their cookies or history browsing logs/records. But there will still be too many candidate ads to display. Furthermore, different advertisements can have different prices, depending on their placement.

Now, the problem is, given the unit profit for each ad, and the historical Web logs of the click-or-not information on different groups of ads, how to compute and recommend the most profitable set of ads for the viewers? Simply displaying the top-k most expensive and profitable ads may not work out well, as few viewers might be interested in them. On the other hand, if enough viewers have browsed a few less profitable (i.e., not top-ranked in terms of individual profits) ads together, the summed revenue from these ads can be more significant than top-k most profitable ads individually.

It is clear that FIM algorithms can not solve the above problem, because they only consider the co-occurrences of ads/items while ignoring their unit profits. Instead, HUIM algorithms can be adopted here to compute/recommend the optimal group of ads to be displayed to the viewers.

2.3. Notations

First, we formally define the key terms in utility mining using the standard conventions followed in the literature. In Table 1, we list the symbols and their notations used in this paper. There are 4 separate groups in the table: the first group contains basic symbols, whereas the second group list the symbols that we use in this paper. The third group concerns functions for different utility definitions, while the last group includes abbreviations for HUI and HUIM that are used in the literature.

A *transactional database* D is a set of $|D|$ *transactions*, $T = \{T_1, T_2, \ldots, T_{|D|}\}$, where each transaction T_t $(1 \le t \le |D|)$ has a unique transaction identifier (TID) and is a finite set of $|I|$ *items* (symbols). I_i, $i = 1, ., |I|$, are the corresponding item identifiers (IID). It holds that $T_t \subseteq I$, where $I = \{I_1, I_2, \ldots, I_{|I|}\}$ is the list of distinct items that

Table 1
List of symbols and notations.

Symbol	Notation
D	A transaction database
$\|D\|$	The number of transactions in D
I	The set with all the distinct items
$\|I\|$	The number of distinct items in D
I_i	The Identifier of an item in D
T_t	The tth transaction in D
$\|T_t\|$	The number of items in T_t
TID	The transaction Identifier
$p(I_i)$	The profit of item I_i (external utility)
$q(I_i, T_t)$	The quantity of item I_i in transaction T_t (internal utility)
k	The cardinality of an itemset X
X	An itemset of k items
support(X)	The number of transactions in D that contains a given itemset X
dbItems	The number of distinct items in D, the same as $\|I\|$
dbTrans	The number of transactions in D, the same as $\|D\|$
density(D)	The density of D, $d(D) = avgTransLen/dbItems$
dbUtil	The total utility of all the transactions in D, the same as *TUD*
maxTransLen	The maximum transaction length in D
avgTransLen	The average transaction length of all the transactions in D
minU	Minumum utility threshold
minUP	Minumum utility threshold in percentage, $minUP = 100 * minU/dbUtil$
*minUP*1	The minUP value where only 1 HUI is generated
TU(X)	The total utility of itemset X in all the transactions that contain X
TWU(X)	The utilities of all the transactions that contain X
TUD	The total utility of all the transactions in D
$U(I_i, T_t)$	The utility of an item I_i in the transaction T_t, $U(I_i, T_t) = p(I_i) * q(I_i, T_t)$
$U(X, T_t)$	The sum of the utility of each item in X in T_t, $U(X, T_t) = \sum U(I_i, T_t)$
$U(T_t)$	The utility of a transaction T_t, $U(T_t) = \sum I_i \in T_t$
HUI	High Utility Itemsets
HUIM	High Utility Itemset Mining
CHUI	Closed HUIs
TKU	Top-k HUIs

can appear in the transactions of the database. Each item I_i in the database is associated with a positive number $p(I_i) \leq 0$, which is called its *external utility* and corresponds to the *unit profit*, i.e. the profit out of a single item. Additionally, every item I_i that occurs in a transaction T_t is associated with positive number $q(I_i, T_t) \geq 0$, which is called its *internal utility* and corresponds to the occurrence quantity, i.e., how many times this item occurs in the given transaction.

An *itemset* (or pattern) X is a non-empty set of k distinct items I_i, where $I_i \in I$, $1 \leq i \leq |I|$. An itemset consisting of k items/elements, i.e. an itemset of *length* or cardinality k, is denoted as *k-itemset*. For instance, itemset $\{a, b, c\}$ is a 3-itemset. Similarly, we can define the length of a transaction, for example, the length of transaction $T_1 = \{a, b, c\}$ is 3. It is apparent that an itemset of length 1 corresponds to a single item. Also, a database transaction is actually an itemset but associated with a TID, which implies that two transactions with different TIDs, can correspond to the same itemset, and an itemset can appear in multiple/different transactions.

The *support* of an itemset X is the frequency of X appearing in the database, i.e., the number of transactions that contain X in the database.

The *utility of an item* I_i in a transaction T_t, indicates the total profit of this item in the transaction. It is defined in Eq. (1).

$$U(I_i, T_t) = p(I_i) \times q(I_i, T_t) \tag{1}$$

Similarly, the utility of an itemset X in a transaction T_t, $X \subseteq T_t$, is defined as

$$U(X, T_t) = \sum_{i \in X} U(I_i, T_t) \tag{2}$$

It is a measure of how "useful" or profitable an itemset is.

The utility of a transaction T_t, is defined as the sum of the utilities of all the items I_i that occur in the transaction T_t:

$$U(T_t) = \sum_{I_i \in T_t} U(I_i, T_t) \tag{3}$$

Table 2
Transactional database.

TID	Items
T_1	(a, 5), (b, 3), (d, 1)
T_2	(a, 2), (c, 2), (d, 3), (g, 1)
T_3	(b, 2), (c, 1), (e, 2)
T_4	(a, 6), (e, 1), (f, 2)
T_5	(e, 3), (g, 2)
T_6	(b, 3), (d, 2), (e, 1), (f, 4)

Table 3
The profit unit values of the items.

Item	Unit profit
a	1
b	2
c	7
d	5
e	5
f	1
g	3

Let us use a transaction database given in Table 2. Transaction T_1 indicates that items a, b, and d were bought in quantities 5, 3 and 1, respectively. Table 3 gives the corresponding unit profit (external utility) of each item.

For example, the utility of item a in T_1 is $U(a, T_1) = p(a) \times q(a, T_1) = 1 \times 5 = 5$. The utility of the itemset $X = \{a, b\}$ in transaction T_1 is $U(X, T_1) = U(a, T_1) + U(b, T_1) = 1 \times 5 + 3 \times 2 = 11$. Similarly, the utility of transaction T_1 is $U(T_1) = U\{a, b, d\} = U(a, T_1) + U(b, T_1) + U(d, T_1) = 5 + 6 + 5 = 16$.

The utility of an itemset X in a transactional database D, denoted as *TU*(X), is defined as the sum of the itemset utilities in all

the transactions of D where X appears, i.e.,

$$TU(X) = \sum_{T_t \in G(X)} U(X, T_t) \quad (4)$$

where $G(X) \subseteq D$ is the set of transactions that contain X.

In the current example, the utility of itemset $X = \{a, d\}$ in D is $TU(X) = U(X, T_1) + U(X, T_2) = U(a, T_1) + U(d, T_1) + U(a, T_2) + U(d, T_2) = 5 + 5 + 2 + 15 = 27$.

The total utility of the database (TUD) is the sum of all the transaction utilities in the database. For example, the total utility of D in Table 2 is $TUD = \sum_{t \in T} U(T_t) = 130$. We refer to *TUD* as *dbUtil* hereafter.

Problem definition of HUIM. The goal of HUIM is to identify all the itemsets with a utility no less than a user-specified minimum utility threshold, denoted as *minU*. That is, for any itemset X that satisfies the above constraint, i.e., $TU(X) \leq minU$. These itemsets are called High-Utility Itemsets (HUIs). Identifying HUIs is equivalent to finding the itemsets that cover the largest portion of the total database utility. In the current example, if we set $minU = 34$, the HUIs in the database are {a, c, d, g} and {e} with utilities 34 and 35, respectively. It is clear that the smaller the *minU* value, the more HUIs are retrieved. The percentage of *minU* to the total utility of the database is denoted as *minimum utility percentage* (*minUP*) hereafter, i.e. $minUP = 100 \times minU/dbUtil$.

The transaction-weighted utilization of an itemset X, denoted as $TWU(X)$, is the sum of the transaction utilities of all the transactions containing X (Liu et al., 2005b):

$$TWU(X) = \sum_{X \subseteq T_t \wedge T_t \in D} U(T_t) \quad (5)$$

For the example in Table 2, $TWU(a, d) = U(T_1) + U(T_2) = 16 + 34 = 50$.

The difference between $TWU(X)$ and the $TU(X)$ is that for $TWU(X)$, we sum the utilities of the whole transactions containing X, while $TU(X)$ only calculates the utilities of X in the transactions where X occurs. Thus, $TWU(X) \geq TU(X)$.

TWU has three important properties that are used to prune the search space.

1. Property 1 (Overestimation). The $TWU(X)$ of an itemset X is always higher than or equal to the utility of X, i.e., $TWU(X) \geq U(X)$.
2. Property 2 (Anti-monotonicity). The TWU measure is anti-monotonic. Let X and Y be two itemsets. If $X \subset Y$, then $TWU(X) \geq TWU(Y)$.
3. Property 3 (Pruning). Let X be an itemset. If $TWU(X) < minU$, then X and all its supersets are not HUIs. This can be derived from Property 1 and Property 2.

An itemset X is called a *candidate itemset*, or *promising* HUI (PHUI), if $TWU(X) \geq minU$, otherwise it is called an *unpromising* itemset. In HUIM, we can easily pre-compute the transaction utility $U(T_t)$ of each transaction, which can be utilised when computing $TWU(X)$ for a given itemset X, for prompt pruning. Moreover, we also need efficient indexing structures to promptly find the transactions that contain X, in order to calculate $TU(X)$.

Actually, if no pruning techniques are adopted, a brute-force approach, in which every itemset is enumerated and evaluated, is computationally expensive/prohibitive and, for large databases, it is even infeasible, as the search space may grow exponentially with regard to the number of distinct items in the database, i.e., as large as $2^{|I|}$.

Typically, the transaction table for the database can be modelled in tuples of (*TID, IID, q*). For readability reasons, the quantities are shown beside the IIDs in parentheses in Table 2.

Based on the above definition of transactional database, it is natural to consider items and transactions as equivalent to term and documents in Information Retrieval (IR). As in IR, where documents can be considered as atomic units of information consisting of smaller indivisible text elements (terms) and we operate on the term-document incidence matrix in either document (column) space or term (row) space (or both), depending on the dimensions of the matrix, we could work similarly in HUIM. For example, gene expression datasets might have 100,000 columns but only 100–1000 rows (Zhu, 2014). This is also true in market basket analysis, if for example we are interested in transactions in a short period of time, where the number of transactions (rows) could be much smaller than the number of available distinct items in the database.

Table 4
Table representation of transactional database.

TID	Items						
	a	*b*	*c*	*d*	*e*	*f*	*g*
T_1	1	2	0	4	0	0	0
T_2	1	0	3	4	0	0	7
T_3	0	2	0	4	5	0	0
T_4	1	0	0	0	5	6	0
T_5	0	0	0	0	5	0	7
T_6	0	2	0	4	5	6	0

Depending on whether the number of transactions (rows) is considerably larger than the number of distinct items (columns/features) or vice versa, we can use row enumeration or column enumeration strategies respectively. In column enumeration, for a moderate increase in columns the running time increases exponentially. Even though the representation in Table 4 is conceptually straightforward, it is evident that it is inefficient, since in most applications the corresponding matrix is very sparse. The typical length of transactions might be a magnitude of order smaller than the number of distinct items in the database, $|T_i| << |I|$, while the size of the database $|D|$ (the number of the transactions, i.e. the number of rows in the matrix representation) can be millions. As a result, there could be extreme CPU and memory requirements and thus limited scalability potential with this model.

Therefore, HUIM algorithms should construct and operate on trees, instead of matrices. As these trees can become long and deep, it is still crucial to be able to construct and traverse these trees efficiently, otherwise the space and time complexity can grow exponentially. In order to do that, a monotonic measure/property associated with the nodes needs to be utilised. This should enable the exploration of the items lattice, and given a threshold, the pruning of the search space. Most HUIM approaches rely on tree structures and *TWU* properties for pruning.

2.4. Relationship with Frequent Itemset Mining

FIM (also called Frequent Pattern Mining) is a major topic in data mining. FIM started as a phase in the discovery of association rules but has since been generalized, independent of these, to many other patterns, such as frequent sequences, periodic patterns, frequent subgraphs, and bridging rules.

The goal of FIM is the discovery of all the itemsets having support larger than a given threshold, which is called *minimum support*. Depending on whether an itemset has higher or lower support than this threshold, it is called frequent or infrequent itemset respectively. Unlike HUIM, the database in FIM is binary and there are no occurrence quantities of items to be maintained.

FIM utilises the monotonicity of the frequent itemsets, that is, if an itemset is not frequent, then all its supersets are infrequent as well. This *downward closure property* allows the pruning of infrequent itemsets.

Apriori (Agrawal & Srikant, 1994) is based on this property. It adopts a bottom-up, level-wise, candidate set generate-and-test approach, where frequent subsets are extended one item at a time, then groups of candidates are tested against the data. But, in order to achieve that, the database has to be scanned multiple times, proportional to the longest frequent itemset.

ECLAT is another FIM algorithm that uses a vertical data layout (instead of a horizontal one) where a list of TIDs is attached to each item and traverses the structure in either depth-first or breadth-first manner. This allows a very fast support counting but at the cost of too large intermediate TID-lists for memory in some cases (Zaki, 2000).

FP-growth is a well-known depth-first algorithm, which compresses the database using FP-tree structures in main memory. Once these structures are constructed, the algorithm applies a recursive divide-and-conquer approach to extract the frequent itemsets. This approach does not require a candidate generation step, but when the number of items is huge, its performance may not always outperform Apriori.

HUIM is directly related to FIM, but it associates weights (utilities) with the items in transactions - if these weights have unit values it then degenerates to FIM. Thus, most algorithms in HUIM extended FIM algorithms. It is known that HUIM is a more challenging problem than FIM, since the utility of an itemset is neither monotonic nor anti-monotonic, as mentioned above. As a result, FIM techniques cannot be directly applied to HUIM.

The problem of HUIM was defined to find the rare frequent itemsets but with high profits (Yao, Hamilton, & Butz, 2004). The corresponding definitions are already given in Section 2.3. Due to the absence of monotonicity and the downward-closure property, the search space for HUIM can not be directly reduced, as in the case of FIM. Early HUIM studies suffered from the overhead of generating too many candidates, which consumes vast memory and storage, and takes very long running time to compute the utilities of the candidates, especially when databases contain lots of long transactions or a low minimum utility threshold is set. A few newer approaches use novel pruning strategies that eliminate unpromising itemsets. This can avoid repeatedly scanning the tree structures built for HUI when computing the corresponding utilities of these itemsets, thus decrease memory consumption and computation time. We will describe the related work in the next section.

Since the downward closure property does not hold for HUIM, the Two-Phase model (Liu et al., 2005b) was presented to keep the transaction-weighted utilization downward closure property for discovering HUIs. Several tree-based algorithms were also proposed, such as the IHUP-tree-based IHUP algorithm for mining HUIs in incremental databases (Ahmed et al., 2009) and the two efficient UP-tree-based mining algorithms, UP-growth (Tseng et al., 2010) and UP-growth+ (Tseng, Shie, Wu, & Yu, 2013). Liu and Qu (2012) then proposed the HUI-Miner algorithm to build utility-list structures and a set-enumeration tree to directly extract HUIs with neither candidate generation nor an additional database rescan. Later, an improved algorithm, FHM, was proposed to enhance HUI-Miner when analysing the co-occurrences among 2-itemsets (Fournier-Viger et al., 2014). In Section 3, we will discuss these related work in depth.

3. State-of-the-art in HUIM

The concept of HUIM was first proposed by Chan, Yang, and Shen (2003) and the mathematical model was formalised by Yao et al. (2004). Since then, many algorithms in HUIM were proposed, most of them being extensions of either Apriori or FP-tree like structures. HUIM algorithms can be typically classified into two categories: approaches that rely on a candidate generation step, and methods that do not need such a candidate generation step.

Algorithms of the first category, hereafter referred to as ***GroupA***, commonly adopt the Transaction-Weighted-Downward closure model approach to pruning the search space. They usually contain two phases. In phase 1, a set of candidate HUIs is generated by overestimating their utility using the TWU properties. This phase typically uses an Apriori-like approach. Then, in phase 2, a database scan is performed in order to calculate the exact utilities of the candidates and filter out low-utility itemsets. These approaches, not only generate too many candidates in the first phase, but also need to scan the database multiple times in phase 2, which can be computation demanding.

In order to alleviate this overhead, latter algorithms introduced new techniques, starting from HUI-Miner (Liu & Qu, 2012), which builds utility-list structures and develops a set-enumeration tree to directly extract HUIs without both candidate generation and additional database rescans. In the same year, Liu at al. proposed an algorithm named d2HUP (Liu et al., 2012), which enumerates an itemset as a prefix extension of another itemset with powerful pruning and, using the TWU properties, it recursively filters out irrelevant items when growing HUIs with sparse data. In this paper, we categorise them as ***GroupB*** algorithms. These algorithms differ from those in ***GroupA*** in that ***GroupB*** algorithms do not need to generate all the possibilities/combinations (as in the case of ***GroupA***), but only consider itemsets present in the structures (however, they do need to recursively check the sub-itemsets present in the paths of the tree structures).

Latest approaches in HUIM are based on ***GroupB*** algorithms, but add enhanced structures and strategies to further reduce the number of unnecessary utility-lists or decrease their joins. These algorithms were just newly proposed in 2017, so for the purpose of an empirical comparison, we group them into a separate category, ***GroupC***.

In the following, we will describe the corresponding algorithms of these three categories in detail.

3.1. Candidate-generating algorithms (GroupA)

The **Two-Phase** algorithm (Liu et al., 2005b) is a classical and one of the earliest approaches for HUIM. It proposed to discover HUIs in two phases, defined the transaction-weighted utilization (TWU), and proved that TWU maintains the downward closure (anti-monotonicity) property. This algorithm adapts the Apriori algorithm (Agrawal & Srikant, 1994) for the HUIM setting by utilising the anti-monotonicity property of TWU, instead of support, during the first phase. Like Apriori, the Two-Phase algorithm overestimates the utility of itemsets in the first phase using the TWU properties 1, 2, and 3 in Section 2.3. Then, in the second phase, it rescans the database to calculate the exact utility of the candidates and filter out low-utility itemsets. In each database scan, the algorithm generates the candidates for k-element TWU itemsets, from the verified HUIs of length $k-1$. In the last scan, it determines the actual HUIs. Two-Phase suffers from the same problem of the level-wise candidate generation-and-test methodology. Erwin, Gopalan, and Achuthan (2008) proposed an algorithm (CTU-Mine) that is more efficient than the Two-Phase method, but only in dense databases when the minimum utility threshold is very low.

IHUP (Incremental High Utility Pattern mining) (Ahmed et al., 2009) is an algorithm that was originally proposed for incremental and interactive HUIM. It can use previous structures and mining results in order to reduce unnecessary calculations when the database is updated or the minimum utility threshold changes. IHUP uses an FP-tree like structure ($IHUP_{TWU}$-tree) that does not require any restructuring operation, in spite of incremental updat-

ing of the databases. When discovering the HUIs, it operates as FP-growth. In comparison with Two-Phase, IHUP only needs one database scan, but it still needs to recursively traverse the FP-tree structure to calculate the TWUs and identify the HUIs. Prefix-tree-based pattern mining techniques, including Two-Phase and IHUP, have shown that the memory requirement for the prefix trees is low enough. IHUP takes into consideration that, in some cases, such as in incremental and interactive mining, transactions have many items in common. Thus, it utilises the path overlapping (prefix-sharing) method. The proposed structure represents useful information in a very compressed form and saves memory space.

UP-Growth (Tseng et al., 2013; Tseng et al., 2010), is an improved version of IHUP. It uses a compact tree structure, called utility pattern tree (UP-tree), which first uses utility values of single items to order them, then constructs a FP-Tree like prefix tree using this order. For each node in the tree, UP-tree maintains the number of transactions that the itemset corresponding to the path (from the root until the current node) appears, as well as the associated utility for this itemset. Based on this UP-tree structure, UP-Growth proposes two strategies, namely DLU (Discarding local unpromising items) and DLN (Decreasing local node utilities). In the DLU strategy, the low utility items are discarded from the paths during the construction of a local UP-tree. In DLN, the minimum item utilities of descendant nodes are decreased during the construction of a local UP-tree. The authors attest that UP-Growth performs well in databases containing lots of long transactions or when a low minimum utility threshold is specified (Tseng et al., 2013). Further improvements to UP-Growth have been proposed by Yun, Ryang, and Ryu (2014) and Goyal and Dawar (2015).

3.2. Algorithms without a candidate generation step (GroupB)

While pattern growth approaches avoid the level-wise candidate generation-and-test methodology, accumulated TWU values are still maintained in node utilities. As a result, a large number of candidates are generated in the mining process and the performance is degraded (Yun et al., 2014). Therefore, researchers came up with improved methods where the generation of potential HUIs is not required.

HUI-Miner (Liu & Qu, 2012) was the first algorithm to discover HUIs without candidate generation, it outperformed previous algorithms by then. It employs a novel vertical data structure to represent utility information, called *utility-list*. It needs only a single database scan to create and maintain such utility-lists of itemsets containing single items. Then, larger itemsets are obtained by performing join operations of utility-lists of itemsets with smaller lengths. The drawback in HUI-Miner is that it resorts to costly join operations between utility-lists of itemsets and smaller itemsets, which can be very costly. With the introduction of newer HUIM algorithms, HUI-Miner's performance has been surpassed and it is not considered efficient anymore, especially in large datasets.

HUP-Miner (Krishnamoorthy, 2015) is an extension of HUI-Miner. It employs two novel pruning strategies; one based on database partitioning and the other (LA-Prune) based on the concept of lookahead pruning. HUP-Miner takes as input a transactional database with utility information, a *minU* threshold and the number of partitions *K*, which determines how many partitions HUP-Miner uses internally. This parameter has an effect on the running time performance and the memory consumption of the algorithm. The authors suggest a value of 512 for *K* but a typical value could be 10 (Fournier-Viger, 2010). However, the optimal value for *K* should be found empirically for a given dataset. It has been demonstrated through experiments that, HUP-Miner is faster than HUI-Miner. Obviously, a shortcoming of HUP-Miner is that the user needs to explicitly set the number of partitions as an additional parameter.

d2HUP (Liu et al., 2012; 2016) is another algorithm that is able to discover HUIs without maintaining candidates. It proposes a novel data structure, named CAUL (Chain of Accurate Utility Lists), which targets the root cause of candidate generation with the existing approaches. Also, it designs a HUIM approach that enumerates an itemset as a prefix extension of another itemset. It efficiently computes the utility of each enumerated itemset and the upper bound on the utilities of the prefix-extended itemsets. This utility upper bound is much tighter than TWU, and it is further tightened by iteratively filtering out irrelevant items when growing high utility itemsets with sparse data. Moreover, it uses less memory space than tree structures used in the above mentioned algorithms. d2HUP was shown to be more efficient than UP-Growth and Two-Phase. But later algorithms (especially EFIM (Zida et al., 2015), which is presented in the following) have been reported to outperform d2HUP (Zida et al., 2015), the authors in Zida et al. (2015) reported through experiments that EFIM is the most efficient algorithm. *However, as we will present in the Section 5, our findings conflict with this claim in cases where low minimum utility threshold values are specified.*

FHM (Fast High-Utility Miner) algorithm (Fournier-Viger et al., 2014) improves over HUI-Miner, by integrating a novel strategy named EUCP (Estimated Utility Co-occurrence Pruning) that reduces the number of join operations when mining HUIs using the utility-list data structure and avoids the costly joins. It employs a structure named EUCS (Estimated Utility Co-Occurenece Structure), which is built with a single database scan. Then, longer patterns/itemsets are obtained by performing join operations on utility-lists of the shorter patterns. The memory overhead of the EUCS structure is low. FHM algorithm performs up to 95% less join operations than HUI-Miner and was shown to be up to six times faster than the latter, especially for sparse datasets (Fournier-Viger et al., 2014).

EFIM (EFficient high-utility Itemset Mining) (Zida et al., 2015) is an one-phase HUIM algorithm that proposes efficient database projection and transaction merging techniques for reducing the cost of database scans. EFIM efficiently merges transactions that are identical in each projected database through a linear time and space implementation. As larger itemsets are explored, both projection and merging reduce the size of the database. Furthermore, the authors propose a new array-based utility counting technique, named Fast Utility Counting (FUC), which helps calculate the new upper-bounds (i.e., *sub-tree utility* and *local utility*) in linear time and space. In general, EFIM's complexity is roughly linear with the number of itemsets in the search space. In their reported results, EFIM was found to be up to 3 times faster than previous state-of-the-art algorithms, such as FHM, HUI-Miner, d2HUP, UP-Growth, while consuming up to 8 times less memory.

3.3. Newest HUIM algorithms (GroupC)

mHUIMiner (Peng et al., 2017) is a HUI-Miner-based algorithm that incorporates a tree structure to guide the itemset expansion process in order to avoid unnecessary utility-list constructions in HUI-Miner and thus avoid considering itemsets that are nonexistent in the transactional database. In contrast to other techniques, it does not have a complex pruning strategy that requires expensive computation overhead. As a result, the algorithm is found to perform best, regarding running time, outperforming IHUP, FHM, HUI-MINER, and EFIM on sparse datasets, while it maintains a comparable performance on dense datasets.

ULB-Miner (Duong et al., 2017) is a utility-list based algorithm that reduces the memory consumption and speeds up the join operation when creating and maintaining utility-lists by proposing an improved structure called utility-list buffer (ULB). The authors re-

Table 5
Theoretical study and analysis of GroupA algorithms under comparison.

Algorithm	Data structure	Approach/exploration	Pruning and optimisations	Performance
Two-Phase (2005, ***GroupA***)		Based on Apriori. Uses TWDC to overestimate itemsets and, in a second phase, filter them.	Uses TWU.	Inefficient because it generates too many candidates in the first phase.
IHUP (2009, ***GroupA***)	tree: *IHUP-tree*, 3 variants, one based on item's lexicographic order, one designed to the transaction frequency in descending order, while the last is using TWU.	Pattern growth, "build once mine many property for incremental HUIM.		IHUP_TF-Tree is the most memory efficient, IHUP_TWU-Tree is the most efficient regarding running time
UP-Growth (2010, ***GroupA***)	tree: *UP-tree*	Based on FP-Growth.	Uses 4 strategies to prune the search space more efficiently than TWU mining. Discarding global unpromising items (DGU), Discarding global node utilities (DGN), Discarding local unpromising items (DLU), Decreasing local node utilities (DLN).	UP-Growth generates much fewer candidates than FP-Growth since strategies DLU and DLN are applied during the construction of a local UP-tree.

Table 6
Theoretical study and analysis of GroupC algorithms under comparison.

Algorithm	Data structure	Approach/exploration	Pruning and optimisations	Performance
mHUIMiner (2017, ***GroupC***)	tree: similar to ID lists used in Eclat and IHUP-tree, uses 2 structures to store different types of information.	Based on HUI-Miner and IHUP. First creates a global tree to maintain transaction information and initial utility-lists for all the distinct items and a global header table of the tree, which contains TWU values for all the items that are in the global tree. The header tree is sorted in descending order of TWU values. Then discards items with $TWU < minU$ and creates local prefix trees and local header tables for the rest. For mining, it identifies HUIs by checking the sum of iutils, decides whether to expand the current itemset by comparing the sum of all iutils and rutils against the threshold, finally creates the prefix tree and local header table for that itemset, and finally its utility-list.	Does not have a complex pruning strategy that requires expensive computational overhead, which usually does not achieve economies of scale in sparse datasets.	Achieves the best running time on sparse datasets, while maintaining a comparable performance to other state-of-the-art algorithms on dense datasets. In general, the performance of mHUIMiner is similar to that of HUIMiner and FHM. The running time and memory consumption of mHUIMiner do not grow exponentially according to the number of transactions in the synthetic dataset.
ULB-Miner (2017, ***GroupC***)	list: *Utility-List Buffer (ULB)*. It is based on the principle of buffering utility-lists to decrease memory consumption. A ULB is designed like a memory pipeline and consists of multiple segments, which are reused to store utility-list information. It has been proposed in order to tackle the limitations of one-phase utility-list-based HUIM algorithms.	Based on FHM. Follows a strategy for memory reutilisation. The search procedure starts from single items and then recursively explores the search space of item- sets by appending single items, while reducing the search space using two properties: Calculating the utility using the sum of iutil values and Pruning using an utility listâs iutil and rutil values.	Uses a linear time method for constructing utility-list segments in a ULB, then efficiently mines all HUIs using the approach of FHM, i.e. the Estimated Utility Co-occurrence Structure (EUCS), which stores the TWU values of all pairs of items, and the corresponding Estimated Utility Co-occurrence Pruning (EUCP) strategy. This considerably reduces the number of join operations. The Early Abandoning approach is used to avoid completely constructing utility-lists.	Algorithms employing ULB are up to 10 times faster than when using standard utility-lists and consume up to 6 times less memory. When the minUP threshold is set to small values, the difference in terms of number of generated utility-lists becomes clear and large.

port that ULB-Miner is up to 10 times faster than FHM and HUI-Miner and consumes up to 6 times less memory.

In Tables 5, 6, and 7, we present the characteristics of the 10 algorithms under comparison that may influence their performance, including data structure and pruning strategies. It should be noted that here we adopt the extended versions of d2HUP and EFIM, i.e. (Liu, Wang, & Fung, 2016) and (Zida, Fournier-Viger, Lin, Wu, & Tseng, 2017).

3.4. Other HUIM algorithms

Besides the algorithms discussed above, there has been more research in HUIM, with an increasing number of newly proposed data structures and concepts, and new algorithms.

UP-Hist (Goyal & Dawar, 2015) is a HUIM algorithm that reduces the number of candidates and runs faster in both dense and sparse DBs, while scaling similarly with regards to database size.

Table 7
Theoretical study and analysis of GroupB algorithms under comparison.

Algorithm	Data structure	Approach/exploration	Pruning and optimisations	Performance
HUI-Miner (2012, ***GroupB***)	list: *utility-list* (vertical data structure)	Avoids costly generation and utility computation of numerous candidate itemsets which requires multiple DB scans. The size of utility-lists is constant, no matter what order items are sorted in.	Heuristic information based on accumulating internal and external/remaining utilities in the utility list.	Inefficient join operations and is also not scalable. Underperforms in sparse transaction databases because it does not have an efficient pruning strategy. Exhaustive search has to check $2^{\|I\|}$ itemsets. Less efficient than UP-Growth when mining large DBs.
FHM (2014, ***GroupB***)	list: *Estimated Utility Co-Occurrence Structure* (EUCS). Implemented as a hashmap of hashmaps.	Based on HUI-Miner. FHM improves HUI-Miner by pre-computing the TWUs of pairs of items to reduce the number of join operations.	EUCP (Estimated Utility Co-occurrence Pruning) pruning approach is based on item co-occurrence. It directly eliminates low-utility extensions of an itemset and all its transitive extensions without constructing their utility-list. Candidate pruning can be very effective by pruning up to 95% (i.e. fir Chainstore) of candidates in real datasets.	95% less join operations than HUI-Miner and is up to six times faster than HUI-Miner in real datasets. Constructing EUCS is fast and occupies a small amount of memory, bounded by $\|I^*\|^2$, where $\|I^*\|$ is the number of HUIs.
HUP-Miner (2015, ***GroupB***)	list: *Partitioned Utility List* (PUL)	Based on HUI-Miner. Improves it by a partition strategy.	Employs 4 pruning strategies, TWU prune, used in the initial mining process, and U-Prune, PU-Prune and LA-Prune during tree/search space exploration to limit the search space.	Improvement vs. HUI-Miner is within a factor of 2 to 6.
d2HUP (2016, ***GroupB***)	tree + hyper-linked list (CAUL, Chain of Accurate Utility Lists)	Uses a reverse set enumeration tree in a depth-first manner for pattern growth and a linear/hyperlinked data structure, CAUL (Chain of Accurate Utility Lists) that maintains the original utility information for each enumerated itemset in order to compute the utility and to estimate tight utility upper bounds of its prefix extensions efficiently.	Utility Upper Bounding. Uses a look-ahead strategy that is based on a closure property and a singleton property, which enhances the efficiency in dealing with dense data. Controlled irrelevant item filtering, i.e. iteratively eliminating the irrelevant items from the utility computation process, using the maximum number of filtering rounds as a parameter. Efficient computation by pseudo projection. Partial materialisation threshold for space-time tradeoff, i.e. between pseudo projection representation and irrelevant item filtering.	HUP-Miner improvement over HUI-Miner is within a factor of 2 to 6, while d2HUP is up to 45 times faster than HUIMiner on the same databases.
EFIM (2017, ***GroupB***)	array: *utility-bin* and tree	All items in the database are renamed as consecutive integers. Then, in a utility-bin array U, the utility-bin U[i] for an item i is stored in the i-th position of the array. This allows to access the utility-bin of an item in O(1) time. Reuses the same utility-bin array multiple times by reinitialising it with zero values before each use. This avoids creating multiple arrays and thus greatly reduces memory usage. Only 3 utility-bin arrays are created to calculate the TWU, sub-tree utility and local utility. When reinitialising a utility-bin array to calculate the sub-tree utility or the local utility of single-item extensions of an itemset, only utility-bins corresponding to items in the extended set of a given itemset are reset to 0, for faster reinitialisation of the utility-bin array.	Uses 2 tight upper bounds, Sub-tree Utility upper bound and Local Utility Upper-Bound, both implemented in linear time. High-utility Database Projection (HDP) prunes the candidate itemsets which are not an alpha-extension of the current itemsets, while High-utility Transaction Merging (HTM) finds identical transactions and combines their utilities, which leads to a compact tree structure.	Memory usage of EFIM is very low compared to other algorithms. The performance advantage of EFIM shrinks on sparse datasets. EFIM is much more memory efficient than the other algorithms over datasets chess, mushroom and retail. This is because while other algorithms rely on complex tree or utility-list structures to maintain information, while EFIM generates projected databases that are often very small in size due to transaction merging.

Since there is no open source implementation of this algorithm, we do not include in the empirical comparisons.

MIP (Mining high utility Itemset using PUN-Lists) (Deng, Ma, & Liu, 2015) introduces a novel data structure, named PUN-list, which maintains both the utility and utility upper bound information of an itemset by summarising this information at different nodes of a prefix utility tree (PU-tree) structure. The algorithm employs an efficient mining strategy, either based on item-support descending order or TWU descending order, in order to construct the PU-tree. Then, it directly discovers high utility itemsets from the search space using the pruning information stored in PUN-lists.

HUIM-GA and HUIM-GA-tree (Kannimuthu & Premalatha, 2013) are algorithms that introduce a novel evolutionary approach, called high utility pattern extraction using genetic algorithm with ranked mutation using minimum utility threshold, in the encoding process. In this approach, optimal HUIs are mined without specifying a *minU* threshold.

HUIM-BPSO (Lin, Yang, Fournier-Viger, Wu et al., 2016) is an algorithm based on binary particle swarm optimization (BPSO) algorithm, where the sigmoid function is adopted in the updating process of the particles to discretely encode the particles as binary variables and integrated with the TWU model to find HUIs. It first sets the number of discovered 1-itemset PHUIs as the particle size then uses TWU to prune unpromising items. This approach can reduce the combinational problem in traditional HUIM especially when the number of distinct items in the database or the size of database is very large. HUIM-BPSO-tree (Lin, Yang, Fournier-Viger, Hong, & Voznak, 2016) is an extension of HUIM-BPSO that exploits an OR/NOR-tree structure to avoid combinations. It should be noted that both algorithms are heuristic and not complete, i.e. they only discover a portion of the actual HUIs.

3.5. Other research directions in HUIM

Top-K HUIs. In HUIM, setting an appropriate minimum utility threshold is a difficult task, so the authors in Wu, Shie, Tseng, and Yu (2012) and Tseng, Wu, Fournier-Viger, and Philip (2016) propose the Top-k Utility itemsets mining algorithm (TKU). TKU first computes the utility for each pair of items (2-itemset), then ranks them by their utilities. The utility of the *k*th pair is first chosen as the initial threshold. Then, it gradually builds the utility pattern tree (UP-tree) and raises the threshold to the *k*th highest node utility in the current tree. Such strategies can reduce the search space and the number of candidates.

Sequential HUIs. In Yin, Zheng, Cao, Song, and Wei (2013) and Wang, Huang, and Chen (2016), the authors propose algorithms to extract the Top-k high utility sequential patterns in a sequence database. Different from HUIM, in high utility sequential pattern mining, when calculating the utility of an itemset, only transactions that have the same order of items as a given itemset will be taken into account.

In Alkan and Karagoz (2015), the authors propose a tighter bound than TWU, named CRoM, to estimate the utility of a candidate HUI, and an efficient algorithm (HuspExt) which is based on CRoM and efficient structures. HuspExt has been reported to have lower execution time and memory consumption than USpan (Ahmed, Tanbeer, & Jeong, 2010).

Closed HUIs. A closed high-utility itemset (CHUI) is a HUI having no proper supersets that are HUIs. CHUI is lossless since it allows deriving all HUIs; it is concise since it only discovers the largest HUIs that are common to groups of customers. In Fournier-Viger, Zida, Lin, Wu, and Tseng (2016), based on EFIM, the authors propose EFIM-closed algorithm, which is an efficient solution for CHUI. It designs checking mechanisms that can determine whether a HUI is closed without having to compare a new pattern with previously found patterns.

Discussions. There are still a few issues in HUIM that have seldom been addressed. One of the issues is how to efficiently find HUIs of a specific length (or smaller than a specific length)? Another issue is that the current definition of HUI only considers itemsets that co-occur in the transactions but with high sum profits; however, the occurrence probability of the itemsets in the transaction database has not been simultaneously considered. It will be interesting to combine both the utility and the frequency of the itemsets to extract itemsets that are both high-utility and frequent.

3.6. Time complexity of HUIM

Let $|D|$, *avgTransLen*, $|I|$ represent the number of transactions in the dataset, the average transaction length, and the number of distinct items, respectively. Unlike FIM, the downward-closure property does not hold in HUIM. Fortunately, with the *TWU* property and other tighter bounds, HUIM algorithms can adopt similar pruning strategies like FIM when extracting HUIs, but the number of candidates will be much greater than the real HUIs. Given a user-specified *minUP*, the time complexity of HUIM algorithms is approximately $O(|D|*avgTransLen*|M|)$, where $|M|$ is the complexity brought by the number of distinct items and the sparsity of the dataset, in the worst cases, $|M| = 2^{|I|}$. For very small and dense datasets, $|M|$ will also be small; on the other hand, if the data is very large and spare, $|M|$ will be greater. Therefore, the worst case time complexity of HUIM algorithms is $O(|D|*avgTransLen*2^{|I|})$, which means that, if the dataset is large enough and covers all the possible combinations of the items (itemsets), to derive HUIs, the complexity will be as large as $O(|D|*avgTransLen*2^{|I|})$. Since the smaller the value of *minUP*, the larger number of candidate HUIs to be derived, thus more computing time is needed. Therefore, the overall time complexity of HUIM algorithms is approximately $O(|D|*avgTransLen*|M|/minUP)$, and the worst case time complexity is $O(|D|*avgTransLen*2^{|I|}/minUP)$.

For EFIM, its complexity also depends on $|D|$, *avgTransLen*, $|I|$, the density of the dataset, and *minUP*. When the dataset is dense and the number of distinct items is small, the time complexity of EFIM will also be small. But when both $|I|$ and $|D|$ are large, while the dataset is sparse, EFIM can be extremely slow, as will be observed in Section 5.2.4.

3.7. Experimental results reported in the literature

When making comparisons in HUIM, it is essential to consider both synthetic and real datasets in order to make a comprehensive performance evaluation. Existing methods typically focus on real datasets and only a few of them use synthetic data.

The issue is that only a few real datasets, which contain both internal utilities (occurrence quantities) and external utilities (unit profits), are available in the domain. In most cases, many authors use FIM datasets by grafting real transactional data with synthetically generated utility values for the items. These utility values are typically generated using a specific distribution (e.g. Log-normal) that is more appropriate for a specific application.

Table 8 summarises the results of 10 state-of-the-art HUIM algorithms, as reported in the cited papers. *It should be noted that there is no comprehensive study that covers all the algorithms in the literature. The minU parameter, which greatly influences the performance of the algorithms, is inconsistently used in different studies, making it very hard to reliably assess their performance.*

Table 8
Summary of experimental comparisons in the literature.

Related work	Algorithms	Real datasets [a]	Synthetic datasets [a]	Conclusions
Ahmed et al. (2009)	FUM, IHUPL, IHUPTF, **IHUPTWU**	Chainstore (0.15–0.35), Kosarak (0.2–1.0), Mushroom (10–30), Retail (0.6–1.4)	T10I6D1M (0.001–0.500), T20I6D1M (0.001–100)	Different versions based on IHUP. IHUPTWU is fastest, IHUPTF requires the least memory, IHUPL is simplest to construct.
Tseng et al. (2010)	IHUP+FPG, **UP-Growth+FPG, UP-Growth-UPG**	valign=tBMS (2.9–4.0), Chess (30–100)	T10I6D100K (0.2–1.0)	Scalability tests on synthetic DBs (dbTrans = D = 200k–1000k). No memory reports. UP-Growth performs faster and scales better than IHUP.
Liu and Qu (2012)	**HUI-Miner**, IHUPTWU, UP-Growth, UP-Growth+	Accidents (15–40), Chainstore (0.004–0.009), Chess (18–28), Kosarak (1.0–3.5), Mushroom (2.0–4.5), Retail (0.020-0.045)	T10I4D100K (0.005–0.030), T40I10D100K (0.35–0.60)	First algorithm without candidate generation. Much faster than other algorithms for dense and sparse DBs, uses less memory. UP-Growth+ generates less candidates than UP-Growth but needs more memory. Processing order of items influences performance (ascending TWU is best).
Fournier-Viger et al. (2014)	**FHM**, HUIMiner	BMS (2.08–2.09), Chainstore (0.04–0.09), Kosarak (0.85–1.41), Retail (0.01–0.20)	–	FHM pruning strategy reduces the search space by up to 95% and it is up to 6 times faster than HUIMiner.
Zida et al. (2015), Zida et al. (2017)	d2HUP, **EFIM**, FHM, HUI-Miner, HUP-Miner, UP-Growth+	Accidents (8.92–14.02), BMS (2.06–2.10), Chainstore (0.077–0.153),Chess (16.22–27.82), Connect (25.49–31.37), Foodmart (0–0.02), Kosarak (0.780–1.064), Mushroom (2.34–2.92), Pumsb (18.210–18.812)	–	EFIM is in general two to three orders of magnitude faster and consumes up to eight times less memory than the rest. Scalability tests were run with regard to dataset size by varying the number of transactions of the real datasets. EFIM has excellent scalability on both sparse and dense datasets.
Krishnamoorthy (2015)	HUI-Miner, **HUP-Miner**	Chainstore (0–0.20), Chess (18–30), Kosarak (0.5–3.5), Mushroom (0–5), Retail (0.02–0.10)	T10I4D100K (0.01–0.05), T20I6D100K (0–0.30), T40I10D100K (0.4–0.6)	Using PU-Prune and LA-Prune strategies, HUP-Miner is found to be better than HUI-Miner.
Goyal and Dawar (2015)	UP-Growth, UP-Growth+, **UP-Hist Growth**	Accidents (scalability), Chess (40–70), Mushroom (20–80), Foodmart (20–70)	–	UP-Hist reduces the number of candidates and runs faster in both dense and sparse DBs, while scaling similarly with regards to DB size. No memory tests.
Deng et al. (2015)	d2HUP, HUI-Miner, **MIP_sup, MIP_twu**, UP-Growth+	Accidents (10–15), Chess (14–15), Chainstore (0.005–0.010), Connect (25,30), Mushroom (1–6), Pumsb (19–20)	T10I4D100K (0.002–0.012), T40I10D100K (5–10)	Both MIP algorithms are the fastest, followed by HUI-Miner. d2HUP is third, while UP-Growth+ is the slowest. In dense datasets MIP consumes less memory than HUI0Miner and has better scalability with varying the number of transactions.
Liu et al. (2016)	**d2HUP**, HUIMiner, IHUP, UP-Growth+	BMS (2–4), Chess (10–70), Chainstore (0.005–0.500), Foodmart (0.01–0.10)	T10I6D1M (0.001–0.500), T20I6D1M (0.001–100)	d2HUP enumerates less patterns than HUIMiner, UPUPG, IHUPTWU, and Two-Phase by up to 4 orders of magnitude. Pseudo CAUL transaction representation needs less memory than UP-tree and Utility-Lists in large DBs.
Peng et al. (2017)	EFIM, FHM, IHUP, HUIMiner, **mHUIMiner**	Accidents (10–40), BMS (2.10–2.70), Chainstore (0.02–0.14), Chess (18–30), Foodmart (0.01–0.07), Kosarak (1–4), Mushroom (2–5), Retail (0.02–0.05)	D = 100K, T = 10.5 minUP = 0.004, d = 0.005–5% D = 100K, T = 10.5, d = 0.005%, minUP = 0.0035–0.0050 D = 100K–500K, I = 20K, minUP = 0.005	3 scalability tests on synthetic DBs. mHUIMiner is best on sparse but slowest on dense DBs. EFIM is the most memory efficient.
Duong et al. (2017)	FHM, HUIMiner, **ULB-Miner**	Chainstore (0.01–0.10), Chess (16–30), Connect (30–70), Foodmart (0.01–0.10), Kosarak (1–4), Retail (0.01-0.10)	D = 100K–500K, T = 10, I = 2K–10K, minUP = 0.05%	3 scalability tests on synthetic DBs. ULB-Miner is up to 10 times faster than when using standard utility-lists and consume up to 6 times less memory.

[a] minUP ranges used are stated in the parentheses next to each dataset. minU values were converted to minUP for works that report only the former. [b] T10I6D1M denotes a database with D = dbTrans = 106 transactions, I = dbItems = 6 items, and T = avgTransLen = 10 average transaction length, d = density.

For synthetic datasets, which are commonly generated using IBM Quest Synthetic Data Generator,[1] the number of transactions in the database and the average or maximum length of transaction need to be specified. *However, these parameters are not picked consistently, varying in different papers.*

According to the comparisons given in Table 8, we observe that EFIM is reported to be the best performer, whereas early HUIM algorithms such as Two-Phase and UP-Growth are the least efficient ones, thus they are impractical for the discovery of HUIs.

In this work, we evaluate the performance of the established HUIM methods in the real datasets commonly used in the literature. More importantly, we also compare their performance in an extensive range of synthetic datasets, which are generated using a diverse coverage of *minU*, number of distinct items, average transaction length, and number of transactions parameters. The goal is to comprehensively and extensively benchmark the state-of-the-art algorithms in HUIM, to offer reliable guidelines for practitioners who need to choose the best HUIM algorithm for their specific applications. The conclusions to be reached in our work, point out that EFIM is the most efficient algorithm in very dense real datasets, in terms of running time, while d2HUP is the best in sparse, large datasets, also including cases where very low minimum utility thresholds are specified.

4. Experiments design

In this section, we present the design of the experiments for performance evaluation of the 10 major algorithms discussed in Section 3. Based on the performance reported in the literature and the chronological order these HUIM algorithms were introduced, we divide them into three groups: ***GroupA***, which includes IHUP, Two-Phase, and UP-Growth, and ***GroupB***, having d2HUP, EFIM, FHM, HUI-Miner, and HUP-Miner. ***GroupA+B*** includes all 8 algorithms. ***GroupC*** includes mHUIMiner and ULB-Miner, proposed in 2017. We will first describe both the real and synthetic datasets used in the experiments, where we also depict the related characteristics of these datasets, including the *minU*, the average length of the transactions, the density of the datasets, etc. This can give significant insight in the experimental outcome when comparing different algorithms. Next, we introduce the evaluation measures for comparing different HUIM algorithms. We then present the test procedure of different algorithms, varying the important parameters in the experiments. Finally, we give information about the hardware specifications and software environment for the experiments.

4.1. Important issues and parameters in the experiments

In some applications, the consideration of the dataset density is very important, since this can help distinguish the variance in the performance of algorithms in datasets with different data characteristics. However, most existing methods, when comparing with other algorithms, have not examined thoroughly the density factor. This parameter is generally known to the practitioners in advance, or can be estimated, depending on the application. It can thus be easily included when choosing the most appropriate HUIM algorithms.

Besides, it is also crucial to consider two other parameters: *maxTransLen*, which refers to the maximum transaction length, and *dbUtil*, which is the total utility of the database, as defined in Section 2. These have been found to directly influence the running time performance and memory requirements. The longer the transactions are or the larger the database size is, the more time is needed by a HUIM algorithm to discover HUIs. *dbUtil* is directly related to the file size of the dataset.

4.2. Real datasets

In order to evaluate and compare the performance of HUIM algorithms in real-world scenarios but also validate our results against those reported in the literature, we include in our tests some commonly used real datasets for HUIM. It must be mentioned that in this work, as in the literature, these datasets have been anonymised and standardised with real item ids (IIDs), real transactions and items occurrences in them, but use synthetically generated utility values. This is typical, as many of these datasets were first derived for research in FIM, where utilities are binary.

To be consistent with the testing methodologies used in the literature, we use real datasets derived from the UCI Machine Learning Repository (Lichman, 2013) and the Frequent Itemset Mining Dataset Repository (Goethals & Zaki, 2003). The list of the real datasets used in this study, alongside their characteristics, is shown in Table 9.

The *Accidents* dataset contains 340,183 (anonymised) lines of traffic accidents data (Geurts, Wets, Brijs, & Vanhoof, 2003). It is moderately dense (7.22%) and has long transactions, having 33.8 items on average. The traffic accidents data is highly skewed. For example, 80% of the accidents in the dataset occurred under normal weather conditions.

BMS (*WebView-1* version) is a real-life dataset of 59,602 sequences of click-stream data from an e-commerce system, with a mix of short and long transactions (Zheng, Kohavi, & Mason, 2001). It contains 497 distinct items. The average length of sequences is 2.51 items. Thus, this dataset is moderately sparse. In this dataset, there are some long sequences. For example, 318 sequences contain more than 20 items. The external utility values have been generated using a Gaussian distribution.

Chess is a small but very dense (50% density) dataset, containing lots of long transactions. The distribution of the lengths of the transactions is degenerate, i.e., every transaction has a length of 37. In *Chess*, the number of distinct items is limited (only 75), compared to other datasets.

Connect is based on the publicly available UCI *connect-4* database (Lichman, 2013). Every transaction contains all legal positions in the game of *connect-4* in which neither player has won yet, and the next move is not forced. It is dense (33.3%) and relatively long (43 items in 67,557 transactions, with *avgTransLen* being 43). As in *Chess*, the distribution of the lengths of the transactions in *Connect* is degenerate.

Foodmart is a quantitative database of products sold by an anonymous chain store. It is a pure real (contains unit profits) and sparse dataset acquired from Microsoft's *Foodmart 2000* sample database. It contains 1559 items and 4141 transactions, the max length is 14, whereas the average length is only 4.4. The density is 0.28, thus this dataset is very sparse. This is very typical in online-shopping scenarios.

The dataset *Mushroom* includes descriptions of hypothetical samples corresponding to 23 species of mushrooms (Goethals & Zaki, 2003). As 19.3% of distinct items are present in every transaction, this dataset is dense and also has long high utility patterns.

Chainstore is a real-life dataset using real utility values. It derives from a major chain in California and contains 1,112,949 of transactions and 46,086 distinct items. It is the sparsest of the 9 real datasets used in the experiments.

The *Kosarak* dataset contains anonymised click-stream data of a Hungarian online news portal. This dataset is large (almost 1 million transactions) and sparse.

[1] IBM Quest Data Mining Project. 1996. Quest Synthetic Data Generation Code. Source: http://www.philippe-fournier-viger.com/spmf/datasets/IBM_Quest_data_generator.zip (Last access 2016-10-30).

Table 9
Real datasets utilised for evaluating different HUIM algorithms.

db	dbTrans	dbItems	maxTransLen	avgTransLen	Density (%)
Accidents	340,183	468	51	33.81	7.22
BMS	59,602	497	267	2.51	0.51
Chainstore	1,112,949	46,086	170	7.23	0.02
Chess	3196	75	37	37.00	49.33
Connect	67,557	129	43	43.00	33.33
Foodmart	4141	1559	14	4.42	0.28
Kosarak	990,002	41,720	2498	8.10	0.02
Mushroom	8124	119	23	23.00	19.33
Retail	88,162	16,470	76	10.31	0.06

Retail contains the anonymised retail market basket data from an anonymous Belgian retail store (Cavique, 2007).

In this work, we used the succinct versions of the aforementioned datasets, available at the repository of the open HUIM software package SPMF (Fournier-Viger, 2010). The internal utility values have been generated using a uniform distribution in the range [1,10], while the external utilities have been generated using a half-normal distribution.

In our experiments, the minimum utility thresholds for the real datasets are selected in the same way as in the literature, in specific with (Zida et al., 2015), so that the results can be verified in a consistent way.

4.3. Synthetic datasets

In addition, we evaluate the running time and maximum memory consumption of all algorithms using synthetic datasets, where all the parameters were hand picked. Synthetic datasets are crucial in determining the performance properties and scalability of HUIM algorithms, since they allow better evaluation/comparison at different parameters, and find hidden properties of the algorithms.

The synthetic datasets were generated using the transaction database generator implemented in SPMF (Fournier-Viger, 2010). The internal utility values have been generated using a uniform distribution in the range [1, 10], while the external utilities have been generated using a half-normal distribution. SPMF also uses a random generator to determine the transaction length for each transaction, sampling from a uniform distribution. As such, the average transaction length in these datasets is approximately half the maximum transaction length.

In reality, the uniform distribution arises in a very limited number of applications, so further analysis would be needed to consider the effects of using various non-uniform distributions, such as the Normal, Log-normal, Exponential, Gamma, Weibull and others.

Table 10 illustrates the different parameter values used for the experiments on the 27 synthetic datasets, named SetD27.

In this work, we notice that, in most cases of SetD27, the three algorithms of ***GroupA*** that need candidate generation (i.e., Two-Phase, IHUP, UP-Growth) are orders of magnitude slower than the rest, they do not terminate in reasonable time, or they exceed memory resources. Thus, we exclude them from the full-set large-scale experiments of SetD27, but we do check their performance in 5 synthetic datasets (d01, d04, d05, d22, d26), in order to attest their inferior performance in efficiency. For the 5 remaining algorithms, i.e., ***GroupB*** (d2HUP, EFIM, FHM, HUI-Miner, HUP-Miner), which do not need to generate candidates, the total number of tests is $27(datasets) \times 5(minU) \times 5(algorithms) = 675$. In each test, we measure the total running time and the maximum memory needed by each algorithm.

It is worth noting that, when generating synthetic datasets, we pick a set of parameters so that they are more appropriate for the real-world cases.

Table 10
Characteristics of synthetic datasets utilised for evaluating different HUIM algorithms (SetD27).

db	dbTrans	dbItems	maxTransLen	Density (%)
d01	5000	100	10	5.40
d02	5000	100	20	10.57
d03	5000	100	30	15.51
d04	5000	500	20	2.11
d05	5000	500	40	4.08
d06	5000	500	60	6.08
d07	5000	1000	30	1.57
d08	5000	1000	60	3.06
d09	5000	1000	90	4.56
d10	10,000	100	10	5.51
d11	10,000	100	20	10.55
d12	10,000	100	30	15.46
d13	10,000	500	20	2.09
d14	10,000	500	40	4.08
d15	10,000	500	60	6.16
d16	10,000	1000	30	1.55
d17	10,000	1000	60	3.06
d18	10,000	1000	90	4.54
d19	20,000	100	10	5.50
d20	20,000	100	20	10.53
d21	20,000	100	30	15.46
d22	20,000	500	20	2.11
d23	20,000	500	40	4.10
d24	20,000	500	60	6.11
d25	20,000	1000	30	1.55
d26	20,000	1000	60	3.07
d27	20,000	1000	90	4.60

It should also be mentioned that, while some authors use raw values of minimum utility thresholds (*minU*), others use minimum utility percentage thresholds (*minUP*), as defined in Section 2.3. In this work, we use the latter. It is clear that, in the case of a retail shop, this value corresponds to the portion of HUIs to the total utility/profit of the database of the shop. The larger the threshold, the less HUIs will be found.

The *minUP* values used for our experiments on SetD27 range from 0.06% to 0.14% with a scale of 0.02%. Thus, we have 5 different *minU*/*minUP* values. The parameters used for the generation of the synthetic datasets of SetD27 (*dbTrans, dbItems, maxTransLen*) were chosen to cover different cases in various retail data analysis scenarios, in order to comprehensively compare the performance of the algorithms and discover the hidden patterns of these algorithms in the specified parameter space. In Table 10, the synthetic datasets are named in increasing order of *dbTrans*, then *dbItems*, and then *maxTransLen*, i.e., d27 has the most transactions, distinct items, and maximum/average transaction lengths in all the datasets in SetD27, while d01 is the smallest one. The file size of the dataset is directly (and equally, for the case of datasets with symmetrically distributed transaction lengths) dependent on *dbTrans* and *maxTransLen*, while *dbItems* only influences the density of the dataset.

Table 11
Characteristics of synthetic datasets utilised for evaluating different HUIM algorithms (SetDH45). *minU*1 is the highest minimum utility threshold where at least one HUI is identified. *minUP*1 is the percentage value that corresponds to *minU*1.

db	dbTrans	dbItems	maxTransLen	Density (%)	minU1	minUP1
dh0110000	10,000	10	0.055	222	0.0541	
dh02	10,000	10,000	50	0.254	735	0.0386
dh03	10,000	10,000	100	0.506	1640	0.0431
dh04	10,000	50,000	10	0.011	131	0.0318
dh05	10,000	50,000	50	0.051	488	0.0257
dh06	10,000	50,000	100	0.101	913	0.0240
dh07	10,000	100,000	10	0.006	136	0.0324
dh08	10,000	100,000	50	0.025	510	0.0268
dh09	10,000	100,000	100	0.051	919	0.0241
dh10	10,000	500,000	10	0.001	154	0.0370
dh11	10,000	500,000	50	0.005	495	0.0260
dh12	10,000	500,000	100	0.01	947	0.0251
dh13	10,000	1,000,000	10	0.001	135	0.0326
dh14	10,000	1,000,000	50	0.003	500	0.0263
dh15	10,000	1,000,000	100	0.005	882	0.0234
dh16	100,000	10,000	10	0.055	1608	0.0389
dh17	100,000	10,000	50	0.255	6610	0.0344
dh18	100,000	10,000	100	0.503	12084	0.0318
dh19	100,000	50,000	10	0.011	485	0.0117
dh20	100,000	50,000	50	0.051	1608	0.0084
dh21	100,000	50,000	100	0.101	2972	0.0079
dh22	100,000	100,000	10	0.006	348	0.0084
dh23	100,000	100,000	50	0.025	936	0.0049
dh24	100,000	100,000	100	0.05	1640	0.0044
dh25	100,000	500,000	10	0.001	169	0.0041
dh26	100,000	500,000	50	0.005	513	0.0027
dh27	100,000	500,000	100	0.01	953	0.0025
dh28	100,000	1,000,000	10	0.001	154	0.0037
dh29	100,000	1,000,000	50	0.003	540	0.0028
dh30	100,000	1,000,000	100	0.005	953	0.0025
dh31	1,000,000	10,000	10	0.055	14690	0.0354
dh32	1,000,000	10,000	50	0.255	59576	0.0311
dh33	1,000,000	10,000	100	0.505	140600	0.0372
dh34	1,000,000	50,000	10	0.011	3295	0.0080
dh35	1,000,000	50,000	50	0.051	13820	0.0072
dh36	1,000,000	50,000	100	0.101	28630	0.0075
dh37	1,000,000	100,000	10	0.006	2015	0.0049
dh38	1,000,000	100,000	50	0.026	6950	0.0036
dh39	1,000,000	100,000	100	0.051	14710	0.0039
dh40	1,000,000	50,0000	10	0.001	584	0.0014
dh41	1,000,000	50,0000	50	0.005	1845	0.0010
dh42	1,000,000	50,0000	100	0.01	3680	0.0010
dh43	1,000,000	1,000,000	10	0.001	425	0.0010
dh44	1,000,000	1,000,000	50	0.003	1165	0.0006
dh45	1,000,000	1,000,000	100	0.005	1824	0.0005

To further evaluate the top-5 algorithms in ***GroupB*** in more adverse cases (regardless of whether HUIs are extracted or not) and check their scalability regarding *minUP* and $|D|$, $|I|$, and *maxTransLen* parameters, we generate a second set of synthetic datasets, named *SetDH*45, where the dataset characteristics vary as follows: *dbTrans* ranges in 3 steps from 1000 transactions up to 1,000,000, *dbItems* from 10,000 to 1,000,000 scaling in 5 steps, while for *maxTransLen* we use the values 10, 50, and 100. This results to $3 \times 5 \times 3 = 45$ datasets. The characteristics of SetDH45 are summarised in Table 11. All datasets in this set are sparse, ranging from 0.0005% to 0.5%. Here, we use a *minUP* value ranging from 0.10% to 0.50% to compare their scalability in a consistent manner. With this *minUP* range, there may not always be HUIs found.

Since d2HUP and EFIM will be demonstrated to be the top-2 performers in running time, we conduct deep comparisons between them in SetDH45 datasets, where we pick a minimum utility threshold value (*minU*) which guarantees that at least one HUI is generated. This usually corresponds to very small *minU* values, especially for large datasets. As will be seen in Section 5.2.4, EFIM needs considerably more running time than d2HUP to accomplish the HUIM task with these *minU* values. If more HUIs need to be identified, the running time can grow exponentially.

Last, we evaluate ***GroupC*** algorithms against d2HUP and EFIM, on all the real datasets and SetD27 synthetic datasets, as well as a few selected datasets from SetDH45.

4.4. Experimental setup

We performed all the experiments on a system with a single Intel i7-5960X CPU (Haswell microarchitecture), originally at 3.0 GHz but overclocked at 4.4GHz, and 128GB DDR-4 RAM at 2400MHz, running Linux Ubuntu 16.04. A low latency Solid State Drive (SSD) was used for storage. For the implementations of the algorithms, we used the SPMF (Sequential Pattern Mining Framework) open-source data mining library at version 2.19 (Fournier-Viger, Lin et al., 2016), on 64bit Java Running Environment (JRE) at build 1.8.0_60-b27. SPMF is a cross-platform and open-source data mining library written in Java, specialised in FIM and HUIM algorithms, and has tools to generate utility transaction databases. This allows comparability and reproducibility of the experimental results. We accept that the HUIM algorithm implementations in SPMF are correct and optimal, since SPMF has been deployed in many research works.

In Java, it is not easy or possible for the programmer to deterministically control memory allocation, since garbage collection

is an integral part of the language. There are many runtime parameters and options for the VM that can affect the performance of a Java application. These are often not well documented, they depend on the hosting Operating System and the hardware platform, and change from one Java Development Kit (JDK) version to another. The java option *-Xmx128g* was selected in all the experiments. This specifies the maximum heap size of the memory allocation pool for the JVM; in this case it is 128GB.

We measured the execution time for each algorithm in every test as provided by SPMF, which simply uses *System.currentTimeMillis* at two time instances, i.e., at the start and at the end of a test, then subtract the values. In order to limit the factor of random access disk activity, in case of datasets with large file size or tests that only need a few milliseconds to finish, we ran the tests sequentially.

We also collected peak memory usage/consumption statistics using the provided memory logger by SPMF, which uses the *totalMemory*() and *freeMemory*() methods of the Java *Runtime.getRuntime()* instance. In our experiments, SPMF's memory logger thread did not consume more than 1% of CPU cycles. In order to limit the factor of undesired automatic garbage collection processes that could impact the measurements of tests that require only a few milliseconds, we used a timeout of 3 s between consecutive tests.

Instead of doing worst-case execution time and memory consumption analysis, we used robust statistics to minimise the effect of outliers. In order to collect robust measurements, we repeat each experiment 7 times and use the Median Absolute Deviation (MAD) method, separately for running time and maximum memory. MAD is defined as the median of the absolute deviations from the data's median and is a robust statistic of measuring the variability of univariate samples of quantitative data.

Last, we counted the number of candidates generated by every algorithm. For HUP-Miner, we used the typical value of $K = 10$ for the number of partitions. All other algorithms were used with the proposed optimizations by the authors, which is done by default in SPMF.

5. Results and analysis

In this section we present and analyse the results from the comparative experiments. We first show and examine the running time and memory consumption results for the real datasets, where we compare the time and memory-consumption efficiency. Then we investigate the results in the synthetic datasets.

5.1. Performance analysis on real datasets

Here, we compare the performance of ***GroupB*** algorithms on 9 real datasets that are commonly used in HUIM research. These include dense (*Accidents, Chess, Connect, Mushroom*) and sparse (*BMS, Chainstore, Foodmart, Retail*) real datasets, as summarised in Table 9.

Fig. 1 illustrates the comparison of the algorithms' running time at different minimum utility percentage (*minUP*) thresholds. The findings are consistent with those reported in Zida et al. (2015), where EFIM is reported to be significantly faster than the others at these *minUP* levels.

However, although EFIM is considerably faster in dense datasets such as *Accidents, BMS, Chess, Connect*, and *Mushroom*, it is the slowest algorithm in sparse datasets, i.e., *Chainstore, Kosarak*, and *Retail*, and it is the second slowest (after HUI-Miner) in *Foodmart*. These datasets have similar sparseness characteristic, as shown in Table 9, and they are associated with retail shopping scenarios, where the inventory of a shop is typically much larger than the average number of distinct items purchased in a transaction. The performance hit on sparse real data sets has also been reported in Zida et al. (2017) and Peng et al. (2017). One of the reasons that EFIM under-performs in these datasets is because the transaction merging strategy that EFIM uses is inefficient in such sparse datasets. It can also be observed that, for EFIM, the required time to identify all the HUIs increases exponentially when *minUP* decreases in *Chainstore* and *Kosarak*, and *Retail*, while, remarkably, in other datasets its running time remains almost constant when *minUP* varies. d2HUP comes second in *Accidents, BMS, Chess, Connect*, and *Mushroom*. But, in sparse datasets, such as *Foodmart, Chainstore*, and *Retail* (all relevant to shopping/market basket analysis), d2HUP is the clear winner. FHM comes either third or fourth in all datasets and *minUP* values, with the exception of *Retail* where it is second, performing very similarly with d2HUP. HUP-Miner is also in the middle of the class, regarding running time performance, with *Foodmart* being its best case, where it ranks second, behind d2HUP. HUI-Miner is typically the slowest algorithm, except in *Chainstore* and *Retail* where it is fourth, surpassing EFIM.

Fig. 2 depicts how the memory footprint of the HUIM algorithms is influenced by *minUP* in different real datasets. Observing the memory consumption results for the real datasets, we can infer that EFIM needs the least amount of memory in most datasets and at most *minUP* values. HUI-Miner and HUP-Miner are also memory efficient, in a few cases (e.g. in *Accidents* at $minUP \leq 10.00$) they are comparable or even better than EFIM. d2HUP and FHM require significantly larger memory than the rest. In specific, in *Accidents* and *Connect*, we observe a mixed pattern, in terms of the related difference of memory consumption between the algorithms, as depicted in Fig. 2. In *BMS, Chess, Connect* and *Mushroom*, d2HUP needs between 8 and 27 times more memory than EFIM. In many cases, FHM is among the two most memory consuming algorithms.

To summarise, in terms of running time, EFIM is the fastest in dense real datasets, but it is among the slowest algorithms in sparse datasets. d2HUP is less efficient than EFIM in the dense datasets, but it is the fastest in sparse datasets. In terms of memory consumption, EFIM is usually the best performer, while d2HUP typically needs the largest amount of memory.

It must be mentioned that, in Zida et al. (2015), the authors tested 6 real datasets, most of them being dense, leaving the sparse datasets untested, as can been in Table 8. Thus, they have not considered the influence of the dataset sparsity factor to the running time efficiency of different algorithms in HUIM.

5.2. Performance analysis on synthetic datasets

5.2.1. Performance analysis on 5 synthetic datasets for GroupA+B algorithms

In order to compare the performance of ***GroupA*** and ***GroupB*** algorithms, we first conduct experiments using the datasets d01, d04, d05, d22, and d26 from SetD27, at two *minUP* (0.10 and 0.12) thresholds. The results of these 10 tests are summarised in Table 12, where all the 8 algorithms are ranked according to their running time and memory consumption performance.

Algorithms in ***GroupA*** materialize candidates and need a second phase to match each candidate with transactions in the database, which causes scalability issue when the number of candidates is large, and have efficiency issues when the database size is big (Liu et al., 2016). As expected, they generally rank last in our results, as can be seen in Table 12.

The relative difference in running time performance, i.e., the ratio between each algorithm's running time and that of the fastest algorithm in every test, is illustrated in Fig. 3. In 4 out of 5 datasets (d01, d04, d05, d22), the relative gap in running time performance between the fastest (in each dataset) algorithm (which is d2HUP) and the algorithms from ***GroupA***, increases as the *minUP* value decreases. For example, in dataset d01, Two-Phase was found to be

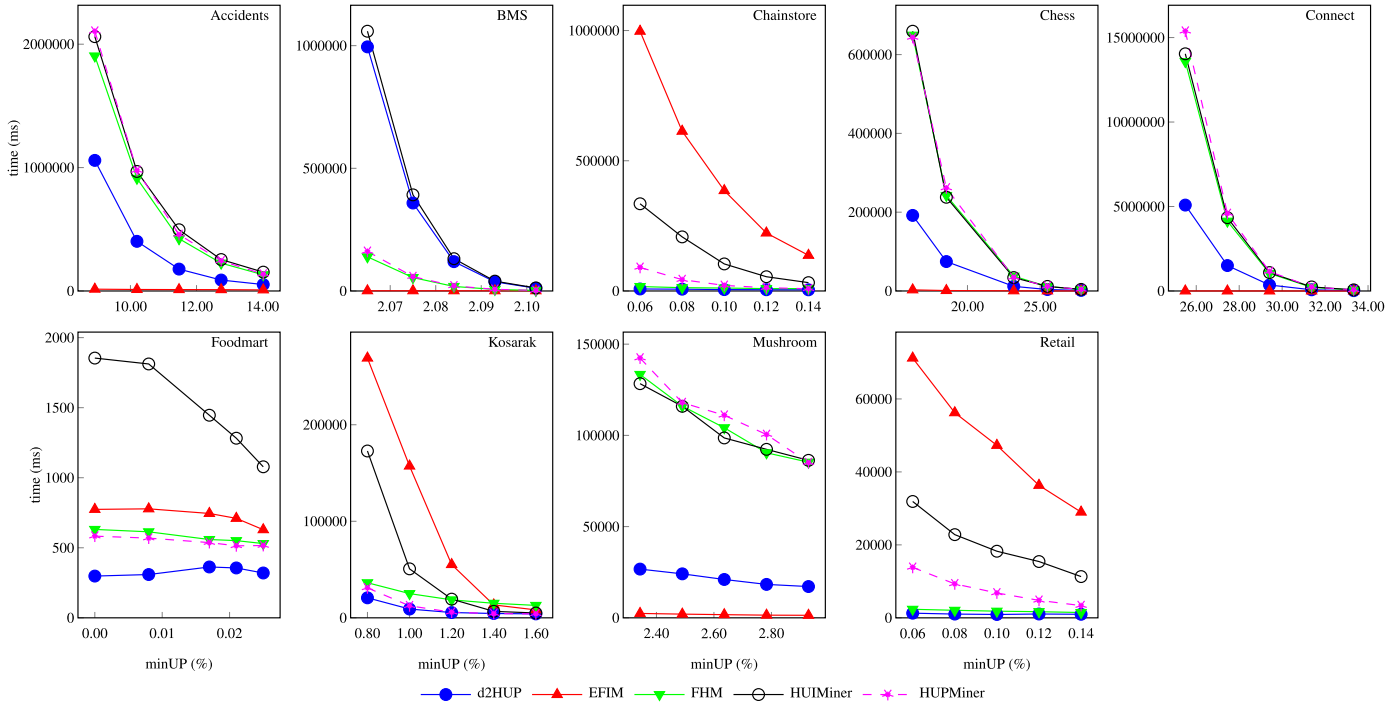

Fig. 1. Running time of GroupB algorithms in real datasets.

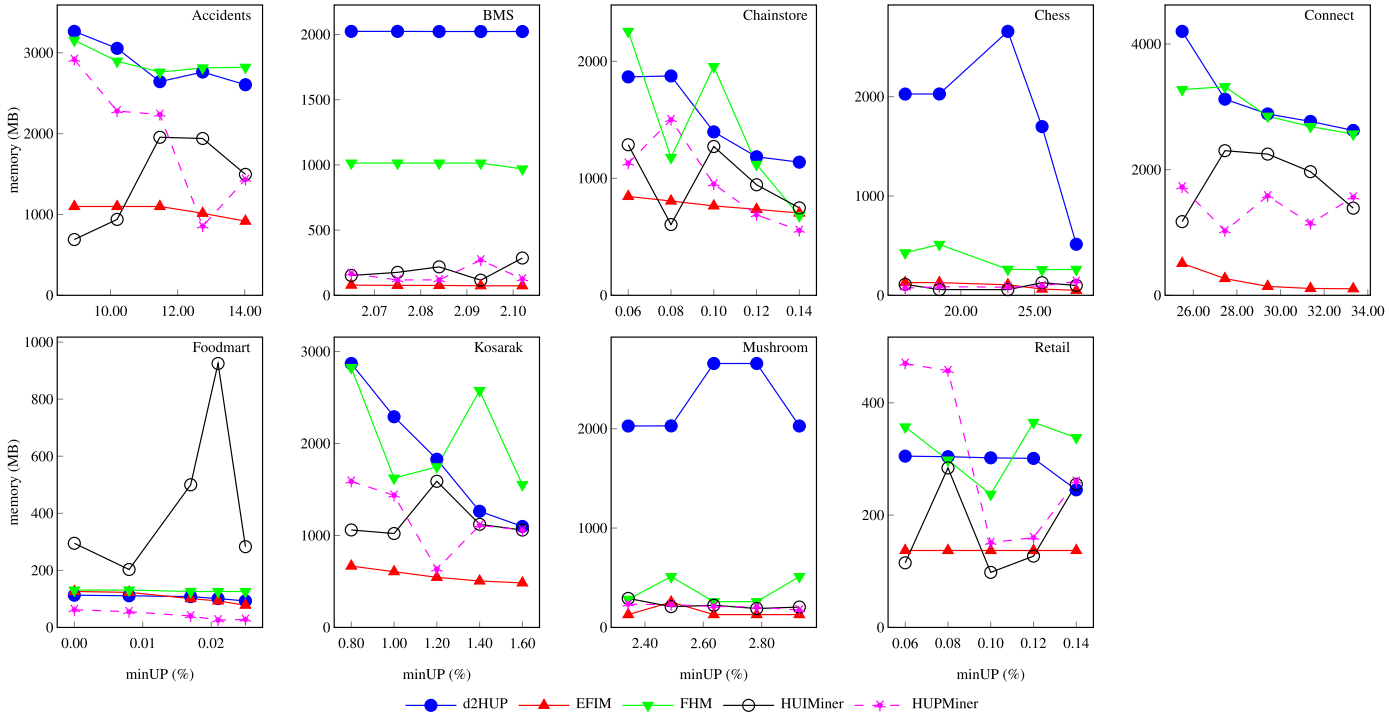

Fig. 2. Peak memory usage of GroupB algorithms in real datasets.

30 times slower than d2HUP at $minUP = 0.12$ and 44 times slower at $minUP = 0.10$. Interestingly, this pattern does not continue in large database d26, since all ***GroupA*** algorithms are relatively faster at $minUP = 0.10$ than $minUP = 0.12$.

At the same time, the difference increases as the datasets become larger, i.e. having either more transactions or transactions of larger maximum/average length. For example, in dataset d01, Two-Phase is 30 times slower than d2HUP, while in dataset d26 it becomes 1781 times slower.

As can be seen in Table 12, Two-Phase is the slowest algorithm in all the 10 tests. Its running time is found to be between 30 and 1781 times slower than the fastest algorithm. It is shown that IHUP ranks better than Two-Phase in running time but, overall, it is the second slowest. UP-Growth, is faster than both Two-Phase and IHUP, which agrees with the results in the literature (Tseng et al., 2010). It is also interesting to mention that in d23 it performs comparably with EFIM, ranking 3rd or 4th. In general, all algorithms of ***GroupA*** are much slower than those in ***GroupB***.

Table 12
Running time and memory ranking for GroupA+B algorithms in d01, d04, d05, d22, and d26 datasets of SetD27 at minUP thresholds 0.10% and 0.12%.

	Time								Memory							
Algorithm/rank	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8
d2HUP (GroupB)	10	0	0	0	0	0	0	0	0	1	1	1	2	1	2	2
EFIM (GroupB)	0	5	3	1	1	0	0	0	7	3	0	0	0	0	0	0
FHM (GroupB)	0	4	0	5	1	0	0	0	0	3	2	0	1	2	1	1
HUI-Miner (GroupB)	0	1	1	0	8	0	0	0	2	1	2	4	0	1	0	0
HUP-Miner (GroupB)	0	0	5	3	0	2	0	0	0	0	4	1	5	0	0	0
IHUP (GroupA)	0	0	0	0	0	0	10	0	0	2	0	0	1	0	3	4
Two-Phase (GroupA)	0	0	0	0	0	0	0	10	1	0	1	3	0	3	0	2
UP-Growth (GroupA)	0	0	1	1	0	8	0	0	0	0	0	1	1	3	4	1

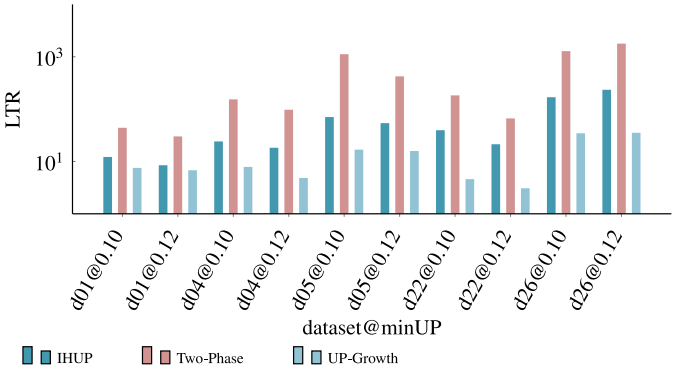

Fig. 3. Running time ratio in base-10 logarithmic scale between GroupA algorithms and d2HUP, in d01, d04, d05, d22, and d26 datasets of SetD27, at minUP=0.10% and 0.12%. Positive values denote slower times than that of d2HUP.

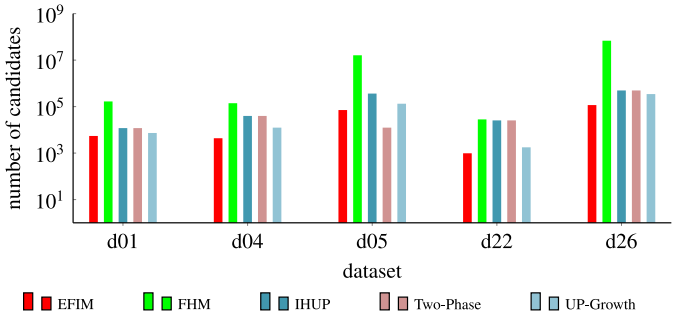

Fig. 4. Generated candidates in base-10 logarithmic scale with base 10 for d01, d04, d05, d22, and d26 datasets of SetD27 at minUP=0.10.

In Table 12, we can observe that EFIM is the most memory efficient algorithm. In general, the memory consumption performance of ***GroupA*** algorithms is average, with the exception of d22, where Two-Phase requires the least memory, and d26 where IHUP is ranked second.

Given the inferior running time performance of the algorithms in ***GroupA***, we opt to neglect them from the remaining experiments.

In Fig. 4, we give the number of the candidate HUIs that EFIM, FHM, IHUP, Two-Phase, and UP-Growth generate in datasets d01, d04, d05, d22, and d26 datasets of SetD27 at minUP thresholds of 0.10% and 0.12%. It can be clearly observed that EFIM generates the least number of candidates. Interestingly, FHM generates the most candidates at these tests. Nevertheless, FHM is still considerably faster than all algorithms of ***GroupA***, especially in larger datasets, such as d26, where it takes 6 times less time than UP-Growth, 30 times less than IHUP, and 225 times less than Two-Phase, in order to complete the task. This can be attributed to the fact that, although FHM generates a lot of candidates, it can prune a very large portion of them (Fournier-Viger et al., 2014) (see Table 8). UP-Growth generates considerably less candidates than the rest of ***GroupA*** algorithms in dataset d22, where it also performs very well in regard to running time (see Fig. 3, being only 3.1–4.6 times slower than d2HUP and on par with other algorithms of ***GroupB***.

Table 13
Summary of results of all tests for GroupB algorithms and SetD27 synthetic datasets, with 5 algorithms, 27 datasets, 5 minUP thresholds (675 total tests). totalTime is the time each algorithm to complete all its $27 * 5 = 135$ tests. Memory values refer to peak memory consumption.

Alg	totalTime (s)	minMemory (MB)	maxMemory (MB)
d2HUP	932	50	2650
EFIM	1431	25	135
FHM	4051	47	2656
HUI-Miner	5925	26	697
HUP-Miner	3908	28	1147

Table 14
Running time and memory ranking for GroupB algorithms in SetD27 synthetic datasets, with 5 algorithms, 27 datasets, 5 minUP thresholds (675 tests).

	Time					Memory				
Algorithm/rank	1	2	3	4	5	1	2	3	4	5
d2HUP	132	3	0	0	0	0	2	5	35	93
EFIM	3	107	21	0	4	54	19	62	0	0
FHM	0	21	34	75	5	0	14	4	88	29
HUI-Miner	0	2	1	12	120	47	54	22	8	4
HUP-Miner	0	2	79	48	6	34	46	42	7	6

5.2.2. Performance analysis on SetD27 synthetic datasets for GroupB algorithms

Here we compare the performance of the 5 ***GroupB*** HUIM algorithms (d2HUP, EFIM, FHM, HUI-Miner, HUP-Miner) in the 27 synthetic datasets of SetD27 which are summarised in Table 10. For this experiment, the total time, as well as the minimum and maximum peak memory consumption for each algorithm are summarised in Table 13. It is evident that in total d2HUP finishes the complete set of tests in considerably less time than the rest but at the same time it needs more memory to do that. EFIM needs 1.535 times more time to complete the tests but it is the most memory efficient by a large margin. The rest of the algorithms are clearly slower. FHM and HUP-Miner perform marginally the same regarding running time, but FHM, as d2HUP, requires the most memory. HUI-Miner is the second most memory efficient algorithm, but it is the slowest one.

Fig. 5 depicts in detail the running time performance of the ***GroupB*** algorithms in synthetic datasets in SetD27, which are moderately sparse, as depicted in Table 10. In the majority of the tests, we observe that d2HUP is the fastest algorithm. Table 14 summarises how all the 5 algorithms of ***GroupB*** rank in the 135 tests of SetD27. d2HUP is the fastest in 132 tests, while in the remain-

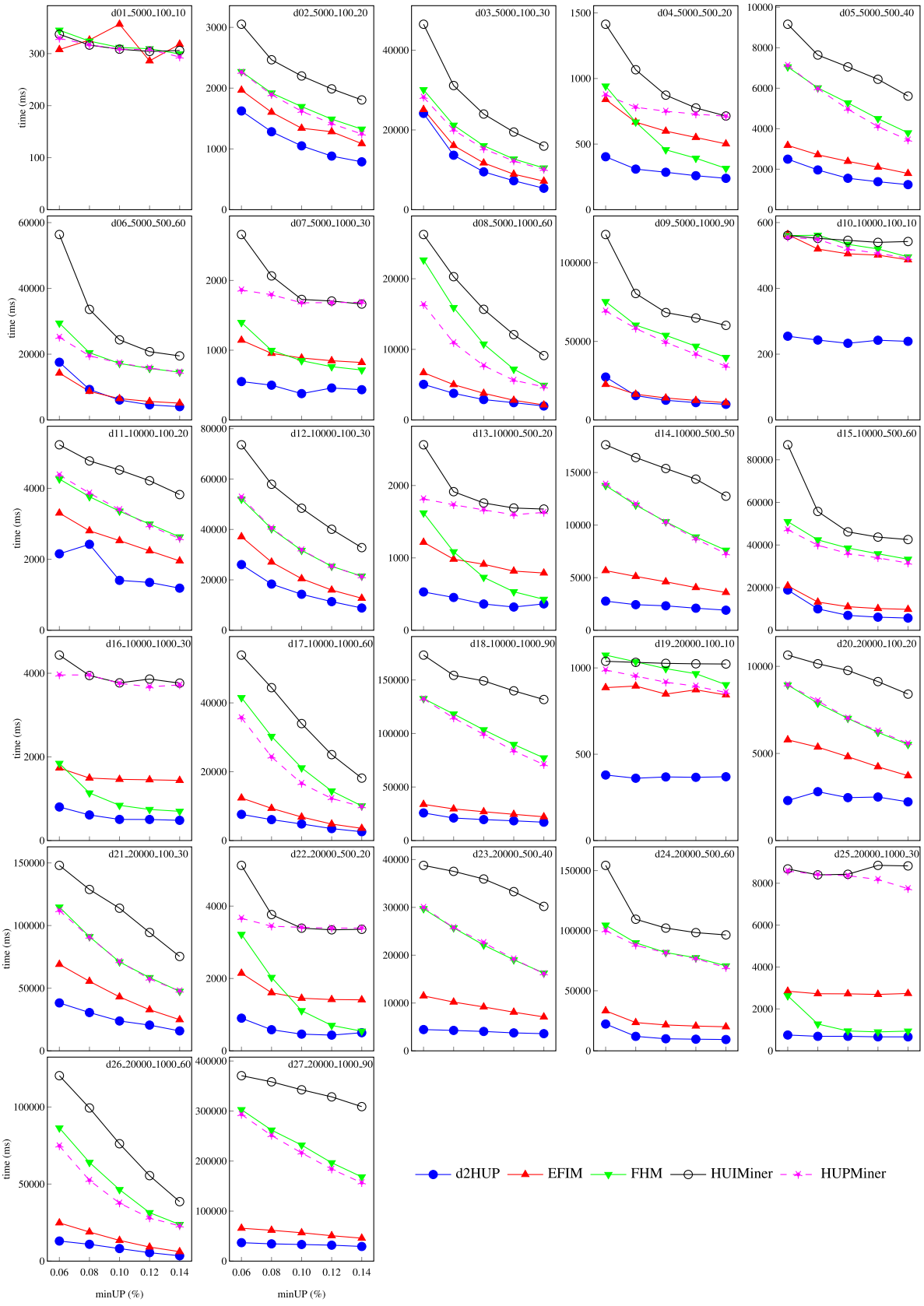

Fig. 5. Running time efficiency for GroupB algorithms on SetD27 synthetic datasets.

ing 3 tests (d06 at $minUP = 0.06$, d06 at $minUP = 0.08$, and d09 at $minUP = 0.06$) it trails EFIM. *This is consistent with our conclusions regarding the running time performance of d2HUP on dense real datasets.* The running time of EFIM is on average 1.7 times of d2HUP. Still, in 32 out of the total 135 tests, the running time of EFIM is at least 2 times of d2HUP.

As can be seen from Table 14 and Fig. 5, EFIM is in general the second best performer in terms of running time, since it ranks second in 107 out of the 135 tests. HUI-Miner is clearly the slowest algorithm. It ranks last in 120 cases and needs 5 times more time than d2HUP to complete the full set of tests in SetD27.

It is easy to visually identify several patterns regarding the relative difference in running time performance for the ***GroupB*** algorithms in the synthetic datasets of SetD27. Through non-rigorous analysis (instead of tensor clustering) we can vectorise the running times of all the 5 algorithms in each dataset (across all 5 *minUP* threshold levels) and perform hierarchical clustering (using Ward's minimum variance method) on these 25-dimensional stacked vectors. The results of the clustering analysis are illustrated in Figs. 7 and 8.

We observe that there are four clusters of datasets, where the running time of the algorithms demonstrate a similar pattern. In the first cluster, which is comprised of datasets d01, d10, d19, we see that d2HUP is the clear winner at all *minUP* values, while the rest of the algorithms are grouped together, as they differ marginally in running time. Also, all five algorithms' running time is linear, independent of *minUP*. The common characteristic in these datasets is the small number of transaction length, where $maxTransLen = 10$. As the number of distinct items is also constant in these datasets, i.e., $dbItems = |I| = 100$, we conclude that, in this case, the variation of the number of transactions and *minUP* do not significantly affect the efficiency of the algorithms.

The second cluster consists of datasets d02, d03, d05, d11, d12, d20, d21. Here, we have all the algorithms scaling in similar way. d2HUP again is the fastest algorithm in all these datasets and at all *minUP* values, with EFIM a close but clear second. Interestingly, FHM and HUP-Miner behave almost the same, while HUI-Miner is the worst performer by far.

Datasets d06, d08, d09, d14, d15, d17, d18, d23, d24, d26, and d27 form the third cluster, where the efficiency of the algorithms generally follow the same pattern, that is, the rankings of the algorithms in different datasets and *minUP* levels are consistent. In these datasets, the fastest algorithms are d2HUP and EFIM, they behave very close, while the performance of FHM and HUP-Miner are also very similar.

In the last cluster, containing d04, d07, d13, d16, d22, d25, we observe that, when the value of *minUP* increases, d2HUP is always the first, while the efficiency of FHM outperforms EFIM; moreover, the running time needed by FHM decreases more substantially than EFIM when *minUP* increases.

In terms of memory performance, as can be seen in Fig. 6 and Table 14, EFIM is the most efficient algorithm, which complies with the findings in Zida et al. (2015). We also observe that, d2HUP is the least memory efficient algorithm, as it typically consumes up to 5 times the memory needed by EFIM. We see that, in a few cases, such as d02 and d25, HUIMiner is better than EFIM in memory consumption. In general, as summarised in Table 14, the top-3 most memory efficient algorithms are EFIM, HUIMiner and HUP-Miner.

5.2.3. Performance analysis on SetDH45 synthetic datasets for GroupB algorithms

Since the densities of the datasets in SetD27 only range from 1.5% to 15%, it is worth comparing the running time performance and scalability of these two algorithms in the sparser datasets in SetDH45. These datasets have densities that range from 0.001% to 0.5%, while the rest of their parameters (i.e. *dbTrans, dbItems, maxTransLen*) vary in a consistent manner for scalability tests.

Table 15
Running time and memory ranking for GroupB algorithms in SetDH45 synthetic datasets, with 5 algorithms, 45 datasets, 5 minUP thresholds (1125 tests).

	Time					Memory				
Algorithm/rank	1	2	3	4	5	1	2	3	4	5
d2HUP	27	147	25	12	14	76	41	41	31	36
EFIM	197	7	14	6	1	44	83	30	40	28
FHM	0	26	24	61	99	33	30	71	42	34
HUI-Miner	1	23	63	58	65	44	35	38	67	26
HUP-Miner	0	41	95	54	20	29	35	30	30	86

Here, we investigate the performance of the 5 algorithms, with a *minUP* value between 0.10 and 0.50, in ***GroupB*** in the sparse datasets of SetDH45 depicted in Table 11. The running time and memory consumption results of these tests, are summarised in Table 15. Datasets dh18, dh21, dh33, and dh36, were excluded from these tests, as all algorithms except d2HUP were very slow there, especially as the *minUP* value decreased. For example, FHM failed to finish the test for dh18 at $minUP = 0.10$ because it ran out of the memory limit we had specified - it required 17GB of RAM to complete it, while other algorithms did not finish some tests after many hours. We observe that EFIM ranks first regarding running time efficiency in the majority of the tests, while d2HUP comes after EFIM, as it ranks first in 15 cases and second in 134 cases.

A common pattern that can be identified for the datasets where d2HUP was found to be the fastest algorithms, while the rest struggled to finish the test, is that for these datasets it holds $dbTrans > dbItems$ and *maxTransLen* is large, either 50 or 100. Surprisingly, this also means that these "hard" datasets are typically denser than the others.

It should be noted that, the above ranking is based on relatively large *minUP* thresholds, which are between 0.10% and 0.50%. Moreover, in datasets that are time-consuming to derive HUIs, e.g., dh03, dh17, and dh32, EFIM ranks lower than d2HUP and, in some cases, it ranks last in ***GroupB***, while in the other cases, where the time needed to finish the HUIM task is short, EFIM is the fastest. Hence, it is interesting and important to investigate the performance of EFIM and d2HUP at lower *minUP* values.

5.2.4. d2HUP vs. EFIM

Given the above experimental results, it is interesting to directly compare EFIM with d2HUP, since in many cases they are the top-2 performers, with regard to running time. While EFIM has been reported in Zida et al. (2015) to consistently outperform all the other algorithms, our experimental results in SetD27 (ref. Section 5.2.2) are not always consistent with their conclusion.

As can be seen in Fig. 1, in 3 real-world datasets (which are *Chainstore, Kosarak, Retail*), EFIM is the slowest algorithm; in comparison, d2HUP is the most efficient algorithm on these three datasets. Moreover, as *minUP* value becomes smaller, the performance gap between them becomes larger and larger. On the *Foodmart* dataset, d2HUP is still the fastest method, but EFIM only ranks 4th. One common characteristic of these four real world datasets is that the density of these datasets is very low, i.e., they are very sparse datasets; the number of the transactions is also large.

To further study the influence of the data characteristics on the running time efficiency of these two algorithms, we use SetDH45 synthetic datasets where the dataset characteristics vary as follows: *dbTrans* ranges in 3 steps from 10,000 transactions up to 1,000,000, *dbItems* ranges from 10,000 to 1,000,000 scaling in 5 steps, while for *maxTransLen* we use the values 10, 50, and 100.

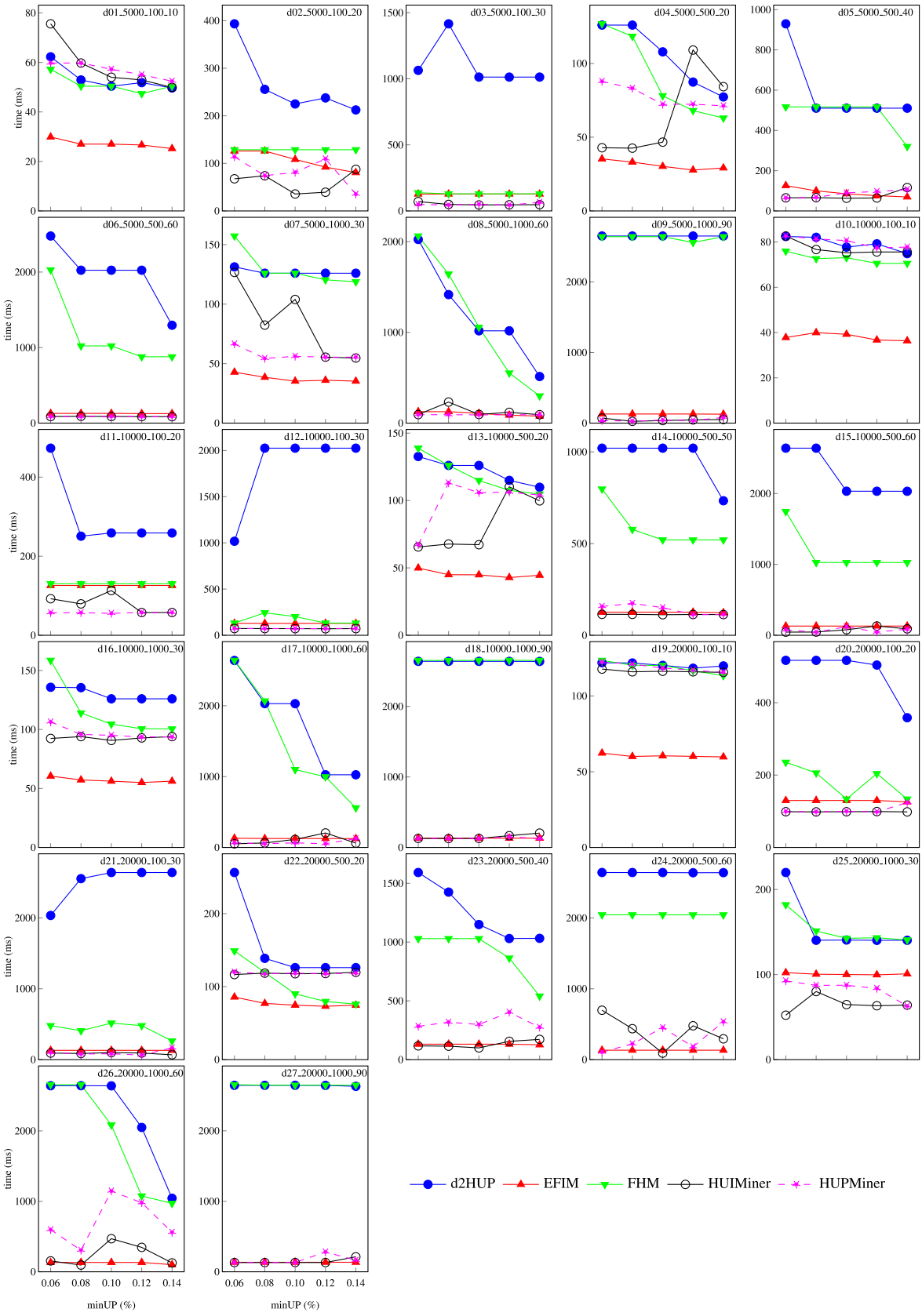

Fig. 6. Maximum memory consumption for GroupB algorithms on SetD27 synthetic datasets.

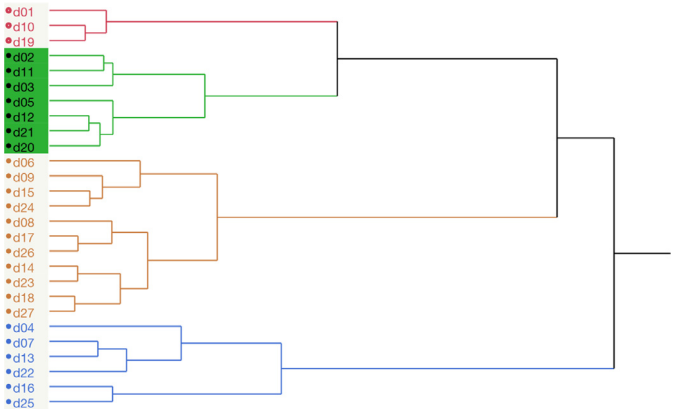

Fig. 7. Hierarchical Analysis - Dendrogram and Distance Graph for the synthetic datasets SetD27.

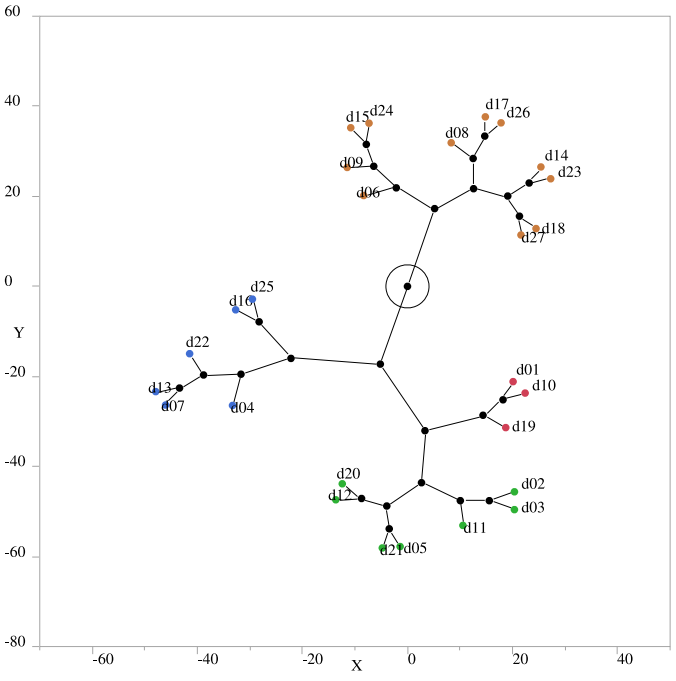

Fig. 8. Hierarchical Analysis - Constellation Plot for the synthetic datasets SetD27.

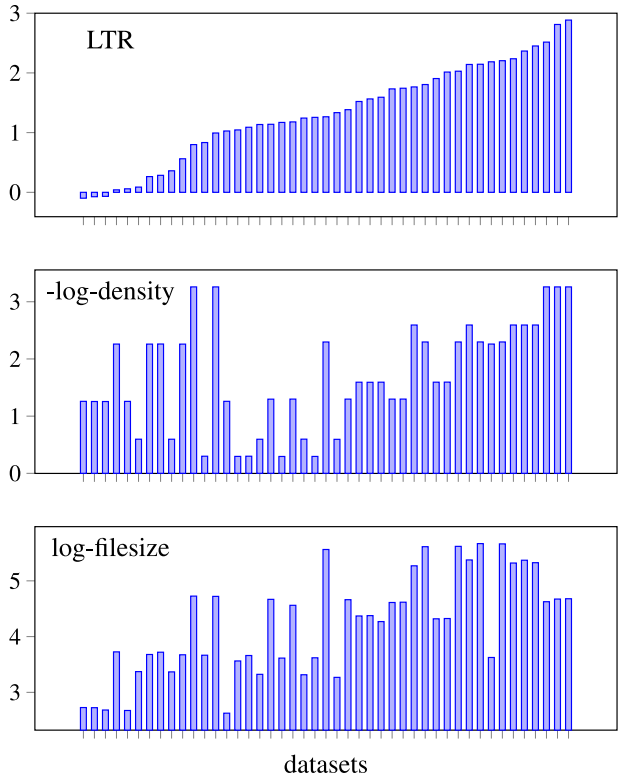

Fig. 9. Correlations between log-ratio running time (*LTR*), dataset density, and dataset file size in SetDH45 datasets. Datasets are ordered ascendingly on *LTR*. Densities are negative log-10 values, i.e., larger values correspond to sparser datasets. File sizes are log-10 values of database size in KBytes.

As can be seen in Table 11, this results to $3 \times 5 \times 3 = 45$ datasets. Their densities range from 0.0005% to 0.5%.

For these experiments, we do not vary the *minUP* values in the range of 0.10%–0.50%, as done in Section 5.2.2. Instead, for each dataset, we pick the minimum utility threshold value that makes sure that one HUI is generated, denoted as *minU*1. The *minU*1 and the corresponding *minUP* values (*minUP*1) used for each dataset can be seen in the last two columns of Table 11. Since the running time is anti-monotonic with *minU*, when *minU* decreases, the running time becomes larger.

When the running time needed by EFIM is more than 1 second, EFIM needs from 1.82 to 758 times the running time of d2HUP. We also observe that, when the datasets require more time to complete, e.g., when EFIM uses more than 10 s to finish, EFIM needs from 13.8 to 758 times of d2HUP's running time. This reveals that, as datasets require more time to accomplish the HUIM task, the performance gap between d2HUP and EFIM increases.

To compare the running time performance of d2HUP and EFIM, we use the relative log10-ratio (denoted as *LTR*) between the running times (in milliseconds) of d2HUP and EFIM (denoted as *LTD* and *LTE*, respectively). It is $LTR = LTE - LTD$. Positive values of *LTR* means that EFIM requires more time to complete a test than d2HUP, i.e., d2HUP is faster in that test.

In Fig. 9, we illustrate the correlation of the relative ratio *LTR* with the dataset density and dataset file size. In general, when excluding datasets where tests require less than a second to finish (this includes the majority of the datasets in the left part of Fig. 9), we can deduce that the gap in the performance between d2HUP and EFIM increases when the datasets become sparser. For the several distinctive sparse datasets on the left of the bar plot, where the running time discrepancy between EFIM and d2HUP is as large as for the datasets on the right, it is mostly $dbItems > dbTrans$. Regarding the file size of the database, *LTR* generally increases monotonically when *maxTransLen* is 50 or 100.

In Fig. 10, we examine the influence of *dbTrans*, *dbItems*, and *maxTransLen* on *LTR*, *LTD*, and *LTE*. In this figure, absolute and related running time performance is grouped based on different parameter values. We observe that the influence of *dbTrans* on the *LTR* ratio is significant. When the number of transactions in the database increases, both running times and *LTR* increase almost exponentially. On the other hand, when the number of distinct items in the database increases, *LTR* does not change in a consistent way.

It can be deduced that, as *maxTransLen* increases, the relative ratio *LTR* also increases. At the same time, *LTR* is anti-monotonic with regard to *dbItems*, when $maxTransLen = 10$. There is a point where this behaviour changes, and *LTR* becomes less dependent on *dbItems*, i.e. the running time performance gain of d2HUP over EFIM is maintained at all *dbItems* values, as can be seen in the right and bottom-right side of the group-plot in Fig. 10.

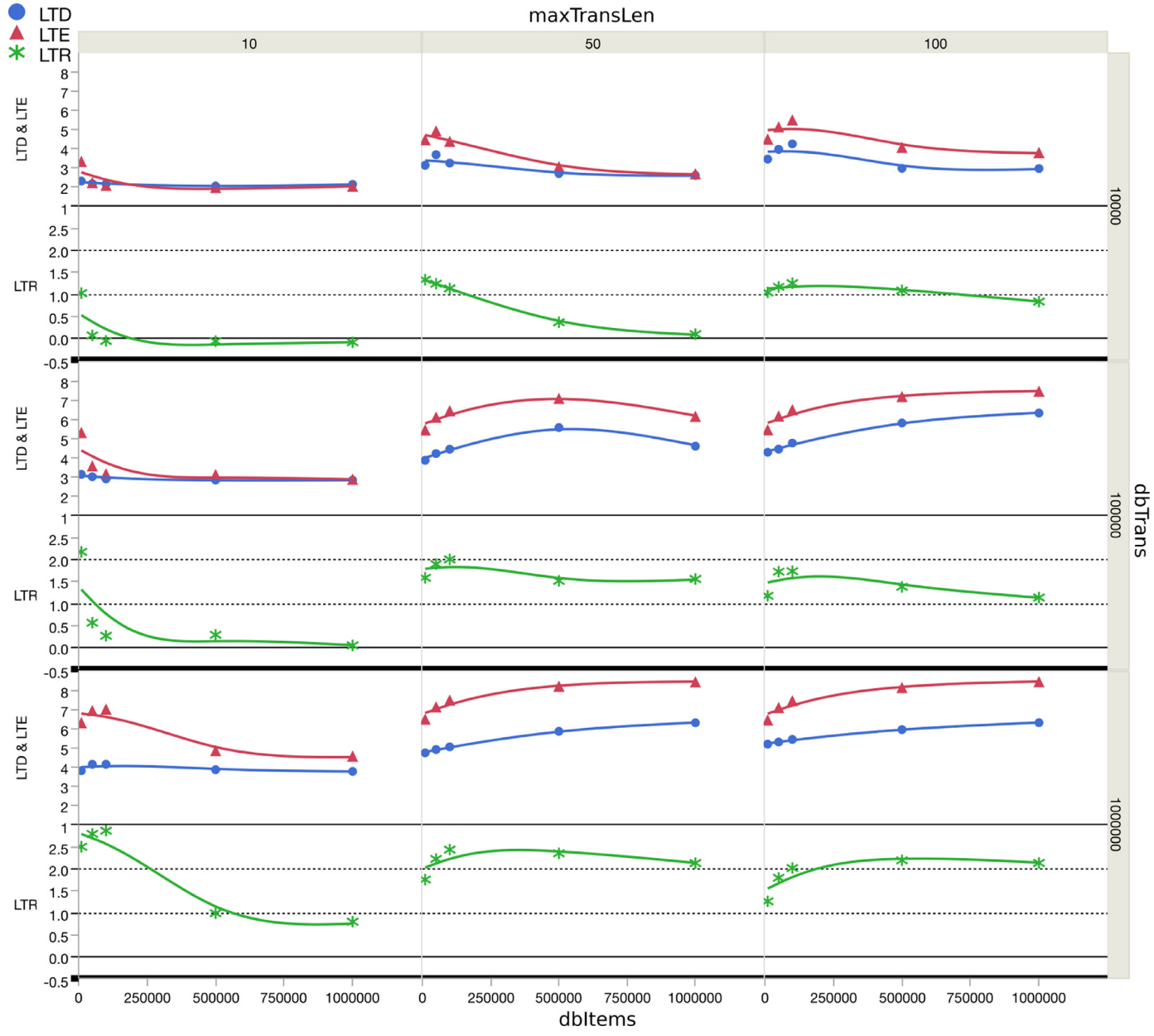

Fig. 10. Running time comparison between d2HUP and EFIM in SetDH45. All tests were run at *minUP*1 thresholds, which correspond to minimum utilities where at least one HUI is identified. *LTD* and *LTE* are the running times (in ms) in log-10 scale for d2HUP and EFIM respectively. $LTR = LTE - LTD$ is the log-time-ratio between the two algorithms. Positive *LTR* values denote that d2HUP is faster than EFIM. Lines have been smoothed using the cubic spline method with a tuning parameter $\lambda = 0.05$ (Eubank, 1988).

5.2.5. Performance of the newest algorithms

In this subsection, we empirically evaluate the performance of the two newest algorithms proposed in 2017, i.e. mHUIMiner and ULB-Miner (***GroupC*** algorithms), versus the top performers found in the above-mentioned experiments, i.e. d2HUP and EFIM. We use the same datasets as in the previous experiments of this work.

It should be noted that the current implementations of HUIM-BPSO and HUIM-BPSO-Tree in SPMF can not generate the same results as reported in the corresponding papers (Lin, Yang, Fournier-Viger, Hong et al., 2016; Lin, Yang, Fournier-Viger, Wu et al., 2016). This has been confirmed by our experiments on the same real datasets with the same minimum utility values, where HUIM-BPSO and HUIM-BPSO-Tree always identify inadequate number of HUIs. For this reason, we do not include them in the evaluation. HUIM-GA and HUIM-GA-Tree algorithms were also unable to work using the latest SPMF version (v2.19) so they were excluded as well from our experiments.

In Figs. 11 and 12, we compare the running time performance of mHUIMiner and ULB-Miner against EFIM and d2HUP. For the datasets in Fig. 11, which are typically dense, EFIM is the best performer, therefore it is used as the reference. We observe that, on all these 5 datasets, mHUIMiner is always the slowest at all the *minUP* values. while both ULB-Miner and d2HUP perform slower than EFIM. There is no constant winner between ULB-Miner and d2HUP.

For Chainstore, Kosarak and Retail in Fig. 12, which are typically sparse datasets, EFIM is the slowest algorithm, while d2HUP is the best performer, therefore it is used as the reference. We observe that EFIM takes 10–100 times the running time needed by d2HUP, while ULB-Miner and mHUIMiner always outperform EFIM at all

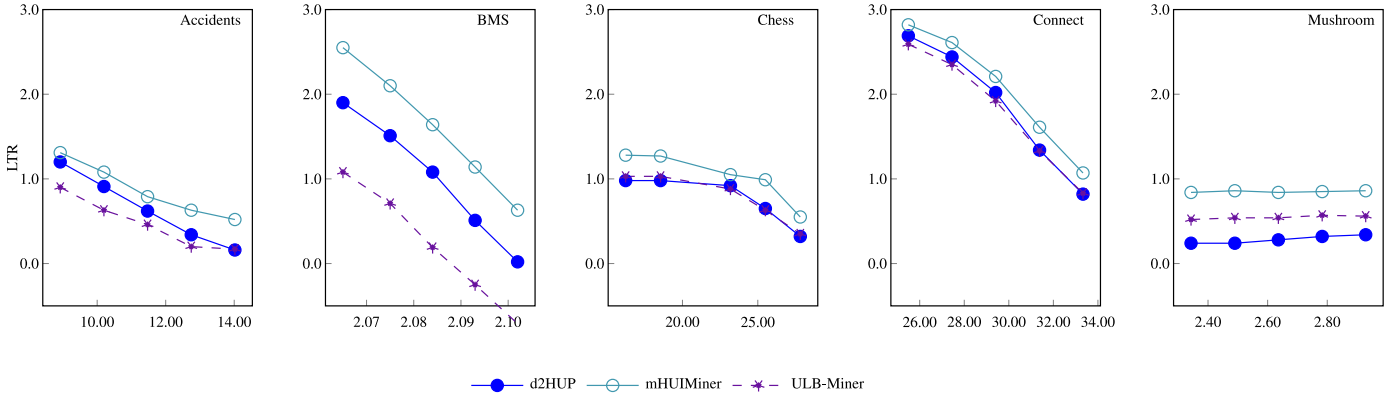

Fig. 11. Relative Running time (LTR) of d2HUP, mHUIMiner, and ULB-Miner vs. EFIM.

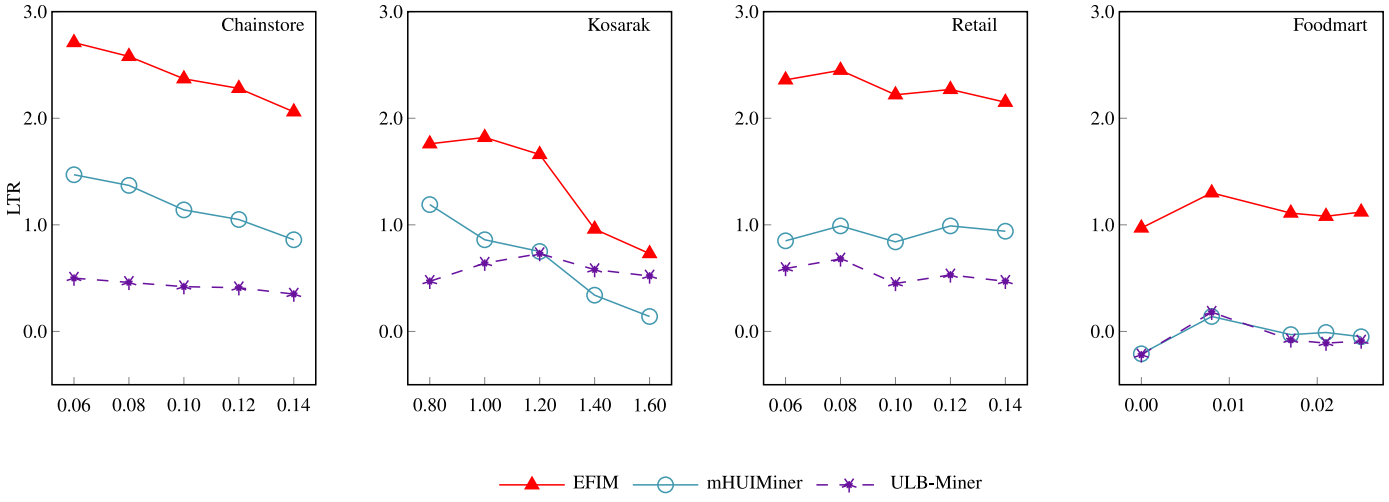

Fig. 12. Relative Running time (LTR) of EFIM, mHUIMiner, and ULB-Miner vs. d2HUP.

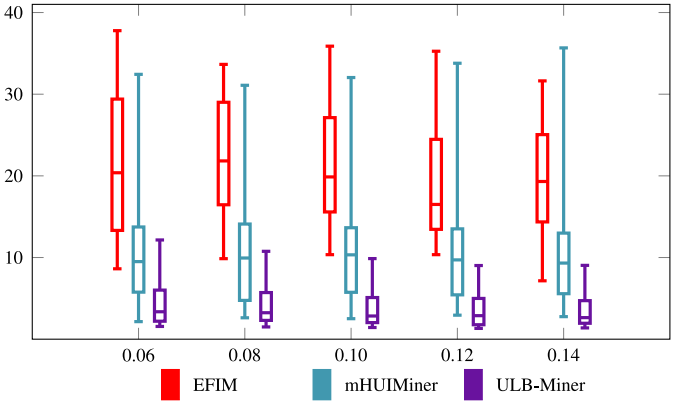

Fig. 13. Running time ratio in base-10 logarithmic scale (LTR) between EFIM, mHUIMiner, ULB-Miner and d2HUP, in dh21 31 datasets of SetDH45, at minUP1. Positive values denote slower times than that of d2HUP.

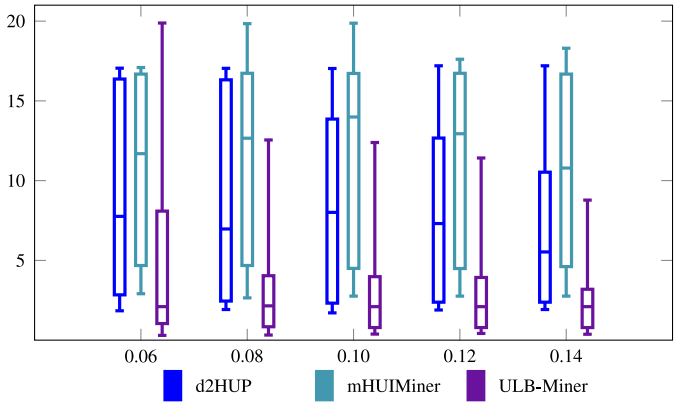

Fig. 14. Relative Memory Consumption Ratios of d2HUP, mHUIMiner and ULB-Miner vs. EFIM at different minUP values.

the *minUP* values. For Foodmart, it is interesting to observe that the most efficient algorithm is ULB-Miner, followed by mHUIMiner; both of them significantly outperform EFIM and their performance is close to d2HUP.

In Fig. 13, we compare the running time performance of EFIM, mHUIMiner, and ULB-Miner on SetD27, with d2HUP as the reference and grouped by *minUP*. We observe that ULB-Miner and mHUIMIner are significantly faster than EFIM, being 2 and 5 times faster on average, respectively. It is also clear that ULB-Miner is more efficient than mHUIMiner, at all *minUP* values. Overall, ULB-Miner, mHUIMIner and EFIM need 4, 11, 21 times of the computation time of d2HUP.

For memory efficiency, in Fig. 14, we report the ratios of memory usage of ULB-Miner, d2HUP and mHUIMiner with respect to EFIM. On average, they consume 3, 8, and 11 times the memory needed by EFIM, respectively. It is also clear that ULB-Miner is much more efficient than the other two, with the exception of d09 and d18 where the maximum transaction length is large and *minUP* threshold is very low.

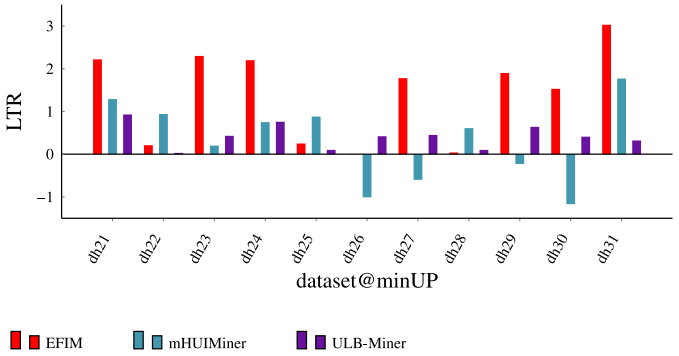

Fig. 15. Running time ratio in base-10 logarithmic scale (LTR) between EFIM, mHUIMiner, ULB-Miner and d2HUP, in dh21 31 datasets of SetDH45, at minUP1. Positive values denote slower times than that of d2HUP.

In Fig. 15, we showcase the LTR of the 4 algorithms with d2HUP as the reference on a selected subset of SetDH45 at *minUP*1 threshold. d2HUP is consistently found to be the top performer in difficult tasks, whereas mHIUMiner is the fastest on 4 datasets (*dh*26, *dh*27, *dh*29 and *dh*30), where mHUIMiner is even faster than d2HUP. These 4 datasets have the same characteristic: $dbItems > dbTrans$, i.e. the number of distinct items in the database is larger than the total number of transactions. In other datasets of SetDH45 with the same property, we also found that mHUIMiner's LTR values are generally much lower than the cases where this property does not hold.

In terms of memory consumption on the same SetDH45 subset, we found that when the maximum transaction length increases to 100, the memory usage of ULB-Miner increases exponentially; it occupies 15–40 times the memory of EFIM. ULB-Miner was unable to finish in reasonable time in 4 SetDH45 datasets (*dh*36, *dh*39, *dh*42, and *dh*45) at *minUP*1 minimum utility thresholds due to excessive memory consumption. All these 4 datasets, which have $maxTransLen = 100$ and large number of distinct items and transactions, ULB-Miner required more than 128GB of RAM. This suggests that it is not appropriate for datasets that are large, sparse at very low minimum utility threshold. This kind of scenario has not been investigated in the original paper (Duong et al., 2017) but may occur in the retail analytics problems.

To summarise, both mHUIMiner and ULB-Miner offer a moderate performance for HUIM, so they could be used in cases when data characteristics are not available. However, when this information is given, it is safe to only choose EFIM or d2HUP, also depending on the running time and memory requirements.

5.3. Summary of results - discussion

In summary, we have the following final remarks for the analysed results of the conducted experiments:

1. The algorithms of ***GroupA***, including IHUP, Two-Phase, and UP-Growth, are orders of magnitude slower than the algorithms in ***GroupB*** (i.e., d2HUP, EFIM, FHM, HUI-Miner, and HUP-Miner), thus ***GroupA*** algorithms are impractical for most applications. The running time performance of ***GroupC*** algorithms usually fall between EFIM and d2HUP, but in a few cases, mHUIMiner has the best performance.
2. In all the experiments, the most efficient HUIM algorithm is either d2HUP or EFIM.
   (a) While EFIM performs exceptionally well in dense real datasets regarding running time, its performance degrades when the datasets are sparse. Moreover, its running time performance trails d2HUP in almost all the synthetic tests of SetD27. While it is faster in 0.10–0.50% *minUP* values in most of the tests in SetDH45, it is slower than d2HUP when *minUP* is much lower.
   (b) EFIM is the most memory efficient algorithm; it consumes the least memory in the majority of the tests, but also scales linearly in space, with regard to *minUP* threshold. d2HUP typically requires the most memory in both real and synthetic datasets, independent of *minUP*. For small and dense datasets, UP-tree like structures have higher compression ratio than d2HUP's proposed structure (CAUL).
3. The minimum utility threshold is a very significant parameter to be considered when assessing the performance of a HUIM algorithm. In transactional databases with many transactions and/or long average length of transactions, low *minUP* values are required in order to retrieve HUIs. In this work, it is showcased that algorithms that perform excellent in high *minUP* values might not scale well with *minUP* value and database size.
4. The characteristics of the dataset should be investigated when making decisions on the best HUIM algorithm.
   (a) The number of transactions (*dbTrans*) increases monotonically with the database size. Thus, time and memory requirements for the HUIM task also increase accordingly, especially for algorithms in ***GroupA***, which depend on multiple database scans. For datasets with many transactions, such as those associated with basket market analysis, HUIs can only be identified in very low *minUP* threshold values. At these values, EFIM requires considerable more time than d2HUP. On the other hand, the choice of d2HUP is not recommended if memory consumption is the major concern. EFIM is more memory efficient because it uses a simple database representation, which does not require to maintain much information in memory (only pointers for pseudo-projections).
   (b) The number of distinct items in the database (*dbItems*) is related with the density of a database, therefore influences the effectiveness of HUIM algorithms that rely on compressed data structures. This parameter also varies greatly with the type of the applications. For example, in big supermarkets and online retail shops, the size of the inventory is many times larger than the average transaction length, while in biological applications or games, the two values can be comparable. In sparse datasets with long transactions, EFIM's ability to merge transactions by reusing some of its data structures (utility bins) is restricted.
   (c) The average/maximum length of the database transactions (*avgTransLen, maxTransLen*) is associated with both the size and the density of the database. For EFIM, as well as for FHM, HUI-Miner, and HUP-Miner, running time rapidly increases when *maxTransLen* is large. This behaviour is further amplified when *minUP* threshold decreases, to a point where these algorithms become impractical. In these cases, d2HUP is many times faster than the rest of the algorithms.
   (d) The total utility of the database (*dbUtil*) is directly associated with the file size of a transactional database that has been generated synthetically, following Gaussian and uniform distributions for utility values and transaction lengths.

The main differences between our work and Zida et al. (2017) are as follows:

1. The goal of our work is to comprehensively compare the performance of different HUIM algorithms, while Zida et al. (2015, 2017) aims to propose a new algorithm, EFIM, and validate its performance over existing methods.
2. In Zida et al. (2017), only 6 algorithms are compared, while in our work, we have empirically evaluated the performance of 10 HUIM algorithms. The 4 extra algorithms we have considered are Two-Phase, IHUP, mHUIMiner, and ULB-Miner, with

the last two being proposed in 2017. We conduct large-scale experiments on 81 datasets in total, both real and synthetic ones, while Zida et al. (2017) only compared EFIM with 5 other algorithms on just 9 real datasets. Moreover, in our work, we have used large-scale synthetic datasets (27+45, 72 in total) to thoroughly and comprehensively study the performance of the 10 HUIM algorithms.

3. Besides EFIM, we have extensively investigated the performance of d2HUP and other algorithms, and revealed that the top-2 performers in terms of running time efficiency are either d2HUP or EFIM, depending on the data characteristics. More importantly, we have reported that d2HUP is the most efficient algorithm in sparse, very large datasets, also including cases where low minimum utility thresholds are specified. Although the authors in Zida et al. (2017) showcased that on the two sparse real datasets (Chainstore and Kosarak), d2HUP outperforms EFIM, the authors have not systematically validated this observation; the scale of these two datasets is also too small, thus less representative to reach a final discovery. Therefore, large-scale experiments need to be carried out to empirically validate the performance difference between EFIM and d2HUP.
4. When we generate the synthetic datasets, we vary the number of transactions (*dbTrans*), the maximum transaction length (*maxTransLen*), which is directly related to the average transaction length, and the number of distinct items (*dbItems*). We use these 72 synthetic datasets of different characteristics to systematically study the influence of these parameters on the performance of different HUIM algorithms and discover the possible patterns/influence of the dataset density on the efficiency of the algorithms (see Section 5.2 and Table 11), under different scales of minimum utility percentage (*minUP*) values.
   Moreover, we consider cases where the number of transactions is smaller than (or equal to) the number of distinct items, which has seldom been addressed in existing works but may exist in the supermarket, online drugstore, furniture retailer, online music/movie store, etc. application scenarios, where the transactional data may be accumulated over a given time period (e.g. 1–2 weeks after the new supermarket opens, during the Christmas/Thanksgiving holidays, etc.). As can be seen in Table 11, the number of distinct items can be 5, 10, 50, or 100 times the corresponding number of transactions. Still, we have included the "default" cases where the number of transactions is larger than the number of distinct items (e.g. datasets dh31...dh42).
   It should also be emphasised that Zida et al. (2017) and other state-of-the-art works on HUIM have not systematically investigated the influence of the average transaction length/maximum transaction length (*maxTransLen*) parameters on the performance of the algorithms. In our work (see Fig. 10 in Section 5.2.4), we study the influence of *maxTransLen, dbTrans*, and *dbItems* on the running time efficiency of d2HUP and EFIM, under a very low minimum utility value (*minU*), which guarantees that one HUI can be generated. We observe that when *maxTransLen* and *dbItems* are both large, d2HUP is usually 10–100 times faster than EFIM. As we have reported, we also observe that the running time and *LTR* (log-time-ratio difference of EFIM and d2HUP running times) increase nearly exponentially with *dbTrans*. Moreover, the running time, as well as *LTR*, also increase with *maxTransLen*. We also report that there is a turning point where *LTR* becomes less dependent on *dbItems*.
5. We have analysed the time complexity of HUIM algorithms, which has not been addressed before. We point out that the key factors that affect the efficiency of the HUIM algorithms, besides the minimum utility parameter, are the average transaction length, the number of distinct items, and the sparsity of the dataset.
6. We have compared EFIM and d2HUP with the newest HUIM algorithms proposed in 2017 (mHUIMiner and ULB-Miner), which have not been considered in Zida et al. (2017).

To our knowledge, all the above points have not been investigated in a systematic manner by the literature.

6. Conclusions

In this study, we empirically evaluate the performance of 10 state-of-the-art HUIM algorithms. The efficiency of 5 more recent HUIM algorithms (i.e., d2HUP, EFIM, FHM, HUI-Miner, and HUP-Miner) is systematically assessed on both synthetic ($27 + 45 = 72$ in total) and real (9 in total) datasets. Three additional well-known algorithms (i.e., IHUP, Two-Phase, and UP-Growth), from earlier works in HUIM literature, are also evaluated on 5 synthetic datasets. Two newest HUIM algorithms, proposed in 2017, are also included. The results of this study show that, although EFIM in general is the most efficient algorithm in memory consumption, it is not the clear winner in terms of running time. Its performance depends highly on the characteristics of the data and degrades exponentially when the dataset is large and sparse, which is often the case in market basket analysis in retail. On the other hand, d2HUP demonstrates superior running time performance (in many cases better than EFIM), especially in large sparse datasets and with low minimum utility percentage values, but it requires more memory than the other HUIM algorithms.

This paper offers in-depth comparisons that have not been studied in the literature. To the best of our knowledge, for the case of the two top-performing HUIM algorithms, d2HUP and EFIM, a direct comparison is only given in Zida et al. (2017), where the authors claim that EFIM is generally faster but accept that transaction merging is not very effective in sparse benefits and its cost exceeds its benefits. Yet, only real datasets are considered there, while the minimum utility percentage values examined lie in a narrow range of (typically high) values. We attest that, d2HUP is faster when either the number of transactions or the average length of the transactions in the dataset is large. Furthermore, in such cases, the performance difference between d2HUP and EFIM increases significantly as the minimum utility percentage threshold decreases.

This discovery enables us to suggest practical HUIM algorithm selection criteria that can associate the memory efficiency and running time performance with the characteristics of the data and, consequently, with the type of applications. If memory resources are very limited, practitioners should select EFIM, as it is generally more memory efficient than all the other algorithms. Furthermore, for small datasets or very dense datasets, EFIM is always among the fastest HUIM algorithms. But when a dataset is very sparse or the average transaction length is large, and running time is favoured over memory consumption, d2HUP should be chosen.

As more HUIM algorithms are proposed, publicly available benchmarks for HUIM under purely synthetic workloads will be needed for repeatable and reliable comparisons, so that researchers can robustly analyse and evaluate their algorithms in HUIM and practitioners can apply the most appropriate algorithm for their specific applications and data.

For future work, we will further study how different HUIM methods and data structures are influenced by the statistical distributions of the item utilities and the average transaction length of a dataset. We will also conduct empirical studies to compare state-of-the-art methods in alternative HUI patterns (e.g., closed, top-k, etc.).

Acknowledgments

This work is partially funded by the National Science Foundation of China (NSFC) under Grant no. 41401466. It is also supported by the National Key Technology R&D Program of China under Grant no. 2015BAK01B06, as well as Henan University under Grant no. xxjc20140005. We greatly appreciate the contributions of the anonymous reviewers for their valuable suggestions and help. We also acknowledge the donations of Tesla K40 GPU from NVIDIA to Henan University for supporting our academic research.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *Proceedings of the 20th international conference on very large data bases, VLDB: Vol. 1215* (pp. 487–499).

Ahmed, C. F., Tanbeer, S. K., & Jeong, B.-S. (2010). A novel approach for mining high--utility sequential patterns in sequence databases. *ETRI Journal, 32*(5), 676–686.

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

Alkan, O. K., & Karagoz, P. (2015). Crom and huspext: Improving efficiency of high utility sequential pattern extraction. *IEEE Transactions on Knowledge and Data Engineering, 27*(10), 2645–2657.

Cavique, L. (2007). A scalable algorithm for the market basket analysis. *Journal of Retailing and Consumer Services, 14*(6), 400–407.

Chan, R. C., Yang, Q., & Shen, Y.-D. (2003). Mining high utility itemsets. In *Proceedings of the IEEE 12th international conference on data mining (ICDM 2003)* (pp. 19–26).

Deng, Z.-H., Ma, S., & Liu, H. (2015). An efficient data structure for fast mining high utility itemsets. arXiv preprint:1510.02188.

Duong, Q.-H., Fournier-Viger, P., Ramampiaro, H., Nørvåg, K., & Dam, T.-L. (2017). Efficient high utility itemset mining using buffered utility-lists. *Applied Intelligence*, 1–19.

Erwin, A., Gopalan, R. P., & Achuthan, N. (2008). Efficient mining of high utility itemsets from large datasets. In *Proceedings of the Pacific-Asia conference on knowledge discovery and data mining* (pp. 554–561). Springer.

Eubank, R. L. (1988). *Spline smoothing and nonparametric regression*. 04; QA278. 2, E8

Fournier-Viger, P. (2010). SPMF data mining library. http://www.philippe-fournier-viger.com/spmf. Accessed on 2017-08-11.

Fournier-Viger, P., Lin, J. C.-W., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., et al. (2016). The spmf open-source data mining library version 2. In *Joint European conference on machine learning and knowledge discovery in databases* (pp. 36–40). Springer.

Fournier-Viger, P., Wu, C.-W., Zida, S., & Tseng, V. S. (2014). Fhm: faster high-utility itemset mining using estimated utility co-occurrence pruning. In *Proceedings of the international symposium on methodologies for intelligent systems* (pp. 83–92). Springer.

Fournier-Viger, P., Zida, S., Lin, J. C.-W., Wu, C.-W., & Tseng, V. S. (2016). Efim-closed: fast and memory efficient discovery of closed high-utility itemsets. In *Machine learning and data mining in pattern recognition* (pp. 199–213). Springer.

Geurts, K., Wets, G., Brijs, T., & Vanhoof, K. (2003). Profiling of high-frequency accident locations by use of association rules. *Transportation Research Record: Journal of the Transportation Research Board, 1840*, 123–130.

Goethals, B., & Zaki, M. J. (2003). Frequent itemset mining dataset repository. http://fimi.cs.helsinki.fi/data. Accessed on 2017-08-11.

Goyal, V., & Dawar, S. (2015). Up-hist tree: An efficient data structure for mining high utility patterns from transaction databases. In *Proceedings of the 19th international database engineering & applications symposium* (pp. 56–61). ACM.

Kannimuthu, S., & Premalatha, K. (2013). Discovery of high utility itemsets using genetic algorithm. *International Journal of Engineering and Technology (IJET), 5*(6), 4866–4880.

Krishnamoorthy, S. (2015). Pruning strategies for mining high utility itemsets. *Expert Systems with Applications, 42*(5), 2371–2381.

Lichman, M. (2013). UCI machine learning repository. http://archive.ics.uci.edu/ml. Accessed on 2017-08-11.

Lin, J. C.-W., Yang, L., Fournier-Viger, P., Hong, T.-P., & Voznak, M. (2016). A binary PSO approach to mine high-utility itemsets. *Soft Computing, 3*, 1107–1135.

Lin, J. C.-W., Yang, L., Fournier-Viger, P., Wu, J. M.-T., Hong, T.-P., Wang, L. S.-L., et al. (2016). Mining high-utility itemsets based on particle swarm optimization. *Engineering Applications of Artificial Intelligence, 55*, 320–330.

Liu, J., Wang, K., & Fung, B. (2012). Direct discovery of high utility itemsets without candidate generation. In *Proceedings of the IEEE 12th international conference on data mining (ICDM 2012)* (pp. 984–989). IEEE.

Liu, J., Wang, K., & Fung, B. C. M. (2016). Mining high utility patterns in one phase without generating candidates. *IEEE Transactions on Knowledge and Data Engineering, 28*(5), 1245–1257.

Liu, M., & Qu, J. (2012). Mining high utility itemsets without candidate generation. In *Proceedings of the 21st ACM international conference on information and knowledge management* (pp. 55–64). ACM.

Liu, Y., Liao, W.-k., & Choudhary, A. (2005a). A fast high utility itemsets mining algorithm. In *Proceedings of the 1st international workshop on utility-based data mining* (pp. 90–99). ACM.

Liu, Y., Liao, W.-k., & Choudhary, A. N. (2005b). A two-phase algorithm for fast discovery of high utility itemsets.. In *Advances in knowledge discovery and data mining: Vol. 3518* (pp. 689–695). Springer.

Peng, A. Y., Koh, Y. S., & Riddle, P. (2017). mHUIMiner: A fast high utility itemset mining algorithm for sparse datasets. In *Proceedings of the Pacific-Asia conference on knowledge discovery and data mining* (pp. 196–207). Springer.

Shie, B.-E., Hsiao, H.-F., Tseng, V. S., & Philip, S. Y. (2011). Mining high utility mobile sequential patterns in mobile commerce environments. In *Proceedings of the international conference on database systems for advanced applications* (pp. 224–238). Springer.

Tseng, V. S., Shie, B.-E., Wu, C.-W., & Yu, P. S. (2013). Efficient algorithms for mining high utility itemsets from transactional databases. *Knowledge and Data Engineering, IEEE Transactions on, 25*(8), 1772–1786.

Tseng, V. S., Wu, C.-W., Fournier-Viger, P., & Philip, S. Y. (2016). Efficient algorithms for mining top-k high utility itemsets. *IEEE Transactions on Knowledge and Data Engineering, 28*(1), 54–67.

Tseng, V. S., Wu, C.-W., Shie, B.-E., & Yu, P. S. (2010). UP-Growth: an efficient algorithm for high utility itemset mining. In *Proceedings of the 16th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 253–262). ACM.

Wang, J.-Z., Huang, J.-L., & Chen, Y.-C. (2016). On efficiently mining high utility sequential patterns. *Knowledge and Information Systems, 49*(2), 597–627.

Wu, C. W., Shie, B.-E., Tseng, V. S., & Yu, P. S. (2012). Mining top-k high utility itemsets. In *Proceedings of the 18th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 78–86). ACM.

Yao, H., Hamilton, H. J., & Butz, C. J. (2004). A foundational approach to mining itemset utilities from databases. In *Proceedings of the 4th SIAM international conference on data mining* (p. 482). SIAM.

Yin, J., Zheng, Z., Cao, L., Song, Y., & Wei, W. (2013). Efficiently mining top-k high utility sequential patterns. In *Data mining (ICDM), 2013 IEEE 13th international conference on* (pp. 1259–1264). IEEE.

Yun, U., Ryang, H., & Ryu, K. H. (2014). High utility itemset mining with techniques for reducing overestimated utilities and pruning candidates. *Expert Systems with Applications, 41*(8), 3861–3878.

Zaki, M. J. (2000). Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering, 12*(3), 372–390.

Zheng, Z., Kohavi, R., & Mason, L. (2001). Real world performance of association rule algorithms. In *Proceedings of the 7th ACM SIGKDD international conference on knowledge discovery and data mining* (pp. 401–406). ACM.

Zhu, F. (2014). Mining long patterns. In C. C. Aggarwal, & J. Han (Eds.), *Frequent Pattern Mining* (pp. 83–104)). Springer.

Zida, S., Fournier-Viger, P., Lin, J. C.-W., Wu, C.-W., & Tseng, V. S. (2015). Efim: a highly efficient algorithm for high-utility itemset mining. In *Proceedings of the mexican international conference on artificial intelligence* (pp. 530–546). Springer.

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