Mining high-utility itemsets from a transaction database

By

Siddharth Dawar

Under the supervision of Dr. Vikram Goyal, Dr. Debajyoti Bera

Computer Science and Engineering

Indraprastha Institute of Information Technology Delhi

New Delhi– 110020

April, 2021

Mining high-utility itemsets from a transaction database

By

Siddharth Dawar

A Thesis

submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Computer Science and Engineering

Indraprastha Institute of Information Technology Delhi

New Delhi– 110020

April, 2021

Certificate

This is to certify that the thesis titled *Mining high-utility itemsets from a transaction database* being submitted by *Siddharth Dawar* to the Indraprastha Institute of Information Technology Delhi, for the award of the degree of Doctor of Philosophy, is an original research work carried out by him under my supervision. In my opinion, the thesis has reached the standard fulfilling the requirements of the regulations relating to the degree.

The results contained in this thesis have not been submitted in part or full to any other university or institute for the award of any degree or diploma.

April, 2021

Dr. Vikram Goyal

Dr. Debajyoti Bera

Indraprastha Institute of Information Technology Delhi

New Delhi 110020

Abstract

Technological advances have enabled organizations to store large amounts of data cost-effectively. Patterns hidden in such massive databases can be valuable to industries to gain actionable knowledge and promote their business. For example, a retail store can utilize information about the products frequently purchased together by its customers for shelf-space management and inventory management. Frequent itemset mining has been studied extensively by the research community to mine such patterns from a transaction database. A transaction represents the set of products purchased together by a customer in the above-mentioned example. An example of a frequent itemset can be products like milk and bread purchased together frequently by customers from a retail store. Frequent itemset mining assumes that the items present in a transaction database have equal importance. However, customers purchase products in different quantities, and products generate different profits for the retail store. The notion of mining high-utility itemsets was formulated by researchers to mine such a set of items. For example, high-utility itemset mining can extract the set of profitable products purchased by customers from a retail store.

High-utility itemset mining is a generalization of the frequent itemset mining problem that associates a positive weight with each item in a transaction. The utility of an itemset in a transaction is the sum of the weights associated with its items. High-utility itemset mining is a more challenging problem compared to frequent itemset mining as the utility measure is neither monotone nor anti-monotone, unlike for frequent itemset mining. During a search space exploration, if the frequency of an itemset is less than the minimum frequency threshold, the itemset and its supersets can not be frequent. However, the superset of a low-utility itemset can have a high-utility, and the subset of a high-utility itemset can be a low-utility itemset. So, the search space can not be pruned solely based on the utility of a partially explored itemset.

In this thesis, we analyze, "how can we improve the performance of exist-

ing high-utility itemset mining algorithms?" The existing algorithms for mining high-utility itemsets can be categorized into one-phase and two-phase algorithms. The two-phase tree-based algorithms for mining high-utility itemsets generate candidate high-utility itemsets in the first phase by constructing a tree data structure recursively, and compute the utility of candidate itemsets through another database scan in the second phase called the verification phase. It has been observed by researchers that the performance of such two-phase tree-based algorithms can be improved by reducing the number of generated candidates. We begin by proposing a novel tree data structure called UP-Hist tree that augments a histogram of item weights frequency and a two-phase tree-based algorithm called UP-Hist Growth. We compare the performance of UP-Hist Growth against the state-of-the-art two-phase tree-based algorithms on several benchmark sparse and dense datasets. Our results demonstrate that the UP-Hist Growth algorithm performs better than those algorithms.

We observe in our experimental study that the two-phase tree-based algorithms, including UP-Hist Growth, run out of memory on some of the dense datasets and the state-of-the-art list-based algorithms like HUI-Miner perform faster than the two-phase tree-based algorithms on dense datasets. We also observe that the tree-based algorithms generate candidates quickly in the first phase, but spend a lot of time in the verification phase. The list-based algorithms construct an inverted-list data structure for every itemset by intersecting the inverted-lists of its immediate subsets. The intersection operation can become a performance bottleneck for list-based algorithms, and list-based algorithms can also generate itemsets that are non-existent in the database during search-space exploration. To combat the limitations of these approaches, we propose a hybrid algorithm that can harness the benefits of both types of algorithms by combining any tree-based algorithm with a list-based algorithm to extract high-utility itemsets. As a case study, we construct two hybrid algorithms by joining two tree-based algorithms named UP-Hist Growth and UP-Growth+ with a list-based algorithm called FHM. Our experimental study validates that the hybrid algorithms have less total execution time compared to the existing two-phase tree-based algorithms on sparse and dense datasets. Additionally, the hybrid algorithms also perform better than the list-based algorithms on sparse datasets.

We observe that a faster high-utility itemset mining algorithm can be designed by augmenting the inverted-list data structure on the top of a tree data structure as it can reduce the amount of information stored within the tree and

also cost of the intersection operation during the search-space exploration. To implement this idea, we propose a tree structure called UT_Mem-tree that augments information compactly in a HashMap with each node of the tree and design the first “one-phase tree-based” algorithm called UT-Miner to mine high-utility itemsets. We also propose a mechanism to construct a lightweight projected database during the mining process for superior performance, especially for dense datasets. We also conduct experiments to compare the performance of UT-Miner with the state-of-the-art high-utility itemset mining algorithms. The results confirm that UT-Miner performs better than the state-of-the-art tree-based, list-based, and hybrid algorithms on sparse and dense datasets.

The existing algorithms and data structures for mining high-utility itemsets are designed for a specific utility function only. We explore the possibility of designing data structures and algorithms that can mine high-utility itemsets for any subadditive monotone utility function. In this scenario, the utility of an itemset in a transaction need not be the sum of its item utilities. We design tighter upper-bounds and algorithms that can mine high-utility itemsets for such utility functions. We believe that generalization of utility functions can be useful. To demonstrate this we identify an application of high subadditive-monotone utility itemsets to find active high-influential groups of users from a Twitter dataset for applications like viral marketing.

In this thesis, we have explored how to improve the state-of-the-art techniques in high-utility itemset mining. We designed tighter bounds to reduce the search space exploration and algorithms for a class of utility functions that can generalize the classical addition function used by the existing algorithms.

Dedication

This thesis is dedicated to my parents and my advisors, Dr. Vikram Goyal, and Dr. Debajyoti Bera, who helped me to continue and complete this research.

Acknowledgements

First and Foremost, I would like to express my sincere gratitude to my research supervisors, Dr. Vikram Goyal and Dr. Debajyoti Bera. Without their continuous support, motivation, valuable guidance, and consistent encouragement throughout my Ph.D. journey, this work would have never been completed.

I am much indebted to Dr. Tanmoy Chakraborty, Dr. V. Raghava Mutharaju, Dr. Chetan Arora, and Dr. Venkata M. Viswanath Gunturi for being a part of the internal review committee and their insightful comments which motivated me to widen my research from different perspectives. I want to express my immense gratitude for the invaluable feedback to the external examiners Dr. Bac Le, Dr. P. K. Reddy, and Dr. Vasudha Bhatnagar.

I would also like to thank Dr. Ganesh Bagler, Dr. Mukesh Mohania, Dr. Pravesh Biyani, Dr. Pushpendra Singh, Dr. Rahul Purandare, Dr. Rajiv Ratn Shah, Dr. Pankaj Jalote, and Dr. Samaresh Chatterji for guiding me during the different stages of my Ph.D. journey towards career opportunities.

I am profoundly grateful to the Indraprastha Institute of Information Technology for providing excellent infrastructure and research environment. I want to thank the Visvesvaraya Ph.D. scheme for Electronics and IT for providing me a research fellowship throughout my Ph.D.

Most importantly, none of this could have happened without my parents and friends who always motivated me to do better and trusted me a lot. I also express my regards to all those who supported me in any aspect during the completion of my Ph.D.

Previously Published Material

- S. Dawar, V. Goyal *"UP-Hist Tree: An efficient data structure for mining high utility patterns from transaction databases "*, International Database Engineering & Applications Symposium, 2015, pp 56-61.
- S. Dawar, V. Goyal, D. Bera *"A hybrid framework for mining high-utility itemsets in a sparse transaction database"*, Applied Intelligence, 2017, pp 809-827.
- S. Dawar, D. Bera, V. Goyal *"High-utility itemset mining for subadditive monotone utility functions"*, arXiv preprint (arXiv:1812.07208), 2018.
- S. Dawar, V. Goyal, D. Bera *"A one-phase tree-based algorithm for mining high-utility itemsets from a transaction database"*, arXiv preprint (arXiv:1911.07151), 2019.

Contents

Abstract **i**

Dedication **iv**

Acknowledgements **v**

Publications **vi**

List of Tables **x**

List of Figures **xii**

1 Introduction **2**

1.1 High-utility itemset mining 5

1.2 Research contributions 8

1.3 Thesis structure 10

2 Literature review **12**

2.1 Frequent itemset mining 13

2.2 High-utility itemset mining 20

2.3 Summary 27

3 UP-Hist Growth: A two-phase tree-based algorithm for mining high-

utility itemsets **28**
3.1 Our proposed UP-Hist Tree and utility estimates 29
3.1.1 Construction of a global UP-Hist tree 30
3.1.2 Construction of a local UP-Hist tree 34
3.2 UP-Hist Growth Algorithm 41
3.2.1 Complexity Analysis 42
3.2.2 An Illustrated Example 44
3.3 Experiments and Results 46
3.4 Summary 53

4 A hybrid algorithm for high-utility itemset mining **54**
4.1 Hybrid algorithm 56
4.1.1 Caveats and Optimizations 58
4.2 Case study: Integration of UP-Hist Growth and UP-Growth+ with FHM 61
4.3 Experiments and Results 64
4.4 Summary 71

5 A one-phase tree-based algorithm for mining high utility itemsets **73**
5.1 UT_Mem-tree Structure 74
5.1.1 The elements of a UT_Mem-tree 75
5.1.2 The construction of a UT_Mem-tree 76
5.1.3 Construction of a lightweight projected database through a local_lists 78
5.2 UT-Miner Algorithm 80
5.3 Experiments and Results 84
5.4 Summary 98

6 High-utility itemset mining for subadditive monotone utility functions 100

6.1 Problem Statement 102

6.1.1 Subadditive and monotone (SM) utility functions 102

6.1.2 High-utility itemset mining for SM functions (HUIM-SM) 104

6.2 Coverage: A graph-based utility function 105

6.3 Bounds for HUIM-SM 111

6.3.1 TU and TWU bounds 112

6.3.2 Exact-utility (EU) and Remaining-utility (RU) bounds . 113

6.4 Algorithms for HUIM-SM 115

6.4.1 List-based algorithm 115

6.4.2 Tree-based algorithm 119

6.4.3 Projection-based algorithm 120

6.5 Case Study of HUIM-SM on a Twitter dataset 121

6.6 Performance evaluation of HUIM-SM algorithms 127

6.7 Summary 134

7 Conclusion 135

7.1 Future Research Directions 138

References 141

List of Tables

1.1 Example database 4
1.2 Frequent itemsets for threshold (f) = 3 4
1.3 Example database 7

2.1 Example database 13

3.1 Example database 33
3.2 TWU of items 33
3.3 Reorganized transactions 33
3.4 Characteristics of real datasets 47
3.5 Candidate generation and verification time (sec) on Kosarak dataset 47

4.1 Example database 61
4.2 TWU of items 61
4.3 Characteristics of real datasets 65

5.1 Information stored with a node N of a UT_Mem-tree 75
5.2 Example database 77
5.3 TWU of items 78
5.4 local_list for the prefix { B } 79
5.5 local_list for the prefix { BA } 84

5.6 Characteristics of real datasets 85

6.1 Transaction database for coverage utility (ucov) 105

6.2 Comparison of ucov(X,T) + ucov(Y,T) with ucov(X∪Y,T) (Lemma 6.1) 109

6.3 Example (Lemma 1) 109

6.4 Example (Theorem 6.3) 110

6.5 Comparison of ucov(X,T) with ucov(X∪Y,T) (Theorem 6.4) . . 111

6.6 SMI-List of $\{A\}$ 116

6.7 SMI-List of $\{B\}$ 116

6.8 SMI-List of $\{AB\}$ 116

6.9 Characteristics of Twitter transaction dataset 122

6.10 Overlap between the top 100 patterns generated by fim, fcov, sum, and sumcov with ucov. 124

6.11 Statistics showing distribution of pattern length for top 100 patterns generated by fim, fcov, sum, sumcov, and ucov. 124

6.12 Statistics showing distribution of Co(·) for top 100 patterns generated by fim, fcov, sum, sumcov, and ucov. 124

6.13 Statistics showing distribution of f(·) for top 100 patterns generated by fim, fcov, sum, sumcov, and ucov. 125

6.14 Comparison of ucov(X,T) with sumcov(X,T) (Theorem 6.5) . . 126

6.15 Characteristics of real datasets 127

List of Figures

1.1 Search-space for I=$\{a, b, c, d, e\}$ [68] 5

2.1 Execution tree of Apriori algorithm [1]. 14

2.2 FP-tree construction [1] 16

3.1 Global UP-Hist tree 33

3.2 Performance evaluation (Time and number of candidates) for UP-Growth+, UP-Growth, IHUP-Growth and UP-Hist Growth on sparse datasets. The UP-Hist Growth and IHUP-Growth algorithms ran out of memory for the NyTimes dataset. The UP-Growth algorithm did not terminate execution for more than 24 hours on the NyTimes dataset for threshold less than 0.45 $\%$. . . 48

3.3 Memory consumption of UP-Growth+, UP-Growth, IHUP-Growth and UP-Hist Growth on sparse datasets. UP-Growth ran out of memory during the candidate generation phase on Retail and Kosarak datasets at lower thresholds. IHUP-Growth ran out of memory during the candidate generation phase on Kosarak dataset. IHUP-Growth also ran out of memory on Retail dataset for threshold less than 0.002$\%$. The UP-Hist Growth and IHUP-Growth algorithms ran out of memory for the NyTimes dataset. 49

3.4 Performance evaluation (Time and number of candidates) for UP-Growth+, UP-Growth, IHUP-Growth, and UP-Hist Growth on dense datasets. All tree-based algorithms did not terminate their execution for more than 24 hours on Chess and Connect datasets. 50

3.5 Memory consumption of UP-Growth+, UP-Growth, IHUP-Growth, and UP-Hist Growth on dense datasets. All tree-based algorithms ran out of memory during the candidate generation phase for Accidents dataset at 2% threshold. 51

3.6 Scalability experiment on ChainStore and Accidents dataset for 0.01% and 2% threshold respectively. All tree-based algorithms did not terminate for more than 24 hours for the Accidents dataset. 51

4.1 Utility-list of itemset $\{E\}$, $\{B\}$ and $\{EB\}$ 61

4.2 EUCS data structure 62

4.3 Global UP-Hist tree 63

4.4 Performance evaluation (Time and number of candidates) on sparse datasets. HUI-Miner and FHM did not terminate for more than 24 hours on the Kosarak dataset for threshold less than 0.7%. 66

4.5 Memory consumption on sparse datasets. The UPHist-Hybrid and the UP-Hist Growth algorithms ran out of memory on the NyTimes dataset. 67

4.6 Performance evaluation (Time and number of candidates) on dense datasets. UP-Growth+ and UP-Hist Growth did not terminate for more than 24 hours on the Accidents dataset at threshold less than 10%, and 8% respectively. The UPHist-Hybrid algorithm ran out of memory on the Accidents dataset. All algorithms did not terminate their execution for more than 24 hours on the Connect dataset. 68

4.7 Memory consumption on dense datasets. The UPHist-Hybrid algorithm ran out of memory on the Accidents dataset. 68

4.8 Scalability experiment on ChainStore and Accidents dataset for 0.01% and 2% threshold respectively. All algorithms did not terminate for more than 24 hours on the Accidents and NyTimes datasets. 70

5.1 Global UT_Mem-tree with gmap associated with each node . . 78

5.2 Performance evaluation (Time and number of candidates) for FHM, mHUIMiner, UPG+-Hybrid and UT-Miner on sparse datasets. FHM did not terminate for more than 24 hours on Kosarak dataset for threshold value less than 0.7 % and mHUIMiner did not terminate on Kosarak dataset for threshold value less than 0.75 %. 86

5.3 Performance evaluation (Time and number of candidates) for FHM, mHUIMiner, UPG+-Hybrid, and UT-Miner on dense datasets. FHM, mHUIMiner, and UPG+-Hybrid did not terminate for more than 24 hours on Connect dataset. FHM, mHUIMiner, and UPG+-Hybrid did not terminate for more than 24 hours on Accidents dataset at 2% threshold. 87

5.4 Memory Consumption by FHM, mHUIMiner, UPG+-Hybrid and UT-Miner on sparse and dense datasets. 88

5.5 Performance evaluation (Time and number of candidates) for EFIM, d^2HUP, HMiner and UT-Miner on sparse datasets. d^2HUP did not terminate for more than 24 hours on Kosarak dataset for threshold less than 0.7 %. 89

5.6 Memory Consumption by EFIM, d^2HUP, HMiner and UT-Miner on sparse datasets. 91

5.7 Performance evaluation (Time and number of candidates) for EFIM, d^2HUP, HMiner and UT-Miner on dense datasets. d^2HUP did not terminate for more than 24 hours on Connect dataset and Accidents dataset at 2% threshold. 92

5.8 Memory Consumption by EFIM, d^2HUP, HMiner and UT-Miner on dense datasets. 93

5.9 Scalability experiment on ChainStore and Accidents dataset for 0.01% and 2% threshold respectively. FHM, mHUIMiner, d^2HUP, and UPG+-Hybrid did not terminate for more than 24 hours on Accidents dataset when more than 60% of the transactions is input to the algorithms. 94

5.10 Memory consumption on ChainStore and Accidents dataset for scalability 94
5.11 Comparison of UT-Miner with UT-Miner-LT on sparse datasets. 95
5.12 Comparison of UT-Miner with UT-Miner-LT on dense datasets. 96

6.1 Graph over items 106
6.2 Performance evaluation on sparse datasets for $ucov$. 128
6.3 Number of candidates and memory consumption on sparse datasets for $ucov$. 129
6.4 Performance evaluation on dense datasets for $ucov$. 130
6.5 Number of candidates and memory consumption on dense datasets for $ucov$. 131

Chapter 1

Introduction

We are living in the age of big data where data is being generated in huge volumes at a very fast pace from different sources like satellites, sensors, wifi hotspots, the world wide web, social media platforms, retail giants like Walmart, Amazon, etc. It is difficult to manually analyze such an enormous amount of data for extracting actionable insights in a timely manner. Hence, there is a need for automated techniques to gather insights from such data. Data mining, also called as knowledge discovery from data (KDD) [38] is the process of discovering interesting patterns and knowledge from large amounts of data. The knowledge discovery pipeline has different phases starting from data collection and cleaning, data integration, data transformation, data mining, and data visualization. Data mining includes techniques like classification [19], clustering [3], association rule mining [4], and outlier detection [2]. In this thesis, we focus on the problem to mine high-utility patterns that can find interesting associations among items from a transaction database. We start by discussing an application of association rule mining on a retail store transaction database below.

Association rule mining [4] generates interesting rules from a transaction database, and the extracted rules can be used for recommending products to a customer in a retail store. Every transaction contains the set of products purchased by a customer from a retail store. Association rule mining algorithms can be applied to the customer transaction database of any retail giant like Walmart to generate rules that can be used for inventory management, shelf-space management, and for recommending products to a customer. The application of association rule mining in retail domain is popularly called market-basket analysis [4] as the retail basket of customers is analyzed to find interesting associations. An example of an association rule might be that 90 $\%$ of the customers who buy bread also purchase butter.

The association rules are generated from a transaction database through a two-step process, In the first step, frequent itemset mining [4] is applied on a transaction database to generate frequent itemsets and the association rules are generated from frequent itemsets in the next step. Frequent itemset mining finds itemsets from a transaction database with a frequency no less than a user-defined minimum frequency threshold denoted by f. An example transaction database is shown in Table 1.1 and frequent itemsets for a minimum frequency threshold (f) equal to 3 is shown in Table 1.2. An example of an itemset can be the set of products, bread, butter, and milk purchased together frequently from a retail store. Frequent itemset mining has several applications outside of market-basket analysis, e.g., in classification [19], clustering [3], bug mining [46], biclustering [41], pattern mining for location prediction [56], outlier detection [81] etc.

Table 1.1: Example database

Transaction	Items
1	A C T W
2	C D W
3	A C T W
4	A C D W
5	A C D T W
6	C D T

Table 1.2: Frequent itemsets for threshold (f) = 3

Frequency	Itemsets
6	C
5	W, CW
4	A, D, T, AC, AW, CD, CT, ACW
3	AT, DW, TW, ACT, ATW, CDW, CTW, ACTW

The search-space for a set of I items contains $2^{|I|} - 1$ itemsets. For example, the itemset lattice for I=$\{a, b, c, d, e\}$ is shown in Figure 1.1. A brute-force approach is to enumerate the complete itemset lattice and find out the frequent itemsets by computing their frequency through a database scan. However, the brute force approach results in exploring an exponential number of itemsets. Several data structures and algorithms [4, 40, 80, 59] were proposed to prune the search space and extract frequent itemsets from a transaction database. The research on frequent itemset mining led to the expansion of pattern mining algorithms to mine complex patterns like a sub-sequence, substructure, etc. A sub-sequence pattern can be purchase of a computer followed by that of a printer. An example of a substructure can be a motif, i.e., a subgraph frequently occurring in a database composed of chemical compounds. The problem of frequent pattern mining can be categorized into frequent itemset mining [4, 40, 80], frequent sequence pattern mining [60, 5, 39], frequent episode mining [84, 70, 55], frequent pattern mining from a data stream [45, 54, 15], and frequent subgraph mining [77, 30, 57] based on the nature of the pattern and database.

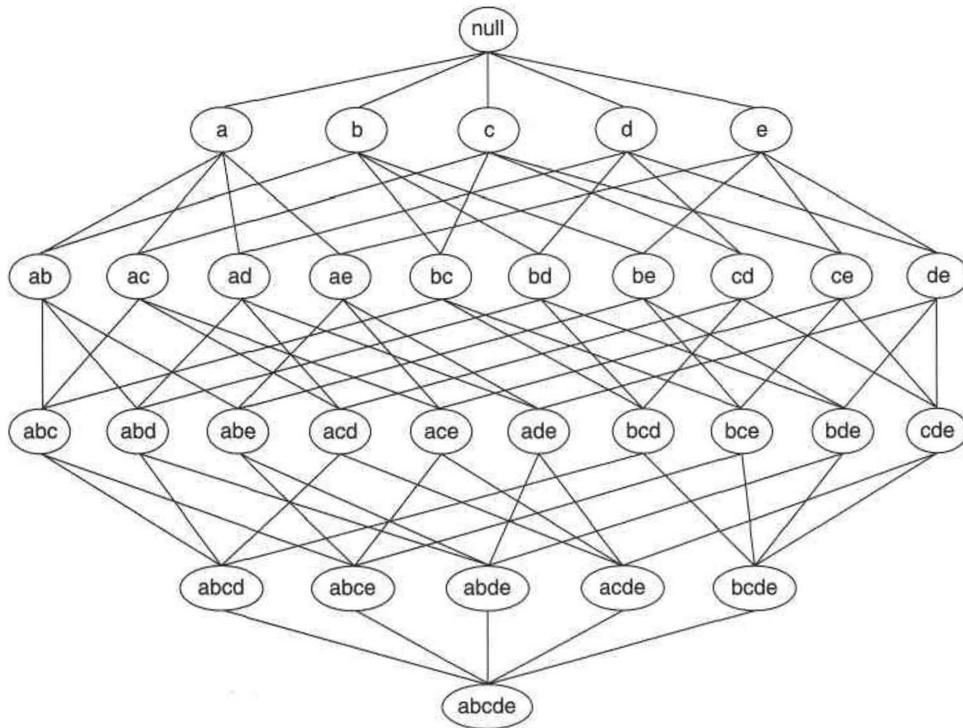

Figure 1.1: Search-space for I={a, b, c, d, e} [68]

1.1 High-utility itemset mining

There are real-life applications where items in a transaction have different importance and have positive weights associated with them. Frequent itemsets can not capture such interesting patterns as frequent itemset mining assumes the binary presence/absence of items in a transaction, and all items are given equal importance. The notion of high-utility itemset mining was designed to capture such an interesting set of items with the weights defined from an application domain. Consider the application of market-basket analysis from the customer transaction logs of a retail store. Customers purchase a product/item in some quantity, and every sold item generates different profit for the retail store. For example, a customer can buy two units of bread and one unit of butter, and the

retail store owner earns a different profit on selling every unit of bread and butter. The weight associated with an item in a transaction can be defined as the product of its quantity and profit. High-utility itemset mining can find the set of products [71] that generate a total profit more than a user-defined threshold when purchased together.

High-utility itemset mining has been applied to mine interesting groups of items from different data sources like gene expression microarray data [52], and spatio-temporal data [43]. A gene expression microarray data can be converted to a transaction database with every transaction created according to a timestamp, and genes with its state as items. The upregulation and downregulation of genes are considered as separate items. The quantity associated with a gene in a transaction is a function of the number of samples that contain the gene with its associated state. The importance of a gene is defined by its neighbors from a gene co-expression network. Liu et al. [52] applied high-utility itemset mining to find the set of genes with different expression levels under two different experimental conditions from human and mice datasets. Kiran et al. [43] applied high-utility itemset mining to mine spatio-temporal high-utility itemsets for the application of congestion detection and pollution monitoring from Tokyo datasets. The transaction database was constructed from the temporal data mined by different sensors, and weights were associated with each item in the transaction according to the application. An additional constraint of proximity among the items extracted from the high-utility itemset mining algorithms was applied to ensure the spatial proximity of the mined patterns.

Table 1.3: Example database

TID	Transaction	TU
T_1	$(B:4)\,(C:4)\,(E:3)\,(G:2)$	13
T_2	$(B:8)\,(C:13)\,(D:6)\,(E:3)$	30
T_3	$(A:5)\,(C:10)\,(D:2)$	17
T_4	$(A:30)\,(B:2)\,(C:1)\,(D:8)\,(H:2)$	43
T_5	$(A:10)\,(C:6)\,(E:6)\,(G:5)$	27
T_6	$(A:10)\,(B:4)\,(D:12)\,(E:6)\,(F:5)$	37

Now we define the problem statement for mining high-utility itemsets formally.

Problem Statement: Consider a set of items $I = \{i_1, i_2, ..., i_m\}$ and a transaction database $DB = \{T_1, T_2, ..., T_n\}$ where every transaction is a subset of I. Every item in a transaction is associated with a positive weight. The utility of an item i in a transaction T, denoted by $u(i, T)$, is the weight associated with the item in T. The utility of an itemset X in a transaction T denoted by $u(X, T)$ is the sum of utility of its items. The utility of itemset X in the database is defined as: $u(X) = \sum_{\substack{X \subseteq T \\ T \in D}} u(X, T)$. An itemset X is called a high-utility itemset if $u(X)$ is no less than a given minimum user-defined threshold denoted by θ. Given a transaction database D, and a minimum user-defined threshold θ, the aim is to find all high-utility itemsets.

For example, consider the transaction database shown in Table 1.3. In our example, $I = \{A, B, C, D, E, F, G, H\}$. The utility of item $\{B\}$ in T_1 is 4. The utility of itemset $\{BC\}$ in T_1 is 8. The utility of the itemset $\{AD\}$ in our example database is 67. The set of high-utility itemsets for θ equal to 50 are $\{AD\}$:67, $\{AC\}$:62, $\{ABD\}$:66, $\{ACD\}$:56, and $\{A\}$:55 respectively.

1.2 Research contributions

High-utility itemset mining is a challenging problem as the search space is exponential in the number of items. It is a harder problem compared to frequent itemset mining as the utility can not be used to prune the search space, unlike frequency. The frequency of an itemset decrease as more items is added to the current itemset. During the search space exploration, if the frequency of an itemset X is less than the minimum frequency threshold, itemset X and its supersets can not be frequent. However, it is not true for the case of high-utility itemset mining algorithms.

There is a need to improve the performance of the existing algorithms for mining high-utility itemsets. We observed that the existing algorithms did not terminate for several days at low utility thresholds on datasets like NyTimes [29], and PubMed [29]. The existing data structures and algorithms are designed to mine itemsets for a specific utility function only. We ask the following questions:

- Can we improve the performance of the existing high-utility itemset mining algorithms?
- Can we design a high-utility itemset mining algorithm for a class of utility functions that can generalize the classical addition function used by the existing algorithms?
- Will the existing bounds designed to reduce the search space exploration

still work?

- Can we demonstrate the significance to mine high-utility itemsets for a class of utility functions through an application on a real-life dataset?

Motivated by these questions, our contributions are the following:

- We propose a novel tree data structure called UP-Hist tree and an algorithm called UP-Hist Growth. Our experimental study validates that UP-Hist Growth is faster in terms of total execution time and generates fewer candidates compared to the state-of-the-art two-phase tree-based algorithms.
- We design a hybrid algorithm that can combine any tree-based algorithm like UP-Growth+ [71] with a list-based algorithm like FHM [32]. We discuss caveats that must be taken into account while designing a hybrid algorithm and techniques to improve the performance of a hybrid algorithm. We present a case study to understand the integration of UP-Growth+ [71] and UP-Hist Growth [21] with FHM [32]. Our experimental study shows that the hybrid algorithms perform better than the state-of-the-art list-based algorithms on sparse datasets and tree-based algorithms on both sparse and dense datasets.
- We propose a novel data structure called UT_Mem-tree and the first one-phase tree-based algorithm called UT-Miner that performs better in terms of total execution time compared to the state-of-the-art tree-based, list-based, and hybrid algorithms across dense and sparse datasets as validated

from our experimental study. We also propose a mechanism to reduce the cost of recursive data structure created during the mining process.

- We propose an inverted-list data structure and an algorithm to mine high-utility itemsets for any arbitrary monotone utility function. We also design a subadditive monotone utility function to find the groups of active and influential users from a Twitter dataset that can influence their followers for applications like viral marketing. Our experimental study suggests that the computation of utility can also play an important role in deciding the relative performance of algorithms belonging to different categories across several sparse and dense datasets.

1.3 Thesis structure

This thesis is organized as follows. In chapter 2, we further review the data structures and algorithms proposed for itemset mining. In chapter 3, we propose a data structure called the UP-Hist tree, and a two-phase algorithm called UP-Hist Growth. We propose a hybrid algorithm that can combine any tree-based algorithm with a list-based algorithm to mine itemsets in one-phase only in chapter 4. In chapter 5, we propose a data structure called UT_Mem-tree and an algorithm called UT-Miner that avoids the construction of complete local trees to improve the total execution time and memory consumed during the mining process. In chapter 6, we formulate the problem of mining high-utility itemsets for the class of subadditive monotone functions. We further propose an

inverted-list data structure and an algorithm to mine high-utility itemsets. We also discuss the changes required in the tree-based and projection-based algorithms to adapt them for mining itemsets for subadditive monotone utility functions. We demonstrate the application of subadditive monotone utility functions by designing a function to capture the active group of influential users from a publicly available Twitter dataset. We conclude this thesis by summarizing the contributions and highlighting several directions for future research in the area of high-utility pattern mining in chapter 7.

Chapter 2

Literature review

The algorithms for itemset mining can be divided into two categories: one-phase and two-phase algorithms. The two-phase algorithms generate candidate high-utility itemsets in the first phase and scan the database again to compute the utility of candidate itemsets in the next phase. The one-phase algorithms generate candidates and compute their utility in a single phase. Several data structures and algorithms have been proposed to mine frequent and high-utility itemsets from a transaction database. These data structures are designed to reduce the total execution time, the number of candidate itemsets explored during the search process, and the memory consumption. We start by reviewing the fundamental concept of association rules and frequent itemset mining in Section 2.1. The data structures and algorithms proposed for high-utility itemset mining are discussed in Section 2.2.

Table 2.1: Example database

TID	Transaction
1	a b c d e
2	a b c d f h
3	a f g
4	b e f g
5	a b c d e h

2.1 Frequent itemset mining

Several algorithms have been proposed for frequent itemset mining. These algorithms explore the search space in different order to find frequent itemsets and several data structures were designed to improve the time and space efficiency. Now, we will discuss some data structures and algorithms proposed to mine frequent itemsets from a transaction database.

Apriori: Agrawal et al. [4] coined the concept of association rules for a sales transactions database. An example of an association might be that 90 $\%$ of the customers who buy bread also purchase butter. An association rule is an implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \bigcap Y = \phi$. Two measures called support and confidence were defined to capture interesting association rules from a transaction database. The rule $X \Rightarrow Y$ has a support s in a transaction database DB if $s\%$ of the transactions contain the itemset $X \bigcup Y$. The rule $X \Rightarrow Y$ has a confidence c if $c\%$ of the transactions in DB that contain X also contain Y. The problem is to extract association rules from a transaction database with support and confidence no less than a user-defined minimum support ($minsup$) and confidence ($minconf$) threshold. The association rules

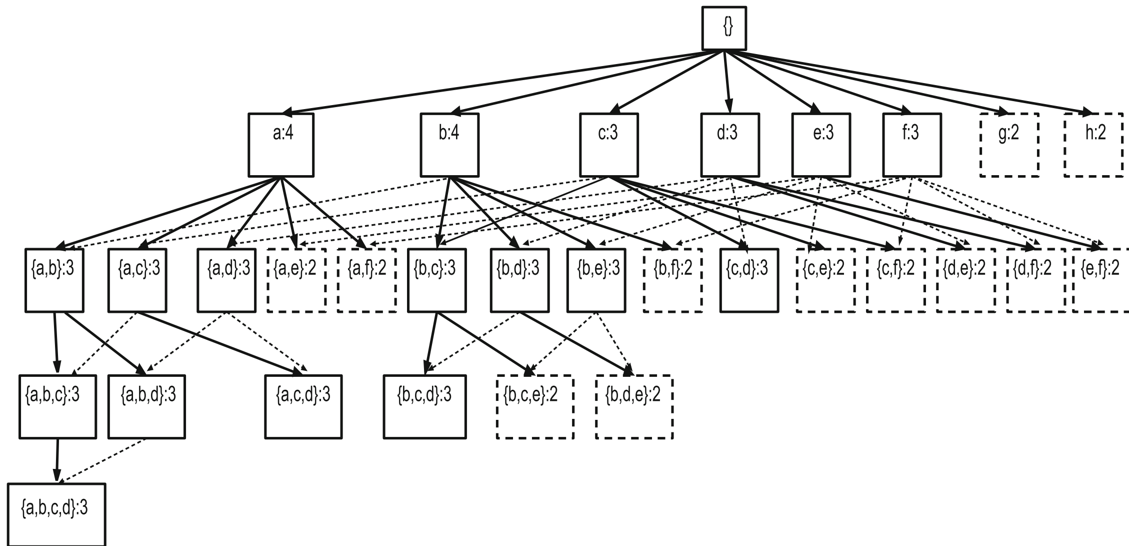

Figure 2.1: Execution tree of Apriori algorithm [1].

can be mined from a transaction database through a two-step process.

The first step is to extract frequent itemsets from DB that have support no less than $minsup$. The next step is to form association rules from the frequent itemsets and remove the association rules with confidence less than $minconf$. An algorithm called *Apriori* was proposed to mine frequent itemsets. The *Apriori* algorithm follows the candidate generation and verification paradigm and explores the search space in a breadth-first or level-wise manner. Consider a transaction database as shown in Table 2.1 and let $minsup$ be 3. The execution tree of Apriori algorithm is shown in Figure 2.1. The execution tree shows the candidate itemsets generated by the Apriori algorithm at different levels. The itemsets within a solid box are the frequent itemsets, and the itemsets within a black box are the candidate itemsets in Figure 2.1. The itemsets within a dotted box are infrequent. Initially, the algorithm scans the database to compute the support for the itemsets of length one shown at the first level of the tree. Only

the items g and h are identified as infrequent at the first level. The candidate itemsets at level two are generated by joining the frequent items from the first level. Another database scan is performed by the Apriori algorithm to find the itemsets that are frequent at the second level. For example, the itemset $\{ab\}$ is generated by combining the frequent itemsets $\{a\}$ and $\{b\}$ from the first level. The Apriori algorithm proceeds further in a similar manner. If the length of longest frequent itemset is k, this algorithm performs k scans of the database. This algorithm uses the downward closure property for pruning the search space. The downward closure property states that the subsets of an frequent itemset are frequent. It can be observed that if an itemset of length l is infrequent, its supersets will be infrequent too as the support measure monotonically decreases with the length of an itemset. For example, the itemset $\{ce\}$ is infrequent in our example database. Therefore, the itemset $\{ce\}$, and its supersets will be infrequent too.

FP-Growth: The *Apriori* algorithm scans the database multiple times to find the frequent itemsets during its breadth-first search. Now, we will discuss an algorithm called *FP-Growth* that can mine frequent itemsets in two database scans without the requirement of a separate verification phase, unlike the Apriori algorithm. Han et al. [40] proposed a new data structure called *FP-tree* that stores the transaction database compactly in the form of a tree. An *FP-tree* is built to avoid scanning the database multiple times for computing support of candidate itemsets. Initially, the FP-Growth algorithm performs a database scan to find the frequent items. The infrequent items are removed from every trans-

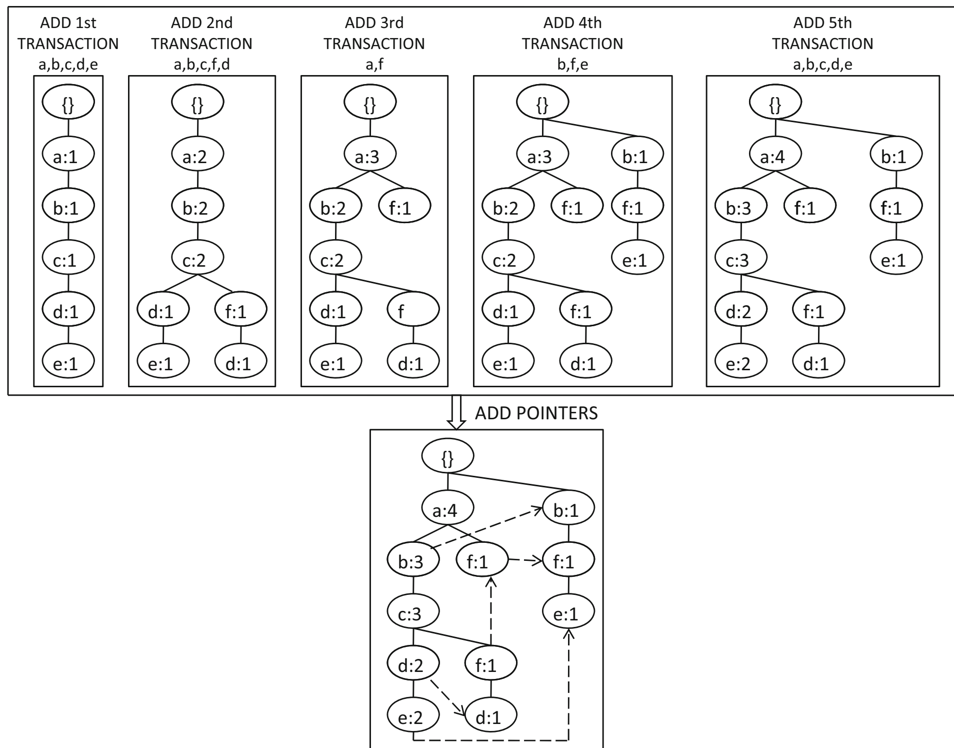

Figure 2.2: FP-tree construction [1]

action, and the items are sorted in decreasing order of support. The transactions are inserted one by one to construct an *FP-tree*. For example, the *FP-tree* for our example database (Table 2.1) and $minsup$ equal to 3 is shown in Figure 2.2. A *FP-tree* contains a special node called root that points to its child nodes. The remaining nodes in the tree represents an item and a path from the root to a node N represents a set of transactions along that path. Each node in an FP-tree contains the following fields: name, support, link to the child nodes and a link to the parent node. The name field of a node n represents the identifier of the item and support value indicates the number of transactions along the path which contain item n. The *FP-Growth* is a recursive depth-first search algorithm which generates frequent itemsets without candidate generation as the support of itemsets

can be computed from the local FP-trees generated during the mining process.

Eclat and dEclat: The *Apriori* and *FP-Growth* algorithms view a database horizontally i.e. a database has transactions and each transaction contains a set of items. Now, we will discuss an algorithm that views a transaction database vertically i.e. for every itemset, a list of transaction identifiers containing the itemset is maintained. Zaki [80] proposed an algorithm named *Eclat* that constructs *Tid-list* data structure associated with every itemset. A *Tid-list* stores a list of transaction identifiers that contains its associated itemset. The support on an itemset can be computed by counting the number of transaction identifiers contained in its *Tid-list*. Initially, the algorithm scans the database to compute the *Tid-list* of single items. The algorithm explores the search space in a breadth-first manner. The frequent items are identified and *Tid-list* of pairs is computed by intersecting the list of individual items. If an itemset is found to be infrequent, its supersets are not explored. The *Tid-list* of k length itemset is computed by interesting the lists of its $\{k-1\}$ length subsets. The advantage of vertical mining based algorithm is the use of simple set intersection operation to compute the support of an itemset. A problem with the vertical mining approach is that the size of *Tid-lists* can be large and therefore the intersection operation is costly. Zaki et al. [82] proposed another representation called diffsets for storing the *Tid-lists* in a compressed form to reduce the memory consumption. The authors proposed an algorithm called dEclat that uses diffsets as a data structure to mine frequent itemsets.

H-Mine: Pei et al. [59] proposed a novel data structure called H-struct, and an

algorithm called H-Mine to mine frequent itemsets. The H-Struct data structure constructs a header table similar to the FP-Growth [40] algorithm and augments a support count and a hyper-link structure to each item. The H-Mine algorithm utilizes the H-Struct data structure to mine frequent itemsets.

FEM: Vu et al. [73] proposed an algorithm called FEM that combines the techniques of *FP-Growth* and *Eclat*. The FEM algorithm constructs the FP-tree and starts the mining process by calling the FP-Growth algorithm. However, the FEM algorithms checks the length of linked-list associated with each item i in the header table. If the length is less than a threshold value K, the algorithm converts the local tree of item i to *Tid-lists* and *Eclat* algorithm is executed. Otherwise, the *FP-Growth* algorithm is executed. The idea is to run the *Eclat* algorithm on dense datasets as they have a smaller number of transactions and few distinct items compared to sparse datasets. A smaller number of transactions results in less cost for performing *Tid-list* intersection operations during the mining process and a small number of distinct items results in less number of generated itemsets. Vu et al. [74] proposed another method to compute dynamically the value of threshold K for switching between *FP-Growth* and *Eclat* algorithm.

Algorithms based on N-list and PPC-tree: Deng et al. [24] proposed a data structure called N-list and a prefix tree structure called PPC-tree like FP-tree [40] to mine frequent itemsets. Each node of the PPC-tree stores the pre-order and post-order rank in addition to the information stored by a FP-tree node. The PPC-tree is a lightweight tree that does not have a header table and a node-link

field with each node. The PPC-tree is constructed only to generate the pre-order and post-order rank with each node. The N-list data structure for a node N is defined as a list of tuples, where each tuple stores a PP-code of the form $\langle (N.pre-order, N.post-order) : count \rangle$. The pre-order and post-order ranks are stored with each node to check for ancestor-descendant relationship during the execution of their proposed algorithm called Prepost. Initially, the N-lists of frequent 1-itemsets is constructed and a PPC-tree is built. The algorithm finds the frequent 2-itemsets by traversing the PPC-tree. The frequent $k+1$-itemsets where $k \geq 2$ are generated by intersecting the $N-lists$ of frequent k-itemsets and the intersection operation is performed with linear time complexity. Deng et al. [27] proposed another algorithm called PrePost+ that employs a strategy called children-parent equivalence pruning to reduce the search space. Deng et al. proposed two algorithm called FIN [26] and dFIN [25] based on data structures similar to N-list and PPC-tree for mining frequent itemsets. Aryabarzan et al. [8] proposed a data structure called NegNodeSet similar to the N-list data structure that employs a novel encoding scheme with bitmap representation and an algorithm called negFIN to mine frequent itemsets. The authors compared the performance of the negFIN algorithm with FP-Growth* [34], Eclat [80], and dFIN [25]. Their experimental study showed that negFIN is the fastest algorithm on all datasets with different minimum support thresholds. However, the dFIN [25] algorithm performed similar to negFIN [8] on some datasets.

2.2 High-utility itemset mining

Frequent itemset mining assumes the binary presence/absence of items in a transaction and the items within a transaction have equal importance. However, items within a transaction can have positive weights associated with them. Consider the customer transaction logs from a retail store. The items within a transaction can be defined as the set of products purchased by a customer. It can be observed in real life that customers purchase a product in some quantity/copies and selling a product generates some profit for the retail store. For example, someone may buy six boxes of DVDs, one video player from a store and furthermore, the store will not make same profit with each item. The positive weight associated with every item in a transaction is defined as the product of its quantity and profit in the retail domain application. The notion of weighted association rule mining (WARM) [13, 75, 69, 10] and utility mining [64, 78, 79] models were defined to capture interesting association rules with measures apart from support and confidence.

It can be observed that frequent itemset mining is a special case of high-utility itemset mining, where the utility of an itemset in a transaction is always equal to one. However, the utility measure is neither monotone nor anti-monotone. The super-set of a low-utility itemset can have utility more than θ and vice-versa can also be true. Liu et al. [50] defined an upper-bound called transaction-weighted utility (TWU) [50] that satisfies the anti-monotonicity property. The transaction utility (TU) for a transaction is defined as the sum of utili-

ties of its items. The transaction-weighted utility of an itemset X is the sum of TU of transactions that contain X. If the TWU of an itemset is less than θ, neither the itemset nor its super-sets can have utility more than θ. This property is known as the transaction-weighted downward closure property (TWDCP) [50]. The TWDCP property holds true as the supersets of an itemset will be contained in a subset of transactions that contains it.

Now, we will discuss some of the algorithms for high-utility itemset mining below.

Two-Phase: Liu et al.[51] proposed a two-phase algorithm to mine high-utility itemsets. In the first phase, a set of candidate high-utility itemsets is generated through a level-wise search of the itemset lattice. The authors observed that only the $\{k-1\}$-itemsets that have a TWU no less than the mininum utility threshold can be combined and added into the set of candidates for the kth level. The database is scanned again in the second phase in order to filter out the high-utility itemsets from the high TWU itemsets.

IHUP-Growth: The Two-phase [51] algorithm discussed above scans the database multiple times to generate candidate high-utility itemsets in the first phase and filters the high-utility itemsets in the last phase. Ahmed et al. [6] proposed the first tree-based two-phase recursive algorithm for mining high-utility itemsets. Three variants of a data structure called $IHUP-tree$ were proposed to incremental mining of high-utility itemsets that allows addition, deletion and modification of existing transactions. The $IHUP_L$-tree arranges the items in a

lexicographic order along every path from the root to a leaf node. Similarly, the $IHUP_{TF}$-tree and $IHUP_{TWU}$-tree sort the items in decreasing order of transaction frequency and TWU respectively. Every node of an $IHUP-tree$ stores the item name, transaction frequency, TWU, a pointer to the parent node, and a list of pointers to its children. The authors analyzed the effect of different item ordering strategies on the performance of their algorithm and observed that sorting items in every transaction by descending order of *TWU* value gives a better performance compared to lexicographic or transaction frequency ordering. They proposed an algorithm called IHUP-Growth [6] that generates candidates that have TWU no less than θ by recursively constructing local IHUP-trees and filters the high-utility itemsets through a database scan in the verification phase.

UP-Growth and UP-Growth+: Tseng et al.[72] proposed a data structure called UP-tree data structure and four strategies to reduce the number of candidates generated in the first phase compared to IHUP-Growth [6]. Each node N of the UP-tree stores the following information, N.name, support, node utility, a pointer to its parent node, a list of pointers to its child node, and a $hlink$ that points to a node whose item name is same as N.name. The node utility is an estimated upper-bound score associated with each node of the UP-tree. The UP-tree also maintains a header table to facilitate traversal of a UP-tree. The header table stores the item name, estimated utility value, and a link that points to the first occurrence of the node which has the same name as the item name in the header table.

Now, we will discuss the proposed strategies to compute a reduced estimated

utility compared to the TWU measure with each node of the UP-tree. Initially, the transaction database is scanned to compute the TWU for every distinct item present in the database. This paper defines an item as unpromising if its TWU is less than the minimum utility threshold. Such items can not be a part of any high-utility itemset due to the transaction-weighted downward closure property. Such items can be removed from the transaction database and this strategy is called the "discarding global unpromising items (DGU)" strategy. The tree-based algorithms like FP-Growth [40] grow a prefix in a bottom-up manner from the leaf towards the root node during the execution of a pattern-growth algorithm. So, the utilities of a node descendants can be removed from its node utility during the construction of a global UP-tree from the transaction database. This strategy is called "discarding global node utilities (DGN)". The remaining strategies called the "discarding local unpromising items (DLU)" and the "discarding local node utilities (DLN)" are similar to DGU, DGN and are applied during the construction of local UP-trees during the mining process.

The process to construct a global UP-tree from the transaction database and local UP-trees is similar to the one followed for the FP-tree [40], and IHUP-tree [6] construction. The utility of items can not be used in the DLU and DLN strategies as it is not available during the generation of local UP-trees. A table called minimum item utility is maintained that stores the minimum utility of global promising items in the database. The authors proposed an algorithm called UP-Growth [72] that constructs the UP-tree data structure to mine high-utility itemsets. Tseng et al. [71] proposed another algorithm called UP-Growth+ that

maintains a minimum node utility in every node of the UP-tree and uses it during the DLU and DLN strategy instead of the minimum item utility table. The minimum node utility associated with a node N is its minimum utility in the set of transactions present in the path from the node N to the root in a UP-tree.

HUI-Miner and FHM: The two-phase [51] and the tree-based [72, 71] algorithms suffer from the computational bottleneck observed in algorithms based on the candidate generation and verification framework. Liu et al. [49] proposed a data structure called utility-list and a list-based algorithm called *HUI-Miner* that can extract high-utility itemsets in one-phase only. The utility-list structure keeps enough information to compute the utility of an itemset generated during the search space exploration. The utility-list associated with an itemset X stores tuples of the form $\langle(Tid, EU, RU)\rangle$. The Tid stores the transaction identifier that contains the itemset X. The exact-utility(EU) is the utility of X in a transaction. The HUI-Miner algorithm sorts the items in the transaction database according to the ascending order of their TWU. The remaining-utility (RU) stores the sum of utilities of remaining items after X in a transaction.

The HUI-Miner algorithm explores the search space in a depth-first search fashion and constructs the utility-list of 1-itemsets before starting the mining process. The utility-list of 2-itemsets are generated by intersecting/joining the utility-lists of the items present in the itemset based on their common transaction identifiers. The utility-list of a k-itemset is generated by intersecting the utility-lists of two $\{k-1\}$-itemsets with the utility-list of the prefix itemset. A typical problem associated with list-based algorithms is the costly operation of

intersecting the utility-lists. Viger et al. [32] proposed a novel data structure *EUCS* and an algorithm *FHM* to reduce the number of intersection operations. The *EUCS* structure maintains the TWU for every pair of items and the FHM algorithm generates utility-list of a new itemset only if the TWU of the item of the current itemset and last appended item to the prefix is greater than the minimum utility threshold.

mHUIMiner: Peng et al. [61] proposed an algorithm called mHUIMiner that constructs an IHUP-tree [6] to avoid generating candidates that are non-existent in the database during the invocation of the HUI-Miner [49] algorithm.

EFIM: Zida et al. [85] proposed an algorithm called EFIM that constructs a projected database in the form of transactions only during the mining process and performs transaction merging to reduce the size of the projected database in every recursive invocation of the algorithm. EFIM is the state-of-the-art algorithm on dense datasets that have very similar transactions with a higher average transaction length with fewer items.

d^2**HUP:** EFIM [85] is not the state-of-the-art algorithm on sparse datasets as it performs binary search during the creation of projected database and transaction merging works better for dense datasets. Liu et al. [48] proposed a hyperlink data structure called CAUL and an algorithm called d^2HUP bundled with strategies like lookahead pruning to efficiently mine high-utility itemsets from sparse datasets. d^2HUP is the state-of-the-art algorithm on sparse datasets.

HMINER: Krishnamoorthy [44] proposed an algorithm called HMINER that

constructs a compact utility-list data structure and mines high-utility itemsets by integrating techniques like transaction merging, lookahead pruning etc. to reduce the search space and total execution time of the algorithm.

Apart from itemset mining, the research community has contributed by embedding constraints into the pattern mining process and this area of research is called constraint pattern mining [58, 42, 66, 23, 37]. Guns et al. [36] introduced a declarative framework named MiningZinc for constraint-based data mining. To the best of our knowledge, MiningZinc is the first framework that can express the high-utility itemset mining problem. MiningZinc uses a cover function that returns the set of transaction identifiers containing an itemset and the actual data to compute its utility. MiningZinc expresses the itemset mining problem over integer and set variables with no explicit data structures. Coussat et al. [20] defined the problem of high-utility itemset mining in uncertain tensors, and showed that an algorithm called multidupehack [14] could be deployed to mine itemsets. The authors studied a generalized version of high-utility itemset mining problem where the utilities can be positive and negative and are not restricted to matrices. It can be an interesting research direction to integrate the constraint pattern mining techniques into the existing itemset mining framework. Interested readers can refer to some recent survey papers [65, 35] for a global view on the integration of constraints in pattern mining.

2.3 Summary

In Section 2.1, we discussed the concept of association rules, frequent itemsets, data structures, and algorithms to mine frequent itemsets. In Section 2.2, we presented a literature review to discuss the data structures and algorithms for mining high-utility itemsets. In the next chapter, we propose a tree data structure called UP-Hist tree, and a two-phase tree-based algorithm called UP-Hist Growth to mine high-utility itemsets.

Chapter 3

UP-Hist Growth: A two-phase tree-based algorithm for mining high-utility itemsets

In this chapter, we propose a data structure called UP-Hist tree and an algorithm called UP-Hist Growth that generates fewer candidates in the first phase compared to existing two-phase tree-based algorithms. We propose a tighter score to estimate the utility of an itemset and its supersets that allows UP-Hist Growth to prune the search space better compared to other two-phase tree-based algorithms. We conduct extensive experiments on several benchmark sparse and dense datasets to compare the performance of UP-Hist Growth with other state-of-the-art two-phase tree-based algorithms. Our results validate the superior performance of UP-Hist Growth on several benchmark sparse and dense datasets.

This chapter is organized as follows. In Section 3.1, we define a histogram that can be stored with every node of our proposed UP-Hist tree structure and estimates to compute a correct bound on the utility of an itemset and its supersets.

We also discuss the process to construct a UP-Hist tree from the transaction database. In Section 3.2, we propose the UP-Hist Growth algorithm. The experimental results are presented in Section 3.3 and Section 3.4 summarizes the research contributions of this chapter.

3.1 Our proposed UP-Hist Tree and utility estimates

A two-phase tree-based algorithm like UP-Growth [72], UP-Growth+ [71] extracts high-utility itemsets in two phase. The tree-based algorithms generate candidate high-utility itemsets by recursively generating tree data structures like a UP-tree [71] in the first phase, and compute the utility of the candidates through another database scan in the second phase. The time spent by the algorithms in the verification phase depends on the number of candidates generated in the first phase. We believe that designing a tighter upper-bound will reduce the number of candidates, and improve the performance of two-phase tree-based algorithms. We augment a histogram with each node of the UP-Hist tree to compute a tighter upper-bound compared to the state-of-the-art two-phase tree-based algorithms. A histogram associated with every node of UP-Hist tree stores the number of transactions that contain the positive utility associated with an item and is formally defined below.

Definition 3.1 (Histogram)**.** *A histogram h for an item-node N_i is a set of pairs $\langle w_i, num_i \rangle$, where w_i is a non-negative utility associated with an item and num_i is the number of transactions that contain the item associated with node N_i and*

the item has w_i utility associated with it.

Each node N of a UP-Hist tree stores the following information: item $N.item$, identifier uid, histogram $N.hist$, node utility nu, support count $N.count$, a pointer to the parent node $N.parent$, and a pointer $N.hlink$ that points to another node with the same item $N.item$. The root of a UP-Hist tree is a special empty node that points to its child nodes. Every path from root to the leaf represents a set of transactions in the transaction database. The support count of a node N along a path from root to N stores the number of transactions that contain $N.item$. The node utility stores an upper-bound score (like TWU) computed for each node of the UP-Hist tree. The field uid stores a randomly generated unique identifier that is associated with each node of the global UP-Hist tree.

A header table is also maintained to efficiently traverse the UP-Hist tree. The header table stores three columns: item name, overestimated utility, and a link to the node with $N.item$ equal to item name. The header table allows to traverse all the nodes associated with a particular item in UP-Hist tree as the nodes are connected through a linked-list.

3.1.1 Construction of a global UP-Hist tree

The global UP-Hist tree is constructed from the transaction database in two database scans. Initially, the transaction database is scanned to compute the TWU [50] for the items present in the database. Transaction utility (TU) of

a transaction [50] is defined as the sum of utility of the items present in the transaction. The transaction-weighted utility (TWU) [50] for an item is defined as the sum of transaction utility for the transactions that contain the item. The items with TWU less than the given minimum utility threshold denoted by θ are labeled as unpromising items. The unpromising items can not be a part of any high-utility itemset due to the transaction-weighted downward closure property [50] and can be removed from the database. The strategy to remove unpromising items and recompute the transaction utility (TU) for every transaction is called the "discarding global unpromising items (DGU)" [72, 71] strategy. The unpromising items are removed from the transaction database, and the items are sorted in descending order according to their TWU values during the second database scan. The transactions after the removal of unpromising items are called reorganized transactions.

We apply another strategy called "discarding global node utility (DGN)" [72, 71] after the application of DGU strategy to compute a tighter upper-bound score. The tree-based algorithms like UP-Growth [72], UP-Growth+ [71] etc. are pattern growth algorithms that grow a prefix in a bottom-up manner. Therefore, the utility of descendants can be removed from the node utility for every node of the UP-Hist tree. The new transactions called reorganized transactions are inserted one by one to construct a global UP-Hist tree through a procedure called Add_transaction_globaltree (Algorithm 3.1). The procedure takes as input a transaction with its transaction utility, and the global UP-Hist tree. Initially a pointer called currentNode points to the root of the tree (line 1). The first item

from the beginning of the transaction is selected, and a variable called RU stores the sum of utility of the remaining items (line 4-10). A new node is created for the selected item by calling a method called createNewNode(), if there is no child node with the same item name as the selected item (line 13-17). The function createNewNode() returns a pointer to the new node created for the selected item from the transaction. The header table associated with the tree T is also updated in the CreateNewNode function. The node utility is computed by applying the DGN strategy (line 16). Else, the histogram and other fields associated with the existing node in the tree are updated (line 19-23), and the procedure is called recursively to process the next item from the transaction.

For example, consider the transaction database shown in Table 3.1 and let the minimum utility threshold (θ) be 75. The TWU for the items present in the transaction database is shown in Table 3.2. The items F, G, and H are identified as unpromising, and can be removed from the database. The items within each transaction are sorted according to the descending order of TWU values. The reorganized transactions for our example database is shown in Table 3.3. The global UP-Hist tree constructing from the reorganized transactions after applying the DGU, and DGN strategy is shown in Figure 3.1. Let us focus on the histogram associated with node C in the global UP-Hist tree. The histogram associated with node C is $h = \{(1:1),(4:1),(6:1),(10:1),(13:1)\}$. It can be observed from the transaction database that the item C appears in five transactions with the utility stored in its associated histogram.

Table 3.1: Example database

TID	Transaction	TU
T_1	$(B:4)\,(C:4)\,(E:3)\,(G:2)$	13
T_2	$(B:8)\,(C:13)\,(D:6)\,(E:3)$	30
T_3	$(A:5)\,(C:10)\,(D:2)$	17
T_4	$(A:30)\,(B:2)\,(C:1)\,(D:8)\,(H:2)$	43
T_5	$(A:10)\,(C:6)\,(E:6)\,(G:5)$	27
T_6	$(A:10)\,(B:4)\,(D:12)\,(E:6)\,(F:5)$	37

Table 3.2: TWU of items

Item	A	B	C	D	E	F	G	H
TWU	124	123	130	127	107	37	40	43

Table 3.3: Reorganized transactions

TID	Reorganized Transaction	TU
T_1	$(C:4)\,(B:4)\,(E:3)$	11
T_2	$(C:13)\,(D:6)\,(B:8)\,(E:3)$	30
T_3	$(C:10)\,(D:2)\,(A:5)$	17
T_4	$(C:1)\,(D:8)\,(A:30)\,(B:2)$	41
T_5	$(C:6)\,(A:10)\,(E:6)$	22
T_6	$(D:12)\,(A:10)\,(B:4)\,(E:6)$	32

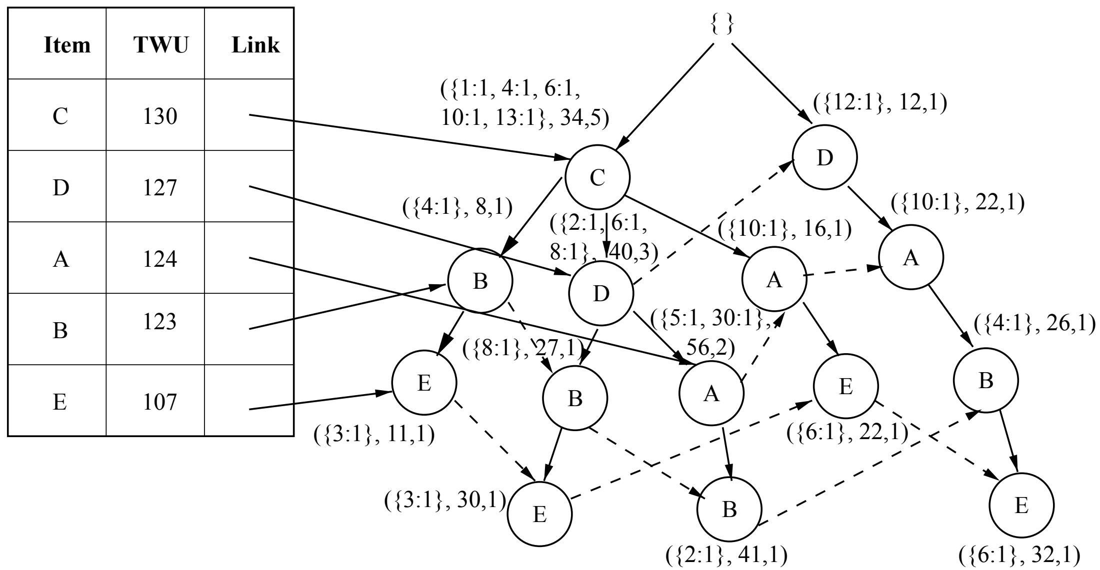

Figure 3.1: Global UP-Hist tree

Algorithm 3.1 Add_transaction_globaltree (Itemset tx, TU,T)

Input: A transaction tx, transaction utility of tx, and a global UP-Hist tree T.
Output: Returns the global UP-Hist tree T after the insertion of transaction tx

```
1:  currentNode = T.root
2:  size = tx.size(), i=0
3:  while i < size do
4:      k = i+1, RU=0
5:      while k < size do
6:          RU = RU + tx.get(k).utility
7:          k = k + 1
8:      end while
9:      item = tx.get(i).item
10:     utility = tx.get(i).utility
11:     childNode = currentNode.getChildWithID(item)
12:     if childNode==NULL then
13:         currentNode = createNewNode(currentNode,item,utility)
14:         currentNode.count = 1
15:         currentNode.updateHist(utility,1)
16:         CurrentNode.nu = TU - RU
17:         currentNode.uid = random_number()
18:     else
19:         childNode.count = childNode.count + 1
20:         childNode.updateHist(utility,childNode.count)
21:         childNode.nu = childNode.nu + TU - RU
22:         currentNode = childNode
23:     end if
24:     i = i+1
25: end while
26: Return T
```

3.1.2 Construction of a local UP-Hist tree

Local trees are constructed by tree-based algorithms during the extraction of candidate high-utility itemsets. The algorithms aim to reduce the number of candidates by computing tighter upper-bound estimates for the utility of the currently explored prefix itemset and its supersets. Tseng et. al. [72, 71] proposed two strategies called "discarding local unpromising items (DLU)", and "discarding local node utility (DLN)" to compute tighter utility estimates. The strategies DLU, and DLN are similar to DGU, and DGN. However, the exact utility of

items in a transaction is not available during the mining process. Therefore, a score called minimum node utility is stored with each node of the UP-tree [71] that is used by the UP-Growth+ [71] algorithm for DLU and DLN. The minimum node utility associated with a node N_i is the minimum utility of item i in the transactions represented by the path from the root to a leaf node containing node N_i as an intermediate node.

We utilize the information stored in the histogram associated with each node of the UP-Hist tree to compute tighter upper-bound scores compared to the state-of-the-art algorithm UP-Growth+ [71]. We will also compute a lower-bound utility estimate to reduce the number of candidates that are passed for verification. Now, we define two functions $minC(I, s)$, and $maxC(I, s)$ that takes as input the histogram associated with a node for item I and return a number as output.

Definition 3.2. *Let h be a histogram, associated with an item-node N_i, consisting of n, $(1 \leq i \leq n)$ pairs $< w_i, num_i >$, sorted in ascending order of w_i. $minC(N_i, s)$ returns the sum of item-copies of k entries of h, i.e., $minC(N_i, s) = \sum_1^k w_i$, such that k is the maximal number fulfilling $k \leq \sum_1^k num_i$.*

Definition 3.3. *Let h be a histogram, associated with an item-node N_i, consisting of n, $(1 \leq i \leq n)$ pairs $< w_i, num_i >$, sorted in descending order of w_i. $maxC(N_i, s)$ returns the sum of item-copies of k entries of h, i.e., $maxC(N_i, s) = \sum_1^k w_i$, such that k is the minimal number fulfilling $k \leq \sum_1^k num_i$.*

In the UP-Hist tree, we maintain a histogram sorted in only ascending order of item-utility (w_i) only. $minC(\cdot)$ will process the pairs from the front of the histogram, and $maxC(\cdot)$ will process pairs from rear to front of the histogram. For example, minC(C,3), and maxC(C,3) for the histogram $h = \{(1:1),(4:1),(6:1),(10:1),(13:1)\}$ associated with C is 11 and 29 respectively.

Definition 3.4. *Given an itemset $I =< a_1, a_2, ..., a_k >$ corresponding to a path in a UP-Hist tree, with support count value of item a_k as s, the UB and LB utility values of I are computed as defined below.*

$$UB(I) = \sum_{i=1}^{k} maxC(a_i, s)$$

$$LB(I) = \sum_{i=1}^{k} minC(a_i, s)$$

Theorem 3.1. *$LB(I)$ and $UB(I)$ are correct lower and upper bound estimates of the exact utility for any itemset I with support s.*

Proof. As per Definition 3.4, the lower bound utility estimate of I i.e. $LB(I)$ is computed as a summation of *minC(a_i, s)* for each item $a_i \in I$. The exact utility of the itemset I is computed as summation of the exact utility of each item $a_i \in I$ associated with each item in the set of transactions containing I. As *minC(I,s)* computes the sum of utility from the front of the histogram, the exact utility of I can not be less than *minC(I,s)*. Therefore, LB(I) is a correct lower-bound estimate for the exact utility of I. Similarly, the summation of the utility associated with each item a_i in I can not be more than computed by *maxC(I,s)*

as the histogram is constructed from the set of transactions containing I only. Therefore, UB(I) is a correct upper-bound estimate for the exact utility of I. $\square$

The method to construct a local UP-Hist tree is presented by Algorithm 3.2. The algorithm takes as input a UP-Hist tree T, an item i, and the user-defined minimum utility threshold θ as input and returns the local tree for item i. Initially, the entry for the item i is fetched from the header table of the input UP-Hist tree T, and a conditional pattern base (CPB) for item i is constructed (line 1-20). The conditional pattern base for item i stores a list of paths, where each path contains the nodes between the root and the node corresponding to item i by traversing the linked-list associated with item i from the header table H. list of paths from the tree T that contain item i. The node corresponding to item i is also stored in the list of paths called PrefixPath, but it will not be inserted in the local UP-Hist tree. A HashMap called itemPathUtility stores the path utility for the ancestors of item i. An empty local UP-Hist tree is created (line 21), and every path stored in the conditional pattern base is processed and inserted to construct a local UP-Hist tree (line 22-39). Only the nodes corresponding to the promising items i.e. items with path utility of at least θ are added to the local tree. The strategy called DLU is applied to remove the contribution of unpromising items (line 33-35). The nodes in every path are sorted in descending order of their path utility scores and the method called Add_transaction_localtree (Algorithm 3.3) inserts the localPath into the local UP-Hist tree. The strategy called discarding local node utility (DLN) is applied in the method called Add_transaction_localtree. We illustrate an example to

construct the local UP-Hist tree for the item $\{A\}$ from the global UP-Hist tree (Figure 3.1) below.

The CPB of $\{A\}$ consists of the following paths: $\langle ADC \rangle$:56, $\langle AC \rangle$:16, and $\langle AD \rangle$:22. The path utility for the ancestors present in CPB(A) i.e. C and D respectively is 72, and 78. Therefore, item C is unpromising and its utility must be removed from the CPB(A). Now, we will calculate the estimated utility using our histogram. The support of the unpromising item C is 2 and *minC(C,2)* is 5. The path utility of path $< CD >$ after applying the DLU strategy is 51. Therefore, the local UP-Hist tree of $\{A\}$ will consist of only node corresponding to the item D. It can be observed that there are two nodes associated with item D in the global UP-Hist tree. Both nodes have a different unique identifier (uid) associated with it. Therefore, the node D in the local tree for the item $\{A\}$ will have the histogram $\{(2 : 1), (6 : 1), (8 : 1), (12 : 1)\}$ associated with it (line 20-22) after the execution of the method called Add_transaction_localtree.

Theorem 3.2. *The recomputed path utility and node utility is an upper-bound on the utility for an itemset and any of its supersets,*

Proof. It has been proved from Theorem 3.1 that $minC(\cdot)$ computes a correct lower-bound estimate for the exact utility of an item. Therefore, the recomputed path utility and node utility after applying the DLU, and DLN strategy computes a correct upper-bound estimate for the utility of an itemset and its supersets. □

Algorithm 3.2 Construct_local_tree (T, i,θ)

Input: A UP-Hist tree T, an item i, a minimum utility threshold θ.
Output: The local tree for the item i.

```
1:  Get a pointer to the first node for item i in T from the header table associated with T and store it in a
    variable called path
2:  Initialize an list called CPB and a HashMap called itemPathUtility to store path utility of the ancestors
    of i in CPB
3:  while path ≠ NULL do
4:     prefixPath.add(path)
5:     parentnode = path.parent
6:     nodeutility = path.nu
7:     while parent ≠ T.root do
8:        prefixPath.add(parentnode)
9:        pu = itemPathUtility.get(parentnode.itemID)
10:       if pu==NULL then
11:          pu = nodeutility
12:       else
13:          pu = pu + nodeutility
14:       end if
15:       itemPathUtility.put(parentnode.itemID,pu)
16:       parentnode = parentnode.parent
17:    end while
18:    CPB.add(prefixPath)
19:    path = path.nodeLink
20: end while
21: Create an empty UP-Hist tree called local_tree
22: for  prefixPath in CPB do
23:    pathCount = prefixPath.get(0),count
24:    pathUtility = prefixPath.get(0).nu
25:    Initialize an empty list called localPath
26:    J = 1
27:    while J < prefixPath.size()  do
28:       value = 0
29:       node = prefixPath.get(J)
30:       if itemPathUtility.get(node.itemID) ≥ θ then
31:          localPath.add(node)
32:       else
33:          value = minC(node,pathCount)
34:       end if
35:       pathUtility = pathUtility - value
36:       J = J+1
37:    end while
38:    Sort localPath in descending order of path utility
39:    local_tree.Add_transaction_localtree(localPath, pathUtility, pathCount)
40: end for
41: Return local_tree
```

Algorithm 3.3 Add_transaction_localtree (localPath, pathUtility,pathCount)

Input: A list of nodes stored in localPath, utility of the path stored in pathUtility, and number of transactions containing the path stored in pathCount.

Output: Inserts a localPath in a local UP-Hist tree

```
1:  currentlocalNode = root
2:  size = localPath.size(), i=0
3:  while i < size do
4:      k = i+1, RU=0
5:      while k < size do
6:          node = localPath.get(k)
7:          RU = RU + minC(node,pathCount)
8:          k = k + 1
9:      end while
10:     item = localPath.get(i).item
11:     childNode = currentlocalNode.getChildWithID(item)
12:     if childNode==NULL then
13:         currentlocalNode = createNewNode(currentlocalNode,item)
14:         currentlocalNode.count = pathCount
15:         currentlocalNode.updateHist(localPath.get(i).hist)
16:         CurrentlocalNode.nu = pathUtility - RU
17:         currentlocalNode.uid = localPath.get(i).uid
18:     else
19:         childNode.count = childNode.count + pathCount
20:         if childNode.uid ≠ localPath.get(i).uid then
21:             childNode.updateHist(localPath.get(i).hist)
22:         end if
23:         childNode.nu = childNode.nu + pathUtility - RU
24:         currentlocalNode = childNode
25:     end if
26:     i = i+1
27: end while
```

3.2 UP-Hist Growth Algorithm

In this section, we propose the UP-Hist algorithm to mine high-utility itemsets. The UP-Hist algorithm (Algorithm 3.4) takes as input a global UP-Hist tree, a prefix α, and a user-defined minimum utility threshold θ as input, and outputs a set of candidate high-utility itemsets. The exact utility of the candidate itemsets is computed in the verification phase by scanning the database.

UP-Hist Growth is a pattern-growth algorithm that starts with an empty prefix, and explores the search space in a depth-first manner. Every item a_i is processed from the header table of the global UP-Hist tree in a bottom-up manner (line 1). The node utility of a_i is computed by traversing the nodes in the linked-list associated with the entry a_i from the header table (line 3). If the estimated node utility is no less than θ, the upper-bound and lower-bound estimates for the current prefix I is computed (line 5). If the lower-bound utility of itemset I is at least θ, I is output as a high-utility itemset as the utility of I in the database will be at least θ too as per Theorem 3.1. Else, if the upper-bound utility of I is no less than θ, I is added to the set of candidate high-utility itemsets (line 10). The conditional pattern base (CPB) for I is constructed, and unpromising items in the CPB of I is identified. The strategy DLU and DLN is applied and a local UP-Hist tree for I is constructed (line 14). If the local UP-Hist tree is not empty, the UP-Hist growth algorithm is called recursively (line 16). The exact utility of the candidate high-utility itemsets generated by UP-Hist Growth is computed to scanning the database and high-utility itemsets are identified (line 20).

Algorithm 3.4 UP-Hist Growth(α,T, H,θ)

Input: Prefix α (initially empty),a global UP-Hist tree T, a header table H for T, a minimum utility threshold θ.

Output: All high-utility itemsets with α as prefix.

```
1: for each entry {i} in H do
2:   Itemset I = α ∪ i.
3:   Compute node.nu by following the links from the header table for {i}.
4:   if node.nu(a_i) ≥ θ then
5:     Compute UB({I}) and LB({I}).
6:     if UB(I) ≥ θ then
7:       if LB(I) ≥ θ then
8:         Output I as a high-utility itemset.
9:       end if
10:      Add I to the list of candidate high-utility itemsets.
11:    end if
12:    T_I = Construct_local_tree (T, i,θ)
13:    if T_I ≠ null then
14:      Call UP-Hist Growth(I,T_I, H_I,θ)
15:    end if
16:  end if
17: end for
18: Compute the exact utility for every generated candidate high-utility itemset by scanning the database
    again and output the high-utility itemsets.
```

3.2.1 Complexity Analysis

The UP-Hist Growth algorithm is a two-phase tree-based algorithm that generates candidates in the first phase, and scans the database to compute the utility of candidates in the verification phase. Let us analyze the complexity of the candidate generation phase.

Let "$|I|$" be the number of items, "n" be the number of transactions, and let the maximum transaction length be "w". We can assume that all transactions have "w" items for the analysis of worst-case time complexity. The initial UP-Hist tree is constructed by inserting each transaction from the database. The cost of initial tree creation is O($n \times w$), where O(w) is the cost of inserting a transaction in the tree either by finding a path from the root node or adding a new

path to the tree. The search-space for itemset mining consists of $2^{|I|}$ itemsets, and a local UP-Hist tree is created for every explored itemset by the UP-Hist Growth algorithm. The length of the header-list for any item in the header table can be maximum "n" assuming that the item is present in every transaction. It takes another O($n \times w$) cost to construct the conditional pattern base. The method named updateHist($\cdot$) is called during the creation of local tree and takes O($m \times log$ (m)) cost to update the histogram at every node. The minC($\cdot$) and maxC($\cdot$) methods perform a linear scan on the histogram and take O(m) cost. The insertion of a path from the conditional pattern base to construct a local tree can take O($w\times$ ($m \times log$ (m)) cost. The total cost of candidate generation phase is O($2^{|I|}\times$ ($nw\times$ ($m \times log$ (m))).

In the verification phase, utilities of the candidate itemsets are computed by scanning the database. It takes O($n \times w$) cost to scan the complete transaction database. For every transaction, the utilities of the candidate itemsets present in it is computed. The worst case complexity of verification phase is O ($n \times w \times 2^{|I|}$).

In terms of space complexity, the main cost is the space used by the global and local UP-Hist trees constructed during the candidate generation phase. The height of a UP-Hist tree is bounded by the longest transaction. Without considering the root node, the number of nodes in a UP-Hist tree is bounded by O($n \times w$) assuming that the transactions share no path in the tree. The histogram associated with a node of the tree can occupy O(m) space assuming that the histogram contains "m" key-value pairs. There can be a maximum of "w"

local trees in memory. The space complexity of UP-Hist Growth algorithm can be bounded by O($w \times$ $(nwm + |I|)$), where $|I|$ factor accounts for the size of header table.

3.2.2 An Illustrated Example

We will now illustrate an example for the execution of UP-Hist Growth on a transaction database. Consider the transaction database as shown in Table 3.1, and let the minimum utility threshold θ be 75. The database is scanned to compute the TWU of items as shown in Table 3.2. Item F, G, and H are identified as unpromising items and are removed from the transaction database. The items in every transaction are sorted in decreasing order of TWU values and inserted to construct the global UP-Hist tree as shown in Figure 3.1. The items in the global header table will be processed in a bottom-up manner. Let us focus on the processing of item A. The linked-list associated with item A will be traversed to compute its total node utility [71]. The total node utility computed by summing up the node utility for every node present in the linked-list for item A is 94. The prefix $\{A\}$ will be processed further as it's total node utility is greater than the minimum utility threshold. The $UB(\{I\})$ and $LB(\{I\})$ is 55. Therefore, prefix $\{A\}$ will not be added to the list of candidate high-utility itemset. The conditional pattern base (CPB) for the current prefix is computed. The CPB of $\{A\}$ consists of the following paths: $\langle ADC \rangle$:56, $\langle AC \rangle$:16, and $\langle AD \rangle$:22. The TWU for the items present in CPB(A) i.e. C and D respectively is 72, and 78. Therefore, item C is unpromising and its utility must be removed from the CPB(A).

The reorganized utility of the path $< CD >$ by UP-Growth+ is computed as shown below.

$pu(< CD >, A - CPB) = 56 - mnu(C) \times s(c) = 56 - 1 \times 2 = 54.$

The estimated utility for the itemset $< AD >$ by UP-Growth+ is equal to the sum of path utility of $< CD >$ and $< D >$ in $\{A\} - CPB$ i.e. 76.

Now, we will calculate the estimated utility using our histogram. The support of the unpromising item C is 2 and *minC(C,2)* is 5. The path utility of path $< CD >$ using the histogram of item $\{C\}$ $\{(1 : 1), (4 : 1), (6 : 1), (10 : 1), (13 : 1)\}$ is given below:

$pu(< CD >, \{A\} - CPB) = 56 - minC(C, 2) = 56 - 5 = 51.$

The estimated utility of itemset $< AD >$ is 73. Therefore, $< AD >$ is a potential high utility itemset according to the UP-Growth and UP-Growth+ algorithm, but a low utility itemset according to UP-Hist Growth.

UP-Hist Growth will not generate any candidate high-utility itemset for verification at minimum utility threshold equal to 75. However, UP-Growth+ generates the following candidate high-utility itemsets: $\{A\}$:94, $\{AD\}$:76, $\{B\}$:102, $\{BD\}$:77, and $\{E\}$:95. This example highlights that the histogram associated with a UP-Hist tree allows to compute a tighter bound compared to the bound computed by UP-Growth+.

3.3 Experiments and Results

In this section, we compare the performance of the proposed UP-Hist Growth algorithm with state-of-the-art two-phase tree-based algorithms, IHUP-Growth [6], UP-Growth [72], and UP-Growth+ [71]. The source code of the algorithms was downloaded from the SPMF library [31]. The total execution time, the number of candidates generated during the mining process, and maximum main memory consumed are used as the metrics to compare the performance of algorithms on several benchmark dense and sparse datasets.

The experiments were performed on an Intel Xeon(R) CPU=26500@2.00 GHz with 16 GB RAM and Windows Server 2012 operating system. All datasets except NyTimes used for our experiments were obtained from the SPMF library [31], and the characteristics of the datasets are shown in Table 3.4. The NyTimes dataset is a bag of words dataset obtained from the UCI Machine Learning repository [29]. The Retail dataset contains customer transactions from an anonymous Belgian retail store. The Kosarak dataset was generated from click-stream data from a Hungarian news portal. The ChainStore dataset contains customer transactions from a grocery store chain in California, USA. The Chess, Mushroom, and Connect datasets were prepared from their corresponding datasets in the UCI machine learning repository. The Accidents dataset was prepared from anonymized traffic accident data. Only the ChainStore and NyTimes datasets contained utility values associated with every item in a transaction. For the remaining datasets, the internal utility values were generated from

Table 3.4: Characteristics of real datasets

Dataset	**#Tx**	**Avg. length**	**#Items (I)**	**Density score R (%) =(A/I)x100 [7]**	**Type**
Retail	88,162	10.3	16,470	0.062	Sparse
Kosarak	9,90,002	8.1	41,270	0.019	Sparse
Chainstore	11,12,949	7.2	46,086	0.015	Sparse
NyTimes	3,00,000	232.2	1,02,660	0.22	Sparse
Chess	3,196	37	75	49.33	Dense
Mushroom	8,416	23	119	19.32	Dense
Connect	67,557	43	129	33.33	Dense
Accidents	3,40,183	33.8	468	7.22	Dense

Table 3.5: Candidate generation and verification time (sec) on Kosarak dataset

Threshold(%)	**UP-Hist Growth**	**UP-Growth+**
0.6	242 sec (20 sec)	51 sec (105 sec)
0.65	226 sec (11 sec)	41 sec (80 sec)
0.7	151 sec (13 sec)	32 sec (68 sec)
0.75	126 sec (11 sec)	22 sec (56 sec)
0.8	96 sec (11 sec)	24 sec (49 sec)

a uniform distribution in the range of 1 to 10, and the external utility values were generated from a Gaussian distribution. The positive weight associated with every item in a transaction is a product of their internal and external utility.

Effect of varying minimum utility threshold: We study the effect of varying minimum utility threshold on the performance of the two-phase tree-based algorithms on different datasets. The results on sparse datasets is shown in Figure 3.2 and Figure 3.3 respectively. The UP-Hist Growth algorithms has the minimum number of candidates generated during the mining process compared

to the other algorithms on every sparse dataset. The UP-Hist Growth algorithm executes at least 19 times faster compared to other algorithms on the Retail dataset for a threshold equal to 0.01 %. The UP-Hist Growth algorithm executes around 3 times faster compared to the UP-Growth+ algorithm at lower

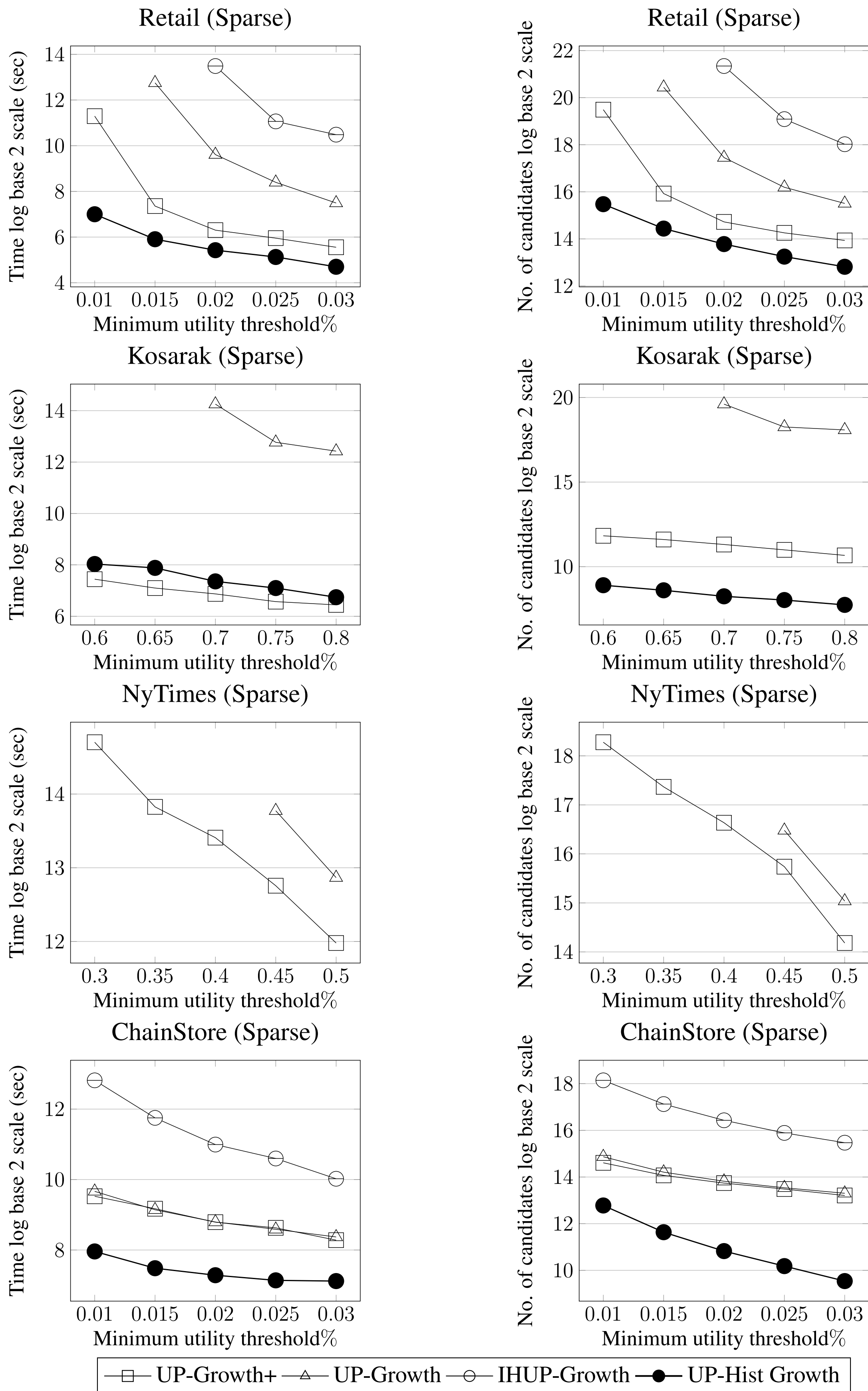

Figure 3.2: Performance evaluation (Time and number of candidates) for UP-Growth+, UP-Growth, IHUP-Growth and UP-Hist Growth on sparse datasets. The UP-Hist Growth and IHUP-Growth algorithms ran out of memory for the NyTimes dataset. The UP-Growth algorithm did not terminate execution for more than 24 hours on the NyTimes dataset for threshold less than 0.45 %.

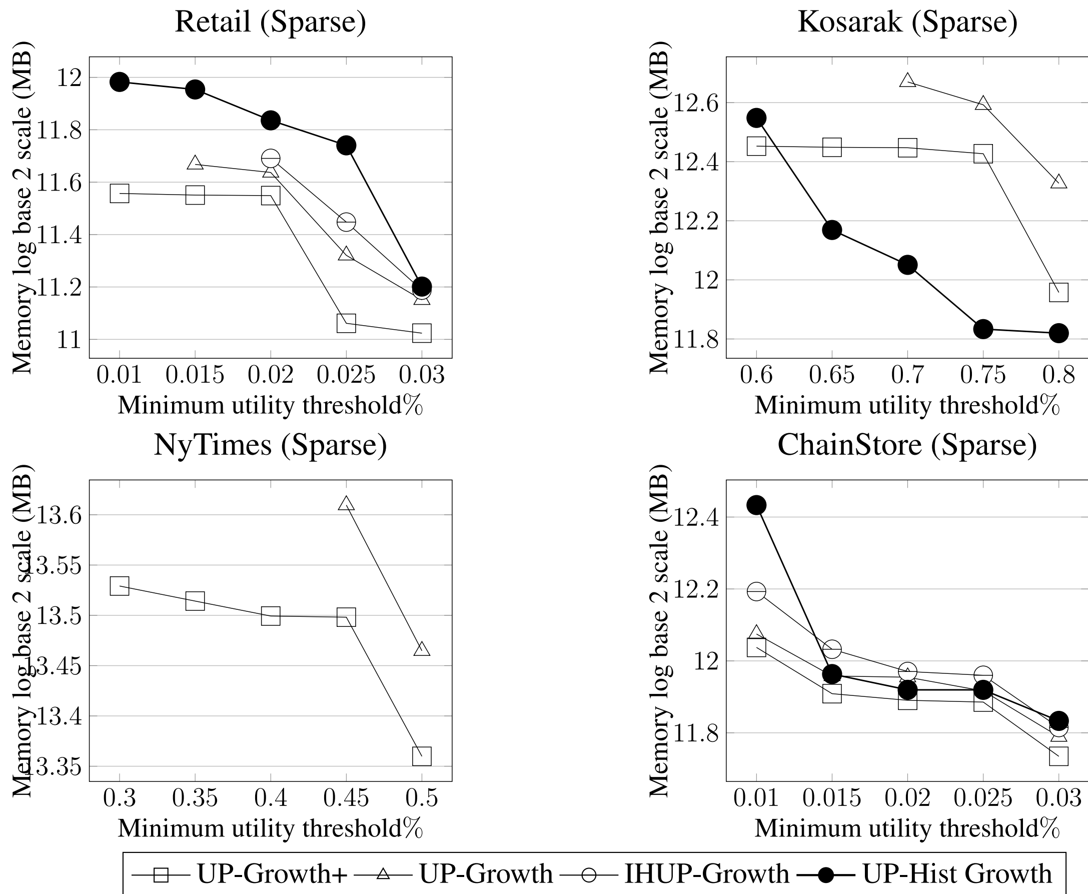

Figure 3.3: Memory consumption of UP-Growth+, UP-Growth, IHUP-Growth and UP-Hist Growth on sparse datasets. UP-Growth ran out of memory during the candidate generation phase on Retail and Kosarak datasets at lower thresholds. IHUP-Growth ran out of memory during the candidate generation phase on Kosarak dataset. IHUP-Growth also ran out of memory on Retail dataset for threshold less than 0.002%. The UP-Hist Growth and IHUP-Growth algorithms ran out of memory for the NyTimes dataset.

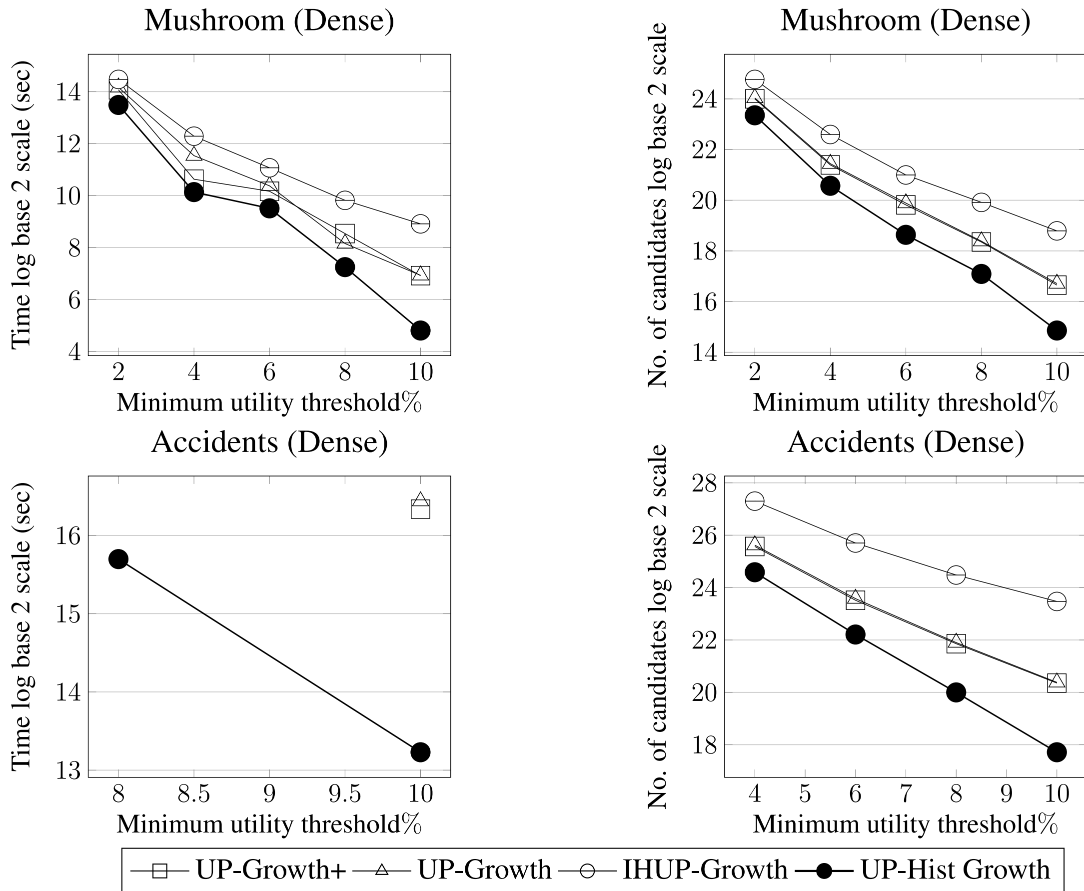

Figure 3.4: Performance evaluation (Time and number of candidates) for UP-Growth+, UP-Growth, IHUP-Growth, and UP-Hist Growth on dense datasets. All tree-based algorithms did not terminate their execution for more than 24 hours on Chess and Connect datasets.

threshold values on the ChainStore dataset. However, it can be observed that the UP-Growth+ algorithm has less total execution time compared to UP-Hist Growth on the Kosarak dataset. We observe from Table 3.5 that the UP-Hist Growth algorithm spends more time in the candidate generation phase compared to the UP-Growth+ algorithm.

Therefore, the UP-Growth+ algorithm executes faster even though it generates more candidates and spends more time in the verification phase for the Kosarak dataset. The UP-Growth and IHUP-Growth algorithms ran out of memory during the candidate generation phase on the Retail dataset for a threshold

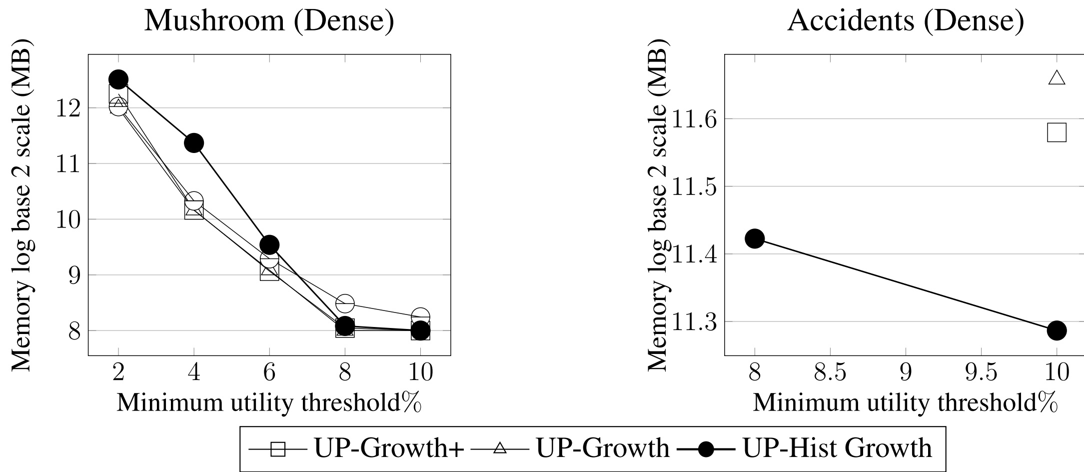

Figure 3.5: Memory consumption of UP-Growth+, UP-Growth, IHUP-Growth, and UP-Hist Growth on dense datasets. All tree-based algorithms ran out of memory during the candidate generation phase for Accidents dataset at 2% threshold.

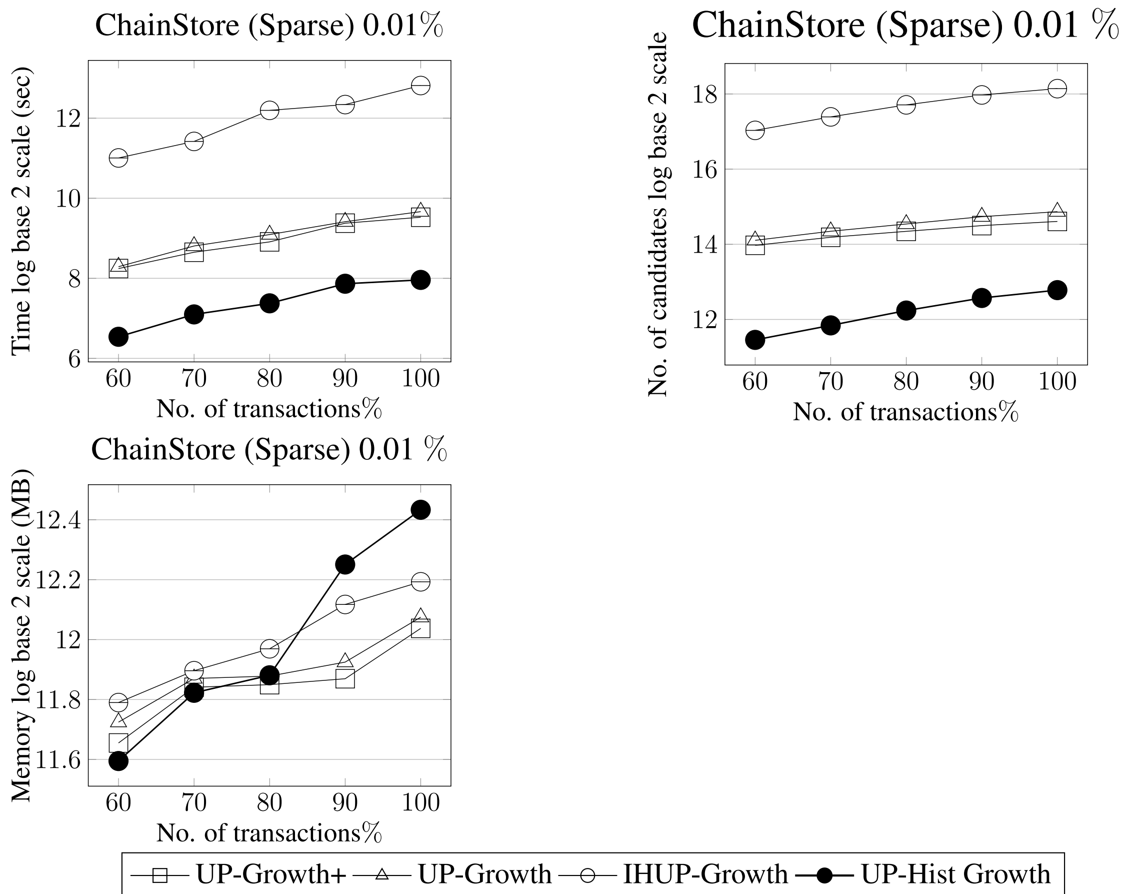

Figure 3.6: Scalability experiment on ChainStore and Accidents dataset for 0.01% and 2% threshold respectively. All tree-based algorithms did not terminate for more than 24 hours for the Accidents dataset.

less than 0.02 and 0.015 %, respectively. The IHUP-Growth ran out of memory during the candidate generation phase on the Kosarak dataset. The UP-Growth ran out of memory for a minimum utility threshold of less than 0.7 % on the Kosarak dataset. The UP-Hist Growth algorithm ran out of memory on the NyTimes dataset during the creation of the global tree.

The results on dense datasets is shown in Figure 3.4 and 3.5. The UP-Hist Growth has the least execution time and the number of candidates than other tree-based algorithms on the Mushroom and Accidents dataset. The UP-Growth+, UP-Growth, and IHUP-Growth algorithms did not terminate their execution on the Accidents dataset for more than 24 hours. Every two-phase tree-based algorithm ran out of memory during the candidate generation phase on the Accidents dataset at 2% threshold. It can be observed that the UP-Hist Growth did not terminate its execution for more than 24 hours on the Accidents dataset for a threshold of less than 8 %. We observe that the performance of two-phase tree-based algorithms depends on the number of candidates generated during the mining process, and there is a need to design faster algorithms that have less memory consumption compared to the two-phase algorithms.

Effect of scalability: We study the effect of increasing the number of transactions on the performance of tree-based algorithms for the ChainStore and Accidents dataset at 0.01% and 2% respectively. The ChainStore and Accidents dataset is the largest dataset in the category of sparse and dense datasets. The result is shown in Figure 3.6. The total execution time, the number of generated candidates, and the memory consumed increases with the number of transac-

tions for the ChainStore dataset. The tree-based algorithms did not terminate their execution for more than 24 hours for the Accidents dataset.

3.4 Summary

In this chapter, we propose a two-phase tree-based algorithm called UP-Hist Growth, and a data structure called UP-Hist tree that stores additional information compared to the UP-tree to compute a better utility estimate to reduce the number of generated candidates in the first phase. Our experimental study in Section 3.3 shows the superior performance of UP-Hist Growth compared to the state-of-the-art two-phase tree-based algorithms on several dense and sparse datasets. However, we observe that the two-phase tree-based algorithms either do not terminate its execution for more than 24 hours or run out of memory for extremely dense benchmark datasets like Chess and Connect. In the next chapter, we design a one-phase algorithm that can combine any tree-based algorithm with a list-based algorithm to mine high-utility itemsets faster compared to the two-phase tree-based algorithms.

Chapter 4

A hybrid algorithm for high-utility itemset mining

Tree-based algorithms like UP-Growth+ [71], UP-Hist Growth [21] generate candidates quickly, but mine high-utility itemsets in two phases. The candidate high-utility itemsets are generated in the first phase, and the high-utility itemsets are filtered in the next phase by computing their utility through another database scan. It has been observed that tree-based algorithms can spend a lot of time in the verification phase, especially for dense datasets that can have a large of candidates generated after the first phase.

The list-based algorithms like HUI-Miner [49], FHM [32] store an inverted-list for every itemset to keep information about the utility of the itemset in every transaction. The list-based algorithms mine high-utility itemsets in one-phase only without any verification phase. However, the joining/intersection of the inverted-lists of two $\{k-1\}$-itemsets to generate the list of a k-itemset can become a performance bottleneck especially on dense datasets that can have a

large of high-utility itemsets for lower threshold values. List-based algorithms also generate itemsets that are non-existent in the database during the mining process.

In this chapter, we propose a hybrid algorithm to combine the benefits of tree-based and list-based algorithms. We propose an algorithm that can combine any tree-based algorithm with a list-based algorithm to design a one-phase algorithm. Such a hybrid algorithm can generate candidates quickly by running a tree-based algorithm and compute the utility of itemsets in one-phase only by switching its execution to a list-based algorithm once a pre-defined switching condition is satisfied. We perform a case study by joining the state-of-the-art tree-based algorithm UP-Growth+ [71] and UP-Hist Growth [21] with the state-of-the-art list-based algorithm FHM [32]. We also compare the performance of hybrid algorithms against state-of-the-art tree-based and list-based algorithms.

In Section 4.1, we propose a hybrid algorithm that can integrate a tree-based algorithm with a list-based algorithm. We further highlight some caveats that must be kept in mind while designing a hybrid algorithm and discuss several techniques that can be integrated to improve the performance of a hybrid algorithm. In Section 4.2, we present a case study to discuss the integration of UP-Growth+ [71] with FHM [32] called UPG+-Hybrid, and UPHist-Growth [21] with FHM [32] called UPHist-Hybrid. Our experimental study is presented in Section 4.3 and the summary of this chapter is presented in Section 4.4.

4.1 Hybrid algorithm

Our proposed hybrid algorithm (Algorithm 4.1) takes as input a prefix α (initially empty), a transaction database D, and a minimum utility threshold θ. It returns the complete set of high-utility itemsets. Initially, the transaction database is scanned to compute the TWU of items, and remove the unpromising items (lines 1-2). The items in every transaction are sorted according to a predefined ordering (like ascending order of TWU), and optimizations like transaction merging [85] can be performed (lines 3-4). The transactions are inserted one by one to form a global tree data structure, and an inverted-list data structure (lines 5-6). The algorithm Tree-Growth that represents the generic structure of a tree-based algorithm is called (line 7). The Tree-Growth algorithm (Algorithm 4.2) takes as input a global tree, a header table, a prefix α (initially empty), and a minimum utility threshold θ. A tree-based algorithm like UP-Growth+ [71], UP-Hist Growth [21], IHUP-Growth [6] generate candidate high-utility itemsets by constructing the local trees recursively and compute a score like TWU [50], node utility [72, 71] to decide whether the current prefix itemset and its supersets can be high-utility or not. Similarly, the Tree-Growth algorithm computes a score (line 4) and proceeds its execution further if the score is no less than θ. A $UB(\cdot)$ function that can compute an upper-bound utility score for the new prefix (itemset I generated in line 2) generated is called. An example of a $UB(\cdot)$ function can be the $maxC(i, s)$ (defined in the Chapter 3) used by the UP-Hist Growth [21] algorithm. If the $UB(\cdot)$ score for the current prefix

I is no less than the minimum utility threshold (θ), execution can be switched to an inverted-list based algorithm (Algorithm 4.3). Else, the local tree corresponding to the current prefix itemset I is constructed and Tree-Growth is called recursively (line 10-14).

Algorithm 4.1 Hybrid algorithm(α, D, θ)

Input: Prefix α (initially empty),Transaction database D, and minimum utility threshold θ.
Output: Complete set of high-utility itemsets.

1: Scan D once to find the unpromising items.
2: Scan the database again to remove the unpromising items from each transaction.
3: Sort the items in the transactions according to a predefined global ordering.
4: Perform transaction merging.
5: Insert all the reorganized transactions to form a tree T with header table H.
6: Construct the inverted-list for the promising items.
7: Call Tree-Growth(T,H,$\{\alpha\}$,θ).

Algorithm 4.2 Tree-Growth(T, H, α, θ)

1: **for** each entry $\{i\}$ in H **do**
2:   Itemset I = $\alpha \cup i$.
3:   Compute $node.nu$ by following the links from the header table for $\{i\}$.
4:   **if** $node.nu(a_i) \geq \theta$ **then**
5:     Compute $UB(\{I\})$.
6:     **if** $UB(I) \geq \theta$ **then**
7:       Construct the inverted-list of I and call List-Miner(I, Extensions of I, θ).
8:       Return.
9:     **else**
10:       Construct the conditional pattern base of itemset I.
11:     **end if**
12:     Construct the local tree (T_I) with header table (H_I).
13:     **if** $T_I \neq null$ **then**
14:       Call Tree-Growth(T_I, H_I,I,θ)
15:     **end if**
16:   **end if**
17: **end for**

A generic list-based algorithm like List-Miner (Algorithm 4.3) takes as input a inverted-list of the current prefix, inverted-lists for the possible extensions of the current prefix, and a minimum utility threshold. If the utility of the itemset is no less than θ, the itemset is output as a high-utility itemset (line 2). List-based algorithms [49, 32] construct an inverted-list for every itemset. An inverted-

Algorithm 4.3 List-Miner(I, Extensions of I (Ext_I), θ)

1: **for** each itemset Ix in Ext_I **do**
2:   **if** $u(Ix) \geq \theta$ **then**
3:     Output Ix as a high-utility itemset.
4:   **end if**
5:   **if** $UpperBound(Ix) \geq \theta$ **then**
6:     Ext_Ix={}.
7:     **for** each itemset Iy in Ext_I such that y comes after x **do**
8:       Ixy=$Ix \cup Iy$.
9:       Construct the list for Ixy.
10:       Ext_Ix=Extensions of the itemset $\{Ix \cup Ixy\}$.
11:     **end for**
12:     List-Miner (Ix,Ext_Ix,θ).
13:   **end if**
14: **end for**

list structure stores information about transactions that can contain the itemset along with its utility and an upper-bound estimate denoted by $UpperBound(\cdot)$ (line 5) on the utility of the itemset and its supersets. For example, the HUI-Miner [49] and FHM [32] algorithm compute the sum of exact-utility [49] and remaining-utility [49] to decide whether the current itemset and its supersets can be high-utility or not. The sum of exact-utility and remaining-utility is an example of $UpperBound(\cdot)$ function (line 5 of Algorithm 4.3). If the score computed by the $UpperBound(\cdot)$ function is no less than θ, the List-Miner algorithm constructs the inverted-lists for the extensions of the current prefix itemset and the algorithm List-Miner is called recursively (line 5-12).

4.1.1 Caveats and Optimizations

In this subsection, we discuss few caveats that must be taken into consideration while merging a tree-based algorithm with a list-based algorithm. We will further discuss some optimizations that can improve the performance of a hybrid

algorithm.

Tree-based algorithms [71, 21] like UP-Growth+ [71], and UP-Hist Growth [21] remove unpromising items during local tree generation and sort the transactions again in descending order of TWU during the construction of local trees. Our proposed hybrid algorithm constructs the global tree, and inverted-lists based on the globally defined ordering of items only. If the ordering of items is changed during the local tree creation by Tree-Growth, it might result in the missing of few valid extensions by a list-based algorithm. For example, list-based algorithms like HUI-Miner [49], FHM [32] order the items in ascending order of TWU before starting the mining process and that ordering of items is not changed during the mining process. Therefore, we keep the ordering of items intact during the execution of a tree-based algorithm. In the next section, we will present a case study with an example to integrate the UP-Hist Growth [21], and UP-Growth+ [71] algorithms with the FHM [32] algorithm.

List-based algorithms [49, 32] generate candidates that are non-existent in the transaction database as they join an itemset with all possible extensions from the defined global ordering of items. However, we can reduce the number of candidates generated by a list-based algorithm by utilizing the information stored in the tree data structure. A tree-based algorithm starts its execution with an item from the header table, and grows the prefix in a bottom-up manner. We can pass the list of extensions for an itemset to a list-based algorithm during the switching procedure.

We further implement few strategies to reduce the computation and memory consumption of a list-based algorithm after the execution is switched from a tree-based algorithm. Recall that the inverted-list for an itemset and its possible extensions is given as an input to a list-based algorithm when the execution is switched from a tree-based algorithm. It is possible to construct that inverted-list from scratch in every recursive call. However, the memoization technique can be used to store the inverted-list of an itemset and avoid the creation of inverted-lists from scratch during the switching process. For example, suppose we want to switch to a list-based algorithm for the prefix $\{ABC\}$ assuming that $\{ABC\}$ is a candidate high-utility itemset. The inverted-list for $\{ABC\}$ can be constructed by joining the list of item A with item B followed by an intersection with the list of item C. The inverted-list of itemset $\{AB\}$ generated during the construction of list for itemset $\{ABC\}$ can be reused to construct the list for another candidate $\{ABD\}$, and many other candidates in the mining process.

List-based algorithms compute an upper-bound score (like the sum of exact-utility and remaining-utility) to decide whether the current prefix and its supersets become high-utility itemset or not. We store those itemsets as blacklisted that have an upper-bound score computed by a list-based algorithm less than the minimum utility threshold. Such a list of itemsets can be used during the mining process to avoid processing any of its supersets by a list-based algorithm. We call the above optimization as early termination strategy.

Table 4.1: Example database

TID	Transaction	TU
T_1	$(B:4)\,(C:4)\,(E:3)\,(G:2)$	13
T_2	$(B:8)\,(C:13)\,(D:6)\,(E:3)$	30
T_3	$(A:5)\,(C:10)\,(D:2)$	17
T_4	$(A:30)\,(B:2)\,(C:1)\,(D:8)\,(H:2)$	43
T_5	$(A:10)\,(C:6)\,(E:6)\,(G:5)$	27
T_6	$(A:10)\,(B:4)\,(D:12)\,(E:6)\,(F:5)$	37

Table 4.2: TWU of items

Item	A	B	C	D	E	F	G	H
TWU	124	123	130	127	107	37	40	43

4.2 Case study: Integration of UP-Hist Growth and UP-Growth+ with FHM

In this section, we will illustrate the integration of tree-based algorithm, UP-Hist Growth [21], and UP-Growth+ [71] with a list-based algorithm, FHM [32] through an example. Consider an example database as shown in Table 4.1, and let θ be 50. The transaction database is scanned to compute the TWU of the the present in it, and the result is shown in Table 4.2. Items F, G, and H are identified as unpromising, and removed from the database. The items in every transaction is sorted according to ascending order of TWU. The ordering of items for our example dataset after the removal of unpromising items is E $\leq$ B $\leq$ A $\leq$ D $\leq$ C.

TID	EU	RU
1	3	8
2	3	27
5	6	16
6	6	26

TID	EU	RU
1	4	4
2	8	19
4	2	39
6	4	22

TID	EU	RU
1	7	4
2	11	19
6	10	22

Figure 4.1: Utility-list of itemset $\{E\}$, $\{B\}$ and $\{EB\}$

The FHM [32] algorithm constructs a utility-list structure for every item. A utility-list stores the following information: a transaction identifier(TID), Exact utility (EU), and remaining utility (RU) for every itemset. The utility-list for promising items is constructed, and the utility-list for a $\{k\}$-itemset is constructed by intersecting the list of two $\{k-1\}$-itemset with the prefix itemset by FHM. The utility-list for itemsets $\{E\}$, $\{B\}$, and $\{EB\}$ is shown in Figure 4.1. FHM also constructs an EUCS data structure to store the TWU for every pair of items. The EUCS structure is utilized to reduce the number of candidates during the mining process by FHM. The EUCS structure for our example database is shown in Figure 4.2. We will first illustrate an example below to integrate UP-Hist Growth with FHM. We call the hybrid algorithm as UPHist-Hybrid.

Item	**C**	**D**	**A**	**B**	**E**
D	88				
A	80	90			
B	82	103	73		
C	63	62	54	73	

Figure 4.2: EUCS data structure

UPHist-Hybrid: The UPHist-Hybrid algorithm constructs the UP-Hist tree and utility-list before starting the mining process. The UP-Hist tree for our example database is shown in Figure 4.3. UPHist-Hybrid processes items from the header table of the UP-Hist tree in a bottom-up manner, and computes node utility, and a $UB(\cdot)$ function for every candidate itemset generated from the UP-Hist tree. The upper-bound function for UPHist-Hybrid is equal to the value returned by the function $maxC(i, s)$ (defined in the Chapter 3) for every item present in the candidate itemset on which the $maxC(i, s)$ function is

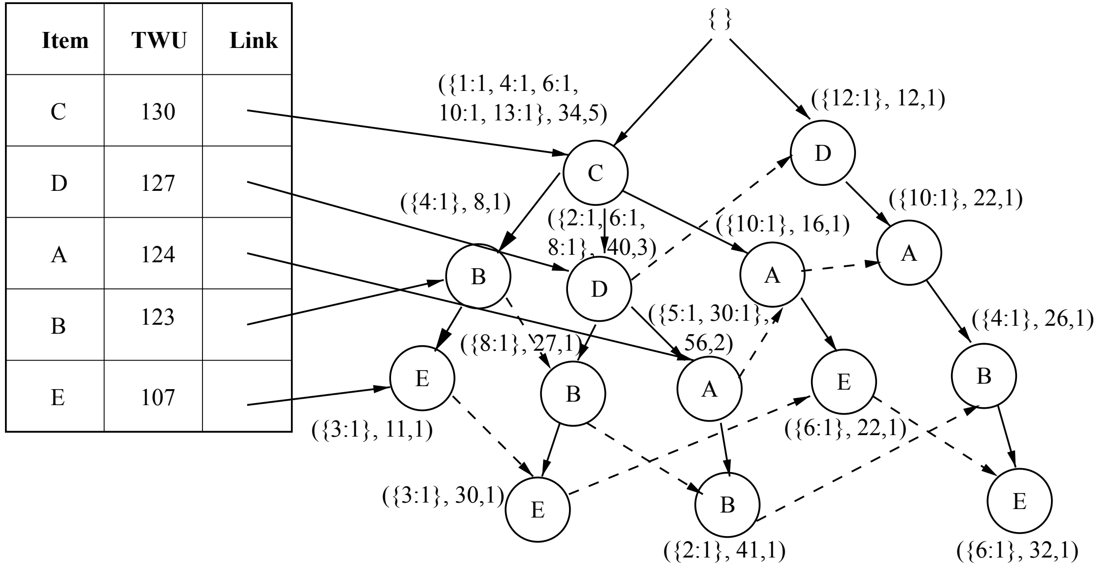

Figure 4.3: Global UP-Hist tree

invoked. The candidate itemsets generated from the UP-Hist tree that have an upper-bound score no less than θ are $\{BA\}$:52, $\{BDC\}$:55, and $\{A\}$:55 respectively. The execution is switched to FHM for further search space exploration. For example, the utility-list of the itemset $\{BA\}$ is constructed, and passed to the FHM algorithm to compute its exact utility, and explore its supersets for high-utility itemsets. The next candidate high-utility itemset generated by FHM with $\{BA\}$ as the prefix itemset is $\{BAD\}$. The set of high-utility itemsets for θ equal to 50 are $\{AD\}$:67, $\{AC\}$:62, $\{BAD\}$:66, $\{ADC\}$:56, and $\{A\}$:55 respectively.

UPG+-Hybrid: The candidates generated from the UP-tree with an upper-bound score no less than θ are $\{EBD\}$:52, $\{EA\}$:72, $\{BA\}$:76, $\{BD\}$:60, $\{BC\}$:63, and $\{A\}$:120 respectively. The UP-Growth [72] defined the minimum item utility as the minimum weight/quantity associated with an item in the transaction database. A similar notion called maximum item utility can be

defined and use it during the implementation of the UPG+-Hybrid algorithm as an $UB(\cdot)$ function. For example, the maximum weight associated with items B, D, and E in our example database is 8, 12, and 6 respectively. The support (s) of the itemset $\{EBD\}$ is 2. Therefore, UB($\{EBD\}$) is 52.

In the next section, we compare the performance of UPG+-Hybrid, and UPHist-Hybrid with the state-of-the-art tree-based and list-based algorithms on several sparse and dense datasets.

4.3 Experiments and Results

In this section, we compare the performance of our proposed algorithms UPG+-Hybrid, and UPHist-Hybrid against the state-of-the-art tree-based algorithms - UP-Growth+ [71], UP-Hist Growth [21], and the state-of-the-art list-based algorithms - HUI-Miner [49], and FHM [32]. The experiments were performed on an Intel Xeon(R) CPU=26500@2.00 GHz with 16 GB RAM and Windows Server 2012 operating system. All datasets except NyTimes used for our experiments were obtained from the SPMF library [31], and the characteristics of the datasets are shown in Table 4.3. The NyTimes dataset was obtained from the UCI Machine Learning repository [29].

Effect of varying minimum utility threshold: We study the effect of varying the minimum utility threshold on the performance of tree-based, list-based, and our proposed hybrid algorithms on several sparse and dense datasets. The result on the sparse datasets is shown in Figure 4.4. The UPG+-Hybrid algorithm

Table 4.3: Characteristics of real datasets

Dataset	#Tx	Avg. length	#Items (I)	Density score R (%) =(A/I)x100 [7]	Type
Retail	88,162	10.3	16,470	0.062	Sparse
Kosarak	9,90,002	8.1	41,270	0.019	Sparse
Chainstore	11,12,949	7.2	46,086	0.015	Sparse
NyTimes	3,00,000	232.2	1,02,660	0.22	Sparse
Chess	3,196	37	75	49.33	Dense
Mushroom	8,416	23	119	19.32	Dense
Connect	67,557	43	129	33.33	Dense
Accidents	3,40,183	33.8	468	7.22	Dense

performs the best in terms of execution time compared to other tree-based and list-based algorithms on sparse datasets. The UPG+-Hybrid algorithm performs at least 10 times faster compared to the tree-based and list-based algorithms on the Retail dataset for the minimum utility threshold equal to 0.01 %. The UPG+-Hybrid algorithm performs at least 15 times faster compared to the list-based algorithms on the ChainStore dataset. The UPG+-Hybrid algorithm performs 1.5 to 2 times faster compared to the UP-Growth+ and UP-Hist Growth algorithm for the minimum utility threshold equal to 0.6 % and 0.01 % on the Kosarak and ChainStore datasets.

The list-based algorithms HUI-Miner and FHM did not terminate for more than 24 hours on the Kosarak dataset for a threshold of less than 7%. Our proposed algorithms UPG+-Hybrid and UPHist-Hybrid, have less execution time compared to other tree-based and list-based algorithms on ChainStore and Retail dataset. The list-based algorithms generate more candidates compared to the tree-based and hybrid algorithms on the sparse datasets as they also generate candidates that are non-existent in the transaction database.

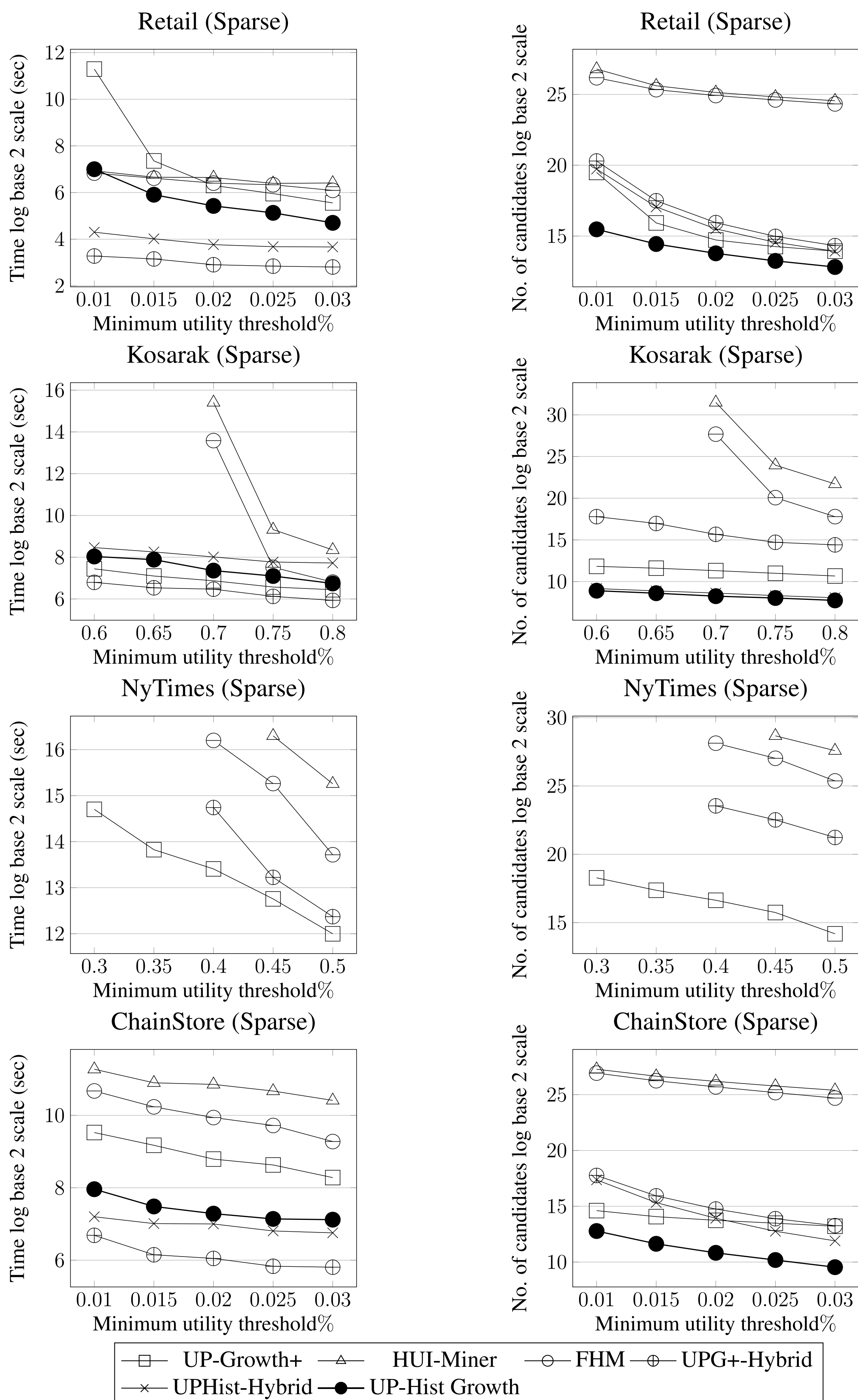

Figure 4.4: Performance evaluation (Time and number of candidates) on sparse datasets. HUI-Miner and FHM did not terminate for more than 24 hours on the Kosarak dataset for threshold less than 0.7%.

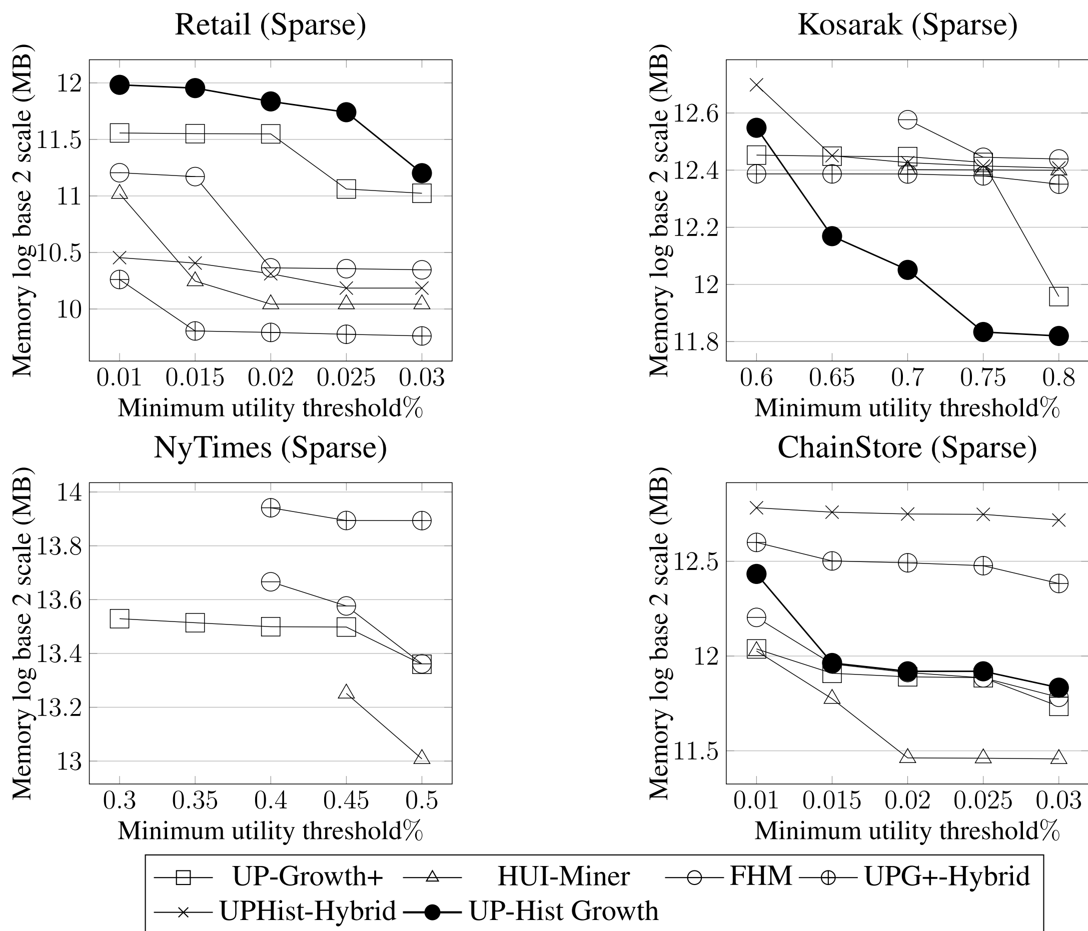

Figure 4.5: Memory consumption on sparse datasets. The UPHist-Hybrid and the UP-Hist Growth algorithms ran out of memory on the NyTimes dataset.

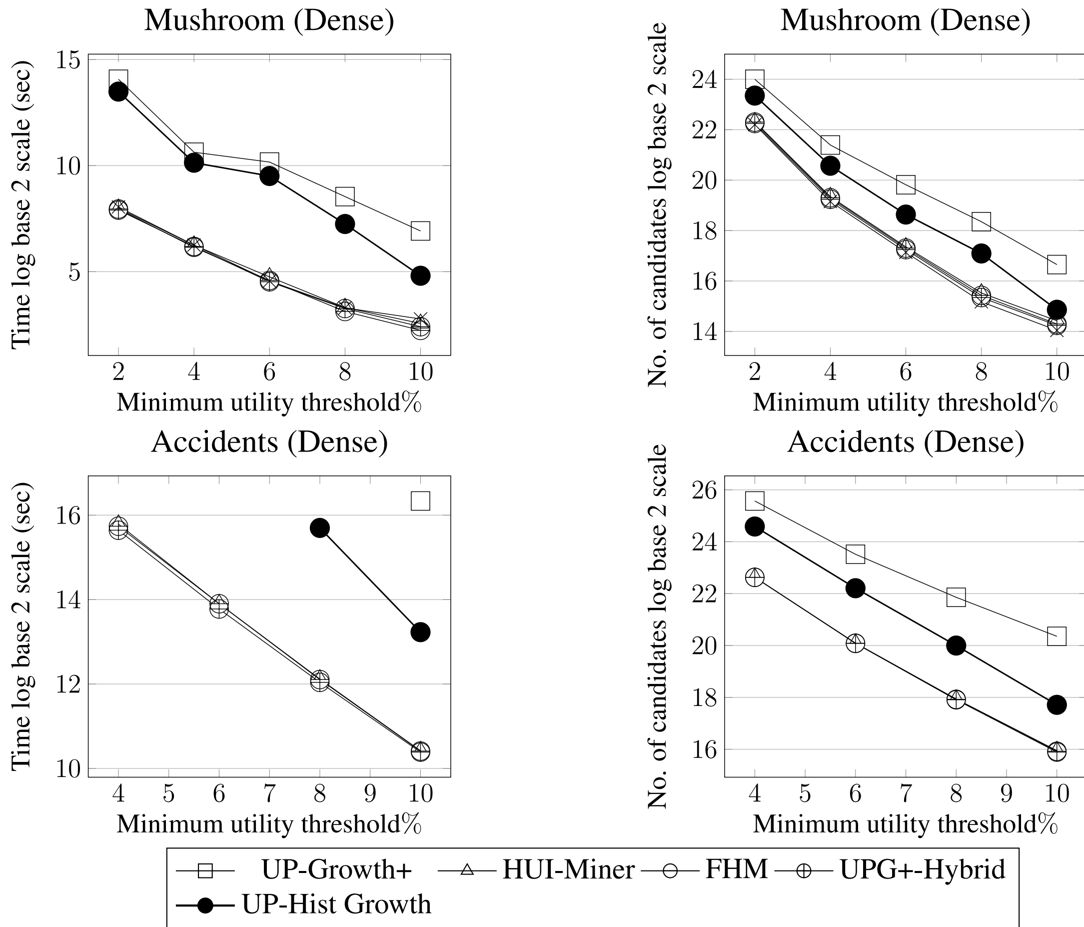

Figure 4.6: Performance evaluation (Time and number of candidates) on dense datasets. UP-Growth+ and UP-Hist Growth did not terminate for more than 24 hours on the Accidents dataset at threshold less than 10%, and 8% respectively. The UPHist-Hybrid algorithm ran out of memory on the Accidents dataset. All algorithms did not terminate their execution for more than 24 hours on the Connect dataset.

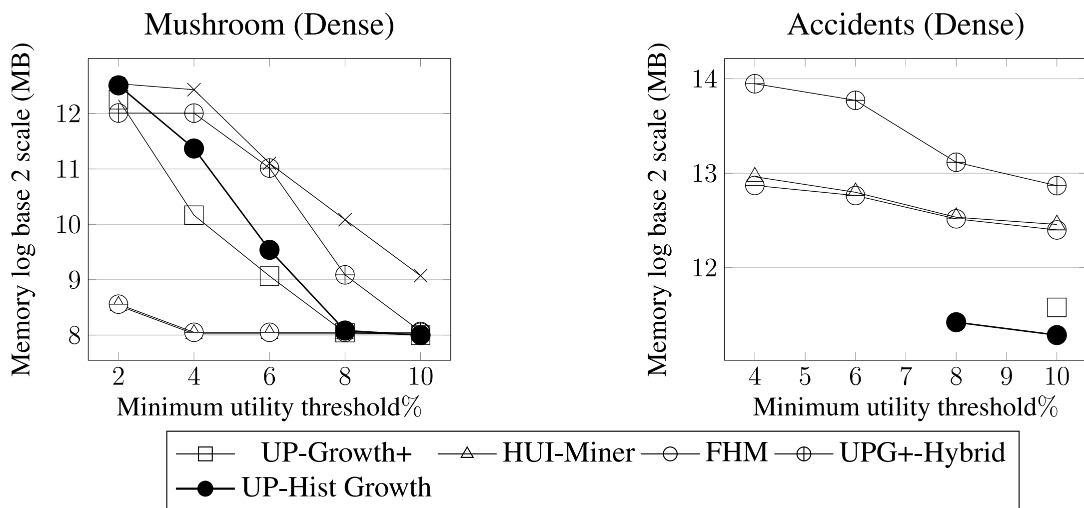

Figure 4.7: Memory consumption on dense datasets. The UPHist-Hybrid algorithm ran out of memory on the Accidents dataset.

It is evident from the results that the tree-based algorithms generate the least candidates on sparse datasets compared to other algorithms as their tree structure guides the algorithm to generate only the candidates existent in the database. The UPG+-Hybrid and UPHist-Hybrid algorithms consume less memory compared to the tree-based algorithms for smaller datasets like Retail and consume more memory than tree-based algorithms for larger datasets like ChainStore as shown in Figure 4.5. We observe that the UPHist-Hybrid generates fewer candidates than the UPG+-Hybrid algorithm, but has total execution time 1.5 to 3 times higher on all sparse datasets. Recall that we observed a similar result in Chapter 3 that the UP-Hist Growth algorithm had more running time even after generating fewer candidates than the UP-Growth+ algorithm on the Kosarak dataset. The time to recursively generate local trees in the candidate generation process can also be an important factor in deciding the relative performance among hybrid algorithms designed from different tree data structures. On the NyTimes dataset, the UP-Hist Growth and the UPHist Hybrid algorithms ran out of memory during the creation of the global tree. The UPG+-Hybrid, FHM and HUI-Miner algorithms did not terminate execution for more than 24 hours on the NyTimes dataset for threshold less than 0.4%, 0.4%, and 0.45%.

The results on dense datasets is shown in Figure 4.6 and 4.7. The list-based and our proposed hybrid algorithms perform at least 15 times faster than the tree-based algorithms on the Mushroom dataset at a threshold of less than 4 %. The UP-Growth+ and UP-Hist Growth did not terminate their execution on the Accidents dataset for more than 24 hours at threshold less than 10 %

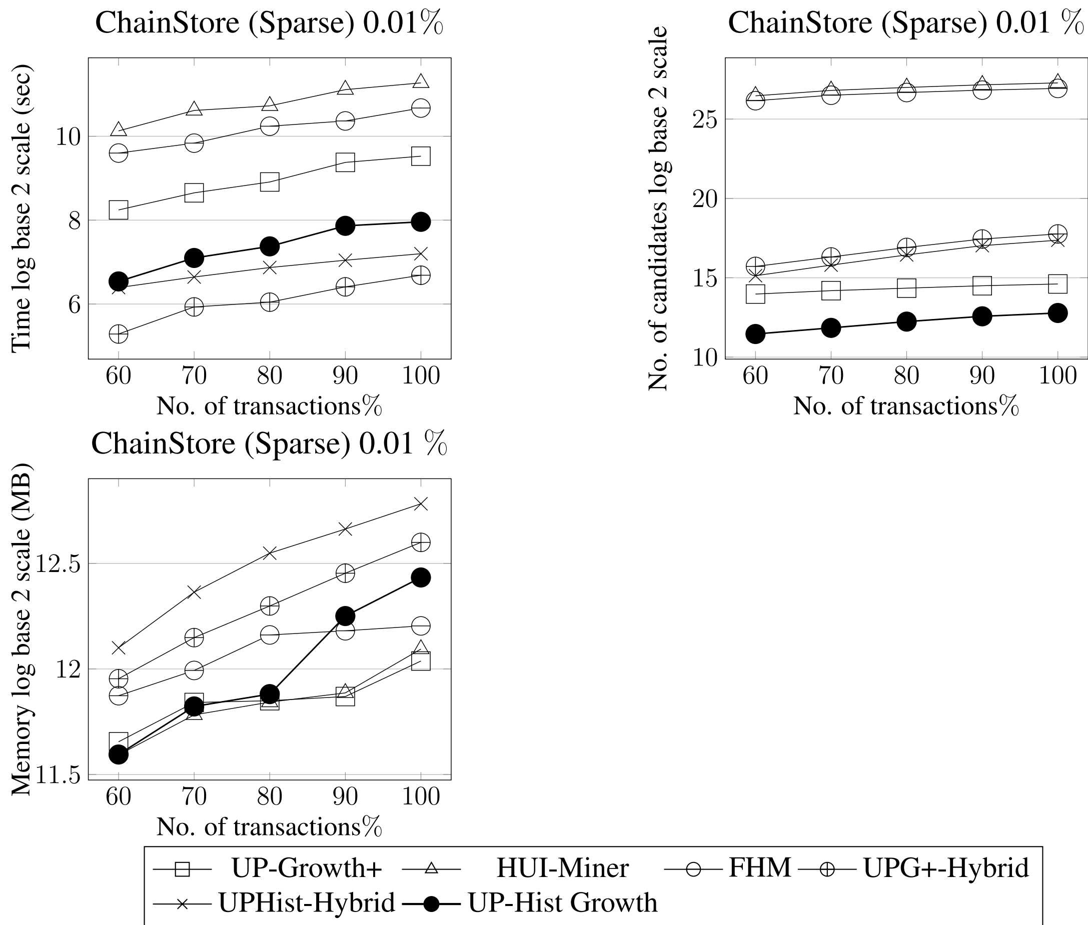

Figure 4.8: Scalability experiment on ChainStore and Accidents dataset for 0.01% and 2% threshold respectively. All algorithms did not terminate for more than 24 hours on the Accidents and NyTimes datasets.

and 8 %, respectively. The performance of list-based and hybrid algorithms is quite similar for dense datasets. The total execution time is correlated with the number of candidates. We observe that the list-based algorithms generate more candidates compared to the hybrid algorithms on sparse datasets, but generate a similar number of candidates on dense datasets.

This behavior can be attributed to the fact that sparse datasets have a large of items with few items in every transaction. Therefore, the list-based algorithms generate a lot more candidates that are non-existent in the dataset on sparse datasets. The UPHist-Hybrid algorithm ran out of memory on the Accidents

dataset. All algorithms did not terminate for more than 24 hours on the Connect dataset. The tree-based and hybrid algorithms did not terminate execution for more than 24 hours on the Chess dataset.

Effect of scalability: We study the effect of scalability on the performance of algorithms and the result is shown in Figure 4.8. The execution time, number of generated candidates, and the memory consumed by every algorithm increases with the number of transactions. UPG+-Hybrid, and UPHist-Hybrid beat the tree-based and list-based algorithms in terms of execution time for the Chain-Store dataset. All algorithms did not terminate the execution for more than 24 hours on the Accidents dataset at 2% threshold.

4.4 Summary

In this chapter, we design an algorithm that can combine any tree-based algorithm with a list-based algorithm to mine high-utility itemsets without a separate verification phase. As a case study, we design and compare the performance of two hybrid algorithms UPG+-Hybrid, and UPHist-Hybrid, against the state-of-the-art tree-based and list-based algorithms on several dense and sparse datasets. Our experiments highlight that hybrid algorithms perform better than the list-based algorithms on sparse datasets and tree-based algorithms on every dataset. We observed that none of the state-of-the-art tree-based, list-based, and our proposed hybrid algorithms terminate its execution in less than 24 hours on the Connect dataset. In the next chapter, we will design an algorithm that maps

the utility-list structure on the top of a tree data structure to mine high-utility itemsets more efficiently on sparse as well as dense datasets.

Chapter 5

A one-phase tree-based algorithm for mining high utility itemsets

Tree-based algorithms like UP-Growth+ [71], and UP-Hist Growth [21] extract the complete set of high-utility itemsets through the candidate generation and verification paradigm. The state-of-the-art tree-based algorithms are two-phase as they do not maintain enough information with each tree node to mine high-utility itemsets in a single phase. Such algorithms perform worse, especially on dense datasets against the state-of-the-art list-based algorithms [49, 32], and projection-based algorithms [85, 48]. Tree-based algorithms also consume more memory due to the construction of complete local trees recursively during the candidate generation phase. In this chapter, we propose a one-phase tree-based algorithms to address the shortcomings mentioned above.

We design a data structure called UT_Mem-tree that stores the complete information compactly with each node and a tree-based algorithm called UT-Miner that can extract high-utility itemsets in one-phase only. We propose

a mechanism to construct a lightweight projected database instead of a complete local tree during the mining process for superior performance, especially for dense datasets. We conduct extensive experiments on several benchmark sparse and dense datasets to compare the performance of our proposed algorithm against the state-of-the-art tree-based, list-based, hybrid, and projection-based algorithms. Our results validate that the proposed UT-Miner algorithm has less total execution time than the tree-based, list-based, and hybrid algorithms on sparse and dense datasets.

5.1 UT_Mem-tree Structure

In chapter 4, we observed that the hybrid algorithms that combine the benefits of tree-based and list-based algorithms perform better than the state-of-the-art tree-based and list-based algorithms. In this section, we propose a data structure called UT_Mem-tree that augments the information stored in an inverted-list data structure like utility-list [49, 32] with each node of the tree. We will observe in the next few sections that the amount of information required to be stored with each node can be reduced during the recursive invocations of the UT-Miner algorithm through a technique called transaction merging [85]. It has been observed that the intersection/join operations to construct data structures like utility-lists [49] is costly and becomes a bottleneck for list-based algorithms like HUI-Miner [49], and FHM [32]. The integration of information stored in inverted-list structures with each node of a tree can reduce the cost of such in-

Table 5.1: Information stored with a node N of a UT_Mem-tree

Field	Description
N.item	item associated with node N
N.gmap	HashMap of key-value pairs (Definition 5.1)
N.local_list	linked-list of local nodes (Definition 5.2)
N.id	unique identifier
N.parent	pointer to the parent node
N.hlink	pointer to a node with same name as N.item

tersection operations. We will observe later in this section that a HashMap can be augmented with each node of the UT_Mem-tree to reduce the cost of such intersection operations by utilizing the following property maintained for each node of the tree structures like UP-tree [71], UP-Hist tree [21], etc.

Property 1: The set of transactions containing a node N of a tree will be present in every ancestor of N. The ancestors for a node N consists of the nodes from the N till the node below the root node of a tree.

5.1.1 The elements of a UT_Mem-tree

Each node N of the UT_Mem-tree stores the information shown in Table 5.1. We define the $gmap$ associated with each node of a UT_Mem-tree, and the procedure to construct the global UT_Mem-tree from a transaction database with an example below.

Definition 5.1 ($gmap$)**.** *N.gmap for an item-node N is a set of tuples* $\langle$ *Tid: (exact-utility (EU),remaining-utility (RU))* $\rangle$*. Tid is the unique transaction identifier associated with a transaction in the database, and the exact-utility (EU) is the utility of the item represented by node* N *in the transaction with identifier*

Tid. The remaining-utility (RU) is the sum of utility of items that appear after the item represented by node N in the transaction with identifier Tid.

5.1.2 The construction of a UT_Mem-tree

The UT_Mem-tree is constructed from a transaction database in two database scans. In the first scan, the TWU [50] score is computed for each item present in the database. The items with a TWU score less than the minimum utility threshold are identified as unpromising items. The unpromising items can not be a part of any high-utility itemset due to the transaction-weighted downward closure property [50] and can be removed from the transaction database. The strategy to remove unpromising items from the transaction database is called the "discarding global unpromising items (DGU)" strategy [71]. The unpromising items are removed and the items within each transaction are sorted according the descending order of their TWU scores. The transactions are inserted one by one to construct the UT_Mem-tree. A header table is also constructed with the UT_Mem-tree for efficient traversal. All nodes with the same label are stored in a linked-list, and $link$ pointer points to the head node in the list. The local_list associated with each node of the UT_Mem-tree is initially empty.

Consider the example database shown in Table 5.2. Let the minimum utility threshold denoted by θ be 50. The TWU for the items present in the database is shown in Table 5.3. The items F, G, and H are identified as unpromising and removed from the transaction database. The item E has the least TWU and item C has the highest TWU. The transactions are sorted and the global UT_Mem-

Table 5.2: Example database

TID	Transaction	TU
T_1	$(B:4)\,(C:4)\,(E:3)\,(G:2)$	13
T_2	$(B:8)\,(C:13)\,(D:6)\,(E:3)$	30
T_3	$(A:5)\,(C:10)\,(D:2)$	17
T_4	$(A:30)\,(B:2)\,(C:1)\,(D:8)\,(H:2)$	43
T_5	$(A:10)\,(C:6)\,(E:6)\,(G:5)$	27
T_6	$(A:10)\,(B:4)\,(D:12)\,(E:6)\,(F:5)$	37

tree is constructed from the transaction database.

Let us analyze the insertion of the first transaction in the tree. The item G is removed from T_1 and the items present in T_1 are sorted according to decreasing order of their TWU score. The transaction labelled T_1 after the removal of unpromising items and sorting of transactions is (C:4, B:4, E:3). The transaction T_1 is inserted into the UT_Mem-tree tree. Initially, the UT_Mem-tree consists of an empty root node only. The node for the first item C is created below the root node and its associated $gmap$ consists of the tuple $\langle$ 1:(4,0) $\rangle$. The exact-utility of item C in T_1 is 4, and the remaining utility is 0. The node for item B is constructed as the child node of C and its associated $gmap$ consists of the tuple $\langle$ 1:(4,4) $\rangle$. The item E is inserted and the $gmap$ associated with it consists of the tuple $\langle$ 1:(3,8) $\rangle$. The remaining-utility for item E in T_1 is the sum of the exact-utility of the items B and C respectively. The global UT_Mem-tree after the insertion of transactions present in the database (Table 5.2) is shown in Figure 5.1. The number shown in brackets with each node of the tree is the unique identifier (ID) associated with it. The $gmap$ (Definition 5.1) associated with each node is also shown in Figure 5.1.

Table 5.3: TWU of items

Item	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**
TWU	124	123	130	127	107	37	40	43

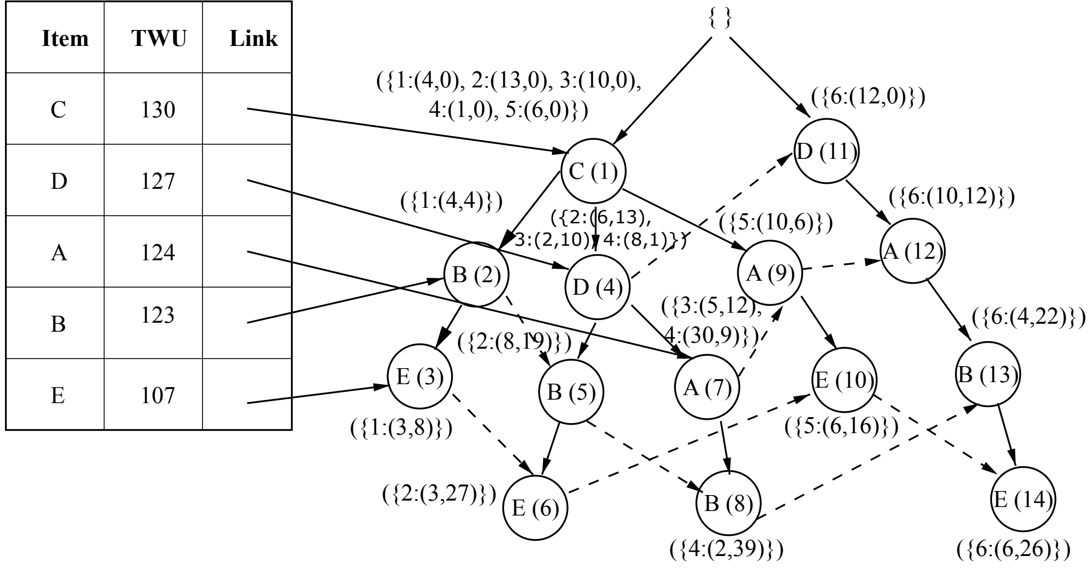

Figure 5.1: Global UT_Mem-tree with gmap associated with each node

5.1.3 Construction of a lightweight projected database through a local_lists

In this subsection, we discuss the structure of a local_list and the procedure to construct it on the top of a UT_Mem-tree to mine high-utility itemsets. The structure of a local_list is defined below.

Definition 5.2 (local_list and lmap)**.** *A linked-list called local_list associated with a node N stores a list of nodes, where each node L present in the local_list stores the following information: 1) a prefix I, 2) a HashMap of key-value pairs L.lmap. A HashMap lmap is a set of tuples ⟨ ID, exact-utility ,remaining-utility, prefix-utility ⟩.*

For example, consider the local_list associated with the ancestors of a node

Table 5.4: local_list for the prefix { B }

Node	Prefix	lmap
C (1)	{ B }	⟨ 2:(4,0,4), 5:(13,0,8), 8:(1,0,2) ⟩
D (4)	{ B }	⟨ 5:(6,13,8), 8:(8,1,2) ⟩
A (7)	{ B }	⟨ 8:(30,9,2) ⟩
D (11)	{ B }	⟨ 13:(12,0,4) ⟩
A (12)	{ B }	⟨ 13:(10,12,4) ⟩

B as shown in Table 5.4. Let us consider the local_list associated with the node $A(7)$ from the global UT_Mem-tree shown in Figure 5.1 to understand the structure through an example. The node $A(7)$ is the ancestor of node $B(8)$. The $lmap$ associated with node $A(7)$ consists of the tuple ⟨ 8:(30,9,2) ⟩, where 8 is the ID of the node $B(8)$, 30 is the exact-utility of $A(7)$, 9 is the remaining-utility of $A(7)$, and 2 is the exact-utility of node $B(9)$. The procedure to construct the local_list on the top of a global UT_Mem-tree is presented as Algorithm 5.1.

The Construct method takes as input a list of nodes associated with an item i, a prefix α, an itemset I, and a UT_Mem-tree T. This method appends a tuple in the local_list for the ancestors of item i. We will explain the Construct method through an example. Let us consider the global UT_Mem-tree as shown in Figure 5.1 and construct The $local_list$ for the ancestors of $I = \{B\}$ with prefix $\alpha = \phi$. The list_nodes_prefix consists of the nodes $B(2)$, $B(5)$, $B(8)$, and $B(13)$. The node $B(2)$ is accessed through the for loop (line 1), and the current_ancestor variable points to the node $C(1)$. The variable id_set = $\{1\}$ and PU is equal to the sum of exact-utility from the $gmap$ associated with the node $B(2)$ in the set of transactions present in the id_set (line 3-5) i.e. 4. The local_list associated with the node $C(1)$ is initially empty and will be ini-

tialized with an entry for the prefix I (line 10-14). The tuple $\langle$ 2:(4,0,4) $\rangle$ is added to the $lmap$ associated with the local_list of the node $C(1)$ (line 15-24), and the current_ancestor variable will be set to the parent of node $C(1)$. The local_list associated with the ancestors of node B is shown in Table 5.4. It can be observed that the set of identifiers present in the id_set will be present in the $gmap$ or local_list associated with every ancestor of the nodes for item i in the UT_Mem-tree due to Property 1. We also compare the performance of creating a lightweight projected database through our Construct (Algorithm 5.1) procedure against a procedure to construct the complete local UT_Mem-tree in Section 5.3. The results validate the superior performance of the proposed Construct method, especially for dense datasets.

5.2 UT-Miner Algorithm

In this section, we present an algorithm called UT-Miner (Algorithm 5.2). The algorithm takes as input a prefix itemset denoted by α, a UT_Mem-tree T, a header table H, a set of possible extensions for α denoted by $hlist$, and a minimum utility threshold indicated by θ. UT-Miner returns the complete set of high-utility itemsets that have α as its prefix. The UT_Mem-tree is constructed from the transaction database in two database scans. In the first scan, the TWU of items present in the database is computed. The items with TWU less than θ called unpromising items will be removed from the database during the next scan. The unpromising items are removed, and transaction merging [85] is per-

Algorithm 5.1 Construct (list_nodes_prefix,I,α,T)

Input: List of nodes associated with item i (list_nodes_prefix), itemset I = $\{\alpha \bigcup i\}$, a prefix α, UT_Mem-tree T
Output: Global UT_Mem-tree with a local_list associated with the ancestors of item i

```
 1: for each node N in list_nodes_prefix do
 2:   current_ancestor = N.parent
 3:   if α == φ then
 4:     id_set=N.gmap.keySet()
 5:     PU = N.sumEU
 6:   else
 7:     id_set=N.local_list.get(α).lmap.keySet()
 8:     PU = N.local_list.get(α).sumEU + N.local_list.get(α).sumPU
 9:   end if
10:   while current_ancestor ≠ T.root do
11:     sumEU=0,sumRU=0
12:     if not(current_ancestor.local_list.contains(I)) then
13:       Create an entry for Itemset I in current_ancestor.local_list and add the tuple ⟨N.id, 0, 0, 0⟩ to
          its lmap
14:     end if
15:     for each tid in id_set do
16:       if α == φ then
17:         sumEU = sumEU + current_ancestor.gmap.get(tid).EU
18:         sumRU = sumRU + current_ancestor.gmap.get(tid).RU
19:       else
20:         sumEU = sumEU + current_ancestor.local_list.get(α).get(tid).EU
21:         sumRU = sumRU + current_ancestor.local_list.get(α).get(tid).RU
22:       end if
23:     end for
24:     Update the tuple ⟨ N.id,sumEU,sumRU,PU ⟩ for the itemset I to current_ancestor.local_list
25:     current_ancestor = current_ancestor.parent
26:   end while
27: end for
```

formed. The transactions are inserted one by one to construct a UT_Mem-tree. A header table is also constructed along with the UT_Mem-tree. The items in the header table are scanned in a bottom-up manner and inserted in the list of extensions denoted as $hlist$. The prefix α passed to UT-Miner after the construction of UT_Mem-tree is empty.

The UT-Miner algorithm picks an item from the $hlist$ (line 1) denoted by $\{i\}$ and appends it to the current prefix α to generate a new itemset denoted by I

(line 2). The nodes associated with the appended item i is scanned by following the links from the header table to compute the exact-utility and remaining-utility for I from the $gmap$ associated with each node in the UT_Mem-tree (line 3). The itemset I will be output as a high-utility itemset if its exact-utility is no less than θ (line 4). The linked-list associated with item $\{i\}$ is scanned again through the header table entry for item $\{i\}$ to compute TWU for the ancestors of $\{i\}$ in the UT_Mem-tree. The ancestors of item $\{i\}$ in the tree will be the set of extensions for the itemset I as tree-based algorithms expand a prefix in a bottom-up manner (line 7). The set of unpromising ancestors are identified and stored in a list called ulist (line 7). The utility of the unpromising items is removed from the sum of exact-utility and remaining-utility upper-bound for itemset I (line 9). If the updated upper-bound score denoted by updatedub is less than θ, the algorithm stops processing for I (line 11) as neither the itemset I nor its supersets can be high-utility itemsets. The computation of a reduced upper-bound score by removing the exact-utility of unpromising items is an application of the DLU strategy [71, 72]. If the updated upper-bound is no less than θ, the $local_list$ for the extensions of I is constructed along with the new $hlist$ and the algorithm UT-Miner is called recursively (line 13-15). After processing the complete set of extensions for a prefix α, the algorithm removes the nodes associated with α in the $local_list$ of its ancestors. We illustrate the working of UT-Miner through an example below.

Consider an example database shown in Table 5.2 and let θ be 50. The TWU of items present in the database is shown in Table 5.3. The items F, G, and H are

Algorithm 5.2 UT-Miner (α,T,H,hlist,θ)

Input: Prefix α (initially empty), a UT_Mem-tree (T), a header table (H) associated with (T), list of extensions to explore (hlist), and a minimum utility threshold θ.
Output: All high-utility itemsets with α as prefix.

```
1:  for each entry {i} in hlist do
2:     Itemset I = α ∪ i.
3:     Store the nodes associated with item i that have α as its descendant in list_nodes_prefix by following
       the links from the header table H. Compute sumEU and sumRU for I from list_nodes_prefix.
4:     if I.sumEU ≥ θ then
5:        Output I as a high-utility itemset.
6:     end if
7:     Compute TWU for the extensions i.e. ancestors of I in T and identify unpromising items (ulist).
8:     Initialize variable ub = I.sumEU+I.sumRU.
9:     Remove the contribution of items in ulist from ub. We represent the updated bound as updatedub.
10:    if updatedub ≤ θ then
11:       Return
12:    end if
13:    T = Construct(list_nodes_prefix,,I,α,T).
14:    Construct the list of extensions for I denoted by hlist_I.
15:    Call UT-Miner(I,T,H,hlist_I,θ).
16: end for
17: Remove the node associated with α from local_list associated with ancestors of α in T.
```

unpromising and will be removed from every transaction before constructing the UT_Mem-tree. The global UT_Mem-tree is shown in Figure 5.1. Let us consider the processing of item $\{B\}$ from the header table. The linked-list associated with the item B is scanned to compute its exact-utility and remaining-utility. The exact-utility and remaining-utility of the itemset $\{B\}$ is 18, and 74 respectively. The itemset $\{B\}$ will be explored further by the UT-Miner algorithm as the sum of its exact-utility, and remaining-utility is more than the minimum utility threshold. Items A, C, and D are extensions for The itemset $\{B\}$ and have TWU values 67, 76, and 94, respectively. The $local_list$ created for the ancestors of prefix $\{B\}$ is shown in Table 5.4.

The UT-Miner algorithm picks the item A from the header table and computes its exact-utility and remaining-utility. The exact-utility and remaining-

Table 5.5: local_list for the prefix { BA }

Node	**Prefix**	**lmap**
C (1)	{ BA }	$\langle$ 7:(1,0,32) $\rangle$
D (4)	{ BA }	$\langle$ 7:(8,1,32) $\rangle$
D (11)	{ BA }	$\langle$ 12:(12,0,14) $\rangle$

utility of the itemset $\{BA\}$ is 46, and 21. The items C and D are the extensions for the itemset $\{BA\}$ and have TWU of 41, and 67 respectively. Item C is identified as an unpromising item, and its utility will be subtracted from the bound (line 10) computed for $\{BA\}$ by UT-Miner. The bound for $\{BA\}$ is 67 after removing the utility of item C. The Construct method (line 13) is called for the itemset I = $\{BA\}$, and $\alpha = \{B\}$. The list_nodes_prefix consists of the nodes A(7), and A(12). The local_list associated with the ancestors for the itemset I is shown in Table 5.5. The itemset $\{BA\}$ will be extended by the item D to generate the itemset $\{BAD\}$. The itemset $\{BAD\}$ is identified as a high-utility itemset by UT-Miner algorithm. The complete set of high-utility itemsets for θ equal to 50 are $\{AD\}$:67, $\{AC\}$:62, $\{BAD\}$:66, $\{ADC\}$:56, and $\{A\}$:55 respectively.

5.3 Experiments and Results

In this section, we compare our proposed algorithm's performance against the state-of-the-art tree-based, list-based, and projection-based algorithms. We compare the performance of UT-Miner against FHM [32], mHUIMiner [61], UPG+-Hybrid [22], HMINER [44], d^2HUP [48], and EFIM [85] on several benchmark sparse and dense datasets. The datasets were downloaded from the SPMF li-

Table 5.6: Characteristics of real datasets

Dataset	#Tx	Avg. length	#Items (I)	Density score R (%) =(A/I)x100 [7]	Type
Retail	88,162	10.3	16,470	0.062	Sparse
Kosarak	9,90,002	8.1	41,270	0.019	Sparse
Chainstore	11,12,949	7.2	46,086	0.015	Sparse
NyTimes	3,00,000	232.2	1,02,660	0.22	Sparse
Chess	3,196	37	75	49.33	Dense
Mushroom	8,416	23	119	19.32	Dense
Connect	67,557	43	129	33.33	Dense
Accidents	3,40,183	33.8	468	7.22	Dense

brary [31], and the UCI Machine Learning repository [29]. The characteristics of the datasets are shown in Table 5.6. The experiments were performed on an Intel Xeon(R) CPU=26500@2.00 GHz with 16 GB RAM and Windows Server 2012 operating system.

Effect of varying minimum utility threshold: We evaluate the performance of UT-Miner against the state-of-the-art high-utility itemset mining algorithms in terms of total execution time, number of candidates, and the main memory consumption.

Comparison with FHM, mHUIMiner, and UPG+-Hybrid: The comparison of UT-Miner with FHM, mHUIMiner, and UPG+-Hybrid algorithms on sparse datasets is shown in Figure 5.2. UT-Miner has the least execution time on all sparse datasets. The UT-Miner algorithm performs at least 4 times and 10 times faster than the mHUIMiner and FHM algorithm at lower threshold values on the Retail dataset. The FHM and mHUIMiner algorithms did not terminate for more than 24 hours at lower thresholds on the Kosarak dataset. The UT-Miner algorithm performs at least 15 times faster than the mHUIMiner and FHM al-

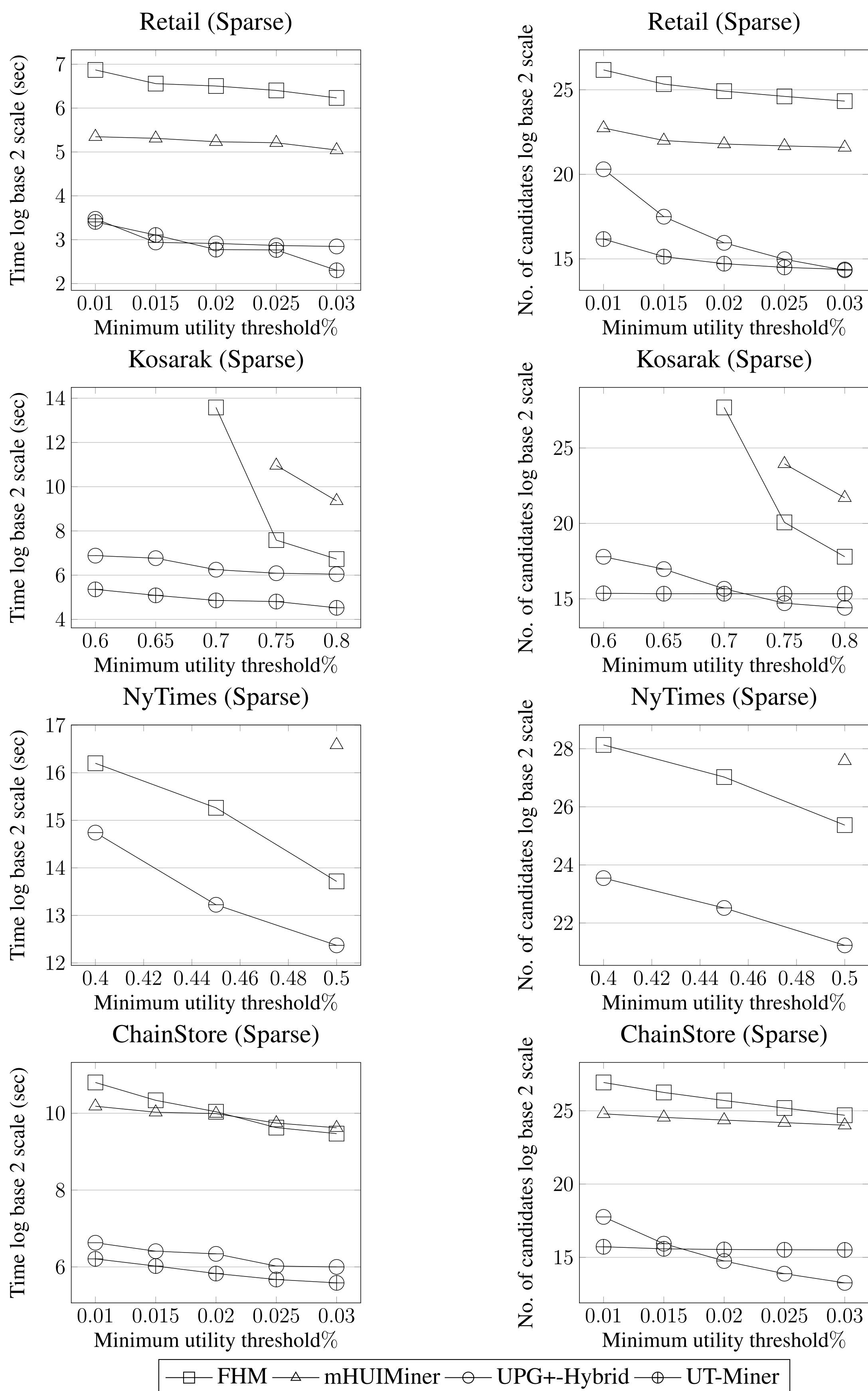

Figure 5.2: Performance evaluation (Time and number of candidates) for FHM, mHUIMiner, UPG+-Hybrid and UT-Miner on sparse datasets. FHM did not terminate for more than 24 hours on Kosarak dataset for threshold value less than 0.7 % and mHUIMiner did not terminate on Kosarak dataset for threshold value less than 0.75 %.

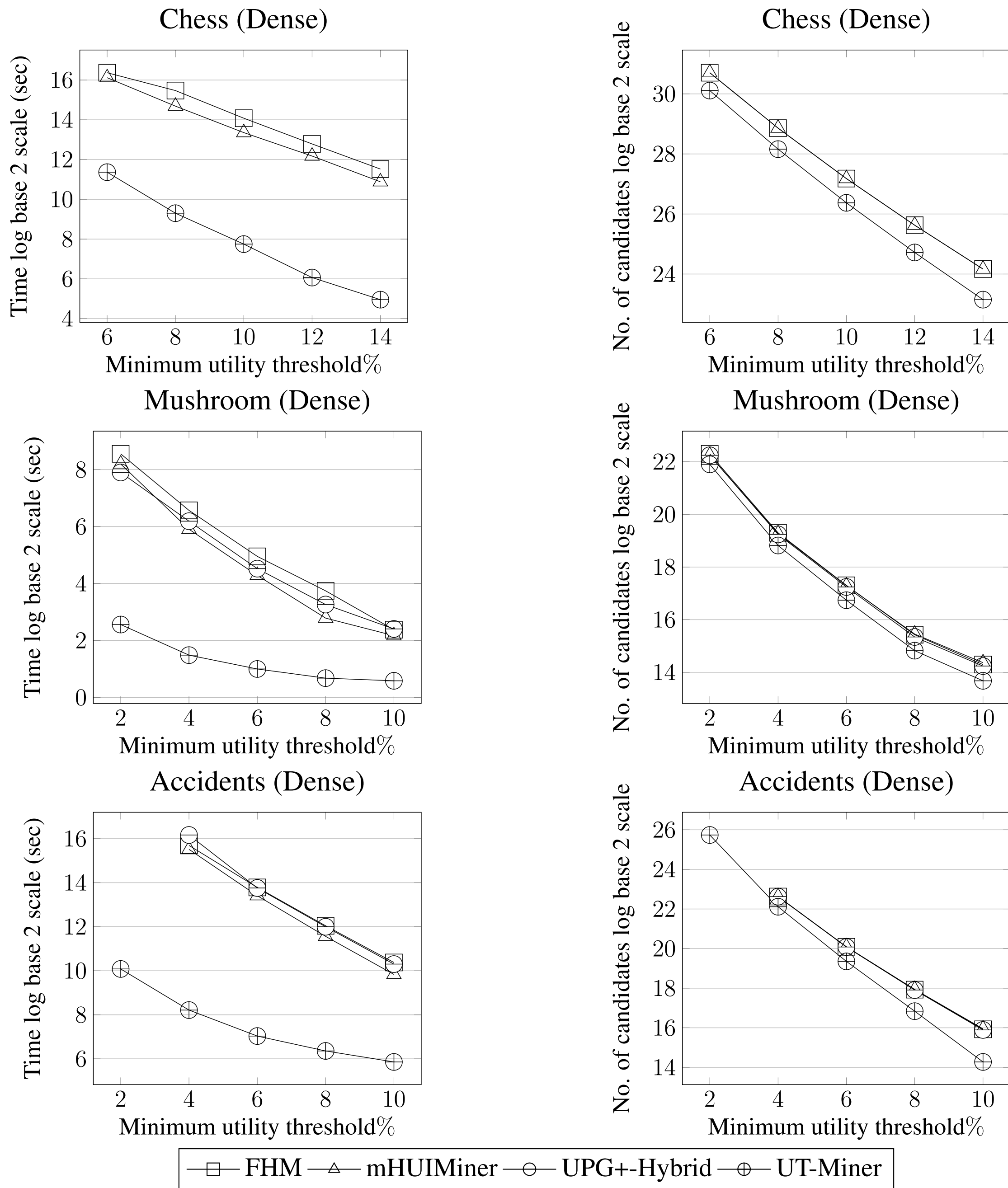

Figure 5.3: Performance evaluation (Time and number of candidates) for FHM, mHUIMiner, UPG+-Hybrid, and UT-Miner on dense datasets. FHM, mHUIMiner, and UPG+-Hybrid did not terminate for more than 24 hours on Connect dataset. FHM, mHUIMiner, and UPG+-Hybrid did not terminate for more than 24 hours on Accidents dataset at 2% threshold.

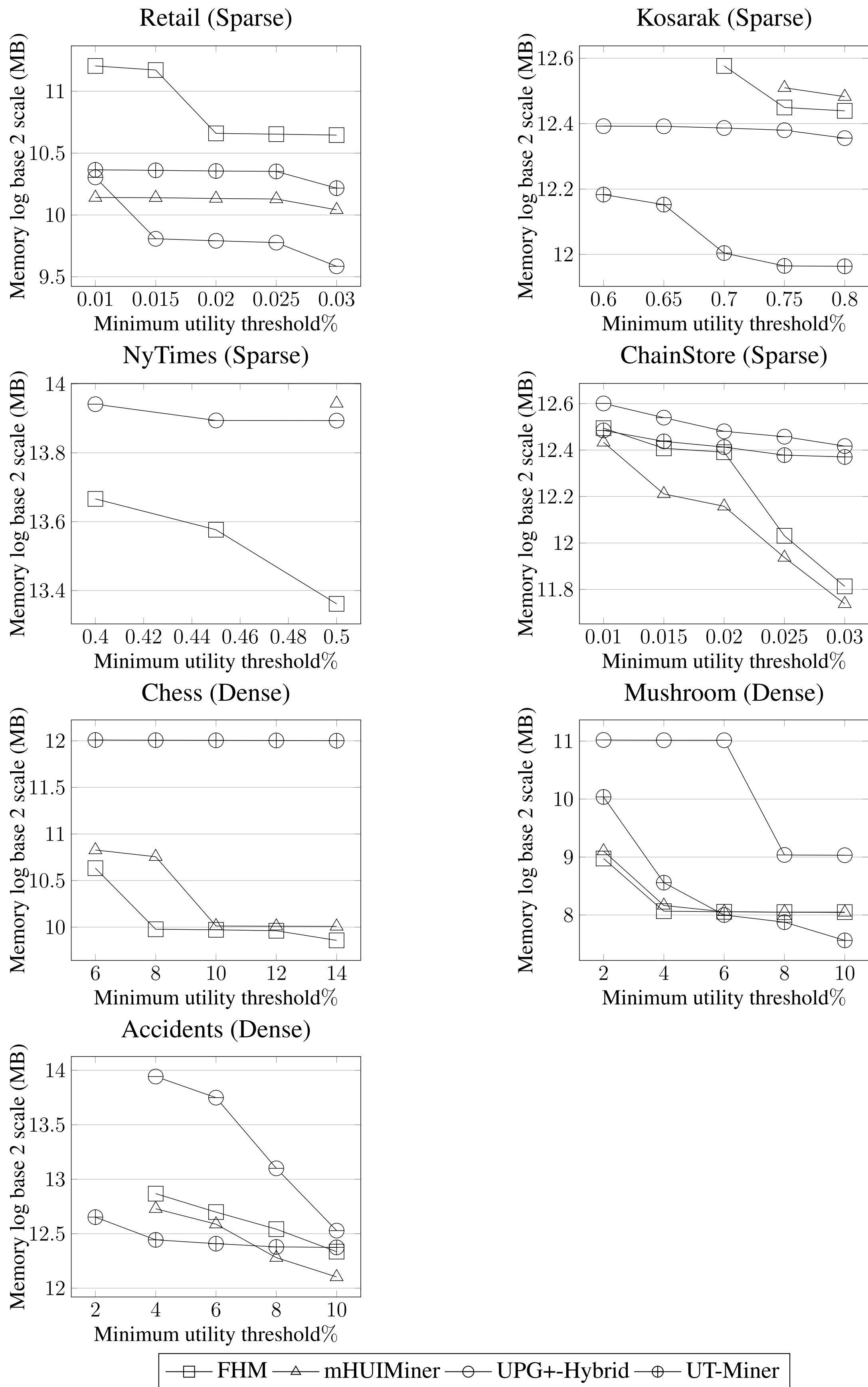

Figure 5.4: Memory Consumption by FHM, mHUIMiner, UPG+-Hybrid and UT-Miner on sparse and dense datasets.

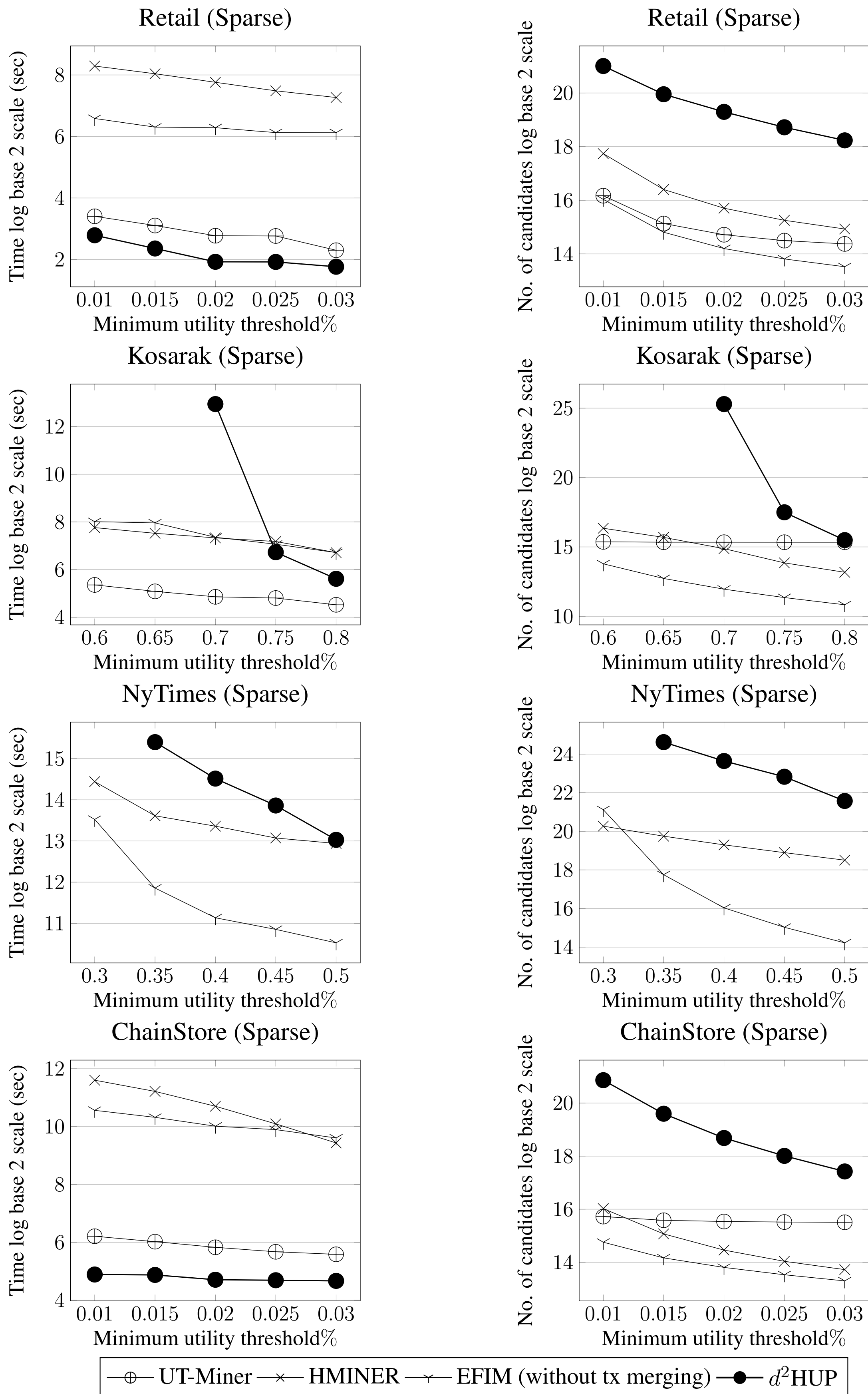

Figure 5.5: Performance evaluation (Time and number of candidates) for EFIM, d^2HUP, HMiner and UT-Miner on sparse datasets. d^2HUP did not terminate for more than 24 hours on Kosarak dataset for threshold less than 0.7 %.

gorithm on the ChainStore dataset at lower thresholds. The UT-Miner algorithm ran out of memory during the construction of the global tree on the NyTimes dataset. The execution time of the algorithms increases with a decrease in the minimum utility threshold as more candidates will be generated for lower thresholds. The UT-Miner algorithm generates the least number of candidates at lower threshold values on sparse datasets. The FHM [32] algorithm generates the maximum number of candidates on the ChainStore and Retail dataset as it also generates candidates that are non-existent in the database. The mHUIMiner [61] algorithm utilizes the IHUP-tree structure to avoid the generation of non-existent itemsets and explores the search space through the HUI-Miner [49] algorithm. The UPG+-Hybrid [22] algorithm switches execution from the UP-Growth+ [71] algorithm to FHM [32] when a candidate high-utility itemset is generated. The UT-Miner algorithm augments the complete information on the UT_Mem-tree to avoid the generation of non-existent candidates in the transaction database and also removes the contribution of unpromising items in every recursive invocation of the algorithm.

The results on dense datasets is shown in Figure 5.3. The UT-Miner algorithm performs at least 25 times faster compared to the mHUIMiner and FHM algorithm on the Chess dataset. The UPG+-Hybrid algorithm did not terminate for more than 24 hours on the Chess dataset. The results for the Connect dataset is not shown as the FHM, mHUIMiner, and UPG+-Hybrid algorithms don't terminate for more than 24 hours. The UT-Miner algorithm performs at least 20 times and 30 times faster than the FHM, and mHUIMiner, and UPG+-Hybrid

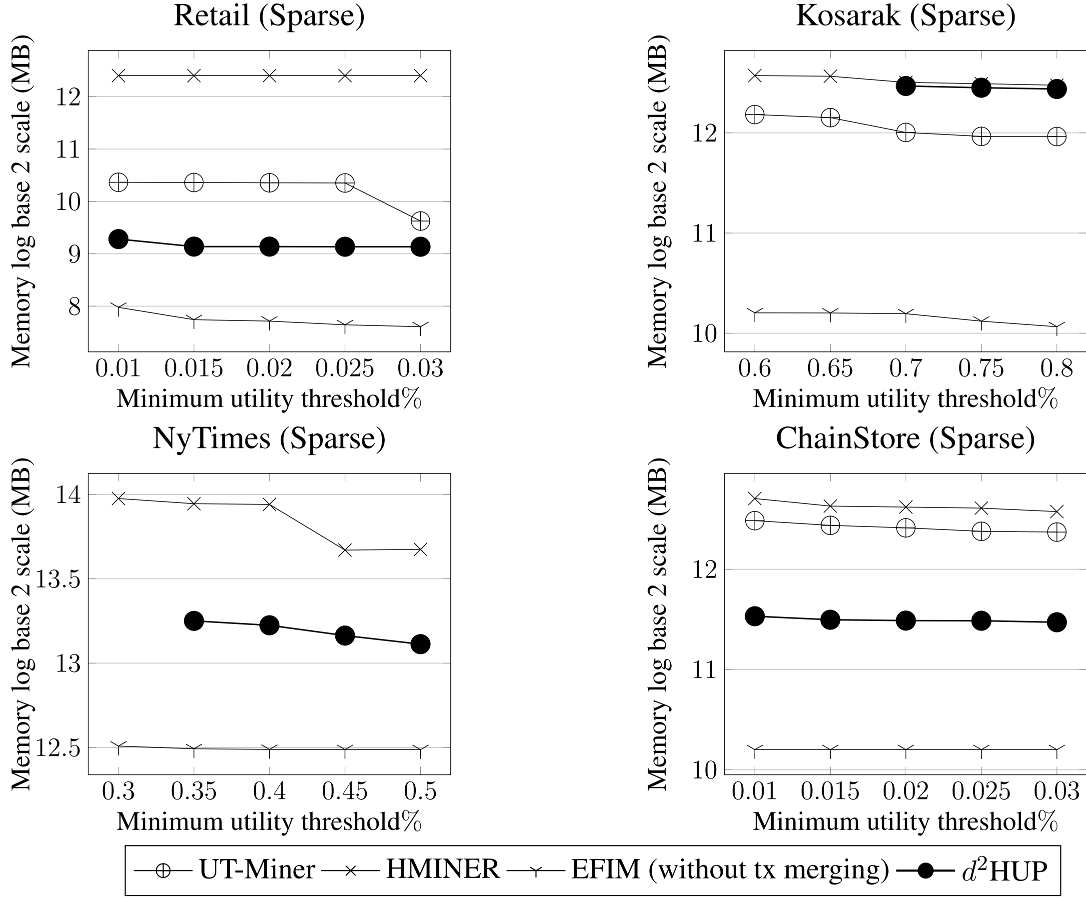

Figure 5.6: Memory Consumption by EFIM, d^2HUP, HMiner and UT-Miner on sparse datasets.

at lower threshold on the Mushroom and Accidents datasets. The UT-Miner algorithm generates the least number of candidates compared to other algorithms on dense datasets. The comparison of different algorithms in terms of memory consumption is shown in Figure 5.4. The UT-Miner algorithm consumes the least amount of memory on the Kosarak and Accidents dataset.

Comparison with EFIM, d^2HUP, and HMINER: The results for sparse datasets is shown in Figure 5.5. UT-Miner has the least execution time for the Kosarak dataset. d^2HUP has the least execution time for the Retail and ChainStore datasets. We compare the performance with transaction merging disabled in

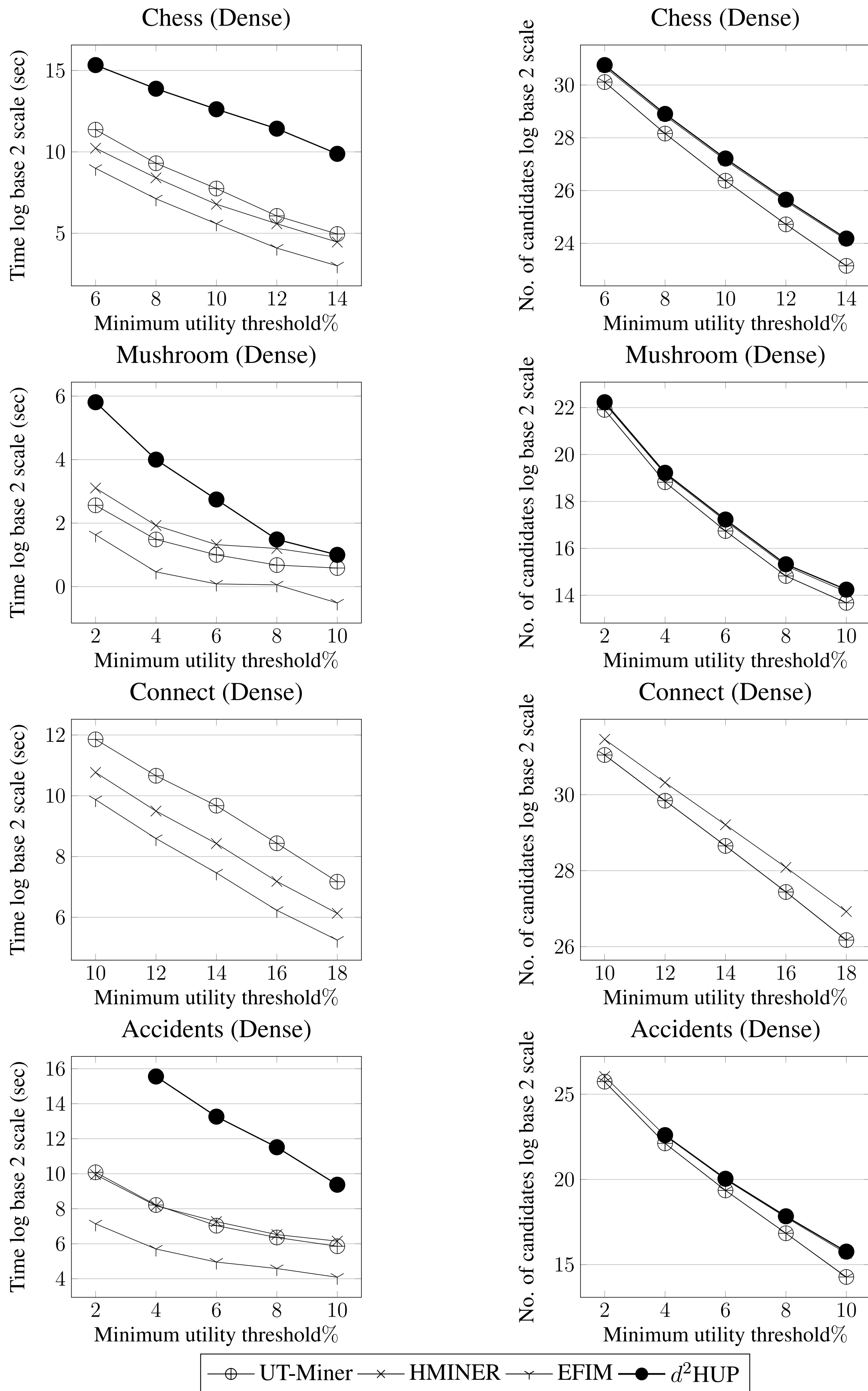

Figure 5.7: Performance evaluation (Time and number of candidates) for EFIM, d^2HUP, HMiner and UT-Miner on dense datasets. d^2HUP did not terminate for more than 24 hours on Connect dataset and Accidents dataset at 2% threshold.

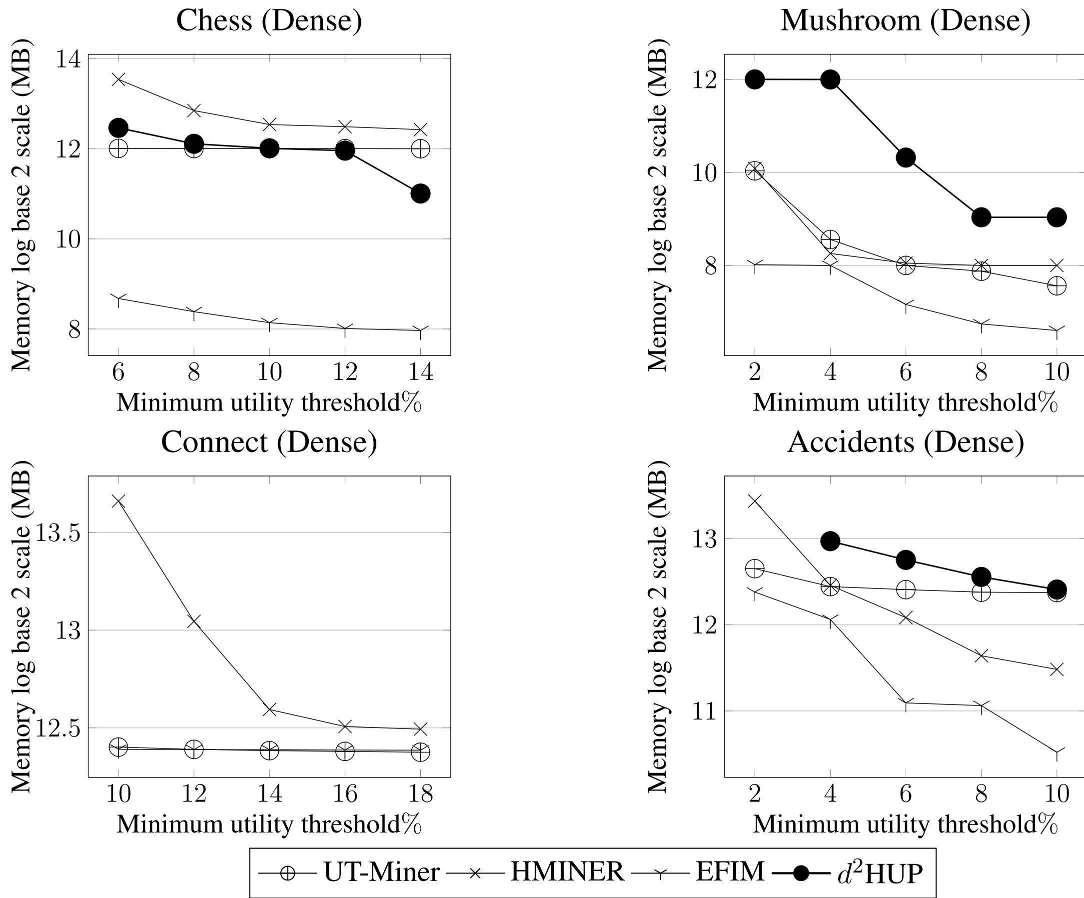

Figure 5.8: Memory Consumption by EFIM, d^2HUP, HMiner and UT-Miner on dense datasets.

the EFIM [85] algorithm. It is known from the literature [85, 22] that EFIM performs better on sparse datasets without its transaction merging feature.

The EFIM algorithm performs a binary search to find an item in every transaction during the projected database creation. d^2HUP [48] is the state-of-the-art algorithm for sparse datasets. It utilizes the hyperlink structure to find the transactions during the projected database creation. The EFIM algorithm generates the least number of candidates for sparse datasets. The memory comparison for sparse datasets is shown in Figure 5.6. The EFIM algorithm consumes the least memory for sparse datasets. The UT-Miner algorithm consumes less memory

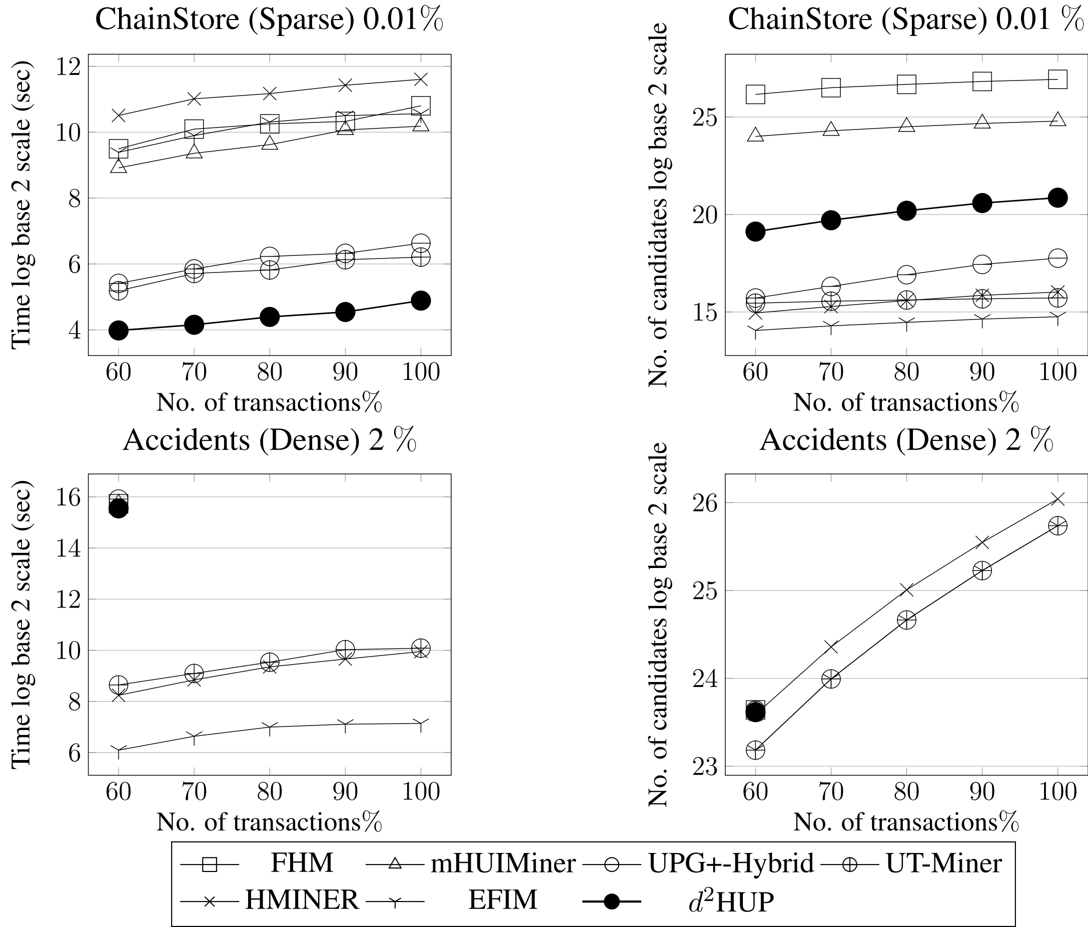

Figure 5.9: Scalability experiment on ChainStore and Accidents dataset for 0.01% and 2% threshold respectively. FHM, mHUIMiner, d^2HUP, and UPG+-Hybrid did not terminate for more than 24 hours on Accidents dataset when more than 60% of the transactions is input to the algorithms.

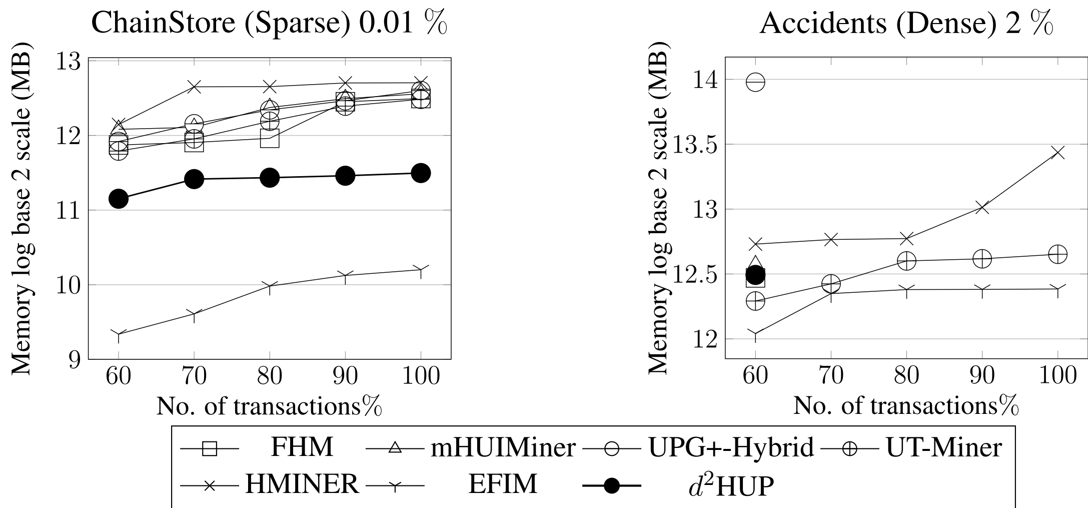

Figure 5.10: Memory consumption on ChainStore and Accidents dataset for scalability

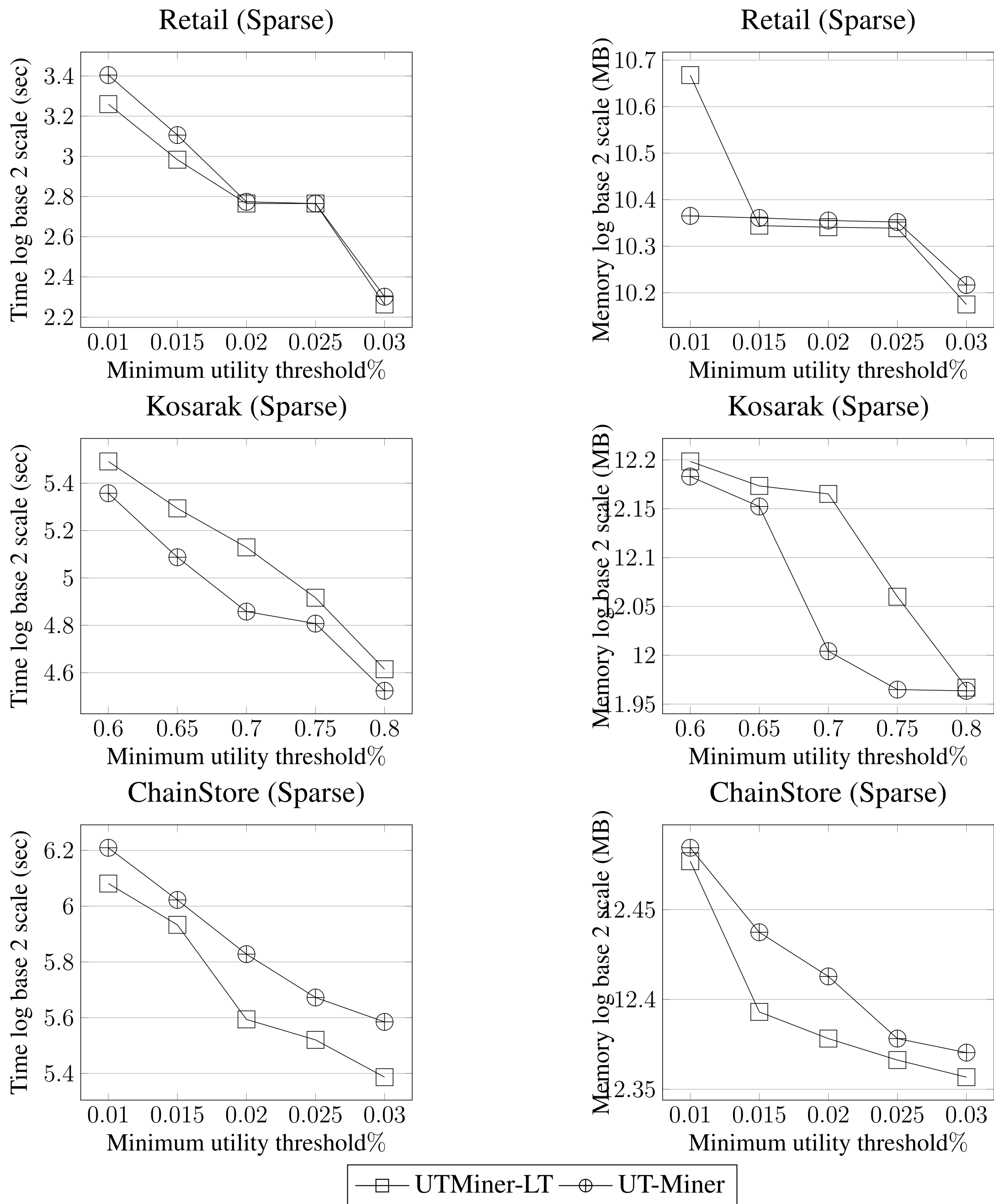

Figure 5.11: Comparison of UT-Miner with UT-Miner-LT on sparse datasets.

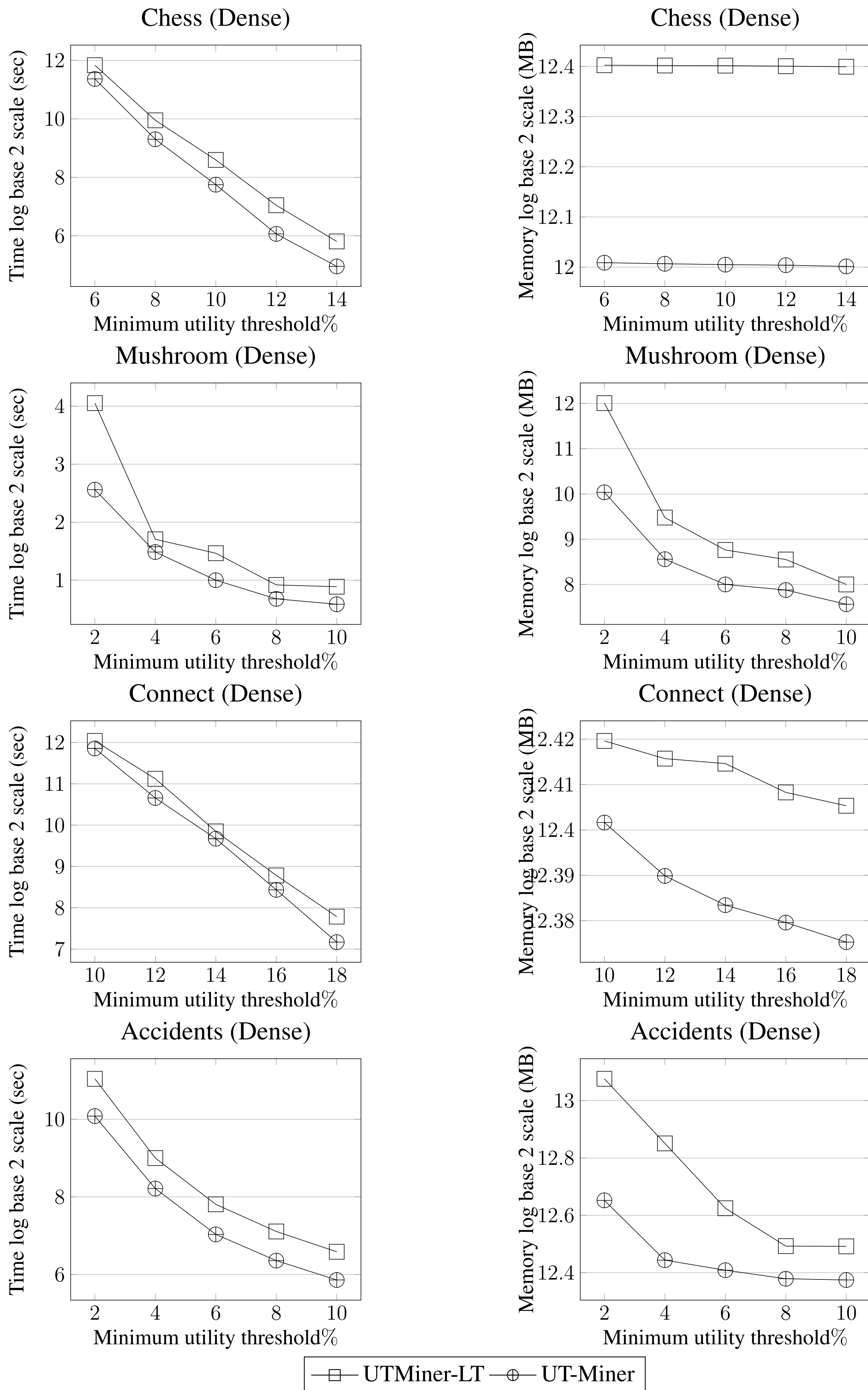

Figure 5.12: Comparison of UT-Miner with UT-Miner-LT on dense datasets.

compared to the HMINER [44] algorithm on all datasets.

The results on dense datasets is shown in Figure 5.7. The d^2HUP algorithm has the highest execution time for dense datasets. The EFIM algorithm has the least execution time as its transaction merging feature works quite well on dense datasets due to higher similarity among transactions. It can be observed from our study that there is no high-utility itemset mining algorithm that performs the best on both sparse and dense datasets. The UT-Miner algorithm is among the top algorithms in terms of total execution time on sparse and dense datasets. The comparison of the algorithms in terms of memory consumption is shown in Figure 5.8. The results demonstrate that EFIM and UT-Miner algorithm have similar memory consumption on the Connect dataset. The EFIM algorithm has the least memory consumption as its transaction merging feature works effectively on dense datasets during the creation of local projected databases.

Effect of scalability: In this set of experiments, we study the influence of varying the number of transactions on the performance of high-utility itemset mining algorithms. The results are shown in Figure 5.9 and 5.10. The total execution time, the number of candidates, and the memory consumption increase with the number of transactions. The FHM, mHUIMiner, d^2HUP, and UPG+-Hybrid algorithms did not terminate their executions for more than 24 hours on the Accidents dataset.

Comparison with UT-Miner-LT: We compare the performance of UT-Miner with a variant of UT-Miner called UT-Miner-LT that constructs a complete

local-tree for every recursive invocation of the algorithm instead of augmenting $local_list$ with each node of the global UT_Mem-tree. The results for sparse and dense datasets is shown in Figure 5.11 and Figure 5.12 respectively. The UT-Miner-LT algorithm executes 1.1 to 1.2 times faster than the UT-Miner algorithm on the Retail and ChainStore dataset. However, the UT-Miner algorithm is 1.4 to 2 times faster compared to UT-Miner-LT on dense datasets. The algorithms generate a large number of candidate high-utility itemsets on dense datasets compared to sparse datasets as dense datasets have average transaction length larger compared to sparse datasets. Theefore, we observe that the UT-Miner algorithm consumes less memory and executes faster compared to UT-Miner-LT on dense datasets.

5.4 Summary

In this chapter, we propose a data structure called UT_Mem-tree and an algorithm called UT-Miner to extract high-utility itemsets in one-phase only. We further propose a lightweight construction procedure for local tree generation during the recursive invocation of UT-Miner. Our experimental study highlight that the UT-Miner algorithm performs better than the tree-based, list-based, and hybrid algorithms on sparse and dense datasets. Our proposed local tree construction procedure reduces the memory consumption and the total execution time of the UT-Miner algorithm on dense datasets.

Till now, we have focused our attention on improving the performance of

high-utility itemset mining algorithms for a specific utility function. In the next chapter, we explore the possibility of designing data structures and algorithms that can mine high-utility itemsets for any subadditive monotone utility function. We design a novel subadditive monotone utility function for mining active, influential groups of users from a Twitter dataset that can be targeted for applications like viral marketing.

Chapter 6

High-utility itemset mining for subadditive monotone utility functions

In this chapter, we generalize the notion of high-utility itemset mining to utility functions that need not be the sum of individual utilities of items. The generalization of the utility function can lead to interesting applications of itemset mining techniques firstly by allowing novel mapping of quantities to utility and, furthermore by allowing the integration of additional information (say, on items) for computing utility. We investigate the problem of high-utility itemset mining for utility functions that are subadditive and monotone (HUIM-SM). Monotonicity is a natural requirement for pattern mining algorithms to prune the search space by defining upper-bound functions like TWU. Subadditivity imposes the constraint that the utility of an itemset X is less than the sum of utility of the itemsets Y and Z where $Y \subseteq X$ and $Z \subseteq X$. The notion of subadditivity appears in the retail domain where customers are offered discounts [9] on purchasing a hamper of products. Subadditive monotone functions have

been used in various domains [16, 76, 17, 11, 62] but not specifically in high-utility itemset mining. So, it is a natural question to ask what such functions can enable us to do in high-utility itemset mining.

In this chapter, we study how itemset mining can be applied for utility functions that are subadditive and monotone (SM). We ask the following questions: Can we design a high-utility itemset mining algorithm for any arbitrary subadditive monotone utility function? Will the existing bounds used for search space exploration still work? We investigate whether the existing tree-based, projection-based, and list-based algorithmic frameworks for HUIM are sufficient to design algorithms for HUIM with SM functions (HUIM-SM). These frameworks are fuelled by “upper bounds” like “Transaction-weighted utility (TWU)" and "Exact-utility, Remaining-utility (EU_RU)". We derive new upper bounds (TSMWU and CU) that are better than TWU and EU_RU and use them to adapt existing tree-based and projection-based algorithms to HUIM-SM. We observe that the information stored in the list structures like utility-list [49, 32] is not sufficient to generalize high-utility itemset for any chosen SM utility function. We define a novel data structure called SMI-list with a lightweight construction method and design a list-based algorithm called SM-Miner.

We show that SM functions can be designed that can capture relationships among items in the form of a relationship graph. We design a function called $ucov$ that can find groups of influential and active users from a social media dataset that can be attractive for applications like viral marketing. We also adapt some of the tree-based, list-based, and projection-based high-utility item-

set mining algorithms and compare their performance on several dense and sparse datasets. Our experimental study reveals that the computation of utility can also be an important factor in the relative performance of algorithms for complex utility functions like $ucov$.

6.1 Problem Statement

Consider the usual setting of HUIM on a transaction database D with n transactions $\{T_1, T_2, \ldots, T_n\}$ over items $I = \{i_1, i_2, ..., i_m\}$. Each transaction can be thought of as a subset of I along with a positive quantity (or weight) associated with every item $i \in T$; we will express T as $\{(i_{j_1} : n_{j_1}), (i_{j_2} : n_{j_2}), \ldots (i_{j_k} : n_{j_k})\}$ where i_j denotes an item in T and $n_j \geq 0$ is its weight in T. By an "itemset X in T" we will mean a set of some items appearing in T and associated with the same weight as that in T. We will use "weighted itemset" to refer to such sets with quantities/weights.

6.1.1 Subadditive and monotone (SM) utility functions

The subadditive monotone functions on a weighted itemset is defined below.

Definition 6.1 (subadditive and monotone function). *Consider functions that map weighted itemsets over I to positive real numbers. Such a function $f(\cdot)$ is subadditive if* $\forall\ X, Y \subseteq T$, $f(X \cup Y) \leq f(X) + f(Y)$. $f(\cdot)$ *is monotone if* $\forall X \subseteq Y \subseteq T$, $f(X) \leq f(Y)$. *A utility function* $u(\cdot, T)$ *is subadditive and monotone if satisfies the above for* every transaction.

We now give a few examples of subadditive and monotone utility functions. First is $Sum(\{x_1 : n_1, \ldots, x_k : n_k\}) = \sum_j n_j$, essentially returning the total quantity of items in an itemset. One should note that this is the utility function used in HUIM.

Theorem 6.1. *The function Sum(·) is monotone and subadditive.*

Proof. Let us consider a transaction T with non-negative real weights associated with items and let us consider a set S ⊆ T and a item v ∈ T such that v ∉ S. $Sum(\cdot)$ is a monotone function as the below equation holds true when the weights associated with items are non-zero real numbers.

$$Sum(S \cup v) \geq Sum(S)$$

We will now prove that $Sum(\cdot)$ is subadditive. Let us consider two sets S and Z such that S ⊆ T and Z ⊆ T. The function $Sum(\cdot)$ is subadditive if.

$$Sum(S \bigcup Z) \ \leq \ Sum(S) \ + \ Sum(Z)$$

The equation below holds When the sets S and Z are disjoint

$$Sum(S \bigcup Z) \ = \ Sum(S) \ + \ Sum(Z)$$

When the sets S and Z are not disjoint, the below equation holds

$$Sum(S \bigcup Z) \ < \ Sum(S) \ + \ Sum(Z)$$

Hence, $Sum(\cdot)$ is subadditive. □

Another example is the $\sqrt{\sum_j n_j}$. It is known that the last function is subadditive and monotone [33].

6.1.2 High-utility itemset mining for SM functions (HUIM-SM)

A utility function $u(\cdot, T)$ is subadditive and monotone for a transaction T if it is a subadditive and monotone function over the items in T. The utility of X in the database is defined as in HUIM: $u(X) = \sum_{\substack{X \subseteq T \\ T \in D}} u(X, T)$. It should be noted that we require $u(X, T)$ to be subadditive and monotone *for any* T but no such property may hold for $u(X)$ i.e. the utility of an itemset X in the database.

HUIM-SM problem: An itemset X is called a high utility itemset if $u(X)$ is no less than a given minimum user-defined threshold denoted by θ. Given a transaction database D, a subadditive monotone utility function $u(\cdot, \cdot)$ and a minimum user-defined threshold θ, the aim is to find all high-utility itemsets.

Current HUIM algorithms work by recursively growing an itemset (called as the prefix) by appending an item from the set of items (I) to the prefix and then determining whether the newly formed prefix (called β) is high-utility or not. For this, a projected database for β is constructed that contains all transactions with itemset β, and the algorithm recursively extends β. Since the search space is exponential in the number of distinct items present in the database, the computational challenge is to efficiently determine whether there could be any high-utility itemset containing β. For this purpose, current HUIM algorithms use upper bound functions of $u(\beta)$ to decide for further exploration.

Table 6.1: Transaction database for coverage utility (ucov)

TID	**Transaction**	TU	$TSMU$	$EU(\{AC\},T)$ + $RU(\{AC\},T)$	$CU(\{AC\},T)$
T_1	$(A:5)\,(C:10)\,(D:2)$	58	37	58	37
T_2	$(A:10)\,(C:6)\,(E:6)\,(G:5)$	97	62	97	62
T_3	$(A:10)\,(B:4)\,(D:12)\,(E:6)\,(F:5)$	139	69	0	0
T_4	$(A:5)\,(B:2)\,(C:3)\,(D:2)\,(H:2)$	47	26	41	26
T_5	$(B:8)\,(C:13)\,(D:6)\,(E:3)$	99	58	0	0
T_6	$(B:4)\,(C:4)\,(E:3)\,(G:2)$	42	27	0	0
T_7	$(F:1)\,(G:2)$	9	7	0	0
T_8	$(F:4)\,(G:3)$	21	15	0	0

Even if the current HUIM algorithms are adapted for HUIM-SM, it is unclear if the current bounds will be correct for a different, general, utility function. Since a correct, and preferably tight, bound is crucial for efficient pruning of search space and also depends on the data structures used, careful attention must be given to come up with an appropriate bound function. We undertake this task in the next few sections.

6.2 Coverage: A graph-based utility function

Now, we describe a subadditive monotone utility function called $ucov$ that not only depends upon the items but also allows us to incorporate any additional information on the relationship between the items into the utility function. We design such a function to highlight that itemset mining can be used by pattern mining researchers and practitioners to find interesting groups in different domains. Such knowledge may be useful to find active communities in a social network.

Given an undirected unweighted graph G with vertices V, graph coverage

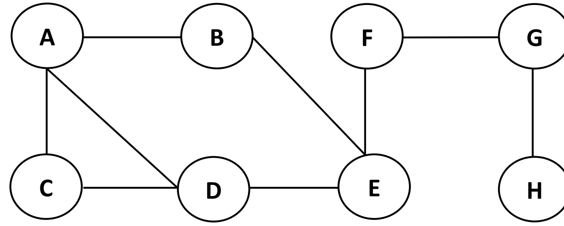

Figure 6.1: Graph over items

$(Co(X))$ of a subset of vertices $X \subseteq V$ is defined as the cardinality of the set containing X [1] and the immediate neighbors of the vertices in X. For example, coverage of $\{A, C\}$ in the graph shown in Figure 6.1 is equal to $|\{A, B, C, D\}|$ = 4.

Theorem 6.2. *The function Co(·) is monotone and subadditive.*

Proof. Let us consider a set S $\subseteq$ I and a item v $\in$ I such that v $\notin$ S. $Co(\cdot)$ is a monotone function if and only if,

$$Co(S \cup v) \geq Co(S)$$

Let us consider Co(S) and v. If item v is not an immediate neighbor for any node in the set S, then

$$Co(S \cup v) > Co(S)$$

If v is a immediate neighbor for any node in S, then Co(S) = Co(S$\cup$v). Hence, $Co(\cdot)$ is a monotone function.

We will now prove that $Co(\cdot)$ is subadditive. Let us consider two sets S and T such that S $\subseteq$ I and T $\subseteq$ I, where I is the set of items which forms vertices of

[1] $Co(X)$ returns the cardinality of a set containing X to ensure that $Co(X)$ of an itemset X is greater than zero.

the graph G. The function $Co(\cdot)$ is subadditive if.

$$Co(S \bigcup T) \leq Co(S) + Co(T)$$

The equation below holds when Co(S) and Co(T) have no vertex in common

$$Co(S \bigcup T) = Co(S) + Co(T)$$

The below equation holds when $Co(S) \bigcap Co(T) \neq \phi$.

$$Co(S \bigcup T) < Co(S) + Co(T)$$

Hence, $Co(\cdot)$ is subadditive. $\square$

Coming back to high-utility itemset mining, suppose in addition to a transaction database we are also given an additional graph G over the items (objects) that capture their pairwise relationships (e.g., see Figure 6.1). Assume that a transaction defines a set of users along with their frequency of activities on a particular day. A follower-followee graph can be constructed with users as vertices and a directed edge from a vertex X to vertex Y denotes that user Y follows user X. Now, we define a utility function called $ucov$ that combines the quantity information in the database and relationship among the users in terms of the coverage information in G. The function $ucov$ captures the notion of minimal coverage/influence of a set of active users on their immediate neighbours and mined patterns can be used as a target for marketing purposes. Functions like $ucov$ can be designed to capture sets of influential users who can be given

incentives to promote a product in their immediate neighbourhood [62].

We do a comparative analysis of the patterns generated by using $ucov$ on Twitter dataset against existing baseline functions like frequency, HUIM and some of our-defined functions in Section 6.5. Please note that $ucov$ is our designed example of a subadditive monotone function. The study in this thesis is valid for any subadditive monotone function.

Definition 6.2 (Coverage utility of an itemset ($ucov$))**.** *Let $T = \{(x_1 : q_1), (x_2 : q_2), \ldots (x_n : q_n)\}$ be a transaction such that $\forall i \in \{1 \ldots n\}$, $q_i > 0$. Suppose that the items are ordered [2] such that $q_i \leq q_{i+1}$ for all i. Let X be an itemset with k items from T: $X = \{x_1, x_2, \ldots, x_k\}$. We define the* coverage utility *of X in T in the following manner.*

$$ucov(X, T) = q_1 \times Co(\{X\}) + \sum_{j=2}^{k} (q_j - q_{j-1}) \times Co(\{x_j \cdots x_k\})$$

For example, $ucov(ACD, T_1)$ in the database in Figure 6.1 can be computed as $2 \times Co(\{DAC\}) + 3 \times Co(\{AC\}) + 5 \times Co(\{C\}) = 2 \times 5 + 3 \times 4 + 5 \times 3 = 37$.

Lemma 6.1. *Let $T = \{(A_1 : q_1), \ldots (A_n : q_n), (B_1 : r_1), \ldots (B_m : r_m)\}$ be a transaction with positive quantity associated with every item in T. Suppose that the items are ordered such that $q_1 \leq \cdots \leq q_n \leq r_1 \leq \cdots \leq r_m$. Let X be an itemset with n items from T: $X = \{A_1, A_2, \ldots, A_n\}$. Let Y be an itemset with m items from T: $Y = \{B_1, B_2, \ldots, B_m\}$. Then, ucov(X,T) + ucov(Y,T) $\geq$ ucov(X$\cup$Y,T).*

[2] We order the items in a transaction T and define $ucov$ as per Definition 6.2 to make the subsequent proofs for $ucov$ easier to understand.

Proof. Let us analyze the terms in ucov(X,T), ucov(Y,T), and ucov(X∪Y,T). The first term in ucov(Y,T) is $(r_1)\times$Co($\{T\}$). The first term in ucov(Y,T) can be expanded such that ucov(X,T) + ucov(Y,T) $\geq$ ucov(X∪Y,T) for the first n+1 terms as shown in Table 6.2 as $(r_1)\times$Co($\{T\}$) = $(q_1)\times$Co($\{T\}$) + $(q_2 - q_1)\times$Co($\{T\}$) + $\cdots$+ $(r_1 - q_n)\times$Co($\{T\}$). The remaining terms in ucov(Y,T) and ucov(X∪Y,T) are equal. □

Table 6.2: Comparison of ucov(X,T) + ucov(Y,T) with ucov(X∪Y,T) (Lemma 6.1)

Quantity	**ucov(X,T)**	+	**ucov(Y,T)**	$\geq$	**ucov(X∪Y,T)**
$m_1 = q_1$	$m_1 \times Co(\{A_1 \cdots A_n\})$	+	$\boldsymbol{m_1 \times Co(\{Y\})}$	$\geq$	$m_1 \times Co(\{A_1 \cdots A_n \cup Y\})$
$m_2 = q_2 - q_1$	$m_2 \times Co(\{A_2 \cdots A_n\})$	+	$\boldsymbol{m_2 \times Co(\{Y\})}$	$\geq$	$m_2 \times Co(\{A_2 \cdots A_n \cup Y\})$
.					
$m_n = q_n - q_{n-1}$	$m_n \times Co(\{A_n\})$	+	$\boldsymbol{m_n \times Co(\{Y\})}$	$\geq$	$m_n \times Co(\{A_n \cup Y\})$
$m_{n+1} = r_1 - q_n$			$\boldsymbol{m_{n+1} \times Co(\{Y\})}$	=	$m_{n+1} \times Co(\{Y\})$
$m_{n+2} = r_2 - r_1$			$m_{n+2} \times Co(\{B_2 \cdots B_m\})$	=	$m_{n+2} \times Co(\{B_2 \cdots B_m\})$
.					
$m_m = r_m - r_{m-1}$			$m_m \times Co(\{B_m\})$	=	$m_m \times Co(\{B_m\})$

Table 6.3: Example (Lemma 1)

Quantity	**ucov(X,T)**	+	**ucov(Y,T)**	$\geq$	**ucov(X∪Y,T)**
$m_1 = 2$	$2 \times Co(\{A_1, A_2, A_3\})$	+	$\boldsymbol{2 \times Co(\{B_1, B_2\})}$	$\geq$	$2 \times Co(\{A_1, A_2, A_3, B_1, B_2\})$
$m_2 = 1$	$1 \times Co(\{A_2, A_3\})$	+	$\boldsymbol{1 \times Co(\{B_1, B_2\})}$	$\geq$	$1 \times Co(\{A_2, A_3, B_1, B_2\})$
$m_3 = 1$	$1 \times Co(\{A_3\})$	+	$\boldsymbol{1 \times Co(\{B_1, B_2\})}$	$\geq$	$1 \times Co(\{A_3, B_1, B_2\})$
$m_4 = 1$			$\boldsymbol{1 \times Co(\{B_1, B_2\})}$	=	$1 \times Co(\{B_1, B_2\})$
$m_5 = 3$			$3 \times Co(\{B_2\})$	=	$3 \times Co(\{B_2\})$

For example, consider a transaction $T = \{(A_1 : 2), (A_2 : 3), (A_3 : 4), (B_1 : 5), (B_2 : 8)\}$, $X = \{A_1, A_2, A_3\}$, and $Y = \{B_1, B_2\}$ respectively. Refer Table 6.3 for an example for Lemma 6.1.

Theorem 6.3. *The function ucov(·,T) is subadditive.*

Proof. Let T be a transaction with positive quantity associated with every item in T. Let the items in T be sorted in ascending order of quantity. Let $X \subseteq T$

and $Y \subseteq T$. ucov(X∪Y,T) will find an item with minimum quantity either from set X or Y in transaction T. ucov(X∪Y,T) will find items with minimum quantity from one set X(or Y) followed by an item from the other set Y(or X) respectively. Using Lemma 1, the terms in ucov(X,T) and ucov(Y,T) can be expanded such that ucov(X,T) + ucov(Y,T) $\geq$ ucov(X∪Y,T). Hence, ucov(·,T) is a subadditive function. □

For example, consider a transaction $T = \{(A_1 : 2), (A_2 : 3), (B_1 : 4), (A_3 : 7), (B_2 : 8), (A_4 : 9), (B_3 : 10)\}$, $X = \{A_1, A_2, A_3, A_4\}$, and $Y = \{B_1, B_2, B_3\}$ respectively. Refer Table 6.4 for an example for Theorem 6.3.

Table 6.4: Example (Theorem 6.3)

Quantity	ucov(X,T)	+	ucov(Y,T)	$\geq$	ucov(X∪Y,T)
$m_1 = 2$	$2 \times Co(\{A_1, A_2, A_3, A_4\})$	+	$\mathbf{2 \times Co(\{B_1, B_2, B_3\})}$	$\geq$	$2 \times Co(\{A_1, A_2, A_3, A_4, B_1, B_2, B_3\})$
$m_2 = 1$	$1 \times Co(\{A_2, A_3, A_4\})$	+	$\mathbf{1 \times Co(\{B_1, B_2, B_3\})}$	$\geq$	$1 \times Co(\{A_2, A_3, A_4, B_1, B_2, B_3\})$
$m_3 = 1$	$\mathbf{1 \times Co(\{A_3, A_4\})}$	+	$\mathbf{1 \times Co(\{B_1, B_2, B_3\})}$	$\geq$	$1 \times Co(\{A_3, A_4, B_1, B_2, B_3\})$
$m_4 = 3$	$\mathbf{3 \times Co(\{A_3, A_4\})}$	+	$\mathbf{3 \times Co(\{B_2, B_3\})}$	$\geq$	$3 \times Co(\{A_3, A_4, B_2, B_3\})$
$m_5 = 1$	$\mathbf{1 \times Co(\{A_4\})}$	+	$\mathbf{1 \times Co(\{B_2, B_3\})}$	$\geq$	$1 \times Co(\{A_4, B_2, B_3\})$
$m_6 = 1$	$\mathbf{1 \times Co(\{A_4\})}$	+	$\mathbf{1 \times Co(\{B_3\})}$	$\geq$	$1 \times Co(\{A_4, B_3\})$
$m_7 = 1$			$1 \times Co(\{B_3\})$	=	$1 \times Co(\{B_3\})$

Theorem 6.4. *ucov(·,T) is a monotone function.*

Proof. Let $T = \{(A_1 : q_1), \ldots (A_n : q_n), (B_1 : r_1)\}$ be a transaction with positive quantity associated with every item in T. Suppose that the items are ordered such that $q_1 \leq \cdots \leq q_n$ and $q_{i-1} \leq r_1 \leq q_i$. Let X be an itemset with n items from T: $X = \{A_1, A_2, \ldots, A_n\}$. Let Y be an itemset from T: $Y = \{B_1\}$. Let us analyze the terms ucov(X,T) and ucov(X∪Y,T) as shown in Table 6.5. It can be quickly verified that the terms from 1st till $i - 1^{th}$ rows in Table 6.5 from ucov(X,T) is less than or equal to ucov(X∪Y,T) as $Co(\cdot)$ is mono-

tone (Theorem 6.2). Let us consider the ith term in ucov(X∪Y,T). It can be observed that the next term from ucov(X,T) can be expanded such that ucov(X,T) $\leq$ ucov(X∪Y,T) as $(q_i - q_{i-1})\times$Co($\{A_i \cdots A_n\}$) = $(q_i - r_1)\times$Co($\{A_i \cdots A_n\}$) +$(r_1 - q_{i-1})\times$Co($\{A_i \cdots A_n\}$). Hence, ucov(·,T) is a monotone function.

□

Table 6.5: Comparison of ucov(X,T) with ucov(X∪Y,T) (Theorem 6.4)

Quantity	**ucov(X,T)**	$\leq$	**ucov(X∪Y,T)**
$m_1 = q_1$	$m_1 \times Co(\{A_1 \cdots A_n\})$	$\leq$	$m_1 \times Co(\{A_1 \cdots A_n \cup Y\})$
$m_2 = q_2 - q_1$	$m_2 \times Co(\{A_2 \cdots A_n\})$	$\leq$	$m_2 \times Co(\{A_2 \cdots A_n \cup Y\})$
.			
$m_{i-1} = q_{i-1} - q_{i-2}$	$m_{i-1} \times Co(\{A_{i-1} \cdots A_n\})$	$\leq$	$m_{i-1} \times Co(\{A_{i-1} \cdots A_n \cup Y\})$
$m_i = r_1 - q_{i-1}$	$\boldsymbol{m_i \times Co(\{A_i \cdots A_n\})}$	$\leq$	$m_i \times Co(\{A_i \cdots A_n \cup Y\})$
$m_{i+1} = q_i - r_1$	$\boldsymbol{m_{i+1} \times Co(\{A_i \cdots A_n\})}$	=	$m_{i+1} \times Co(\{A_i \cdots A_n\})$
$m_{i+2} = q_{i+1} - q_i$	$m_{i+2} \times Co(\{A_{i+1} \cdots A_n\})$	=	$m_{i+2} \times Co(\{A_{i+1} \cdots A_n\})$
.			
$m_{n+1} = q_n - q_{n-1}$	$m_{n+1} \times Co(\{A_n\})$	=	$m_{n+1} \times Co(\{A_n\})$

6.3 Bounds for HUIM-SM

Consider the $Sum()$ function used by HUIM algorithms. The $Sum()$ function does not follow the anti-monotonicity property. A superset of a low-utility itemset can have high-utility. Hence, high-utility itemset mining algorithms define upper-bounds like TWU [50], exact-utility summed with remaining utility [49] to prune the search space. We derive new upper bounds that are better than the bounds designed for HUIM. We now discuss these bounds in the context of a general subadditive and monotone utility function $u(X, T)$.

6.3.1 TU and TWU bounds

The TU of a transaction is the *sum* of the utilities of its items. TWU of an itemset X is the sum of TU for transactions that contains X. The *transaction-weighted downward closure* property of TWU ensures that if $TWU(X)$ is less than the threshold, X and its super-sets cannot be high-utility itemsets [50]. We now define a related concept of transaction subadditive-monotone utility.

Definition 6.3 (TSMU and TSMWU)**.** *The transaction subadditive monotone utility of a transaction T is defined as $TSMU(T) = u(T, T)$. The transaction subadditive monotone weighted utility (TSMWU) of an itemset X is the sum of $TSMU(T)$ for all the transactions containing X.*

For example, consider transaction T_1 from Table 6.1. TU for T_1 for $ucov$ is equal to $ucov(\{A\}, T_1) + ucov(\{C\}, T_1) + ucov(\{D\}, T_1)$ which evaluates to 20 + 30 + 8 = 58. TSMU for T_1 for $ucov$ is equal to $ucov(\{ACD\}, T_1)$ which evaluates to 37. Observe that TSMU(T) = TU(T) when $Sum()$ is used as the utility function. Now we show that $TSMWU$ satisfies the downward-closure property making it useful in mining algorithms and a tighter bound compared to TWU for an arbitrary $u(\cdot, \cdot)$.

Lemma 6.2. *If TSMWU(X) is less than the threshold, X and its super-sets cannot be high-utility itemsets.*

Proof. For any transaction T and any $X \subseteq X' \subseteq T$, $u(X', T) \leq TSMU(X)$ as the set of transactions containing X' is always a subset of the transactions con-

taining X and $u(\cdot, T)$ is a monotone function. Therefore, $u(X') \leq TSMWU(X)$ for all $X \subseteq X'$. □

Lemma 6.3. *$TSMWU(X)$ is a tighter upper-bound compared to $TWU(X)$.*

Proof. This can be easily proved since $TU(X) = \sum_{T \in D} \sum_{x \in T} u(x, T)$ $\geq \sum_{T \in D} u(T, T) = TSMU(X)$ due to subadditivity. Therefore, $TSMWU(X) \leq TWU(X)$. □

6.3.2 Exact-utility (EU) and Remaining-utility (RU) bounds

List-based [49, 32] and projection-based [85, 48] HUIM algorithms use the Exact-utility (EU) and Remaining-utility (RU) bounds that are tighter compared to the TU bound explained earlier. These algorithms process items according to some fixed order and items in each transaction are sorted accordingly; let X be some itemset that is being processed, T be some transaction containing X and T/X be the *items appearing after* X in T. One should note that $X \cup (T/X)$ is not T, in particular, items appearing in T *before* X in the processing order are not in T/X. The exact utility $EU(X, T)$ is the *sum* of the utilities of the items in X in T and the remaining utility $RU(X, T)$ is defined as the *sum* of the utilities of the items in T/X. It is known that X and its extensions (according to the processing order) cannot be high utility if $EU_RU(X) = \sum_{T \supseteq X} EU(X, T) + RU(X, T)$ is less than the threshold [49]; this fact is used in HUIM algorithms to decide whether to examine extensions of the currently explored itemset X. Now, we define combined utility in an

analogous manner for subadditive monotone utility functions and analyze its applicability for itemset mining.

Definition 6.4 (Combined utility (CU)). *Given an itemset X and a transaction T with $X \subseteq T$, the combined utility of X is defined as $CU(X,T) = u(X \cup T/X, T)$ and $CU(X) = \sum_{T \supseteq X} CU(X,T)$.*

For example, consider transaction T_1 from Table 6.1. Let us compute $EU(\{AC\}, T_1)$ + $RU(\{AC\}, T_1)$, and $CU(\{AC\}, T_1)$.
$EU(\{AC\}, T_1) + RU(\{AC\}, T_1)$ evaluates to 58, and $CU(\{AC\}, T_1)$ evaluates to 37.

Lemma 6.4. *If $CU(X)$ is less than a threshold, any extension X' of X in the processing order of items is not a high-utility itemset.*

Proof. Let X' be some extension of X in the processing order of items; since transactions are also (implicitly) stored in the same order, $X' \subseteq X \cup T/X$. The proof of the lemma follows easily since the set of transactions containing X' is always a subset of the set of transactions containing X and $u(\cdot, T)$ is a monotone function which implies that $u(X', T) \leq u(X \cup T/X, T) = CU(X,T)$. □

Lemma 6.5. *$CU(X)$ is a tighter upper-bound compared to $EU_RU(X)$.*

Proof. First observe that subadditivity implies that $EU(X,T) = \sum_{x \in X} u(x,T)$ $\geq u(X,T)$. Similarly it follows that $RU(X,T) \geq u(T/X,T)$. Therefore, we immediately get $EU(X,T) + RU(X,T) \geq u(X,T) + u(T/X,T) \geq u(X \cup$

$T/X, T) = CU(X, T)$ (second inequality is again due to subadditivity). Therefore it follows that $EU_RU(X) \geq CU(X)$. $\square$

It should be noted that $CU(X) = EU_RU(X)$ when Sum is used as the utility function. The results of Subsections 6.3.1 and 6.3.2 show that we can use $TSMWU$ in place of TWU and CU in place of EU_RU in HUIM algorithms. We illustrate these bounds in our example transaction database presented in Table 6.1 for $ucov$.

6.4 Algorithms for HUIM-SM

Existing algorithms can be categorized into tree-based, list-based and projection-based algorithms. We show that the data structures and the corresponding algorithms can be adapted with suitable change in the upper-bound function.

6.4.1 List-based algorithm

List-based algorithms like HUIM and FHM [49, 32] etc. construct a data structure called utility-list for every itemset explored during the mining process that consists of tuples (Transaction id (T), Exact-utility (EU(X,T)), Remaining-utility (RU(X,T))). The utility-list for a $\{k\}$-itemset (an itemset consisting of k items) is constructed by intersecting lists of two $\{k-1\}$ itemsets Px and Py that have a prefix itemset P in common. Another intersection operation with the utility-list of the prefix itemset P is also performed to avoid the double count-

Table 6.6: SMI-List of $\{A\}$

TID	CWI	RWI
1	$\{(A:5)\}$	$\{(C:10),(D:2)\}$
2	$\{(A:10)\}$	$\{(C:6),(E:6),(G:5)\}$
3	$\{(A:10)\}$	$\{(B:4),(D:12),(E:6),(F:5)\}$
4	$\{(A:5)\}$	$\{(B:2),(C:3),(D:2),(H:2)\}$

Table 6.7: SMI-List of $\{B\}$

TID	CWI	RWI
3	$\{(B:4)\}$	$\{(D:12),(E:6),(F:5)\}$
4	$\{(B:2)\}$	$\{(C:3),(D:2),(H:2)\}$
5	$\{(B:8)\}$	$\{(C:13),(D:6),(E:3)\}$
6	$\{(B:4)\}$	$\{(C:4),(E:3),(G:2)\}$

ing of EU(P) in EU(Pxy), i.e. EU(Pxy,T)=EU(Px)+EU(Py)-EU(P). Two variables sumEU(X), and sumRU(X) store the sum of EU(X,T), and RU(X,T) for all transactions which contain itemset X.

Table 6.8: SMI-List of $\{AB\}$

TID	CWI	RWI
3	$\{(A:10),(B:4)\}$	$\{(D:12),(E:6),(F:5)\}$
4	$\{(A:5),(B:2)\}$	$\{(C:3),(D:2),(H:2)\}$

To support the efficient calculation of utility for any sub-additive and monotone function, we propose a data structure called SMI-list. For each transaction T that contains X, the SMI-list stores a tuple of the form (Transaction Id (T), Current Weighted Itemset (CWI), Remaining Weighted Itemset (RWI)). The Current Weighted Itemset of an SMI-list for an itemset X stores the item-quantity information for all items in X. The Remaining Weighted Itemset stores the items with their quantities which appear after X in a transaction. The variables SumEU, and SumCU accumulate the the exact EU and CU value during the construction of the inverted-list for an itemset X. The variable SumEU is

used to decide whether to report the itemset at the output using the utility threshold value. And, the variable SumCU is used to decide whether to explore further. A variable CU stores the Combined Utility (CU(X)) bound with the utility-list for every itemset X. The construction process is presented as Algorithm 6.1. We observe that there is no need to scan the SMI-list of the prefix while constructing the list for a k-itemset from $\{k-1\}$-itemsets, unlike required by some algorithms like HUI-Miner, FHM [49, 32] for HUIM. It is due to the reason that a set union operation is performed when the SMI-list of two $\{k-1\}$-itemsets are joined. The SMI-List for the itemsets $\{A\}$, $\{B\}$, and $\{AB\}$ are shown in Table 6.6, 6.7, and 6.8 respectively.

Algorithm 6.1 Construct-SMI-List (Ix,Iy,$f(\cdot)$)

Input: L_{Ix}: SMI-List of k-1 itemset Ix, L_{Iy}: SMI-List of k-1 itemset Iy, $f(\cdot)$: subadditive monotone utility function.
Output: L_{Ixy}: SMI-List of k itemset Ixy.

```
1: L_Ixy = {}
2: for each element Ex in L_Ix do
3:    if ∃Ey ∈ L_Iy and Ex.Tid == Ey.Tid then
4:       Exy = ⟨Ex.Tid, Ex.CWI ∪ Ey.CWI, Ey.RWI⟩
5:       L_Ixy.append(Exy)
6:       Ixy.sumEU+ = f(Ex.CWI ∪ Ey.CWI)
7:       Ixy.sumCU+ = f(Ex.CWI ∪ Ey.CWI ∪ Ey.RWI)
8:    end if
9: end for
10: Return Ixy
```

Our proposed algorithm called SM-Miner (Algorithm 6.2) takes as input a prefix (say I), a list of SMI-Lists for the 1=extensions of the prefix (L_{Ext_I}, a minimum utility threshold (θ), and a subadditive monotone utility function ($f(\cdot)$) and mines pattern using depth first search strategy. The 1-extensions of a prefix are the items that can extend the currently explored prefix itemset by an itemset mining algorithm. Initially, the database is scanned to compute

the TSMWU of the items present in the transaction database as TSMWU is a tighter bound then TWU as per Lemma 6.3. Another database scan is performed to remove items with TSMWU less than θ (Lemma 6.2) and construct the SMI-List of the remaining items present in the transaction database to get L_{Ext_I} with I being empty. The algorithm SM-Miner explores the search space in a depth-first search manner and returns the complete set of high-utility itemsets. During each step, SM-Miner constructs the SMI-List of a $\{k\}$-itemsets from SMI-List of two $\{k-1\}$-itemsets. Two variables SumEU, and SumCU accumulate the EU and CU values during the construction of the SMI-List of an itemset as CU is a tighter bound (Lemma 6.5). If SumCU for an itemset X is less than θ, X and its supersets can not be high-utility as per Lemma 6.4.

Algorithm 6.2 SM-Miner (I, L_{Ext_I}, θ,$f(\cdot)$)

Input: Prefix I (initially empty), L_{Ext_I}:List of SMI-Lists of I 1-extensions, θ: a user-specified threshold, $f(\cdot)$: subadditive monotone utility function.
Output: the set of high-utility itemsets with I as prefix.

1: **for** each SMI-List L_{Ix} in L_{Ext_I} **do**
2:    **if** $Ix.sumEU \geq \theta$ **then**
3:       Ix is a high-utility itemset and report in output
4:    **end if**
5:    **if** $Ix.sumCU \geq \theta$ **then**
6:       L_{Ext_Ix}=$\emptyset$
7:       **for** each SMI-List L_{Iy} after L_{Ix} in Ext_I **do**
8:          L_{Ixy}=Construct-SMI-List(Ix,Iy,$f(\cdot)$)
9:          L_{Ext_Ix}=$L_{Ext_Ix} \cup L_{Ixy}$
10:       **end for**
11:       SM-Miner (Ix,L_{Ext_Ix},θ,$f(\cdot)$)
12:    **end if**
13: **end for**

6.4.2 Tree-based algorithm

A two-phase tree-based algorithm like UP-Growth and UPGrowth+ [71] generate candidate high-utility itemsets in the first phase and compute their utility through another database scan in the second phase. The tree-based algorithms construct a global tree data structure from transaction database and store an upper-bound estimate like TWU with each node to generate candidates quickly. Tree-based algorithms remove unpromising items during the mining process through techniques like DGU, DLU [71]. An itemset X is unpromising in a transaction database if $TWU(X)$ is less than the minimum utility threshold θ. Local trees are recursively created from the global tree to generate candidates in the first phase. Tree-based algorithms remove unpromising items during global and local tree creation to compute better utility estimates. We observe that unpromising items can be removed during global tree creation as it is possible to recompute the utility of a transaction after removing the unpromising items for any arbitrary subadditive function.

However, we observe that removing unpromising items during local tree creation may not give correct upper-bound estimates for any arbitrary subadditive function, even though it works for the $Sum()$ function. Imagine a tree-based algorithm at an intermediate stage and let Y denote a node (rather, a path) in the tree. Further, consider a case where an item A appears on the path Y and, at that intermediate stage, A was found to be unpromising. The tree-based algorithm at this point removes A from the local tree and re-adjusts the utility upper bounds

of the path Y. To update the utility upper-bound of node Y, the algorithm subtracts the u(A) from the utility of path Y. This happens to be correct when the $u = Sum()$ since the $Sum(Y/A, T) = Sum(Y, T) - Sum(A, T)$. However, it may not necessarily hold for other SM utility function. If we use the same procedure and update $u(Y/A, T) = u(Y, T) - u(A, T)$, we may incorrectly prune itemsets with high-utility; since the subadditivity property only guarantees that $u(Y/A, T) > u(Y, T) - u(A, T)$ which clearly shows that the updated value, which appears on the right-hand side, could be lesser than the actual value on the left.

We propose that the tree-based algorithms for HUIM can be adapted with minimal change for an arbitrary subadditive monotone function. There is no change in the tree data structure, and the only change is that removing unpromising items during local tree creation must be disabled during the candidate generation phase. The tree construction and verification phase remain the same. This leads to direct use of utility values associated with the tree nodes in the mining process. However, the absence of removing unpromising items during local tree creation can result in the generation of a large set of potential candidates.

6.4.3 Projection-based algorithm

Projection-based algorithms like EFIM [85] merge two identical transactions to reduce the cost for database scan during the construction of the projected database for an itemset. Let $T = \{(A_1 : q_1), \ldots (A_n : q_n)\}$ be a transaction with positive quantity associated with every item in T. Let $S = \{(A_1 :$

$r_1), \dots (A_n : r_n)\}$ be another transaction with positive quantity associated with every item in S. Let $M = \{(A_1 : q_1 + r_1), \dots (A_n : q_n + r_n)\}$ be another transaction with positive quantity associated with every item in M. Merging transactions does not change the utility of the itemset $\langle A_1, A_2, ...A_n \rangle$ in the database for the Sum function defined by HUIM. However, utility of an itemset in the merged transaction can be less than the sum of its utility in individual transactions for an arbitrary subadditive function. Consider the transactions T_7 and T_8 in Figure 6.1. The utility ($ucov$) of the itemset $\{FG\}$ in T_7 and T_8 is 7 and 15 respectively. Consider merging transactions T_7 and T_8 to a single transaction $M = \{(F : 5), \dots (G : 5)\}$. The utility of itemset $\{FG\}$ in this merged transaction i.e. u($\{FG\}$,M) is 20. Therefore, transaction merging can change the utility of an itemset in the database. We disable transaction merging when adapting projection-based algorithms like EFIM for an arbitrary subadditive monotone function.

6.5 Case Study of HUIM-SM on a Twitter dataset

We conduct an experimental study on a publicly available Twitter dataset to emphasize that utility functions like $ucov$ can be designed that can combine information from a transaction database and domain knowledge from an external data source in the form of a graph. We extract the top-100 patterns extracted by $ucov$ and other mining methodologies like frequent itemset mining and high-utility itemset mining.

Table 6.9: Characteristics of Twitter transaction dataset

Dataset	#Tx	Avg. length	#Items	Max. length
Twitter	2,631	87	14,766	381

The Twitter-Dynamic-Net dataset[3] was constructed by selecting "Lady Gaga" on Twitter, and randomly selecting 10,000 of her followers. The selected followers were taken as seed users, and all their followers were collected by a crawler. Every tweet has information like user name, tweet id, timestamp, and retweet by user name, etc. fields associated with it. We construct a transaction database and a directed follower-followee graph from the dataset. Every transaction captures the users who were active during the period associated with the transaction. The quantity associated with a user is the number of tweets posted by that user in the time interval associated with the transaction. The follower-followee graph has users as vertices, and a directed edge from a user 'A' to user 'B' if 'B' has retweeted at least one tweet posted by user 'A' in the complete dataset. We fix the time duration to three hours and construct a transaction database whose statistics are given in Table 6.9. The followee-follower graph constructed from the dataset contains 14,766 followees and 20,423 followers with 46,164 edges between followees and followers. The average degree of a followee is 3, and the maximum degree is 48. We ensured that every user in the constructed graph has at least one follower by removing nodes that do not meet this criteria. Even though we discussed coverage on an undirected graph earlier, it is straightforward to adapt it for directed graphs and we do the same.

[3] The dataset is publicly available as a part of the AMiner repository.(https://aminer.cn/data-sna)

We compare the top-100 patterns extracted by $ucov$ with the existing baselines: frequent itemset mining (fim), and HUIM (sum). We also define two new baseline functions by integrating domain knowledge captured in the form a graph with frequent itemset mining, and HUIM. We call our-defined baseline as frequent itemset mining with coverage ($fcov$), and HUIM with coverage ($sumcov$) respectively.

Frequent itemset mining (fim) is applied to the transaction dataset only to extract top-k frequent itemsets. $fcov$ for an itemset returns the frequency of the itemset in the transaction database multiplied by its graph coverage $Co(\cdot)$. sum function captures the high-utility itemsets from the transaction database only. To integrate the coverage graph in the existing framework of high-utility itemset mining, we define the utility of an item in a transaction as the product of its quantity and coverage from the directed follower-followee graph. The utility of an itemset in a transaction is simply the sum of the utility of its items. We call this function as $sumcov$. It can be observed that $fim(\cdot, T)$, $fcov(\cdot, T)$, $sum(\cdot, T)$, and $sumcov(\cdot, T)$ for a transaction T are a subadditive monotone functions.

We compare the overlap among top-100 patterns generated by $ucov$ with fim, $fcov$, sum and $sumcov$ in Table 6.10. It can be observed that $ucov$ extracts different patterns compared to fim, and sum functions. There is a significant overlap in the top-100 patterns generated by $sumcov$, and $ucov$. We claim that the patterns generated by $ucov$ will always be a subset of the patterns generated by $sumcov$ for the same minimum utility threshold. We present a for-

Table 6.10: Overlap between the top 100 patterns generated by fim, fcov, sum, and sumcov with ucov.

Function	**Total number of patterns**	**Patterns common with** $ucov$
fim	100	21
$fcov$	100	37
sum	100	16
$sumcov$	100	71

Table 6.11: Statistics showing distribution of pattern length for top 100 patterns generated by fim, fcov, sum, sumcov, and ucov.

Statistics	fim	$fcov$	sum	$sumcov$	$ucov$
Minimum	1	1	1	1	1
Maximum	1	2	316	7	5
Mean	1	1.4	249	2.5	2.06
Median	1	1	315	2	2

mal proof for our claim in Theorem 6.5. We observed empirically that $sumcov$ generates a large number of patterns compared by $ucov$ for a fixed utility threshold. For example, $sumcov$ generated 10,29,921 patterns while $ucov$ generated only 681 patterns when the minimum utility threshold was set to 10,000 for the Twitter dataset used in our experimental study.

We also analyze the pattern length and $Co(\cdot)$ for the top-100 patterns generated by the different utility functions. Table 6.11 shows the distribution of pattern length i.e. number of items contained in a pattern by different functions for the top-100 patterns. It can be observed that fim generates patterns containing one item only.

Table 6.12: Statistics showing distribution of Co(·) for top 100 patterns generated by fim, fcov, sum, sumcov, and ucov.

Statistics	fim	$fcov$	sum	$sumcov$	$ucov$
Minimum	1	12	7	16	16
Maximum	40	64	1798	105	79
Mean	14.97	30.62	1421	42.7	38.9
Median	15.5	29	1793	38	38

Table 6.13: Statistics showing distribution of f(·) for top 100 patterns generated by fim, fcov, sum, sumcov, and ucov.

Statistics	fim	$fcov$	sum	$sumcov$	$ucov$
Minimum	179	68	1	8	15
Maximum	482	462	462	462	462
Mean	231	181	40.9	99.6	115
Median	213	170	1	67.5	73

The sum function is applied on the transaction database only and it generates very long patterns with a length greater than 300. However, our preference is to find patterns with a reasonable length as it's infeasible to give a discount to many active users so that they influence their set of followers for applications like viral marketing. Table 6.12 shows the distribution of graph coverage $Co(\cdot)$ for different functions. Frequent itemset mining and high-utility itemset mining ignores the domain knowledge captured in the form of a graph completely and only considers the transaction database to generate patterns. It can observed that the patterns generated by $ucov$ have better $Co(\cdot)$ compared to the patterns generated by fim, and sum. Table 6.13 shows the distribution of the frequency i.e. the number of transactions for the top-100 patterns generated by different functions. The sum function generates patterns with very low frequency as longer patterns are usually present in few transactions in a transaction database. The frequent itemset mining functions generate patterns that have frequency higher than sum, $sumcov$ and $ucov$. We end this section with a proof that the ucov function selects only a small subset of patterns generated by the sumcov utility function. Itemset mining is known to have the problem of generating too-many-patterns, and ucov appears to be doing a better filtering of available information in that context.

Table 6.14: Comparison of ucov(X,T) with sumcov(X,T) (Theorem 6.5)

Quantity	**ucov(X,T)**	$\leq$	**sumcov(X,T)**
$m_1 = q_1$	$m_1 \times$Co($\{A_1 \cdots A_n\}$)	$\leq$	$m_1 \times$ (Co($\{A_1\}$) + Co($\{A_2\}$) +. . .+Co($\{A_n\}$))
$m_2 = q_2 - q_1$	$m_2 \times$Co($\{A_2 \cdots A_n\}$)	$\leq$	$m_2 \times$ (Co($\{A_2\}$) + Co($\{A_3\}$) +. . .+Co($\{A_n\}$))
.			
.			
.			
$m_n = q_n - q_{n-1}$	$m_n \times$Co($\{A_n\}$)	=	$m_n \times$Co($\{A_n\}$)

Theorem 6.5. *For a fixed minimum user-defined threshold θ, the set of high-utility patterns generated by $ucov(\cdot)$ will always be a subset of the patterns generated by $sumcov(\cdot)$.*

Proof. Let $T = \{(A_1 : q_1), \ldots (A_n : q_n)\}$ be a transaction with positive quantity associated with every item in T. Suppose that the items are ordered such that $q_1 \leq \cdots \leq q_n$. Let X be an itemset with n items from T: $X = \{A_1, A_2, \ldots, A_n\}$. Let's analyze the functions $ucov(X,T)$ and $sumcov(X,T)$. $ucov(X,T)$ = $(q_1)\times$Co(X) + $(q_2 - q_1)\times$Co($\{A_2 \cdots A_n\}$)+ . . . +$(q_n)\times$Co($\{A_n\}$). $sumcov(X,T)$ = $(q_1)\times$Co($\{A_1\}$) + $(q_2)\times$Co($\{A_2\}$) + $(q_3)\times$Co($\{A_3\}$) + . . . +$(q_n)\times$Co($\{A_n\}$). It can be quickly verified that the first term in $ucov(X,T)$ i.e.
$(q_1)\times$Co($\{A_1 \cdots A_n\}$) $\leq$ $(q_1)\times$ (Co($\{A_1\}$) + Co($\{A_2\}$)+...+Co($\{A_n\}$)).
Similarly, $(q_2 - q_1)\times$Co($\{A_2 \cdots A_n\}$) $\leq$ $(q_2 - q_1)\times$ (Co($\{A_2\}$) + Co($\{A_3\}$)+ . . .+Co($\{A_n\}$)). For the proof refer to Table 6.14 where we compare the expressions of $ucov(X,T)$ and $sumcov(X,T)$ term-by-term. The table shows that $sumcov(\cdot)$ can assign an equal or higher utility score compared to $ucov(\cdot)$. All patterns with $ucov(\cdot)$ no less than θ will have $sumcov(\cdot)$ no less than θ and this proves the theorem. In fact, $sumcov(\cdot)$ can overestimate the coverage of patterns if the set of vertices covered by individual items present in the pattern

have lots of common neighbors. □

For example, $ucov(ACD, T_1)$ in the database in Figure 6.1 is 37. $sumcov(\{ACD\}, T_1)$ can be computed as $5 \times Co(\{A\}) + 10 \times Co(\{C\}) + 2 \times Co(\{D\}) = 5 \times 4 + 10 \times 3 + 2 \times 4 = 58$.

6.6 Performance evaluation of HUIM-SM algorithms

In this section, we compare the performance of algorithms from the category of tree-based, list-based, and projection-based algorithms. We specifically ask which category algorithms performs best on sparse and dense datasets? Do we get the same performance trends as we get for HUIM?

Table 6.15: Characteristics of real datasets

Dataset	**#Tx**	**Avg. length**	**#Items (I)**	**Density score R (%) =(A/I)x100 [7]**	**Type**
Retail	88,162	10.3	16,470	0.062	Sparse
Kosarak	9,90,002	8.1	41,270	0.019	Sparse
Chainstore	11,12,949	7.2	46,086	0.015	Sparse
Chess	3,196	37	75	49.33	Dense
Mushroom	8,416	23	119	19.32	Dense
Accidents	3,40,183	33.8	468	7.22	Dense

Experimental Setup: We choose UP-Growth+ [71] from tree-based, SM-Miner, EFIM [85] and D2HUP [48] from projection-based algorithms. We obtained the Java source code of UP-Growth+, EFIM and D2HUP algorithm from the SPMF library [31] and adapt them to mine patterns for any arbitrarily subadditive monotone function as described in Section 6.4. We call the algorithms adapted for SM functions as UPG+SM, EFIMSM, and D2HUPSM respectively.

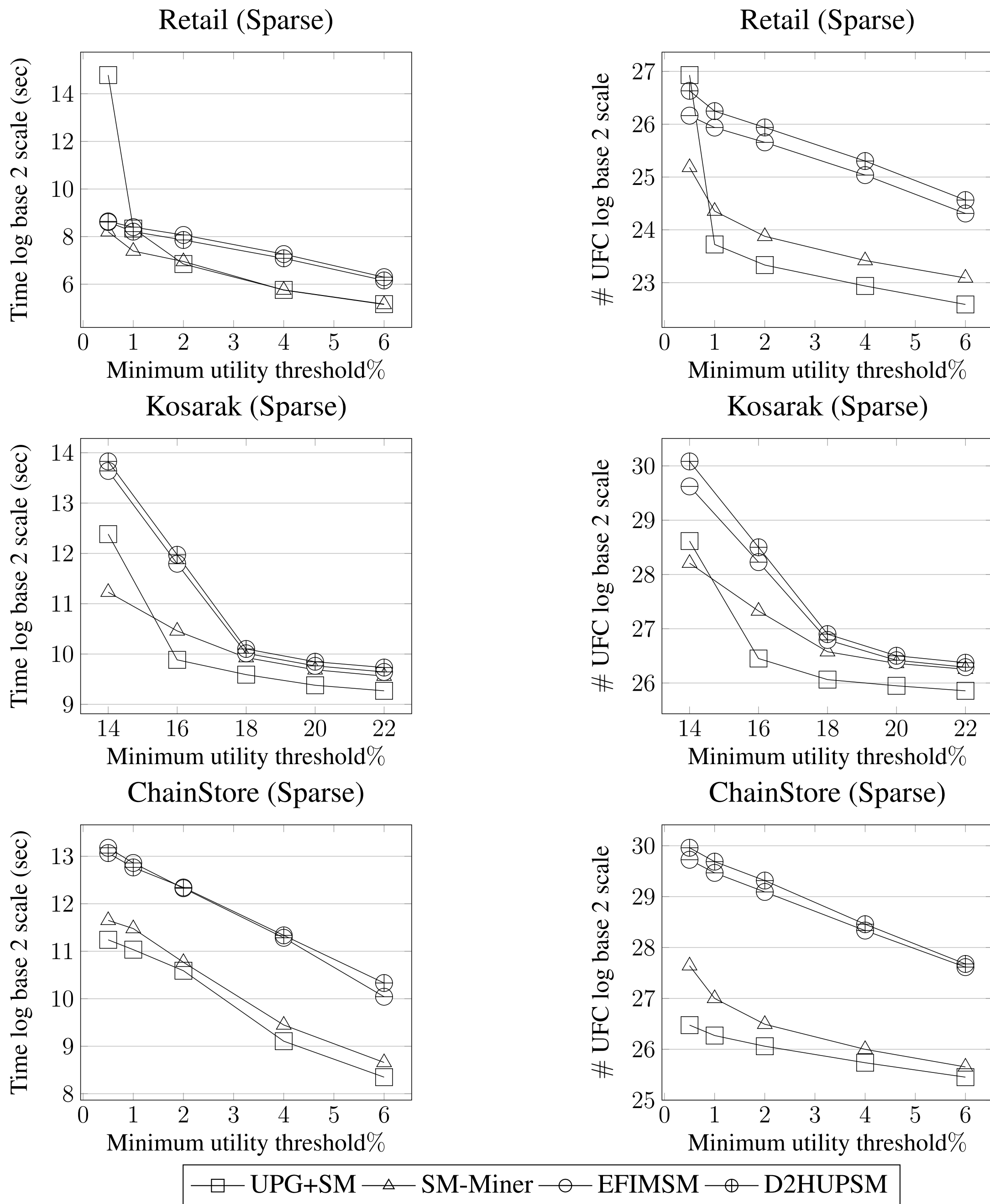

Figure 6.2: Performance evaluation on sparse datasets for $ucov$.

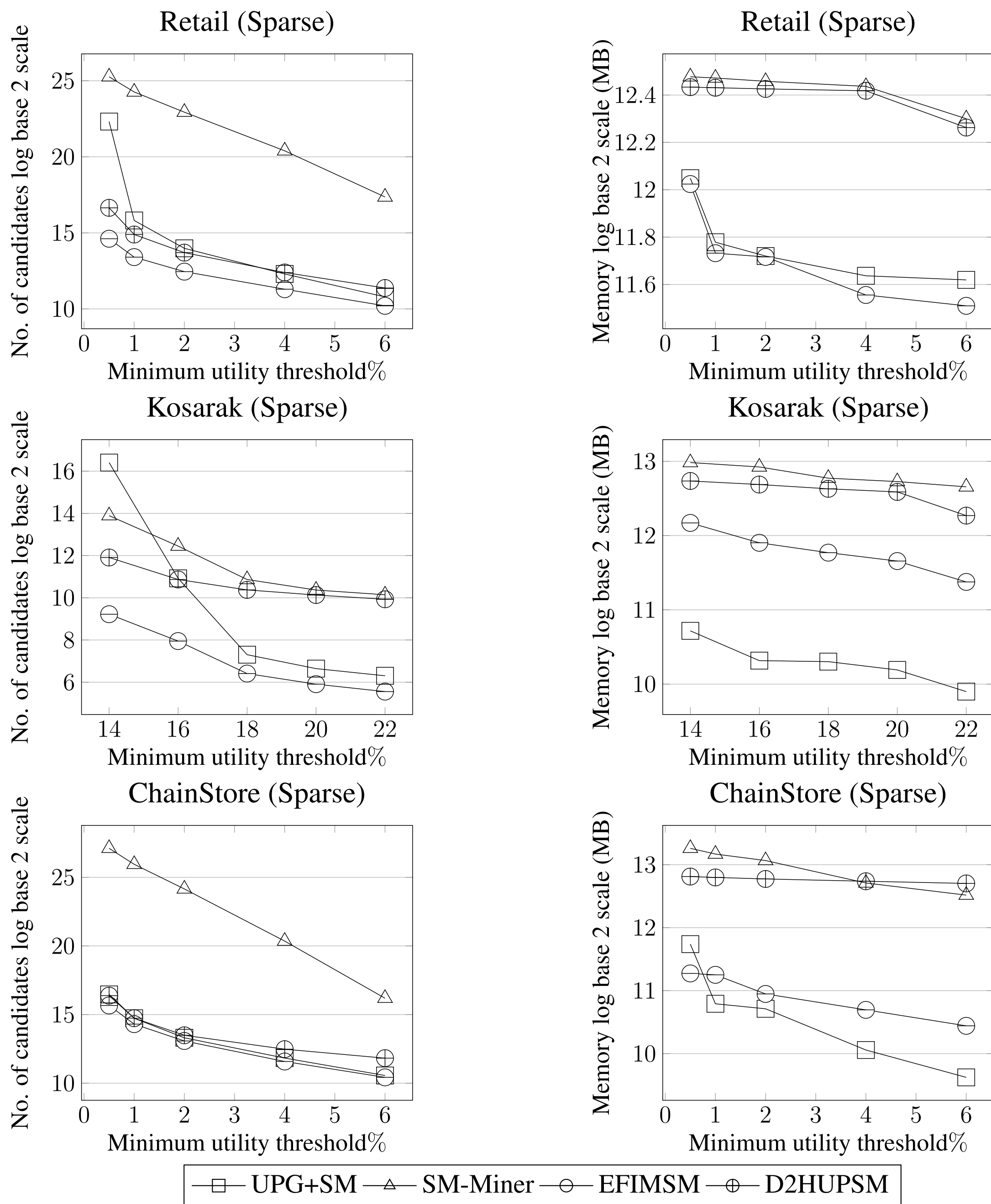

Figure 6.3: Number of candidates and memory consumption on sparse datasets for $ucov$.

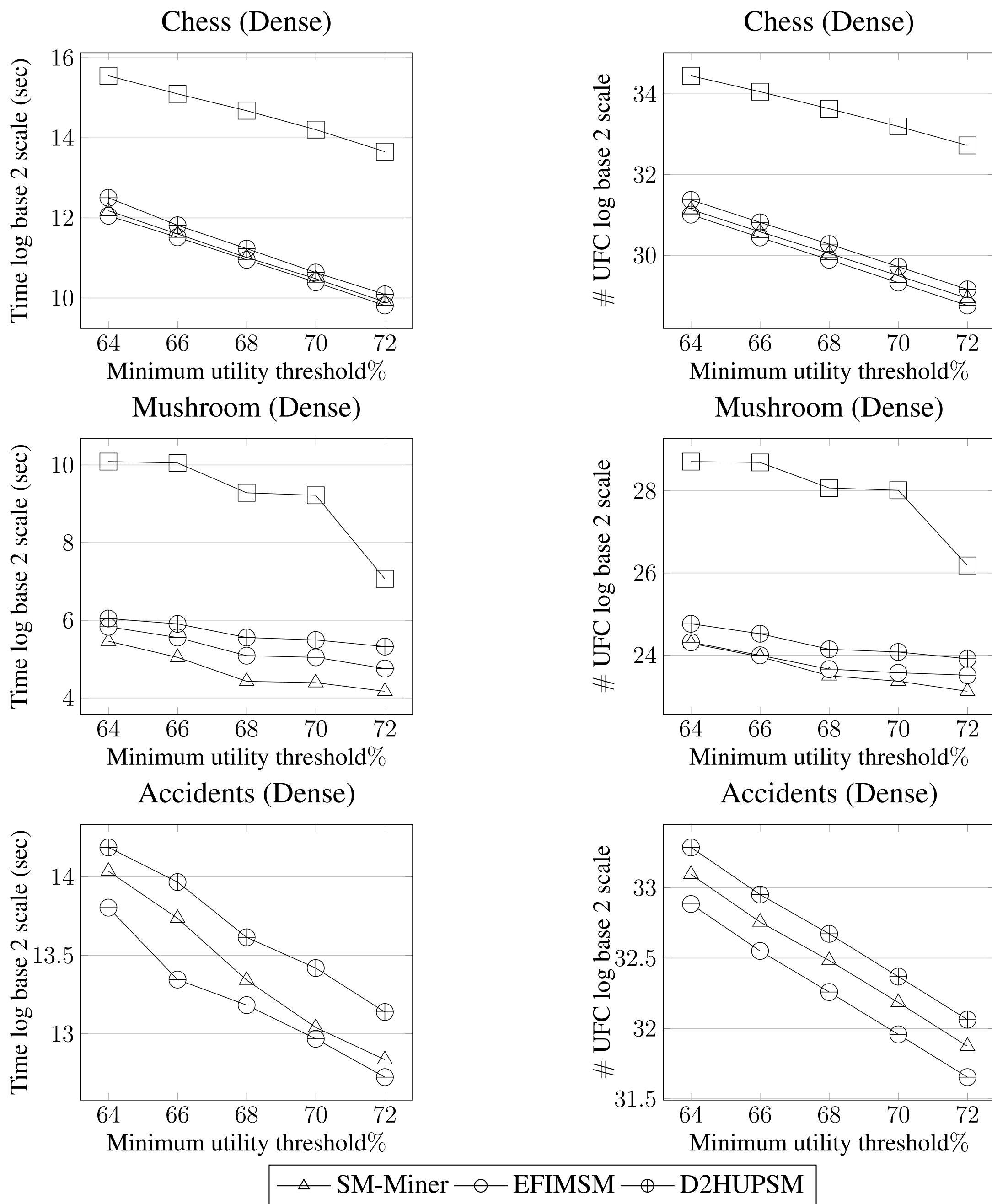

Figure 6.4: Performance evaluation on dense datasets for $ucov$.

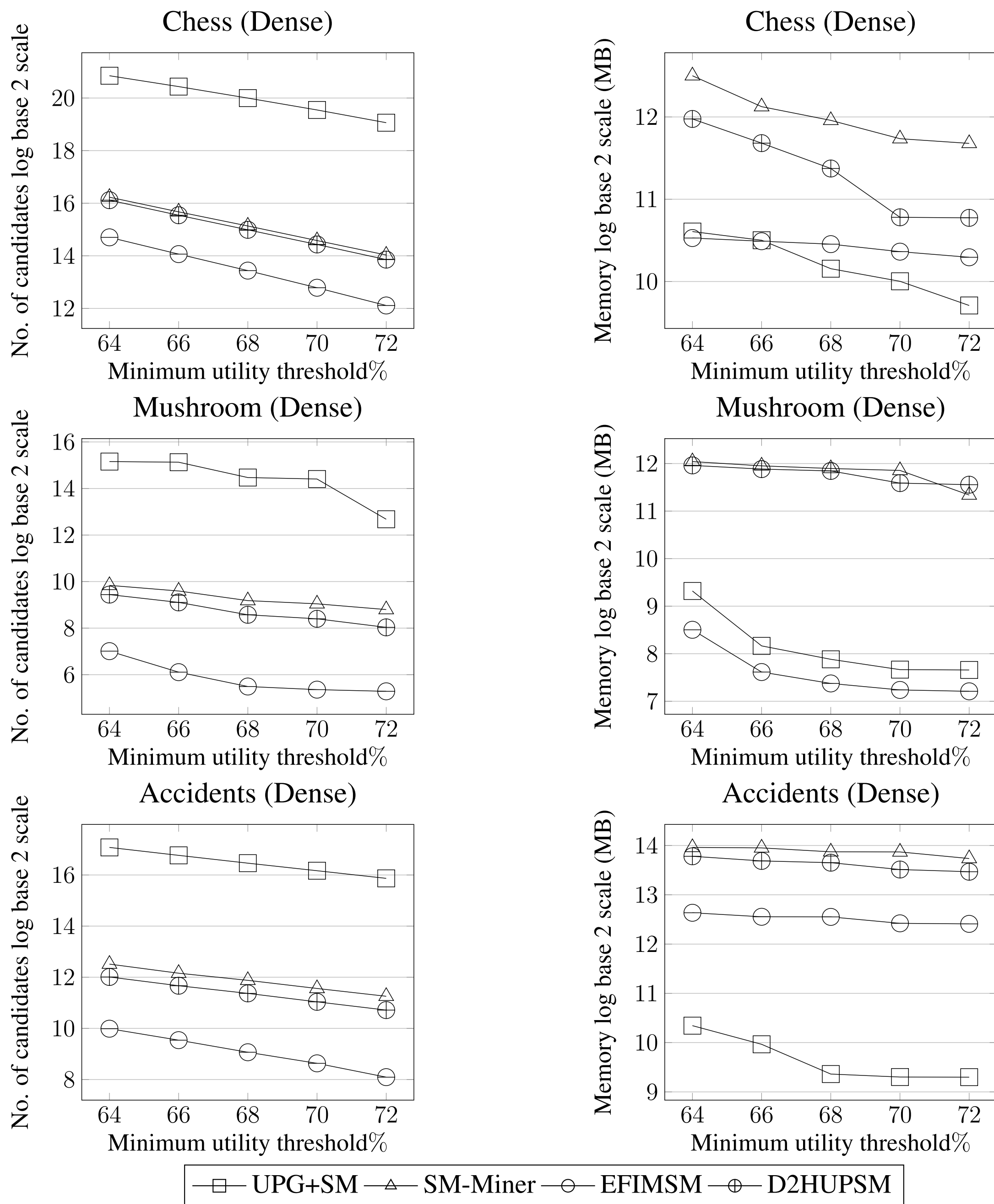

Figure 6.5: Number of candidates and memory consumption on dense datasets for *ucov*.

The experiments were performed on an Intel Xeon(R) CPU=26500@2.00 GHz with 64 GB RAM and Windows Server 2012 operating system. The datasets are obtained from the SPMF library [31] and vary in the number of transactions, the number of items, and the average transaction length as shown in Table 6.15. The coverage graph was constructed from the transaction database by taking the set of distinct items present in the database as vertices and linking them by edges. The average degree of a vertex in the coverage graph is four. The metrics for performance measure are total execution time, the number of explored candidates, the number of utility function calls (UFC) and main memory consumption.

Result-1: Which category algorithm performs better on Sparse datasets? The results for sparse datasets are shown in Figure 6.2 and 6.3. We observe that tree-based algorithm UPG+SM and list-based algorithm SM-Miner perform better than projection based algorithms EFIMSM and D2HUPSM on Sparse datasets. The reason for this behavior is less number of utility function calls by UPG+SM. The UPG+SM algorithm does not call the utility function during the candidate generation phase. The function calls are made only during the tree construction and verification phase. However, EFIMSM and D2HUPSM identify promising items during every recursive call, unlike SM-Miner and UPG+SM. Hence, the number of function calls by EFIMSM, D2HUPSM is more compared to SM-Miner and UPG+SM. There is an outlier at 14 % threshold on Kosarak dataset which is due to a large number of candidates generated by UPG+SM and the overhead for candidate verification dominates the execution time. D2HUP

is the state-of-the-art algorithm on sparse datasets for high-utility itemset mining. However, we observe that the tree-based and list-based algorithms perform better compared to projection-based algorithms on sparse datasets. We observe that the total execution time of the algorithms is more correlated with the number of utility function calls compared to the number of candidates generated during the mining process.

Result-2: Which category algorithm performs better on Dense datasets? The results for dense datasets is shown in Figure 6.4 and 6.5. SM-Miner performs the best on the Mushroom dataset and EFIMSM performs the best on Accidents dataset. UPG+SM performs the worst for Mushroom and Accidents dense datasets as it generates lots of candidates, and the verification phase dominates the runtime performance for tree-based algorithms. The execution for UPG+SM did not complete on Accidents dataset even after 24 hours and was terminated. EFIMSM performs the best for Accidents dataset as it generates the least number of candidates as well as the number of calls to the utility computation function. SM-Miner performs the best for Mushroom dataset followed by EFIMSM. EFIM is known to be the state-of-the-art algorithm in high-utility itemset mining for dense datasets. We observe that the SM-Miner competes with EFIMSM for the best performance on dense datasets and the total execution time is more correlated with the number of utility function calls compared to the number of candidates.

6.7 Summary

In this chapter, we formalize the problem of high-utility itemset mining for the class of subadditive monotone utility functions. We focus on designing upper-bounds for high-utility itemset mining problem with subadditive monotone functions (HUIM-SM). We propose a new inverted-list data structure called SMI-List with a lightweight construction method and an algorithm called SM-Miner to find high-utility itemsets for arbitrary subadditive monotone functions. We also adapt the existing tree-based and projection-based high-utility itemset mining algorithms for SM functions. We compare the performance of several HUIM-SM algorithms on dense and sparse datasets. Our results demonstrate that the computation of utility can play an important role in the performance of algorithms for complex utility functions like $ucov$. Subadditive monotone utility functions like $ucov$ can be designed to find high-influence active groups that can be attractive for applications like viral marketing.

Chapter 7

Conclusion

Frequent itemset mining [4, 40, 80, 27] is a very important problem that has been studied extensively by the data mining community in the last three decades. Frequent itemset mining assumes the binary presence/absence of items within a transaction. The notion of high-utility itemset mining was coined to associate positive weights with each item in a transaction. It is a harder problem to solve compared to frequent itemset mining as the utility measure is neither monotonic nor anti-monotonic. Despite recent attention and significant advances, several challenges remain to be resolved with respect to efficiency and applicability of high-utility itemset mining. We were able to address some of these which are summarized below. We conclude with several interesting research directions that are left to be pursued.

In Chapter 3, we designed a data structure called UP-Hist tree and an algorithm called UP-Hist Growth that performed better than the state-of-the-art two-phase tree-based algorithms [6, 72, 71]. We observed that two-phase tree-

based algorithms take a lot of time in the verification phase for dense datasets, and take more computational resources like memory, execution time compared to the one-phase list-based algorithms. We further observed that the intersection operation to construct an inverted-list data structure like utility-list [49, 32] is a costly operation and can impact the performance of a high-utility itemset mining algorithm. List-based algorithms can also generate itemsets that are non-existent in the database. In Chapter 4, we proposed an approach to construct a high-utility itemset mining algorithm by starting the execution with a tree-based algorithm and switching to a list-based algorithm when switching criteria is met.

The advantage of using a hybrid approach is that it avoids constructing the inverted-lists of smaller itemsets that have large inverted-lists as the number of transactions containing an itemset decrease with an increase in itemset length. The tree data structure can also guide the search space exploration of a list-based algorithm, and non-existent itemsets generation can be avoided. We further embedded several optimizations to improve the performance of a hybrid algorithm and presented a case study that integrated UP-Growth+ [71] and UP-Hist Growth [21] with FHM [32]. Our experimental study validated that the hybrid algorithms have superior performance compared to the state-of-the-art tree-based and list-based algorithms.

However, even the state-of-the-art hybrid tree-based, list-based, and the hybrid algorithms presented as a case study did not terminate for more than 24 hours on very dense datasets like Connect. It motivated us to design a novel data structure called UT_Mem-tree and the first one-phase tree-based algorithm

called UT-Miner presented in Chapter 5. The information stored in an inverted-list data structure like utility-list [49, 32] is augmented on a tree structure and we propose a lightweight tree construction mechanism to improve the performance of UT-Miner on dense datasets. We observed in our experimental study that the lightweight tree construction mechanism made the UT-Miner algorithm about 1.4 to 2 times faster on dense datasets. We observed that the UT-Miner, and UP-Hist Growth algorithm ran out of memory during the global tree creation on the NyTimes [29] dataset. However, the UPG+-Hybrid algorithm performs better than the FHM and mHUIMiner algorithms on NyTimes as shown in Section 5.3. We conclude that it is better to use lightweight tree structures like UP-Tree compared to other tree structures for datasets like NyTimes that have longer transactions.

The above approaches led to significant improvement in the running time required for high-utility itemset mining. For example, the UT-Miner algorithm performed at least 20 and 30 times better than the state-of-the-art tree-based, list-based and hybrid algorithms on the Mushroom and Accidents datasets. Our next focus was on enhancing the applicability of HUIM.

A lot of effort has been made by the pattern mining community to design efficient algorithms where the utility of an itemset is defined as the sum of the utility of its items. We wanted to explore if the high-utility itemset mining framework can incorporate utility functions that are subadditive and monotone. The generalization of utility can lead to interesting applications that allow integration of domain knowledge with itemsets generated from a transaction database. In

Chapter 6, we defined the problem of high-utility itemset mining for any arbitrary subadditive monotone utility function. We presented a case study to highlight the application of designing a novel utility function that can capture groups of active and influential users from a real Twitter dataset for applications like viral marketing. We showed that the existing upper-bounds and high-utility itemset mining algorithms could be adapted to mine itemsets for any subadditive monotone utility function. We also pointed out some caveats that must be kept in mind by data mining researchers and practitioners while designing an algorithm for their designed utility function.

7.1 Future Research Directions

The high-utility itemset mining problem requires extensive efforts to design an efficient algorithm that performs well on sparse and dense datasets. There is a possibility of coming up with an algorithm that can utilize the hybrid framework similar to the one in Chapter 4 to combine algorithms like EFIM [85] and D2HUP [48]. The EFIM algorithm creates a compact projected database by implementing the transaction merging technique that makes it the state-of-the-art algorithm on dense datasets. The D2HUP algorithm uses it hyperlink data structure during projected database creation that improves its performance on sparse datasets. It will be interesting to design an algorithm that can take advantage of both algorithms. An extensive study can be performed to decide the switching criteria in the hybrid framework. There is a need for an adaptive

high-utility itemset mining algorithm that can dynamically tune its search-space exploration strategy according to the dataset characteristics observed during the mining process. Techniques to reduce the projected database size similar to the one proposed in Chapter 5 and storing information in a compressed form similar to a histogram proposed in Chapter 3 can be investigated to improve the efficiency and memory requirements of high-utility mining algorithms.

In Chapter 6, we integrated the constraint of subadditive monotone utility functions in the high-utility itemset mining framework. The objective of the constraint pattern mining [65, 35] field of research is to develop general systems and programming languages that provide a framework to identify the general classes of constraints that can be processed by pattern mining systems during the search. It can be an interesting research direction to integrate the constraint pattern mining techniques into the itemset mining framework. We designed a subadditive monotone function that can integrate knowledge from a Twitter network in the form of a follower-followee graph with a transaction database. We believe that designing novel applications [43, 53, 63] to mine interesting patterns will lead to a much wider acceptance and usability of pattern mining algorithms.

Another interesting research direction can be to estimate the number of potential high-utility itemsets for a given transaction dataset and a user-defined minimum utility threshold. We can investigate techniques like data clustering [28], and locality-sensitive hashing [83, 12] to identify the set of items that can generate high-utility itemsets. We believe that estimating the number of poten-

tial high-utility itemsets can help to improve the load balancing for high-utility itemset mining algorithms applied on big data platforms [47, 18, 67]. The algorithms designed on parallel architectures need to be evaluated on datasets big enough to validate their performance at the big data scale. An extensive study can be conducted to evaluate and improve the performance of dynamic load balancing strategies.

References

[1] Aggarwal, C. C., Bhuiyan, M. A., and Al Hasan, M. Frequent pattern mining algorithms: A survey. In *Frequent pattern mining*. (2014), pp. 19–64.

[2] Aggarwal, C. C., and Yu, P. S. Outlier detection for high dimensional data. In *Proceedings of the ACM International Conference on Management of Data* (2001), pp. 37–46.

[3] Agrawal, R., Gehrke, J., Gunopulos, D., and Raghavan, P. Automatic subspace clustering of high dimensional data for data mining applications. In *Proceedings of ACM SIGMOD International Conference on Management of Data* (1998), pp. 94–105.

[4] Agrawal, R., and Srikant, R. Fast algorithms for mining association rules. In *Proceedings of the 20th International Conference on Very Large Databases, VLDB* (1994), pp. 487–499.

[5] Agrawal, R., and Srikant, R. Mining sequential patterns. In *Proceedings of the Eleventh International Conference on Data Engineering* (1995), pp. 3–14.

[6] AHMED, C. F., TANBEER, S. K., JEONG, B., AND LEE, Y. Efficient tree structures for high utility pattern mining in incremental databases. *IEEE Transactions on Knowledge and Data Engineering* (2009), pp. 1708–1721.

[7] AHMED, C. F., TANBEER, S. K., JEONG, B.-S., AND LEE, Y.-K. HUC-Prune: An efficient candidate pruning technique to mine high utility patterns. *Applied Intelligence* (2011), pp. 181–198.

[8] ARYABARZAN, N., MINAEI-BIDGOLI, B., AND TESHNEHLAB, M. negFIN: An efficient algorithm for fast mining frequent itemsets. *Expert Systems with Applications* (2018), pp. 129 – 143.

[9] BANSAL, R., DAWAR, S., AND GOYAL, V. An efficient algorithm for mining high-utility itemsets with discount notion. In *International Conference on Big Data Analytics* (2015), pp. 84–98.

[10] BARBER, B., AND HAMILTON, H. J. Extracting share frequent itemsets with infrequent subsets. *Data Mining and Knowledge Discovery* (2003), pp. 153–185.

[11] BAVELAS, A. Communication patterns in task-oriented groups. *The Journal of the Acoustical Society of America* (1950), pp. 725–730.

[12] BERA, D., AND PRATAP, R. Frequent-itemset mining using locality-sensitive hashing. In *International Computing and Combinatorics Conference* (2016), pp. 143–155.

[13] CAI, C. H., FU, A. W. C., CHENG, C. H., AND KWONG, W. W. Mining association rules with weighted items. In *Proceedings of International Database Engineering and Applications Symposium* (1998), pp. 68–77.

[14] CERF, L., AND MEIRA, W. Complete discovery of high-quality patterns in large numerical tensors. In *2014 IEEE 30th International Conference on Data Engineering* (2014), pp. 448–459.

[15] CHANG, J. H., AND LEE, W. S. Finding recent frequent itemsets adaptively over online data streams. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2003), pp. 487–492.

[16] CHEN, C., WANG, W., AND WANG, X. Efficient maximum closeness centrality group identification. In *Australasian Database Conference* (2016), pp. 43–55.

[17] CHEN, M., KUZMIN, K., AND SZYMANSKI, B. K. Community detection via maximization of modularity and its variants. *IEEE Transactions on Computational Social Systems* (2014), pp. 46–65.

[18] CHEN, Y., AND AN, A. Approximate parallel high utility itemset mining. *Big Data Research* (2016), pp. 26 – 42.

[19] CHENG, H., YAN, X., HAN, J., AND HSU, C. Discriminative frequent pattern analysis for effective classification. In *IEEE 23rd International Conference on Data Engineering* (2007), pp. 716–725.

[20] Coussat, A., Nadisic, N., and Cerf, L. Mining high-utility patterns in uncertain tensors. *Procedia Computer Science* (2018), pp. 403 – 412.

[21] Dawar, S., and Goyal, V. 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* (2015), pp. 56–61.

[22] Dawar, S., Goyal, V., and Bera, D. A hybrid framework for mining high-utility itemsets in a sparse transaction database. *Applied Intelligence* (2017), pp. 809–827.

[23] De Raedt, L., Guns, T., and Nijssen, S. Constraint programming for itemset mining. In *Proceedings of the 14th ACM International Conference on Knowledge Discovery and Data Mining* (2008), pp. 204–212.

[24] Deng, Z., Wang, Z., and Jiang, J. A new algorithm for fast mining frequent itemsets using n-lists. *Science China Information Sciences* (2012), pp. 2008–2030.

[25] Deng, Z.-H. Diffnodesets: An efficient structure for fast mining frequent itemsets. *Applied Soft Computing* (2016), pp. 214 – 223.

[26] Deng, Z.-H., and Lv, S.-L. Fast mining frequent itemsets using nodesets. *Expert Systems with Applications* (2014), pp. 4505 – 4512.

[27] Deng, Z.-H., and Lv, S.-L. Prepost+: An efficient n-lists-based algorithm for mining frequent itemsets via children–parent equivalence pruning. *Expert Systems with Applications* (2015), pp. 5424–5432.

[28] Djenouri, Y., Chun-Wei Lin, J., NÃŸrvÃĕg, K., and Ramampiaro, H. Highly efficient pattern mining based on transaction decomposition. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)* (2019), pp. 1646–1649.

[29] Dua, D., and Graff, C. UCI machine learning repository, 2017.

[30] Elseidy, M., Abdelhamid, E., Skiadopoulos, S., and Kalnis, P. Grami: Frequent subgraph and pattern mining in a single large graph. *Proceedings of the VLDB Endowment* (2014), pp. 517–528.

[31] Fournier-Viger, P., Gomariz, A., Gueniche, T., Soltani, A., Wu, C.-W., and Tseng, V. S. SPMF: A java open-source pattern mining library. *The Journal of Machine Learning Research* (2014), pp. 3389–3393.

[32] Fournier-Viger, P., Wu, C.-W., Zida, S., and Tseng, V. S. FHM: Faster high-utility itemset mining using estimated utility co-occurrence pruning. In *Foundations of Intelligent Systems* (2014), pp. 83–92.

[33] García-Moratilla, S., Martínez-Muñoz, G., and Suárez, A. Evaluation of decision tree pruning with subadditive penalties. In *Intelligent Data Engineering and Automated Learning* (2006), pp. 995–1002.

[34] Grahne, G., and Zhu, J. Fast algorithms for frequent itemset mining using fp-trees. *Knowledge and Data Engineering, IEEE Transactions on* (2005), pp. 1347–1362.

[35] GROSSI, V., ROMEI, A., AND TURINI, F. Survey on using constraints in data mining. *Data mining and knowledge discovery* (2017), pp. 424–464.

[36] GUNS, T., DRIES, A., NIJSSEN, S., TACK, G., AND RAEDT, L. D. Miningzinc: A declarative framework for constraint-based mining. *Artificial Intelligence* (2017), pp. 6 – 29.

[37] GUNS, T., NIJSSEN, S., AND RAEDT, L. D. Itemset mining: A constraint programming perspective. *Artificial Intelligence* (2011), pp. 1951 – 1983.

[38] HAN, J., PEI, J., AND KAMBER, M. *Data mining: concepts and techniques*. 2011.

[39] HAN, J., PEI, J., MORTAZAVI-ASL, B., CHEN, Q., DAYAL, U., AND HSU, M.-C. Freespan: frequent pattern-projected sequential pattern mining. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining* (2000), pp. 355–359.

[40] HAN, J., PEI, J., AND YIN, Y. Mining frequent patterns without candidate generation. In *ACM SIGMOD Record* (2000), pp. 1–12.

[41] HENRIQUES, R., ANTUNES, C., AND MADEIRA, S. C. A structured view on pattern mining-based biclustering. *Pattern Recognition* (2015), pp. 3941 – 3958.

[42] JIAN PEI, JIAWEI HAN, AND LAKSHMANAN, L. V. S. Mining frequent itemsets with convertible constraints. In *Proceedings 17th International Conference on Data Engineering* (2001), pp. 433–442.

[43] Kiran, R. U., Zettsu, K., Toyoda, M., Fournier-Viger, P., Reddy, P. K., and Kitsuregawa, M. Discovering spatial high utility itemsets in spatiotemporal databases. In *Proceedings of the 31st International Conference on Scientific and Statistical Database Management* (2019), pp. 49–60.

[44] Krishnamoorthy, S. HMiner: Efficiently mining high utility itemsets. *Expert Systems with Applications* (2017), pp. 168 – 183.

[45] Li, H.-F., and Lee, S.-Y. Mining frequent itemsets over data streams using efficient window sliding techniques. *Expert Systems with Applications* (2009), pp. 1466–1477.

[46] Li, Z., Lu, S., Myagmar, S., and Zhou, Y. Cp-miner: A tool for finding copy-paste and related bugs in operating system code. In *OSDI* (2004), pp. 289–302.

[47] Lin, Y. C., Wu, C.-W., and Tseng, V. S. Mining high utility itemsets in big data. In *Advances in Knowledge Discovery and Data Mining*. 2015, pp. 649–661.

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

[49] Liu, M., and Qu, J. Mining high utility itemsets without candidate generation. In *Proceedings of the* 21st *ACM International Conference on Information and Knowledge Management* (2012), pp. 55–64.

[50] Liu, Y., Liao, W.-k., and Choudhary, A. A fast high utility itemsets mining algorithm. In *Proceedings of the 1st International Workshop on Utility-based Data Mining* (2005), pp. 90–99.

[51] Liu, Y., Liao, W.-k., and Choudhary, A. A two-phase algorithm for fast discovery of high utility itemsets. In *Advances in Knowledge Discovery and Data Mining* (2005), pp. 689–695.

[52] Liu, Y.-C., Cheng, C.-P., and Tseng, V. S. Mining differential top-k co-expression patterns from time course comparative gene expression datasets. *BMC Bioinformatics* (2013).

[53] Ma, F., Meng, C., Xiao, H., Li, Q., Gao, J., Su, L., and Zhang, A. Unsupervised discovery of drug side-effects from heterogeneous data sources. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), pp. 967–976.

[54] Manku, G. S., and Motwani, R. Approximate frequency counts over data streams. In *Proceedings of the 28th international conference on Very Large Data Bases* (2002), pp. 346–357.

[55] Mannila, H., Toivonen, H., and Verkamo, A. I. Discovery of frequent episodes in event sequences. *Data mining and knowledge discovery* (1997), pp. 259–289.

[56] Monreale, A., Pinelli, F., Trasarti, R., and Giannotti, F. WhereNext: A location predictor on trajectory pattern mining. In *Proceed-*

ings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (2009), pp. 637–646.

[57] NIJSSEN, S., AND KOK, J. N. A quickstart in frequent structure mining can make a difference. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining* (2004), pp. 647–652.

[58] PEI, J., AND HAN, J. Can we push more constraints into frequent pattern mining? In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2000), pp. 350–354.

[59] PEI, J., HAN, J., LU, H., NISHIO, S., TANG, S., AND YANG, D. H-Mine: Hyper-structure mining of frequent patterns in large databases. In *Proceedings IEEE International Conference on Data Mining* (2001), pp. 441–448.

[60] PEI, J., HAN, J., MORTAZAVI-ASL, B., PINTO, H., CHEN, Q., DAYAL, U., AND HSU, M.-C. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *ICCCN* (2001), pp. 215–224.

[61] PENG, A. Y., KOH, Y. S., AND RIDDLE, P. mHUIMiner: A fast high utility itemset mining algorithm for sparse datasets. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2017), pp. 196–207.

[62] PEROZZI, B., AND AKOGLU, L. Discovering communities and anomalies in attributed graphs: Interactive visual exploration and summarization. *ACM Transactions on Knowledge Discovery from Data* (2018), pp. 1–40.

[63] QIN, X., KAKAR, T., WUNNAVA, S., RUNDENSTEINER, E. A., AND CAO, L. MARAS: Signaling multi-drug adverse reactions. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2017), pp. 1615–1623.

[64] RAYMOND CHAN, QIANG YANG, AND YI-DONG SHEN. Mining high utility itemsets. In *Third IEEE International Conference on Data Mining* (2003), pp. 19–26.

[65] SILVA, A., AND ANTUNES, C. Constrained pattern mining in the new era. *Knowledge and Information Systems* (2016), pp. 489–516.

[66] SOULET, A., AND CRÉMILLEUX, B. An efficient framework for mining flexible constraints. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining* (2005), pp. 661–671.

[67] TAMRAKAR, A. High utility itemsets identification in big data (2017).

[68] TAN, P.-N., STEINBACH, M., AND KUMAR, V. *Introduction to Data Mining, (First Edition).* (2005).

[69] TAO, F., MURTAGH, F., AND FARID, M. Weighted association rule mining using weighted support and significance framework. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (2003), pp. 661–666.

[70] TATTI, N., AND CULE, B. Mining closed episodes with simultaneous events. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (2011), pp. 1172–1180.

[71] TSENG, V. S., SHIE, B., WU, C., AND YU, P. S. Efficient algorithms for mining high utility itemsets from transactional databases. *IEEE Transactions on Knowledge and Data Engineering* (2013), pp. 1772–1786.

[72] TSENG, V. S., WU, C.-W., SHIE, B.-E., AND YU, P. S. 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* (2010), pp. 253–262.

[73] VU, L., AND ALAGHBAND, G. A fast algorithm combining fp-tree and tid-list for frequent pattern mining. *Proceedings of Information and Knowledge Engineering* (2011), pp. 472–477.

[74] VU, L., AND ALAGHBAND, G. Mining frequent patterns based on data characteristics. In *Proceedings of the International Conference on Information and Knowledge Engineering (IKE)* (2012).

[75] WANG, W., YANG, J., AND YU, P. S. Efficient mining of weighted association rules (WAR). In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2000), pp. 270–274.

[76] YAN, Q., HUANG, H., GAO, Y., LU, W., AND HE, Q. Group-level influence maximization with budget constraint. In *International Conference on Database Systems for Advanced Applications* (2017), pp. 625–641.

[77] YAN, X., AND HAN, J. gspan: Graph-based substructure pattern mining. In *IEEE International Conference on Data Mining* (2002), pp. 721–724.

[78] YAO, H., HAMILTON, H. J., AND BUTZ, C. J. *A Foundational Approach to Mining Itemset Utilities from Databases*. pp. 482–486.

[79] YAO, H., HAMILTON, H. J., AND GENG, L. A unified framework for utility-based measures for mining itemsets. In *Proceedings of ACM SIGKDD 2nd Workshop on Utility-Based Data Mining* (2006), pp. 28–37.

[80] ZAKI, M. J. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering* (2000), pp. 372–390.

[81] ZAKI, M. J., AND AGGARWAL, C. C. Xrules: an effective structural classifier for xml data. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (2003), pp. 316–325.

[82] ZAKI, M. J., AND GOUDA, K. Fast vertical mining using diffsets. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining* (2003), pp. 326–335.

[83] ZHANG, C., TIAN, P., ZHANG, X., JIANG, Z. L., YAO, L., AND WANG, X. Fast eclat algorithms based on minwise hashing for large scale transactions. *IEEE Internet of Things Journal* (2019), pp. 3948–3961.

[84] ZHU, H., WANG, P., HE, X., LI, Y., WANG, W., AND SHI, B. Efficient episode mining with minimal and non-overlapping occurrences. In *IEEE 10th International Conference on Data Mining* (2010), pp. 1211–1216.

[85] ZIDA, S., FOURNIER-VIGER, P., LIN, J. C.-W., WU, C.-W., AND TSENG, V. S. EFIM: a fast and memory efficient algorithm for high-utility itemset mining. *Knowledge and Information Systems* (2017), pp. 595–625.