An Efficient Approach for Mining Weighted Frequent Patterns with Dynamic Weights

Umama Dewan[1], Chowdhury Farhan Ahmed[1], Carson K. Leung[2][0000−0002−7541−9127](✉), Redwan Ahmed Rizvee[1], Deyu Deng[2], and Joglas Souza[2]

[1] University of Dhaka, Dhaka, Bangladesh
farhan@du.ac.bd
[2] University of Manitoba, Winnipeg, MB, Canada
kleung@cs.umanitoba.ca

Abstract. Weighted frequent pattern (WFP) mining is considered to be more effective than traditional frequent pattern mining because of its consideration of different semantic significance (weights) of items. However, most existing WFP algorithms assume a static weight for each item, which may not be realistically hold in many real-life applications. In this paper, we consider the concept of a dynamic weight for each item and address the situations where the weights of an item can be changed dynamically. We propose a novel tree structure called compact pattern tree for dynamic weights (CPTDW) to mine frequent patterns from dynamic weighted item containing databases. The CPTDW-tree leads to the concept of dynamic tree restructuring to produce a frequency-descending tree structure at runtime. CPTDW also ensures that no non-candidate item can appear before candidate items in any branch of the tree, and thus speeds up the construction time for prefix tree and its conditional tree during the mining process. Furthermore, as it requires only one database scan, it can be applicable to interactive, incremental, and/or stream data mining. Evaluation results show that our proposed tree structure and the mining algorithm outperforms previous methods for dynamic weighted frequent pattern mining.

Keywords: Data mining · Knowledge discovery · Weighted frequent pattern mining · Dynamic weights.

1 Introduction

Discovery of meaningful and hidden knowledge from a large collection of data is the main goal of data mining [4, 7, 10, 13, 16, 19]. Frequent pattern mining [12,17,18] is an important data mining problem where the patterns that occur frequently in a database are mined. However, in real-life scenarios, the frequency of a pattern cannot be considered as a sufficient indicator to find the meaningful patterns in large transaction databases. It is because, through frequency, only the number of transactions in the database containing the pattern is reflected. In many cases, the items in a transaction can be considered to have different

degrees of importance (weight). For example, in retail applications, an expensive product generally contributes a large portion of overall revenue although it may not appear in many transactions. For this reason, *weighted frequent pattern* (*WFP*) *mining* [6, 21–23] was introduced to discover more useful knowledge by considering different weights for different items. Some real life examples where weight-based pattern mining can be applied are market data analysis where the price of products is an important factor, web traversal pattern mining where different web pages have different strength of impact.

Even though WFP mining considers diverse application specific weights for different items, still it cannot reflect real world environment where the significance (weight) of items vary with time. Most of the existing WFP algorithms consider static weight for an item. But in real life, the significance of an item can be affected by many factors. Consumer behaviors change with time which affect the significance of products in retail market. For example, the demand of jackets increase in winter, but their demand are quite likely to decrease in summer. Again, at one period of time, the demand of a particular design or material of jacket (e.g., jean jacket) can increase or decrease considering the current trend and other factors. It signifies that considering different weights for a particular item for different times is a requirement for several real-life applications.

Motivated by these real world scenarios, a new strategy for handling dynamic weights in WFP mining was introduced [2]. However, the algorithm uses a less compact tree structure (CanTree [14, 15]) and requires long mining time. The main focus of the current work is to solve these problems all together. Our *key contributions* of this paper include:

- A novel, highly compact tree structure—namely, *Compact Pattern Tree for Dynamic Weights* (*CPTDW*)—is proposed to mine frequent patterns with dynamic weights that significantly improves the performance with a single database scan.
- A phase-by-phase tree restructuring method—namely, path adjusting method—is proposed for dynamic weighted frequent pattern mining, which improves the degree of prefix sharing in the tree structure.
- A single-scan mining algorithm is developed based on the above tree structure, that can be applied for finding dynamic weighted frequent patterns over a data stream.
- Performance characteristics of a pattern-growth mining approach for dynamic weighted frequent pattern mining were observed through extensive experimental study.

The remainder of this paper is organized as follows. The next section discusses the required preliminary concepts with related works. Section 3 describes our proposed methodology with proper examples. Section 4 presents our evaluation results to show the supremacy of our proposed approach. Finally, conclusions are drawn in Section 5.

2 Preliminary Concepts and Related Works

2.1 Frequent Pattern Mining

The support/frequency of a pattern signifies the number of transactions that contain the pattern in the transaction database. Frequent pattern mining is used to find the complete set of patterns that satisfies a minimum support threshold in the transaction database. The *downward closure property* states that, if a pattern is infrequent, then all of its super patterns must be infrequent and can be pruned.

The Apriori algorithm [1] is the first solution for the frequent pattern mining problem. However, it needs several database scans and suffers from the level-wise-candidate-generation-and-test problem. *Frequent Pattern* (*FP*)*-Growth* [9] solves this problem by using an FP-tree based technique which requires only two database scans. There has been several research works which are being used to devise new algorithms or to improve the existing works for finding frequent patterns.

2.2 Weighted Frequent Pattern Mining

The weight of an item is a non-negative real number that is assigned to reflect the importance of that item in the transaction database. For a set of items $I = \{i_1, i_2, ..., i_n\}$, the weight of a pattern, P $\{x_1, x_2, ..., x_n\}$ is given as follows:

$$Weight(p) = \frac{\sum_{q=1}^{length(P)} Weight(x_q)}{length(P)} \tag{1}$$

For example, consider (i) an item "a" has weight 0.7 and frequency 2, and (ii) an item "b" has weight 0.3 and frequency 5. Then, according to Eq. (1), the weight of itemset "ab" will be $\frac{0.7+0.3}{2} = 0.5$. The weighted support of a pattern is the result of multiplying the pattern's support with the weight of that pattern:

$$WSupport(P) = Weight(P) \times Support(P) \tag{2}$$

A *weighted frequent pattern* is the pattern whose weighted support is at least the minimum threshold.

Example 1. If (i) an item "a" has weight 0.7 and frequency 2 and (ii) item "b" has weight 0.3 and frequency 5, then $WSupport(\text{"a"}) = 0.7 \times 2 = 1.4$ and $WSupport(\text{"b"}) = 0.3 \times 5 = 1.5$ according to Eq. (2). If the minimum support threshold is 1.2, then both "a" and "b" are weighted frequent patterns.

In real-life applications, normalized weight values are assigned to each item based on their price. Normalization process is required to adjust the differences among data from various sources so that a common basis of comparison is being created [22,23]. Based on the normalization process, a specific weight range can be determined so that the final item weights can be within that range.

Some weighted frequent pattern mining algorithms [6, 21] have been developed based on Apriori technique, which makes the use of candidate generation-and-test paradigm. These algorithms require multiple database scans, and result in poor mining performance. Moreover, WFP mining is more challenge because the weighted frequency of a pattern does not satisfy the *downward closure property.*

Example 2. Continue with Example 1. With Eqs. (1) and (2), $Weight$("ab") $= \frac{0.7+0.3}{2} = 0.5$ and $WSupport$("ab") $= 0.5 \times 3 = 1.5$, but $WSupport$("a") $= 1.4$ and $WSupport$("b") $= 0.9$. For the minimum support threshold of 1.2, pattern "b" is infrequent but item "ab" is frequent, which means *downward closure property* is not satisfied.

WFIM [23] and its extension WIP [22] maintain the property by multiplying each item's frequency by the overall maximum weight.

Example 3. Continue with Example 2. Item "a" has the maximum weight of 0.7. By multiplying it with the support count of item "b", 2.1 is obtained. As a result, "b" will not be pruned at an early stage and pattern "ab" will not be missed. However, at the final stage, the overestimated pattern "b" will be pruned by using its actual weighted support.

2.3 Dynamic Weighted Frequent Pattern Mining

In dynamic weighted frequent pattern mining, weight of each item changes dynamically in each batch based on the importance of that particular item. The dynamic weighted support of a pattern is the result of adding the weighted supports of that pattern in each batch. A dynamic weighted frequent pattern is the pattern whose dynamic weighted support is greater than or equal to the minimum threshold. Dynamic weighted support of a pattern P is:

$$DWSupport(P) = \sum_{j=1}^{N} Weight_j(P) \times Support_j(P) \tag{3}$$

where N is the number of batches. Consider an item "a" has weight 0.7 and frequency 2 in first batch and weight 0.3 and frequency 5 in the second batch. Then, according to Eq. (2), the weighted support of pattern "a" in the first and second batches are $0.7 \times 2 = 1.4$ and $0.3 \times 3 = 0.9$, respectively. So, the total $DWSupport$("a") $= 1.4 + 0.9 = 2.3$ according to equation (3). If the minimum support threshold is 1.2, then "a" is a dynamic weighted frequent pattern.

Zhang et al. [24] proposed a strategy to find association rules in dynamic databases by weighting. However, they considered one weight for a database containing a group of transactions. By doing so, the recently added groups of transactions are highlighted over the previously added groups. However, this assumption is not realistic because the importance of an item or itemset can vary with time.

Ahmed et al. [2] proposed a *dynamic weighted frequent pattern mining* (*DWFPM*) *algorithm* to dynamically handle the changing item weights. It exploits pattern growth mining technique that removes the level-wise candidate generation-and-test methodology of the dynamic weight algorithm [24]. Furthermore, it requires only one database scan which makes it eligible for using in incremental, inter-active and stream data mining. However, the CanTree structure [14, 15] used in this algorithm results in a less compact tree structure and incurs very high mining time due to the canonical order of its tree structure.

3 Our Proposed Approach

3.1 Tree Construction

To capture transactions having items with dynamic weights, we construct a *compact pattern tree for dynamic weights* (*CPTDW*). A header table is maintained with the tree structure. The first value of the header table is the item ID. The second value of the header table contains each item's weight value in a batch-by-batch fashion, and the third value of header table contains the I-list of items which contains the current frequency value of each item in a batch-by-batch manner. Our CPTDW builds an FP-tree [9] like compact frequency-descending tree structure with a single-scan of transaction database. At first, transactions of the first batch are inserted into the CPTDW tree one by one according to a predefined item order (e.g., lexicographic order). After inserting a batch of transactions, the CPTDW tree structure is dynamically restructured by the current frequency descending item order and I-list is updated accordingly using the path adjusting method [3, 11]. In summary, CPTDW tree can be constructed in two phases:

1. **Insertion phase:** Transaction(s) of a batch is scanned, according to the current item order of I-list, transactions are inserted into the tree and the frequency count of the respective items is updated in the I-list.
2. **Restructuring phase:** The I-list is rearranged according to the frequency descending order of the items and the tree nodes are restructured according to the newly arranged I-list.

The construction of CPTDW starts with the insertion phase. The first insertion phase begins by inserting the first transaction of the first batch in a lexicographic item order into the tree. The tree will be restructured after the insertion of all the transactions of the first batch. The tree is restructured by using the *path adjusting method* [11]. The paths in a prefix-tree are adjusted through recursive swapping of the adjacent node in the path until the path completely achieves the new sorted order. Thus, bubble sort technique is used to process the swapping between two nodes. One of the basic properties of FP-tree is that the frequency count of a node cannot be greater than the frequency count of its parent node. To maintain this property, the *path adjusting method* inserts a new node of the same name as a sibling of the parent node in the tree when

Algorithm 1 Path adjusting method algorithm.

Input: Let X, Y and Z be three nodes in a path in a prefix tree where X is the parent of Y, Y is the parent of Z and Y and Z nodes are needed to be exchanged for path adjustment. Consider *nodeName.name*, *nodeName.count* and *nodeName.child* refer to the name, the support count (in the referred path) and a child of a node. Therefore, the path adjusting is performed according to the following algorithm:
function EXCHANGE
    Exchange parent and children links of Y and Z
function INSERTION
    Insert Y' to X as a new child node such that $Y'.name = Y.name$
    Set $Y'.count = Y.count - Z.count$
    Assign all children of Y except Z to Y'
    Set $Y.count = Z.count$
function MERGE_NODE(P,Q)
    Set $Q.count = Q.count + P.count$
    for each child node of Q **do**
        for each child node of P **do**
            if $Q.child == P.child$ **then**
                Call MERGE_NODE($Q.child, P.child$)
            else if $Q.child$ not equals $P.child$ **then**
                Add $P.child$ and its sub tree to $Q.child$ list
function MERGE
    if C is another child node of X and $C.name = Z.name$ **then**
        Call MERGE_NODE(C,Z)
        Delete C and its sub tree
function PATH_ADJUST
    if Y and Z need to be swapped and $Y.count > Z.count$ **then**
        Call INSERTION( )
    Call EXCHANGE( )
    Call MERGE( )
Repeat to **Call** PATH_ADJUST with next two nodes of Y and Z in another path to be exchanged and **Terminate** when no further node exchange is required.

the parent node needs to be exchanged with any child node which has a smaller count value. Otherwise, if the frequency counts of both the nodes are equal, then a simple exchange operation between the two nodes is sufficient. However, after swapping, if two sibling nodes contain the same item due to the exchange operation, then they should be merged. This insertion and restructuring phases are executed alternatively until all the transactions of all batches are inserted into the tree and restructured in the frequency descending order according to a batch-by-batch fashion. The pseudo-code for path adjusting method is shown in Algorithm 1.

Consider the example database of Table 1. At first, the items of the first batch are inserted according to the lexicographic order as shown in Fig. 2(a). Then, they are sorted in frequency descending order based on path adjusting method [3,11]. Figure 2(b) shows the tree structure after restructuring the transactions of

Table 1: An example of transaction database with dynamic weights

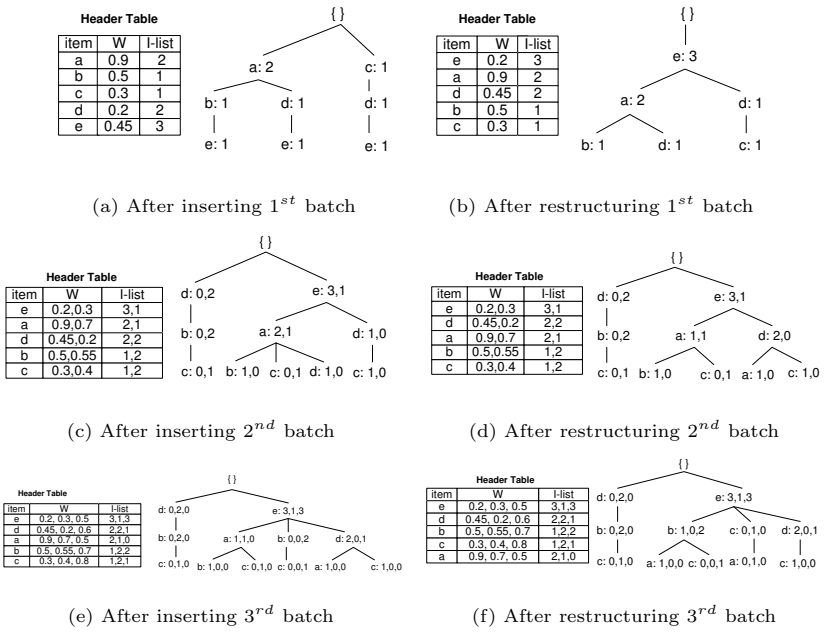
Batch	TID	Transactions	Weights
1^{st}	T1	a, d, e	a: 0.9 b: 0.5 c: 0.3 d: 0.45 e: 0.2
	T2	c, d, e	
	T3	b, a, e	
2^{nd}	T4	b, d	a: 0.7 b: 0.55 c: 0.4 d: 0.2 e: 0.3
	T5	a, c, e	
	T6	b, c, d	
3^{rd}	T7	d, e	a: 0.5 b: 0.7 c: 0.8 d: 0.6 e: 0.5
	T8	b, e	
	T9	c, b, e	

the first batch. Figure 2(c) shows the tree after inserting the transactions of the second batch based on the item order which is achieved after restructuring the transactions of the first batch. Then, the items are sorted in frequency descending order based on their current frequency which includes their frequency count of both the batches. Figure 2(d) shows the tree structure after restructuring. As frequency information for each batch is kept separately in each node of the tree, it can be easily discovered which transactions have occurred in which batch. At the same way, Fig. 2(e) shows the tree structure after inserting the transactions of the third batch. Figure 2(f) is our final CPTDW tree structure which is achieved by inserting and restructuring all the transactions of all batches. Our proposed CPTDW tree structure has the following properties:

- **Property 1:** The items in the tree are sorted according to the frequency descending order.
- **Property 2:** The total frequency count of any node in the tree is greater than or equal to the sum of total frequency counts of its children.
- **Property 3:** The tree structure can be constructed in a single database scan.

3.2 Mining Process

According to the FP-growth mining algorithm [9], while creating a prefix-tree for a particular item, all branches prefixing that particular item are taken with their frequency value. After that, the conditional tree is built from the prefix tree by deleting the nodes containing infrequent items. CPTDW algorithm performs the same type of mining. The mining operation of CPTDW is done in a top-down approach [20]. As discussed in Section 2.2, the main challenge in weighted frequent pattern mining is that, the weighted frequent pattern of an item does not hold the *downward closure property*. So, to maintain this property, global maximum weight GMAXW has to be used. GMAXW is the maximum weight of all the items in the global database. In our case, this is the highest weight among every weight in all of the batches. For example, in Table 1, item “a”



(a) After inserting 1^{st} batch (b) After restructuring 1^{st} batch

(c) After inserting 2^{nd} batch (d) After restructuring 2^{nd} batch

(e) After inserting 3^{rd} batch (f) After restructuring 3^{rd} batch

Fig. 2: CPTDW Tree construction

has the GMAXW of 0.9. The local maximum weight LMAXW is needed while doing the mining operation for a particular item, and it is not always equal to GMAXW.

As our CPTDW tree is sorted according to the frequency descending order, LMAXW could be anywhere for a particular item. We start our pattern growth mining operation from the top-most item of the CPTDW tree structure. So, for this case, LMAXW is the weight of the first item for sure. After that, for the second item, we compare its weight with the previous LMAXW and consider the larger one as the current LMAXW. By moving in this way, LMAXW calculation for each time can be saved.

We consider the database presented in Table 1, the tree constructed for that particular database in Fig. 2(f) and minimum support threshold 1.2. Here, GMAXW is 0.9. After multiplying GMAXW with the total frequency of each item, we get a: $3 \times 0.9 = 2.7$, b: $5 \times 0.9 = 4.5$, c: $4 \times 0.9 = 3.6$, d: $5 \times 0.9 = 4.5$, and e: $7 \times 0.9 = 6.3$. So, all items are single element candidates. We start the mining process with the top-most item of the CPTDW tree, "e". For "e", LMAXW is 0.5, frequency of "e" is $3+1+3 = 7$. By multiplying the frequency of "e" with LMAXW, we get $0.5 \times 7 = 3.5$, which is greater than minimum support threshold 1.2. So, single element pattern "e" is generated.

After that, we consider item "d" as it is the second top most item in Fig. 2(f). So, the prefix tree of "d" is created by taking all the branches prefixing item "d"

Algorithm 2 Mining and Test_Candidate Procedure.

```
1: procedure PROCEDURE_MINING(T, H, α, LMAXW)
2:     for each item β of H do
3:         if (frequency(β) × LMAXW <δ) then
4:             Delete β from H and T
5:     Let CT be the Conditional tree of α
6:     Let HC be the Header table of Conditional tree CT
7:     for each item β in HC do
8:         Call TEST_CANDIDATE(αβ, frequency(αβ),δ)
9:         Create Prefix tree PT_αβ with its Header table HP_αβ for pattern αβ
10:        Call Mining(PT_αβ, HP_αβ, αβ, LMAXW)
11: procedure TEST_CANDIDATE(X,B,δ)
12:    Let Dynamic weighted support of X be DW_X
13:    Let frequency(X_k) denotes frequency of pattern X in kth batch
14:    Let weight(X_k) denotes weighted average of pattern X in kth batch
15:    Set DW_X = 0
16:    for each Batch B_i in B do
17:        DW_X = DW_X + (frequency(X_{B_i}) × weight(X_{B_i})
18:    if DW_X ≥ δ then
19:        Add X in the Dynamic weighted frequent pattern list
```

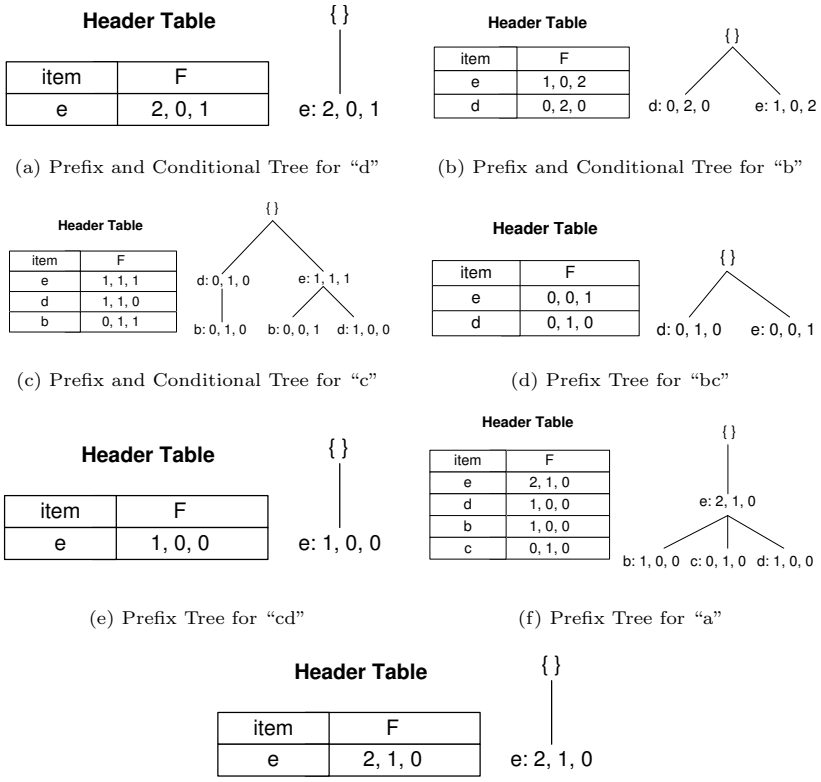
as shown in Fig. 4(a). For creating conditional tree, the nodes that cannot be candidate patterns must be deleted from the prefix tree. For item "d", LMAXW is 0.6. After multiplying the frequency of the item in the header table shown in Fig. 4(a), we get e: $3 \times 0.6 = 1.8$. As the value is greater than the minimum support threshold value, that is 1.2, so no node should be deleted from the prefix tree which signifies that, the prefix and conditional tree for item "d" is same. So, the candidate patterns "de" and "d" are generated at this point.

The same procedure is conducted for all the items in the I-list of Fig. 2(f) for finding out all the candidate patterns of our example database. The pseudo-code of the mining procedure is illustrated in Algorithm 2 and the operations are shown in Fig. 4. After generating all the candidate patterns, we calculate the actual DWSupport of each candidate pattern according to Eq. (2) to check whether they are actually frequent or not. This calculation is shown in Table 2.

4 Performance Evaluation

In this section, we present the overall performance of our proposed algorithm CPTDW over several datasets. The performance of our proposed algorithm CPTDW in compared with the existing DWFPM algorithm [2].

Experimental environment and datasets. To evaluate the performance of our proposed tree structure and algorithm for dynamic weighted frequent pattern mining, we have performed several experiments on IBM synthetic dataset (e.g., T10I4D100K) using synthetic weights and real life datasets (e.g., retail, chess, mushroom, pumsb*, connect, pumsb) using synthetic weights and real-life



(a) Prefix and Conditional Tree for "d"

(b) Prefix and Conditional Tree for "b"

(c) Prefix and Conditional Tree for "c"

(d) Prefix Tree for "bc"

(e) Prefix Tree for "cd"

(f) Prefix Tree for "a"

(g) Conditional Tree for "a"

Fig. 4: Mining Operation

dataset (e.g., chain-store) with real weight values. All the datasets are collected from frequent itemset mining dataset respiratory [5]. The performance of the proposed algorithm is compared with the existing algorithm DWFPM [2], with respect to runtime (aka execution time) and memory usage. Our programs are written in Java programming language. Programs were run in a time sharing environment with the Linux 16.04 operating system on a HP Notebook, 64 bits machine, Intel(R) Core(TM) i3-6100U CPU, 2.30GHz processor, 4GB RAM, 100MHz clock, and 500 GB of main memory. We have divided all the datasets in such a way that each batch contains at most 10 transactions. The minimum support threshold values of 2%, 3%, 10% and 15% are used to conduct the experiments.

Table 3 shows some important characteristics of synthetic and real-life datasets. Dense and sparse natures of datasets are very useful properties. A dense dataset contains many items per transaction and small number of distinct items. For the *chess* dataset in Table 3, it has a total of 75 distinct items, an average

Table 2: DWSupport calculation of the candidate patterns of CPTDW algorithm

No	Candidate patterns	DW support calculation	Results
1	e: 3,1,3	$(0.2 \times 3) + (0.3 \times 1) + (0.5 \times 3) = 2.4$	Passed
2	de: 2,0,1	$(\frac{0.45+0.2}{2} \times 2) + (\frac{0.6+0.5}{2} \times 1) = 1.2$	Passed
3	d: 2,2,1	$(0.45 \times 2) + (0.2 \times 2) + (0.6 \times 1) = 1.9$	Passed
4	be: 1,0,2	$(\frac{0.5+0.2}{2} \times 1) + (\frac{0.7+0.5}{2} \times 2) = 0.95$	Pruned
5	bd: 0,2,0	$\frac{0.55+0.2}{2} \times 2 = 0.75$	Pruned
6	b: 1,2,2	$(1 \times 0.5) + (0.55 \times 2) + (0.7 \times 2) = 3$	Passed
7	ce: 1,1,1	$(\frac{0.3+0.2}{2} \times 1) + (\frac{0.4+0.3}{2} \times 1) + (\frac{0.8+0.5}{2} \times 1) = 1.25$	Passed
8	cd: 1,1,0	$(\frac{0.3+0.45}{2} \times 1) + (\frac{0.4+0.2}{2} \times 1) = 0.675$	Pruned
9	bc: 0,1,1	$(\frac{0.55+0.4}{2} \times 1) + (\frac{0.7+0.8}{2} \times 1) = 1.225$	Passed
10	c: 1,2,1	$(0.3 \times 1) + (0.4 \times 2) + (0.8 \times 1) = 1.9$	Passed
11	ae: 2,1,0	$(\frac{0.9+0.2}{2} \times 2) + (\frac{0.7+0.3}{2} \times 1) = 1.6$	Passed
12	a: 2,1,0	$(0.9 \times 2) + (0.7 \times 1) = 2.5$	Passed

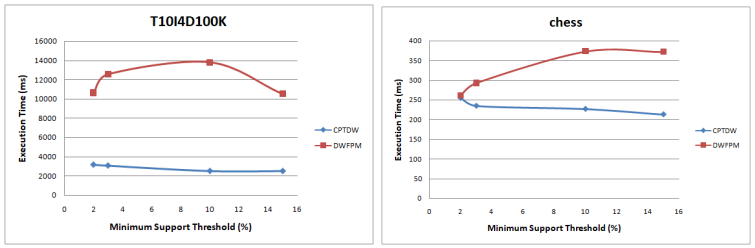
Table 3: Dataset characteristics

Datasets	#trans.	Avg. trans. len. (A)	#distinct items (D)	Dense & sparse characteristics ratio $R = \frac{A}{D} \times 100(\%)$
T10I4D100K	100,000	10.1	870	1.16
mushroom	8,124	23	119	19.327
chess	3,196	37	75	49.33
pumsb*	49,046	50.48	2,088	2.42
retail	88,162	10.3	16,470	0.0625
Chain-store	1,112,949	7.2	46,086	0.0156

transaction length of 37, and 49.33% items are present in every transaction. If $R > 10\%$, then the dataset is considered *dense*, which may generate many long frequent patterns and dynamic weighted frequent patterns. If $R \leq 10\%$, then the dataset is *sparse*. The dataset *chess* is too dense and the dataset *mushroom* is moderately dense. Similarly, datasets *T10I4D100k* and *pumbsb** are moderately sparse datasets; datasets *retail* and *Chain-store* are too sparse datasets.

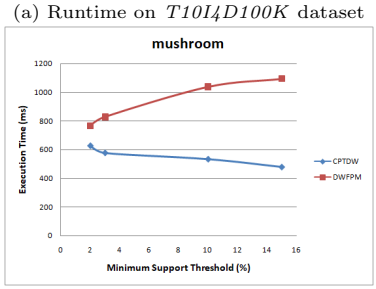
Synthetic dataset with synthetic weight. We used IBM synthetic dataset *T10I4D100k* developed by the IBM Almaden Quest research group. The dataset was obtained from the frequent itemset mining dataset respiratory [5]. The dataset do not provide weight values. According to the real world scenario, the weight values of each item was heuristically chosen to be in the range from 0.1 to 0.9, and randomly generated by using a log-normal distribution. The pattern generation times for this dataset for both the existing DWFPM algorithm and our proposed CPTDW algorithm are shown in Fig. 6(a).

Real life datasets with synthetic weight. We used real-life datasets *chess, mushroom, pumsb** and *Retail* obtained from the frequent itemset mining

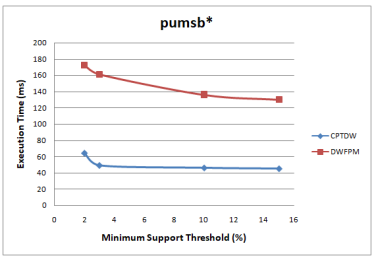


(a) Runtime on *T10I4D100K* dataset

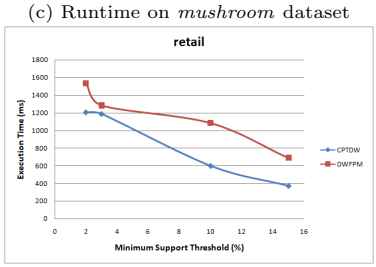
(b) Runtime on *chess* dataset



(c) Runtime on *mushroom* dataset



(d) Runtime on *pumsb** dataset



(e) Runtime on *retail* dataset

(f) Runtime on *Chain-store* dataset

Fig. 6: Experimental Results

dataset respiratory [5]. These datasets do not provide weight values. So, weights for items were generated randomly by using log-normal distribution. The pattern generation times for these datasets for both the existing DWFPM algorithm and our proposed CPTDW algorithm are shown in Figs. 6(b)–6(e).

Real life dataset with real weight. We used real-life dataset *Chain-store* obtained from SPMF, an open-source data mining library [8] consisting of multiple data mining applications and databases. This dataset was taken from a major chain store in California. We have taken real weight values for items from their utility table. The experiment is conducted on the first half transactions of the total transactions of the dataset. The pattern generation times of the DWFPM and CPTDW algorithms for this dataset are shown in Fig. 6(f).

Scalability of CPTDW. The experimental results on different datasets show that our proposed algorithm can easily handle large number of transaction containing databses (e.g., *T10I4D100k*, *Chain-store*). Hence, these experimental results demonstrate the scalability of our proposed algorithm to handle large

Table 4: Node count of CPTDW and DWFPM algorithms

Dataset	**N**ode Count (CPTDW)	**N**ode Count (DWFPM)
mushroom	12	21
pumsb	21	61
pumsb*	36	70
Chain-store	789	804

Table 5: Runtime Distributions of CPTDW and DWFPM Algorithms

Dataset	Tree construction time of CPTDW (ms)	Overall runtime of CPTDW (ms)	Tree construction time of DWFPM (ms)	Overall runtime of DWFPM (ms)
T10I4D100K	1,033	3,925	283	6,001
chess	19	37	3	40
mushroom	13	38	2	44
pumsb*	21	285	3	424
retail	36	367	10	518

number of transactions and distinct items. Our CPTDW algorithm outperforms the existing DWFPM algorithm by using an efficient tree structure and pattern growth mining technique in terms of runtime and memory usage.

Memory usage. Research on prefix-tree based frequent pattern mining shows that, the memory requirement for the prefix tress is low enough to use the gigabyte range memory available nowadays. Table 4 shows the total number of nodes of the prefix-trees at the time of generating dynamic weighted frequent patterns for different datasets for both the CPTDW and DWFPM algorithms. We have handled our tree structure very efficiently. Our CPTDW tree can represent transaction information in a very compressed form because transactions have many items in common. By using more prefix-sharing, our tree structure can save memory space.

Runtime distribution. Recall from Section 3.1 about our CPTDW tree construction process, CPTDW requires several swapping operations to restructure the tree structure in frequency descending order after the insertion of the transactions of each batch. So, there might arouse an issue that, CPTDW should require more time for the tree construction which is true. However, observed from Table 5, although CPTDW requires more time to construct the tree structure, we get a significant gain in overall runtime due to the frequency descending compact structure of the tree. For the following experiments presented in Table 5, the datasets were divided into 15 unique symbols and the minimum support threshold value of 3% is used.

5 Conclusions

Although there have been several efforts in mining weighted frequent patterns, they are not designed for handling many real-life situations where the importance of an item varies dynamically over time. Our key contribution of this paper is to provide a new tree-based approach to efficiently mine dynamic weighted frequent patterns. By storing batch-by-batch frequency and weight information, our compact pattern tree for dynamic weights (CPTDW) algorithm discovers accurate knowledge about dynamic weighted frequent patterns. CPTDW is applicable to real time data processing because it requires only one database scan. By using an efficient tree structure and mining approach, our CPTDW saves memory space and time consumption. Extensive performance analyses shows our algorithm is efficient when applied to both sparse and dense datasets, and can handle a large number of distinct items and transactions. As ongoing and future work, we are extending the current work for (i) incremental and interactive mining on databases with dynamic weights and for (ii) sliding window based dynamic weighted frequent patterns mining over data streams.

Acknowledgements

This project is partially supported by NSERC (Canada) and University of Manitoba.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules in large databases. In: VLDB 1994, pp. 487–499 (1994)
2. Ahmed, C.F., Tanbeer, S.K., Jeong, B., Lee, Y.: Handling dynamic weights in weighted frequent pattern mining. IEICE TIS E91-D(11), 2578–2588 (2008)
3. Ahmed, C.F., Tanbeer, S.K., Jeong, B., Lee, Y., Choi, H.: Single-pass incremental and interactive mining for weighted frequent patterns. ESWA 39(9), 7976–7994 (2012)
4. Barkwell, K.E., Cuzzocrea, A., Leung, C.K., Ocran, A.A., Sanderson, J.M., Stewart, J.A., Wodi, B.H.: Big data visualisation and visual analytics for music data mining. In: IV 2018, pp. 235–240 (2018)
5. Bayardo, R., Goethals, B., Zaki, M.J. (eds.): Proc. IEEE ICDM Workshop on FIMI 2004 (2004)
6. Cai, C.H., Fu, A.W., Cheng, C.H., Kwong, W.W.: Mining association rules with weighted items. In: IDEAS 1998, pp. 68–77 (1998)
7. Fan, C., Hao, H., Leung, C.K., Sun, L.Y., Tran, J.: Social network mining for recommendation of friends based on music interests. In: IEEE/ACM ASONAM 2018, pp. 833–840 (2018)
8. Fournier-Viger, P., Lin, J.C., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., Lam, H.T.: The SPMF open-source data mining library version 2. In: ECML PKDD 2016, Part III. LNCS (LNAI), vol. 9853, pp. 36–40 (2016)
9. Han, J., Cheng, H., Xin, D., Yan, X.: Frequent pattern mining: current status and future directions. DMKD 15(1), 55–86 (2007)

10. Hoi, C.S.H., Khowaja, D., Leung, C.K.: Constrained frequent pattern mining from big data via crowdsourcing. In: BigDAS 2017. AISC, vol. 770, pp. 69–79 (2017)
11. Koh, J., Shieh, S.: An efficient approach for maintaining association rules based on adjusting FP-tree structures. In: DASFAA 2004. LNCS, vol. 2973, pp. 417–424 (2004)
12. Leung, C.K., Hoi, C.S.H., Pazdor, A.G.M., Wodi, B.H., Cuzzocrea, A.: Privacy-preserving frequent pattern mining from big uncertain data. In: IEEE BigData 2018, pp. 5101–5110 (2018)
13. Leung, C.K., Jiang, F.: Efficient mining of 'following' patterns from very big but sparse social networks. In: IEEE/ACM ASONAM 2017, pp. 1025–1032 (2018)
14. Leung, C.K., Khan, Q.I., Hoque, T.: CanTree: a tree structure for efficient incremental mining of frequent patterns. In: IEEE ICDM 2005, pp. 274–281 (2005)
15. Leung, C.K., Khan, Q.I., Li, Z., Hoque, T.: CanTree: a canonical-order tree for incremental frequent-pattern mining. KAIS 11(3), 287–311 (2007)
16. Liu, J., Chang, Z., Leung, C.K., Wong, R.C.W., Xu, Y., Zhao, R.: Efficient mining of extraordinary patterns by pruning and predicting. ESWA 125, 55–68 (2019)
17. Perner, P.: Mining frequent subgraph pattern over a collection of attributed-graphs and construction of a relation hierarchy for result reporting. In: ICDM 2017. LNCS (LNAI), vol. 10357, pp. 323–344 (2017)
18. Phan, H., Le, B.: A novel parallel algorithm for frequent itemsets mining in large transactional databases. In: ICDM 2018. LNCS (LNAI), vol. 10933, pp. 272–287 (2018)
19. Rahman, M.M., Ahmed, C.F., Leung, C.K.: Mining weighted frequent sequences in uncertain databases. Inf. Sci. 479, pp. 76–100 (2019)
20. Tanbeer, S.K., Ahmed, C.F., Jeong, B., Lee, Y.: Efficient single-pass frequent pattern mining using a prefix-tree. Inf. Sci. 179(5), 559–583 (2009)
21. Wang, W., Yang, J., Yu, P.S.: WAR: weighted association rules for item intensities. KAIS 6(2), 203–229 (2004)
22. Yun, U.: Efficient mining of weighted interesting patterns with a strong weight and/or support affinity. Inf. Sci. 177(17), 3477–3499 (2007)
23. Yun, U., Leggett, J.J.: WFIM: weighted frequent itemset mining with a weight range and a minimum weight. In: SIAM SDM 2005, pp. 636–640 (2005)
24. Zhang, S., Zhang, C., Yan, X.: Post-mining: maintenance of association rules by weighting. Inf. Syst. 28(7), 691–707 (2003)