International Journal of Innovative
Computing, Information and Control
Volume **11**, Number **4**, August **2015**

ICIC International ©2015 ISSN 1349-4198
pp. 1421–1439

WSWF: A WEIGHTED SLIDING WINDOW FILTERING ALGORITHM FOR FREQUENT WEIGHTED ITEMSETS MINING

Mohammed M. Fouad[1,2], Mostafa G. M. Mostafa[1], Abdulfatah S. Mashat[2] and Tarek F. Gharib[2,1]

[1]Faculty of Computers and Information Sciences
Ain Shams University
Cairo 11566, Egypt
mgmostafa@cis.asu.edu.eg

[2]Faculty of Computing and Information Technology
King Abdulaziz University
Jeddah 21589, Saudi Arabia
{ mmfouad; tfgharib; asmashat }@kau.edu.sa

Received October 2014; revised March 2015

Abstract. *Weighted itemset databases are introduced to overcome the problem of neglecting important itemsets when mining traditional databases for frequent itemsets based only on their support. In this paper, we present a weighted sliding window filtering (wSWF) algorithm for efficiently mining frequent weighted itemsets. The proposed algorithm uses Diffsets strategy for fast weighted support calculation. The main objective of the proposed algorithm is to minimize the number of generated candidates and reduce memory requirements without using any extra data structures. Experimental results show that the proposed algorithm outperforms recently cited algorithms in terms of runtime and memory requirements with a significant performance enhancement. In addition, the experiments show that wSWF algorithm is scalable to run on large databases even at low threshold values.*

Keywords: Data mining, Frequent weighted itemsets, Sliding window filtering, Diffsets strategy

1. **Introduction.** Frequent itemset mining (FIM) is one of the major tasks in the data mining field that has been introduced and applied to different types of data, such as transactional databases [1-3], temporal database [4,5], time series databases [6], utility databases [7-9], and streaming databases [10]. It also has been applied in many application areas, such as bioinformatics [11], web-usage analysis [12]. Over the past twenty years, there are many presented FIM algorithms and techniques starting with Apriori algorithm by [1]. There are also different techniques appearing like incremental mining [13,14], and interactive mining [15,16].

The main problem with FIM is that it is interested only in itemset frequency. However, some items are infrequent but more significant and important than frequent ones. For example, in a market analysis, furniture is not a frequent item in sales, but it produces a higher profit than some frequent items like tea and sugar. Therefore, in some applications, the benefits associated with the item (e.g., profit, cost or weight value) are more interesting and require extracting the relative benefits between these items. To overcome this problem, weighted association rule's mining was introduced to consider the items' weights in transactional databases. Ramkumar et al. [17] were the first to propose an Apriori-based algorithm for frequent weighted itemset (FWI) mining with an application to weighted association rules mining.

Mining frequent weighted itemsets can be defined as finding the significant itemsets with higher weights. The weight of each transaction is calculated based on the weights of its items. This leads to "Weighted support" as a redefined measure of itemset weight based on the weight of the transactions that contain this itemset. An itemset is considered frequent weighted itemset (FWI) if its weighted support is no less than a user-defined threshold called minimum weighted support (minWS). Frequent weighted itemsets mining can be applied to various applications such as market analysis [18,19], e-commerce management [20] and some biomedical data analysis [21].

Existing studies [22-25] adopt the trend of utilizing the traditional FIM approaches to cope with mining frequent weighted itemsets. The main problem with these techniques is that they produce a large number of candidates during mining process that requires a huge block of memory and extra running time to check these candidates. This could be worst when working with very large and dense databases or at low minWS thresholds.

To solve this issue, we proposed an effective algorithm for mining frequent weighted itemsets from weighted transaction databases. The main contributions of this work are summarized as follows.

1. A new algorithm, named *weighted sliding window filtering* (wSWF) is presented for mining frequent weighted itemsets with efficient memory usage and enhanced performance.
2. Both real and synthetic datasets are used in the experimental studies to compare the performance of the proposed algorithm with recently cited algorithms. Experimental results show a significant performance enhancement in both running time and memory usage aspects. In addition, the experiment shows that wSWF algorithm is scalable to run on large databases even at low threshold values.

The rest of the paper is organized as follows. Section 2 has introduced some frequent weighted itemset (FWI) preliminaries and definitions with recently cited related work. The proposed wSWF algorithm is presented in detail in Section 3. Experimental results and discussion are presented in Section 4. Finally, conclusions are drawn in Section 5.

2. Preliminaries and Related Work.

2.1. Preliminaries.

A weighted transaction database $DB = \{T_1, T_2, \ldots, T_m\}$ contains a set of transactions where each transaction T_p has a unique identifier called Transaction ID (TID). $I = \{i_1, i_2, \ldots, i_n\}$ is the set of all items in DB, where each item i_k has a positive weight w_k. For example, consider the database presented in Table 1 that contains six transactions and five items, and the weight of each item is shown in Table 2.

Table 1. Example of the transactional database DB

TID	Transaction	tw
T_1	A, C, D, E	0.28
T_2	B, C, E	0.36
T_3	A, B, E	0.43
T_4	A, C, D	0.30
T_5	A, B, C, E	0.40
T_6	A, B, D	0.40

TABLE 2. The weights table of items in DB

Item	A	B	C	D	E
Weight	0.5	0.6	0.3	0.1	0.2

Definition 2.1. *Transaction weight of a transaction T_p with N_p items is denoted as $tw(T_p)$ and defined as:*

$$tw(T_p) = \frac{\sum_{i_k \in T_p} w_k}{N_p} \tag{1}$$

For example, from Tables 1 and 2, the transaction weight of T_2 that contains 3 items (B, C and E) is calculated as $tw(T_2) = (0.6 + 0.3 + 0.2)/3 = 0.36$. Table 1 shows the transaction weight values of all the transactions in example DB.

Definition 2.2. *Transaction ID list of an itemset X denoted as $TID(X)$, is the set of all transactions IDs that contain X and $|TID(X)|$ is the number of transactions that contains X in the database [26].*

Definition 2.3. *The weighted support of an itemset X is denoted as $ws(X)$ and defined as:*

$$ws(X) = \frac{\sum_{T_p \in TID(X)} tw(T_p)}{\sum_{T_p \in DB} tw(T_p)} \tag{2}$$

Definition 2.4. *An itemset is called a frequent weighted itemset (FWI) if its weighted support is no less than a user-defined minimum weighted support value denoted as minWS. Otherwise, it is called rare or infrequent weighted itemset.*

For example, for a given itemset $X = \{AB\}$, $TID(X) = \{T_3, T_5, T_6\}$, then $ws(X) = (0.43 + 0.40 + 0.40)/2.17 \approx 0.56$. If minWS threshold is set to 0.4, then X is a frequent weighted itemset.

Problem Statement. Given a transactional database DB with a pre-specified weight w_k value for each distinct item i_k in DB and a user-defined minimum weighted support value denoted as minWS, the problem of mining frequent weighted itemsets from DB is to discover all the itemsets with weighted support values not less than minWS threshold.

Lemma 2.1. *Traditional frequent itemset mining is a special case of frequent weighted itemset mining when all the items have the same weight.*

Proof: Let all the items in the database have equal weight value, i.e., $w_k = 1$ for each item i_k in DB. Then:

$$tw(T_p) = \frac{\sum_{i_k \in T_p} w_k}{N_p} = \frac{\sum_{i_k \in T_p} 1}{N_p} = \frac{N_p}{N_p} = 1, \quad \forall T_p \in DB$$

Moreover, the weighted support for a given itemset X is:

$$ws(X) = \frac{\sum_{T_p \in TID(X)} tw(T_p)}{\sum_{T_p \in DB} tw(T_p)} = \frac{|TID(X)|}{|DB|} = support(X)$$

For example, assuming that all the items in Table 2 have the same weight value equal to 1, then the weight of transaction T_2 is calculated as $tw(T_2) = (1 + 1 + 1)/3 = 1$. All the transactions in the database will have the same transaction weight value (equal to 1). The weighted support $ws(AB) = (1 + 1 + 1)/6 = 0.5$, which equals the traditional support value that considers the frequency of the itemset in the database.

Property 2.1 (Downward Closure Property)**.** *For any itemset X, if X is an FWI, any subset of X is an FWI also.*

Proof: Let X be a superset of Y and X is an FWI, i.e., $ws(X) \geq \min WS$.

$$\begin{aligned}
&\text{if } Y \subset X, \text{ then } TID(X) \subset TID(Y)\\
&\Rightarrow \textstyle\sum_{T_p \in TID(Y)} tw(T_p) \geq \sum_{T_p \in TID(X)} tw(T_p)\\
&\Rightarrow \frac{\sum_{T_p \in TID(Y)} tw(T_p)}{\sum_{T_p \in DB} tw(T_p)} \geq \frac{\sum_{T_p \in TID(X)} tw(T_p)}{\sum_{T_p \in DB} tw(T_p)} \Rightarrow ws(Y) \geq ws(X)\\
&\Rightarrow ws(Y) \geq ws(X) \geq \min WS \Rightarrow Y \text{ is an FWI}
\end{aligned}$$

For example, in the example database shown in Table 1, at $minWS = 0.4$, itemset $\{AB\}$ is an FWI because $ws(\{AB\}) = 0.56$. The sub-itemsets $\{A\}$ and $\{B\}$ are also FWIs because their weighted support is 0.83 and 0.73 respectively.

2.2. **Related work.** Frequent weighted itemset mining was issued to overcome the discovery problem of frequent itemsets in the weighted transaction database. Ramkumar et al. [17] introduced this problem, firstly, and proposed an Apriori-based algorithm for FWI with an application to weighted association rules mining (WARM). Their proposed algorithm, WIS, simply starts with level-1 candidates and uses Apriori candidate generation technique to generate level-2 candidates and so on until no more candidates were found. The main problem of this algorithm is that it generates a huge number of candidates who required multiple database scans to calculate their weighted support leading to poor performance.

Later, Tao et al. [22] proposed a model for weighted support measure and weighted downward closure property. They presented WARM algorithm for weighted association rule mining. The algorithm is based on the Apriori algorithm and the number of candidates is maintained using the weighted downward closure property. Muyeba et al. [27] adopted the same approach in their work of fuzzy weighted association rules mining to reduce some steps in candidate generation phase.

Yun et al. [28] were interested in mining maximal weighted frequent patterns in transaction databases. They proposed MWFIM algorithm using a prefix-tree with descending weight order to minimize search space and prune infrequent patterns. MWFIM algorithm discovers all weighted frequent patterns, and then mines the maximal frequent pattern from it. The pattern is considered maximal weighted frequent if it has no weighted frequent superset.

Recently, Le et al. [23] proposed a new method for frequent weighted itemset mining using WIT-trees, as an extension of IT-trees proposed by Zaki and Hsiao in [29]. The original one, called WIT-FWI, simply used downward closure property for early pruning and TID list notation for weighted support counting. Vo et al. [24] enhanced the performance of WIT-FWI algorithm by utilizing some features in the ID list implementation and presented WIT-FWI-MODIFY algorithm. Finally, they used Diffset strategy proposed by Zaki and Gouda in [30] for fast support counting and memory space reduction to present WIT-FWI-DIFF algorithm. The main problem of this method appears when mining a large database with long transactions. In this case, the number of level-2 candidates is huge, i.e., for N distinct items, there are $(N * (N-1)/2)$ candidates, which is high computationally expensive because the Diffset strategy is applied over TID lists for each item.

In some applications, such as a fraud detection and statistical disclosure risk assessment of the census, the infrequent or rare weighted itemsets are more interesting than frequent ones. Cagliero and Garza [31] were the first to address this problem and proposed two algorithms for mining infrequent weighted itemsets, called MIWI, using a frequent pattern

growth technique. They compared their performance with a traditional, non-weighted, infrequent itemsets mining algorithms, called MINIT [32]. Their experimental studies showed a great performance enhancement over MINIT algorithm because of the efficient pruning strategy in FP-growth technique.

3. **Mining Frequent Weighted Itemsets.** In this section, we present the proposed weighted sliding window filtering (wSWF) algorithm for efficient frequent weighted itemsets mining. The proposed algorithm is based on two main techniques. The first is the sliding window filtering technique which is used to generate the candidate weighted frequent itemsets. The second technique is the Diffsets strategy, which is used for counting weighted support of generated candidates. The first two subsections present a brief discussion about these two techniques. In the later section, the proposed wSWF algorithm is illustrated with a detailed example.

3.1. **Sliding window filtering.** Lee et al. [33] introduced the technique of sliding-window filtering (SWF) for incremental mining of association rules. The database is partitioned into several parts, and SWF algorithm applied filtering threshold for each part to handle generated level-2 candidates (denoted as $C2$) in each part. The filtering threshold is based on the local support of the $C2$ candidates in each part. The idea of SWF is to reduce the number of $C2$ that are generated from each part. $C2$ candidates are carried over from one part to the next one until all the database parts are processed.

The final $C2$ candidates list is small and close to the frequent level-2 itemsets because the filtering threshold is applied to all parts. This will enhance the overall mining process in both running time and required memory because later candidates are generated based on the small list of $C2$ candidates.

3.2. **Diffsets strategy.** For fast support counting, each itemset is attached to a list of transactions; it occurs in (denoted as *TID* list). Simply, the database is scanned once to find a level-1 itemsets and their *TID* lists. For later levels, the itemsets in level k is joined together to generate level $k+1$ candidate itemsets. In the joint operation, the *TID* list for the candidate itemset is derived by the intersection of the *TID* lists of its parents. For example, from the database in Table 1, $TID(A) = \{T_1, T_3, T_4, T_5, T_6\}$ and $TID(B) = \{T_2, T_3, T_5, T_6\}$, then $TID(AB) = \{T_3, T_5, T_6\}$ as the intersection of $TID(A)$ and $TID(B)$. The weighted support of each candidate itemset is calculated as illustrated in Definition 2.3.

The main problem with this technique is the memory requirements, especially in the large and dense databases in which the *TID* list of each item is very long, and the intersection operation requires more processing time.

Zaki and Gouda [30] proposed Diffsets strategy for fast support counting and minimum memory usage. Diffset computes the difference set between two *TID* lists with the same equivalence class which is much more less than the size of *TID* lists in large or dense databases. Their proposed technique starts again by scanning the database and holds *TID* list for level-1 itemsets. For next levels, the following definition is used to enhance *TID* list notation.

Definition 3.1 (Itemset Diffset)**.** *Consider two itemsets PX and PY with size k that have the same equivalence class $\{P\}$. The Diffset of their join PXY of size $k+1$ is calculated by:*

$$DIFF(PXY) = \begin{cases} TID(PX) - TID(PY), & k = 1 \\ DIFF(PY) - DIFF(PX), & k > 1 \end{cases} \tag{3}$$

where "−" is the set difference operator, i.e., $A - B$ is the set of all items in A and not in B. The weighted support of a given candidate itemset $\{PXY\}$ is defined as:

$$ws(PXY) = ws(PX) - \frac{\sum_{T_p \in DIFF(PXY)} tw(T_p)}{\sum_{T_p \in DB} tw(T_p)} \tag{4}$$

For example, letting itemset $\{AB\}$ be a candidate itemset generated by joining itemsets $\{A\}$ and $\{B\}$, i.e., $P = \{\ \}$, $X = \{A\}$ and $Y = \{B\}$, then $ws(\{AB\})$ is calculated as follows:

$$TID(A) = \{T_1, T_3, T_4, T_5, T_6\}, \quad TID(B) = \{T_2, T_3, T_5, T_6\}$$
$$DIFF(AB) = TID(A) - TID(B) = \{T_1, T_4\}$$
$$ws(\{AB\}) = ws(A) - \frac{\sum_{T_p \in DIFF(AB)} tw(T_p)}{\sum_{T_p \in DB} tw(T_p)} = 0.83 - \frac{0.28+0.3}{2.17} \approx 0.56$$

3.3. **The proposed wSWF algorithm.** The main objective of wSWF algorithm is to reduce the memory usage, computation of weighted support and minimize the candidate generation phase running time. This could be achieved by generating level-2 candidates ($C2$) that are approximately close to frequent level-2 itemsets (FWI_2). In the proposed algorithm, the database is divided into some parts while each part is processed sequentially. $C2$ candidates are propagated and filtered from each part to the next one until all the parts are processed. During mining step, Diffsets strategy is used to handle weighted support counting of the generated candidates and minimize required memory usage. Figure 1 shows the algorithm steps in detail.

3.4. **Illustrative example.** Considering the database shown in Table 3 which is split into two parts, we shall apply the wSWF algorithm at minimum weighted support $\min WS = 0.4$ to find frequent weighted itemsets.

TABLE 3. Example of the transactional database DB

TID	**Transaction**	***tw***	***TW***
T_1	A, C, D, E	0.28	$TW(P_1) = 1.07$
T_2	B, C, E	0.36	
T_3	A, B, E	0.43	
T_4	A, C, D	0.30	$TW(P_2) = 2.17$
T_5	A, B, C, E	0.40	
T_6	A, B, D	0.40	

The algorithm starts by calculating the transaction weight (tw) of each transaction using Equation (1). The total transaction weight for each part is calculated as:

$$TW(P_n) = \sum_{T_p \in P_n} tw(T_p) + \sum_{k=1}^{n-1} TW(P_k) \tag{5}$$

In the example database shown in Table 3, there are two parts P_1 and P_2. Then:

$$TW(P_1) = tw(T_1) + tw(T_2) + tw(T_3) = 1.07, \text{ and}$$
$$TW(P_2) = tw(T_4) + tw(T_5) + tw(T_6) + TW(P_1) = 2.17$$

Note that total transaction weight of the database DB, denoted as $TW(DB)$, is equal to $TW(P_2)$ because part P_2 is the last part in the database. The transaction weight of each part will be used to calculate the local weighted support threshold while the transaction weight of the database will be used for the global weighted support threshold.

The algorithm processes each part (as in procedure **ProcessPart(P_n, TW)**) to generate the local $C2$ (denoted as $C2_n$) list and merge it with the global $C2$ list (denoted

Input:
$DB \Rightarrow$ transactional database with M items split into N Parts
$W_i \Rightarrow$ weights for items $(i = 1, 2, \ldots, M)$
minWS $\Rightarrow$ minimum weighted support threshold

Output:
The frequent weighted itemsets (FWI)

wSWF ( )
1. $C1 = C2 = \emptyset$, FWI $= \emptyset$
2. Calculate $\boldsymbol{TW}$
3. foreach part P_n in DB, $n = 1, 2, \ldots, N$
4. Call **ProcessPart($\boldsymbol{P_n, TW}$)**
5. foreach itemset X in $C1$
6. if $ws(X) \geq \min WS * TW(DB)$ Then Add X to FWI_1
7. foreach itemset X in $C2$
8. $DIFF(X) = TID(X_1) - TID(X_2)$ // Equation (3), $k = 1$
9. $ws(X) = ws(X_1) - CalculateWS(DIFF(X))$ // Equation (4)
10. if $ws(X) \geq \min WS * TW(DB)$ Then
11. Add X to FWI_2
12. Call **CandidateGeneration($\boldsymbol{FWI_2}$)**

ProcessPart($\boldsymbol{P_n, TW}$)
13. $C2_n = \emptyset$
14. foreach transaction T in P_n
15. foreach item X_i in T
16. $C1[X_i] += TW(T)$
17. Add $ID(T)$ to $TID(X_i)$
18. foreach item X_j in T, $j > i$
19. $C2_n[X_i, X_j] = C2[X_i, X_j] + TW(T)$
20. Add $ID(T)$ to $TID(X_j)$
21. foreach itemset A in $C2_n$
22. if $ws(C2_n[A]) \geq \min WS * TW(P_n)$ Then
23. if $A \in C2$ Then
$ws(C2[A]) = ws(C2[A]) + ws(C2_n[A])$
24. else Add A to $C2$

CandidateGeneration($\boldsymbol{FWI_k}$)
25. $FWI_{k+1} = \emptyset$
26. foreach itemset X_i in FWI_k
27. foreach itemset X_j in FWI_k, $j > i$
28. if $Prefix(X_i) = Prefix(X_j)$ Then
29. $Y = X_i \cup X_j$
30. $DIFF(Y) = DIFF(X_j) - DIFF(X_i)$ // Equation (3)
31. $ws(Y) = ws(X_i) - CalculateWS(DIFF(Y))$
32. if $ws(Y) \geq \min WS * TW(DB)$
33. Add Y to FWI_{k+1}
34. if $FWI_{k+1} \neq \emptyset$ Then
35. Call **CandidateGeneration($\boldsymbol{FWI_{k+1}}$)**

FIGURE 1. The proposed wSWF algorithm

as $C2$). For example, in part P_1, the first transaction $\{A, C, D, E\}$ is scanned. For item $\{A\}$, $C1(A)$ is updated to be 0.28 and $TID(A) = \{T_1\}$. In the inner loop in lines 18 to 20, the algorithm passes next items to generate candidate $C2_1$. In this case, there are three candidates, $\{AC, AD, AE\}$ each one with weighted support equal to 0.28. The same is applied to the rest of the items in the first transaction. After the whole transaction is processed, three more candidates $\{CD, CE, DE\}$ are added to $C2_1$. The algorithm continues to process all the transactions in part P_1, updates the support and the *TID* list of the candidate itemsets in $C1$, and produces the final candidate $C2_1$ list for this part.

After all the transactions in part, P_1 is being processed, and the algorithm checks the itemsets in $C2_1$ and adds the frequent ones (according to current part TW) to $C2$, or updates its weighted support if it already exists in $C2$ in lines 21 to 24. For example, the support of all the candidates in $C2_1$ is checked against the local threshold of part P_1, which is $\min WS * TW(P_1) = 0.4 * 1.07 = 0.43$. Only $\{AB, AE, BE$ and $CE\}$ are moved to $C2$ while the others are removed because their weighted support is less than the local threshold.

The algorithm continues with processing the transactions in part P_2 and updates the candidates in $C1$ with updated *TID* lists and generates the candidate itemsets in $C2_2$ in lines 13 to 20. The support of all the candidates in $C2_2$ is checked against the local threshold of part P_2, which is $\min WS * TW(P_2) = 0.4 * 2.17 = 0.87$. If the candidate itemset is found in $C2$, then update its weighted support, otherwise add it to $C2$. For example, itemset $\{AB\}$ already exists in $C2$ after processing part P_1 and appears in $C2_2$, so its weighted support is updated to be $0.43 + 0.8 = 1.23$. This is also applied to $\{AE\}$, $\{BE\}$ and $\{CE\}$ itemsets. Figure 2 shows the detailed steps of processing parts P_1 and P_2 in the example database.

Processing part P1

$C2_1$			C2		
Itemset	**support**		**Itemset**	**start**	**support**
{AC}	0.28		{AE}	1	0.71
{AD}	0.28		{CE}	1	0.64
{AE}	0.71		{BE}	1	0.79
{CD}	0.28	==>	{AB}	1	0.43
{CE}	0.64				
{DE}	0.28				
{BC}	0.36				
{BE}	0.79				
{AB}	0.43				

Processing part P2

$C2_2$			C2		
Itemset	**support**		**Itemset**	**start**	**support**
{AC}	0.7		{AE}	1	1.11
{AD}	0.7		{CE}	1	1.04
{CD}	0.3		{BE}	1	1.19
*{AB}	0.8	==>	{AB}	1	1.23
*{AE}	0.4		{AC}	2	0.7
{BC}	0.4		{AD}	2	0.7
*{BE}	0.4				
*{CE}	0.4				
{BD}	0.4				

FIGURE 2. Processing DB parts in wSWF algorithm

After processing all parts, there are five candidates in $C1$ with their weighted support and TID list as shown in Table 4. The global minimum weighted support threshold is calculated as $\min WS * TW(DB) = 0.4 * 2.17 = 0.87$. The algorithm checks the support of the candidate itemsets in $C1$ and adds the ones with support greater than or equal to 0.87 to the output frequent weighted itemsets (FWI) list in lines 5 and 6. As shown in Table 4, all the itemsets have support greater than 0.87, so all the candidates in $C1$ candidates are frequent.

TABLE 4. $C1$ candidates and their weighted support

Itemset	***TID* List**	**Weighted Support**
A	$\{T_1, T_3, T_4, T_5, T_6\}$	1.81
B	$\{T_2, T_3, T_5, T_6\}$	1.60
C	$\{T_1, T_2, T_4, T_5\}$	1.34
D	$\{T_1, T_4, T_6\}$	0.97
E	$\{T_1, T_2, T_3, T_5\}$	1.47

As shown in Figure 2, there are six candidates in $C2$ after processing part P_2. The algorithm calculates the diffsets of these candidates using Equation (3) with $k = 1$ to calculate their actual weighted support using Equation (4), and then adds the frequent ones to FWI_2 for candidate generation step in lines 7 to 11. For example, consider the candidate itemset $\{AC\}$, $DIFF(AC) = TID(A) - TID(C) = \{T_1, T_3, T_4, T_5, T_6\} - \{T_1, T_2, T_4, T_5\} = \{T_3, T_6\}$. Then $ws(AC) = ws(A) - \sum_{Tp \in DIFF(AC)} tw(T_p) = 1.81 - (0.43 + 0.4) = 0.98$, which is ≥ 0.87, $\{AC\}$ is added to FWI_2 itemsets.

wSWF algorithm generated only six $C2$ candidates, while WIT-based algorithms will produce 10 candidates because there are five frequent levels-1 itemsets. Table 5 shows that all the generated $C2$ candidates by wSWF algorithm are frequent itemsets (FWI_2) after calculating their weighted support.

TABLE 5. $C2$ candidate itemsets

C2				***FWI_2***	
Itemset	**Diffset**	***ws***		**Itemset**	***ws***
$\{AB\}$	$\{T_1, T_4\}$	1.23		$\{AB\}$	1.23
$\{AC\}$	$\{T_3, T_6\}$	0.97		$\{AC\}$	0.97
$\{AD\}$	$\{T_3, T_5\}$	0.97	$\Rightarrow$	$\{AD\}$	0.97
$\{AE\}$	$\{T_4, T_6\}$	1.11		$\{AE\}$	1.11
$\{BE\}$	$\{T_6\}$	1.19		$\{BE\}$	1.19
$\{CE\}$	$\{T_4\}$	1.04		$\{CE\}$	1.04

In candidates generation step in lines 25 to 35, the algorithm joins the frequent itemsets with size k to generate candidates of size $(k + 1)$. Only the itemsets that have the same prefix can be joined together to generate the new candidate itemset. The Diffsets of the new candidate itemset is calculated using Equation (3) with $k > 1$ and its weighted support is calculated using Equation (4). If this candidate is found to be frequent, then it is added to FWI_{k+1}, otherwise, it is removed. This process is repeated until no new candidates are generated.

Table 6 shows the detailed steps for candidate generation phase for the FWI_2 itemsets shown in Table 5. Only four frequent itemsets $\{AB, AC, AD, AE\}$ can be joined together because they have the same prefix $\{A\}$. The remaining candidates $\{BE, CE\}$ do not share

any common prefix with any other FWI_2 itemsets so they are neglected during candidate generation phase. For example, when $X_i = \{AB\}$ and $X_j = \{AC\}$ are joined together, a new candidate $Y = \{ABC\}$ is generated in line 29. The Diffset $DIFF(Y)$ is calculated by $DIFF(X_j) - DIFF(X_i) = DIFF(AC) - DIFF(AB) = \{T_3, T_6\} - \{T_1, T_4\} = \{T_3, T_6\}$. The weighted support $ws(Y) = ws(X_i) - \sum_{Tp \in DIFF(Y)} tw(T_p) = 1.23 - (0.43 + 0.4) = 0.4$, which is less than 0.87, so $\{ABC\}$ is not frequent and will not be added to FWI_3 itemsets.

The support of the generated candidates is checked against a global threshold (0.87) but no one satisfies this threshold. Because FWI_3 does not contain any frequent weighted itemsets, the algorithm stops running and produces the output *FWIs* as $\{A\}$, $\{B\}$, $\{C\}$, $\{D\}$, $\{E\}$, $\{AB\}$, $\{AC\}$, $\{AD\}$, $\{AE\}$, $\{BE\}$, $\{CE\}$.

TABLE 6. Example of the candidate generation phase

Itemset		**Candidate**	**Diffset**	$\boldsymbol{ws}$
$\{AB\}$	$\{AC\}$	$\{ABC\}$	$\{T_3, T_6\}$	$1.23 - (0.43 + 0.4) = 0.40$
	$\{AD\}$	$\{ABD\}$	$\{T_5, T_6\}$	$1.23 - (0.43 + 0.4) = 0.40$
	$\{AE\}$	$\{ABE\}$	$\{T_6\}$	$1.23 - (0.4) = 0.83$
$\{AC\}$	$\{AD\}$	$\{ACD\}$	$\{T_5\}$	$0.97 - (0.4) = 0.57$
	$\{AE\}$	$\{ACE\}$	$\{T_4\}$	$0.97 - (0.3) = 0.67$
$\{AD\}$	$\{AE\}$	$\{ADE\}$	$\{T_4, T_6\}$	$0.97 - (0.7) = 0.27$

4. **Experimental Results.** Some experiments are performed to evaluate the performance of the proposed wSWF algorithm in comparison with recently cited FWI algorithms presented in [24]. The comparisons are interested in different aspects, including algorithm running time, memory usage and number of generated $C2$ itemsets. In addition, the scalability of the proposed algorithm is tested with respect to variable database sizes. All the algorithms are implemented using Visual C++ 2008 Express Edition on Windows 7 operating systems with 2.3 GHz PC and 3GB memory.

4.1. **Datasets.** Five real datasets (Chess, Mushroom, Connect, Accidents and BMS-POS) are used in the experiments. These real datasets are obtained from frequent itemset mining dataset repository [34]. These datasets do not include the weight values of each item. We generated random weight value for each item ranging from 1 to 10 as in the previously published work [35,36].

Table 7 shows the statistical information about the datasets used in the experiments. For each dataset, it shows the number of transactions (*#Trans*), number of distinct items (*#Items*) and average items per transaction. This information gives a clear view of the density of the dataset. For example, Chess dataset has 75 distinct items while each transaction contains about 37 items, which means about 50% of the items are in every transaction because Chess dataset is a very dense database. BMS-POS dataset is an example of sparse datasets because it contains about 515K transactions and 1.6K distinct items but each transaction holds only 6.5 items on average.

Table 8 summarizes the experimental results with respect to the number of output frequent weighted itemsets (*#FWIs*) for each dataset with a different minWS threshold. The results presented in Table 8 show that the number of frequent weighted itemsets of BMS-POS is small, for Accidents and Mushroom are medium, and for Chess and Connect are large. For example, BMS-POS database produces about 130 *FWIs* while Accidents and Chess produce about 1,400 and 13,800 *FWIs* respectively. In addition, we can notice that Connect dataset is considered very sensitive to minWS threshold because when minWS

TABLE 7. Frequent itemsets mining datasets [34]

Dataset	***#Trans***	***#Items***	**Avg. Items per Trans**
Chess	3,196	75	37
Mushroom	8,124	119	23
Connect	67,557	129	43
Accidents	340,183	468	33.8
BMS-POS	515,597	1,656	6.5

TABLE 8. Number of $FWIs$ for each database

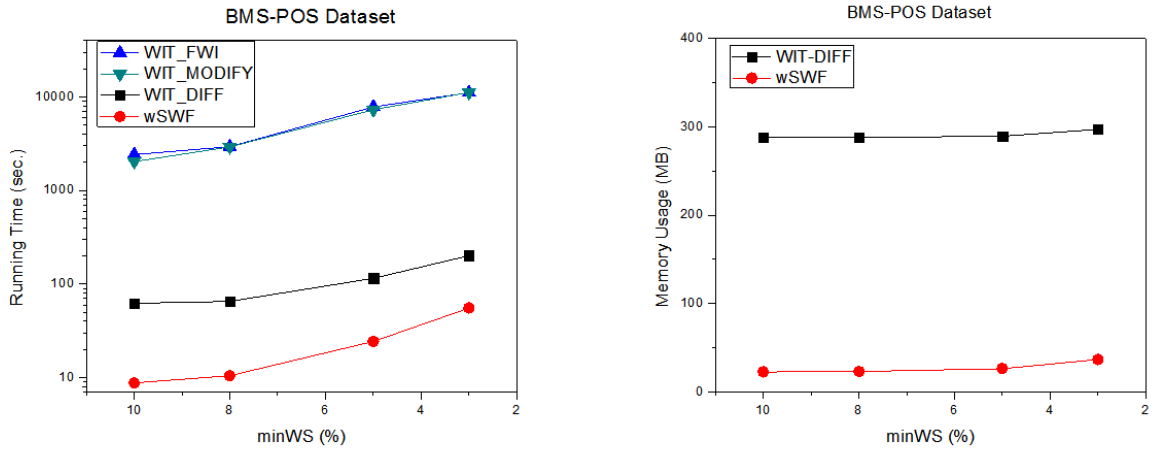
Dataset	**minWS (%)**	***#FWIs***
BMS-POS	10	15
	8	18
	5	53
	3	131
Accidents	95	15
	90	71
	88	314
	85	1,412
Mushroom	60	45
	50	141
	40	485
	30	2,223
Chess	90	402
	85	1,705
	80	5,379
	75	13,845
Connect	96	1,209
	94	4,739
	92	12,823
	90	30,079

changes slightly from 96% to 90%, *#FWIs* increased dramatically from 1209 to 30079 (about 25 times).

4.2. **Runtime and memory usage analysis.** In this experiment, the performance of wSWF algorithm is compared with three recent algorithms (WIT-FWI, WIT-MODIFY and WIT-DIFF) presented in [24]. The WIT-based algorithms use both WIT tree data structure, to store and generate candidates, and TID list representation for weighted support counting. The main problem of these algorithms is in the number of generated candidates during mining operation. WIT-based algorithms generate huge number of candidates which requires much memory space and also extra running time to compute the weighted support of these candidates and remove the infrequent ones.

Figures 3(a)-7(a) show the measured running time (in seconds) for these algorithms and the proposed wSWF algorithm with different minWS threshold values. We also measured the required memory usage for wSWF and WIT-DIFF algorithm only as it overcomes other two algorithms in both running time and memory usage. Figures 3(b)-7(b) show the memory usage comparison between wSWF and WIT-DIFF algorithms.

In Figure 3(a), although BMS-POS dataset produces a small number of *FWIs*, the mining process at minWS ranging from 10% to 3% requires long running time. For example, in WIT-DIFF the running time ranges from 62 to 201 seconds while wSWF needs about 8 to 55 seconds only with average speed-up ratio 80.3%. For memory usage, we notice that WIT-DIFF requires huge memory to create WIT-Tree structure. In contrast, the needed memory for wSWF is very low because it stores only the *FWIs* that are generated from the reduced list of candidates as shown in Figure 3(b). On average, wSWF reduces more required memory than WIT-DIFF with about 90%.



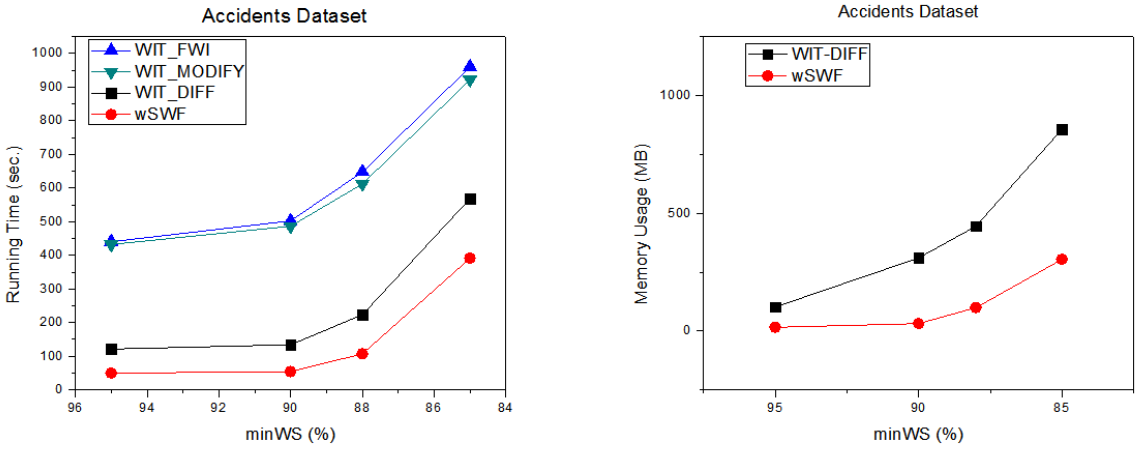
(a) (b)

FIGURE 3. Runtime comparison and memory usage – BMS-POS dataset

For Accidents dataset shown in Figure 4(a), the running times at 95% and 90% are the same, and there is no big difference in *#FWIs*. When minWS changed to 85%, the number of *FWIs* increased to 1412 that required much time to extract from the database. In all minWS values, wSWF algorithms overcome WIT-DIFF with a reasonable average speed-up ratio about 50%. In Figure 5(a), at 60% minWS, WIT-DIFF requires 8 seconds to extract *FWIs* while wSWF requires only 0.5 seconds. When the minWS reaches 30%, wSWF and WIT-DIFF need 38 and 86 seconds respectively. The average speed-up ratio over all minWS values is about 75%. As shown in Figures 4(b) and 5(b), wSWF memory requirements are relative to the number of generated *FWIs*. For WIT-DIFF algorithm, the required memory is an issue, especially at low minWS values. When the number of *FWIs* increased, the number of nodes in WIT-Tree also increased, which requires more memory to store. wSWF achieves a high reduction ratio in both Accidents and Mushroom datasets with about 79% and 78% respectively.

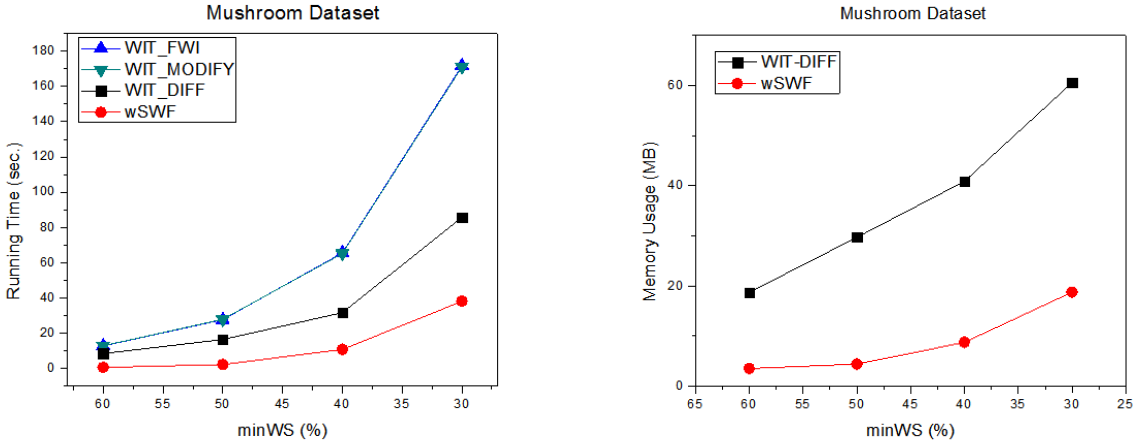
For Chess dataset, wSWF runtime is only 0.3 and 1.1 seconds at minWS 90% and 75% respectively. On the other hand, WIT-DIFF needs about 3.5 and 11.4 seconds in the same minWS values as shown in Figure 6(a). On average, the speed-up ratio of *wSWF* is about 83%. This occurs because Chess dataset is very dense, and the length of the generated *FWIs* is long. In this case, extra time is required to construct WIT-Tree and extract *FWIs* from it. This also is reflected in memory usage as shown in Figure 6(b). We can notice the big gap in memory requirements between WIT-DIFF and wSWF because the extra memory is required to store WIT-Tree.

Finally, Connect dataset as we stated earlier is very sensitive to minWS value. This also can be noticed in its runtime and memory usage analysis shown in Figures 7(a) and 7(b). At high minWS values, such as 96% and 94%, the performances of wSWF and WIT-DIFF



(a) (b)

FIGURE 4. Runtime comparison and memory usage – Accidents dataset



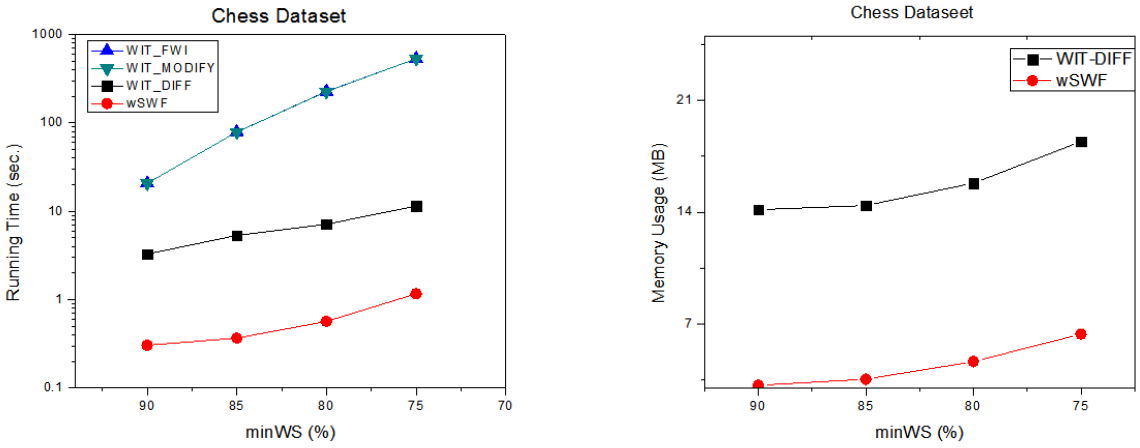
(a) (b)

FIGURE 5. Runtime comparison and memory usage – Mushroom dataset

algorithms are reasonable for the small FWIs number. When the *FWIs* increased by 92% and 90%, WIT-DIFF needs more time and memory to handle this huge number of *FWIs*. In the same minWS values, wSWF shows good performance and produces the same *FWIs* in small runtime compared to WIT-DIFF runtime. For Connect dataset, wSWF achieves an average speed-up ratio about 58% with 70% less memory than WIT-DIFF.

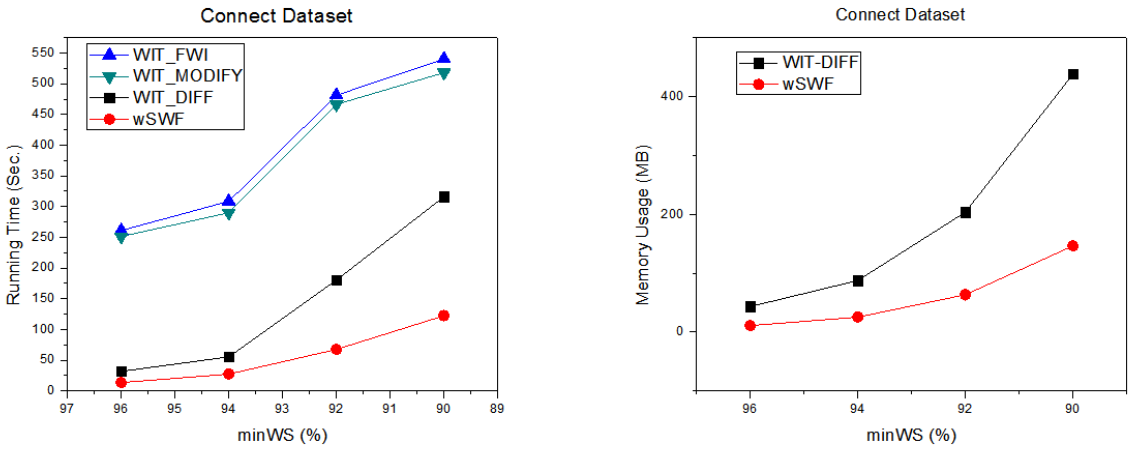
These comparisons show that wSWF algorithm outperforms other WIT-based algorithms in generating *FWIs* with different database characteristics and different minWS values. They also show that WIT-based algorithms suffer from high memory usage due to the used tree date structure. wSWF does not have this issue because there are no extra data structures used. This leads to achieving small running time for wSWF compared with other algorithms, especially for the datasets that produce large numbers of FWIs.

4.3. **Number of $C2$ candidates.** One of the main goals of wSWF algorithm is to use sliding window filtering technique to reduce the generated level-2 ($C2$) candidates. To



(a) (b)

FIGURE 6. Runtime comparison and memory usage – Chess dataset



(a) (b)

FIGURE 7. Runtime comparison and memory usage – Connect dataset

make sure that this goal is achieved, the number of $C2$ candidates is counted for both wSWF and WIT-DIFF algorithms for all datasets and plotted in Figures 8-12.

As shown in Figures 8-12, wSWF actually generated a reduced set of $C2$ candidates in all the datasets with different minWS values. The number of $C2$ candidates generated by wSWF algorithm is affected by the characteristics of the database. Sparse datasets, such as BMS-POS, wSWF reduce the number of $C2$ efficiently especially at low minWS values. For example, at 3% minWS, wSWF generates 65 candidates while WIT-DIFF generates about 950 candidates for BMS-POS dataset.

In the other datasets, wSWF also reduces the generated $C2$ candidates than WIT-DIFF but with a smaller reduction ratio than BMS-POS dataset. This is because these datasets are dense, and most of $C2$ candidates are frequent itemsets. wSWF achieves a good reduction ratio in the number of $C2$ candidates with about 26%, 52%, 24% and 18% for Accidents, Mushroom, Chess and Connect datasets respectively on average.

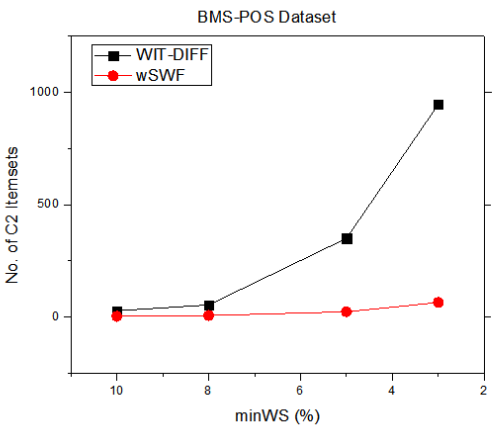


FIGURE 8. $C2$ candidates – BMS-POS dataset

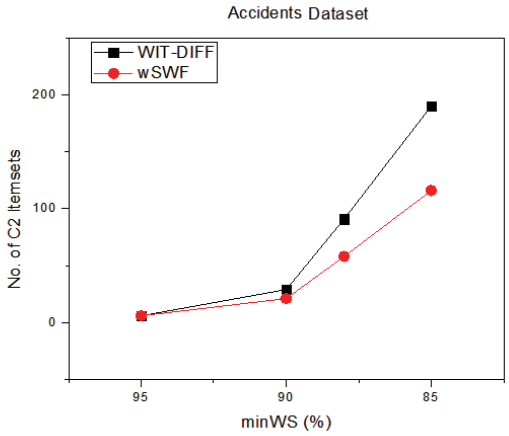


FIGURE 9. $C2$ candidates – Accidents dataset

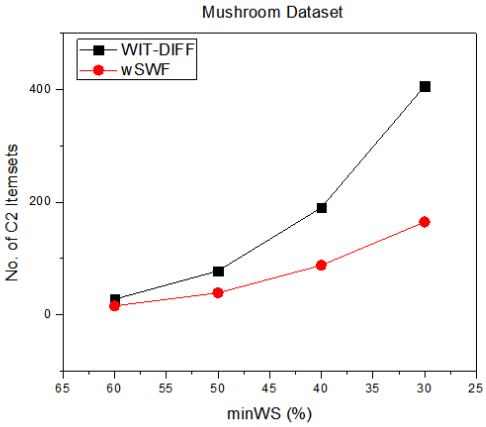


FIGURE 10. $C2$ candidates – Mushroom dataset

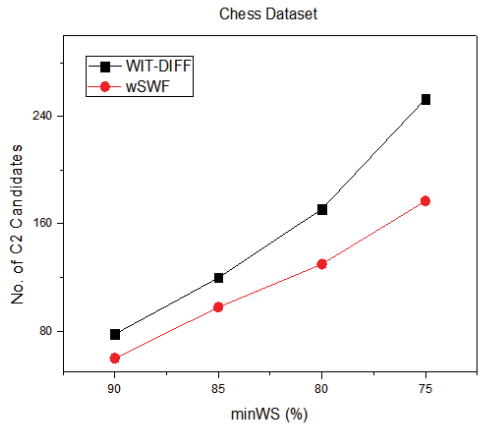


FIGURE 11. $C2$ candidates – Chess dataset

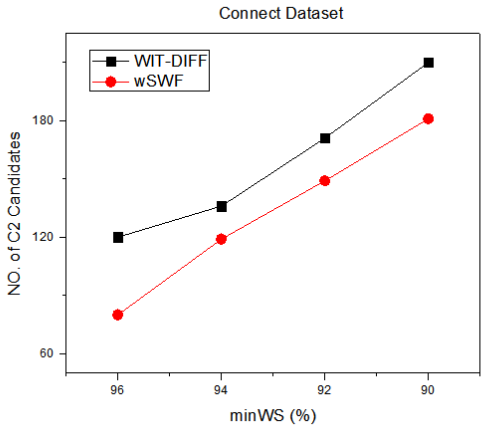


FIGURE 12. $C2$ candidates – Connect dataset

4.4. **Scalability.** In this experiment, the scalability of the proposed algorithm was examined with respect to different database sizes. The IBM synthetic data generator [37] is used to generate three synthetic datasets with 10K, 50K and 100K transactions. The runtime of wSWF algorithm is measured at two minWS values, 1% and 0.5%, as shown in Figure 13.

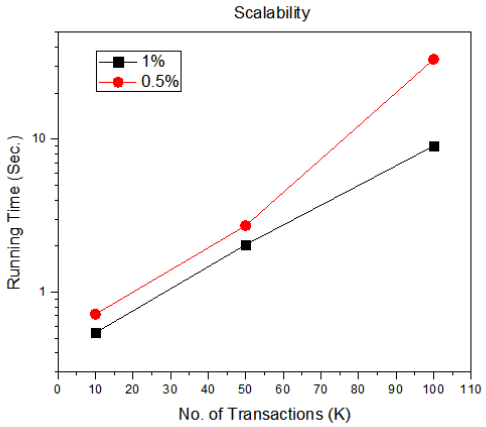


FIGURE 13. wSWF algorithm scalability

We can notice that wSWF algorithm is scalable and its running time is nearly linear with respect to database size. For example, at 1% minWS, when moving from 10K to 50K transactions, the running time increased by 4 times. In addition, the runtime for 100K database is about 4.5 times of runtime for 50K transactions. At 0.5% minWS, which is very small value, wSWF runtime increased by 3.7 times when size increased from 10K to 50K, but needs about 12 times when database size is 100K because the number of FWIs increased significantly.

4.5. **Comparison summary.** In this section, we present a brief summary for the comparison between wSWF algorithm and WIT-based algorithms [24]. As shown in Table 9, wSWF running time scalability is linear with respect to database size, while WIT-based algorithms tend to require exponential time especially with large and dense databases. The number of generated candidates is one of the main aspects of this comparison. wSWF utilizes sliding window filtering technique in order to reduce the total number of generated candidates while WIT-based algorithms produce a very large set of candidates. The memory requirements are greatly affected by the number of candidates. This leads to the fact that wSWF requires small memory space while WIT-based algorithms require large amounts of memory space to hold the large number of generated candidates. The shown values and percentages are calculated for the real dataset results at the lowest minWS value in each database.

TABLE 9. Comparison summary

	wSWF Algorithm	**WIT-based Algorithms [24]**
Scalability	Linear	Exponential
Memory Requirements	Small (102MB)	Large (334MB)
Generated Candidates	wSWF algorithm generates 42% less candidates than WIT-based algorithms on average.	

5. **Conclusions.** Weighted databases are transactional databases in which different weights are assigned to different items in the transaction. Frequent weighted itemset (FWI) mining is to maintain and discover frequent itemsets in weighted databases. Recent algorithms face computation problems due to the large number of generated candidates and the process of computing weighted support for these candidates. In this paper, we proposed a weighted sliding window filtering (wSWF) algorithm that overcomes these problems. The proposed algorithm reduced the number of generated candidates by using a sliding window filtering technique and optimized the weighted support counting process by using Diffsets strategy.

The results of the proposed wSWF algorithm have shown a significant performance improvement relative to recently cited FWI mining algorithms in three main aspects: running time, memory requirements and number of generated candidates. The main key point of this improvement is the number of generated candidate. wSWF algorithm produces the same *FWIs* that WIT-based algorithms produce but with 42% less candidates. This reduction affects directly both running time and required memory space. For running time, wSWF achieves 75% speed-up ratio on average than WIT-DIFF algorithm. Besides, the required memory space is also reduced by one-third on average; wSWF algorithm required about 102MB while WIT-based algorithms required about 334MB at the lowest minWS value in each database. Large synthetic weighted datasets were used to test the scalability of the wSWF algorithm, which is found to be linear even at small minWS thresholds.

Incremental mining was introduced to overcome the problem of updated database in which new transactions are added to the original database after the mining results [14-16]. For the future work, we shall study how to adapt wSWF algorithm to deal with the incremental mining problem in weighted databases.

REFERENCES

[1] R. Agrawal and R. Srikant, Fast algorithms for mining association rules in large databases, *Proc. of the 20th Int'l Conf. on Very Large Data Bases*, pp.487-499, 1994.

[2] J. Han, J. Pei and Y. Yin, Mining frequent patterns without candidate generation, *Proc. of ACM-SIGMOD Int'l Conf. Management of Data*, pp.1-12, 2000.

[3] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang and D. Yang, H-mine: Fast and space-preserving frequent pattern mining in large databases, *IIE Trans. Inst. of Industrial Engineers*, vol.39, no.6, pp.593-605, 2007.

[4] A. Tansel and S. Imberman, Discovery of association rules in temporal databases, *Proc. of the 4th IEEE Int'l Conf. on Information Technology*, pp.371-376, 2007.

[5] J. W. Huang, B. R. Dai and M. S. Chen, Twain: Two-end association miner with precise frequent exhibition periods, *ACM Trans. Knowledge Discovery from Data*, vol.1, no.2, 2007.

[6] M. Y. Eltabakh, M. Ouzzani, M. A. Khalil, W. G. Aref and A. K. Elmagarmid, Incremental mining for frequent patterns in evolving time series databases, *Technical Report CSD TR#08-02*, Purdue Univ., 2008.

[7] A. Erwin, R. P. Gopalan and N. R. Achuthan, Efficient mining of high utility itemsets from large data sets, *Proc. of the 12th Pacific-Asia Conf. Advances in Knowledge Discovery and Data Mining*, pp.554-561, 2008.

[8] C. F. Ahmed, S. K. Tanbeer, B.-S. Jeong and Y.-K. Lee, Efficient tree structures for high utility pattern mining in incremental databases, *IEEE Trans. Knowledge and Data Engineering*, vol.21, no.12, pp.1708-1721, 2009.

[9] V. S. Tseng, B. E. Shie, C. W. WU and P. S. Yu, Efficient algorithms for mining high utility itemsets from transactional databases, *IEEE Trans. Knowledge and Data Engineering*, vol.25, no.8, pp.1772-1785, 2013.

[10] S. K. Tanbeer, C. F. Ahmed, B.-S. Jeong and Y.-K. Lee, Efficient frequent pattern mining over data streams, *Proc. of the 17th ACM Conf. Information and Knowledge Management*, pp.1447-1448, 2008.

[11] R. Martinez, N. Pasquier and C. Pasquier, GenMiner: Mining nonredundant association rules from integrated gene expression data and annotations, *Bioinformatics*, vol.24, pp.2643-2644, 2008.
[12] C.-H. Yun and M.-S. Chen, Using pattern-join and purchase-combination for mining web transaction patterns in an electronic commerce environment, *Proc. of the 24th IEEE Ann. Int'l Computer Software and Application Conf.*, pp.99-104, 2000.
[13] C. W. Lin, T. P. Hong and W. H. Lu, The pre-FUFP algorithm for incremental mining, *Expert System with Applications*, vol.36, no.5, pp.9498-9505, 2009.
[14] T. F. Gharib, M. Taha and H. Nassar, An efficient technique for incremental updating of association rules, *International Journal Hybrid Intelligent Systems*, vol.5, no.1, pp.45-53, 2008.
[15] Z. G. Huai and M. H. Huang, A weighted frequent itemsets incremental updating algorithm base on hash table, *Proc. of the 3rd IEEE International Conf. on Communication Software and Networks*, Xi'an, China, pp.201-204, 2011.
[16] C. F. Ahmed, S. K. Tanbeer, B. S. Jeong, Y. K. Lee and H. J. Choi, Single-pass incremental and interactive mining for weighted frequent patterns, *Expert Systems with Applications*, vol.39, no.9, pp.7976-7994, 2012.
[17] G. D. Ramkumar, R. Sanjay and S. Tsur, Weighted association rules: Model and algorithm, *Proc. of the 4th ACM International Conf. on Knowledge Discovery and Data Mining*, 1998.
[18] Y. Liu, W. Liao and A. Choudhary, A fast high utility itemsets mining algorithm, *Proc. of Utility-Based Data Mining Workshop*, 2005.
[19] S. J. Yen and Y. S. Lee, Mining high utility quantitative association rules, *Proc. of the 9th Int'l Conf. Data Warehousing and Knowledge Discovery*, pp.283-292, 2007.
[20] B.-E. Shie, H.-F. Hsiao, V. S. Tseng and P. S. Yu, Mining high utility mobile sequential patterns in mobile commerce environments, *Proc. of the 16th Int'l Conf. Database Systems for Advanced Applications*, vol.6587, pp.224-238, 2011.
[21] E. Georgii, L. Richter, U. Ruckert and S. Kramer, Analyzing microarray data using quantitative association rules, *Bioinformatics*, vol.21, pp.123-129, 2005.
[22] F. Tao, F. Murtagh and M. Farid, Weighted association rule mining using weighted support and significance framework, *Proc. of the 9th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pp.661-666, 2003.
[23] B. Le, H. Nguyen and B. Vo, Efficient algorithms for mining frequent weighted itemsets from weighted items databases, *Proc. of IEEE International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future*, Hanoi, Vietnam, pp.1-6, 2010.
[24] B. Vo, F. Coenen and B. Le, A new method for mining frequent weighted itemsets based on WIT-trees, *Expert Systems with Applications*, vol.40, no.4, pp.1256-1264, 2013.
[25] B. Vo, N. Y. Tran and D. H. Ngo, Mining frequent weighted closed itemsets, *Advanced Computational Methods for Knowledge Engineering, Studies in Computational Intelligence*, vol.479, pp.379-390, 2013.
[26] M. J. Zaki, Mining non-redundant association rules, *Data Mining and Knowledge Discovery*, vol.9, no.3, pp.223-248, 2004.
[27] M. Muyeba, M. S. Khan and F. Coenen, Fuzzy weighted association rule mining with weighted support and confidence framework, *Proc. of the 1st International Workshop on Algorithms for Large-Scale Information Processing in Knowledge Discovery*, pp.52-64, 2008.
[28] U. Yun, H. Shin, K. H. Ryu and E. Yoon, An efficient mining algorithm for maximal weighted frequent patterns in transactional databases, *Knowledge Based Systems*, vol.33, pp.53-64, 2012.
[29] M. J. Zaki and C. J. Hsiao, Efficient algorithms for mining closed itemsets and their lattice structure, *IEEE Trans. Knowledge and Data Engineering*, vol.17, no.4, pp.462-478, 2005.
[30] M. J. Zaki and K. Gouda, Fast vertical mining using Diffsets, *Proc. of the 9th ACM SIGKDD International Conf. on Knowledge Discovery and Data Mining*, pp.326-335, 2003.
[31] L. Cagliero and P. Garza, Infrequent weighted itemset mining using frequent pattern growth, *IEEE Trans. Knowledge and Data Engineering*, vol.26, no.4, pp.903-915, 2014.
[32] D. J. Haglin and A. M. Manning, On minimal infrequent itemset mining, *Proc. of Int'l Conf. Data Mining*, pp.141-147, 2007.
[33] C. H. Lee, C. R. Lin and M. S. Chen, Sliding-window filtering: An efficient algorithm for incremental mining, *Proc. of the 10th International Conf. on Information and Knowledge Management*, pp.263-270, 2001.
[34] *Frequent Itemset Mining Dataset Repository, 2003*, http://fimi.ua.ac.be/data, 2014.
[35] W. Wang, J. Yang and P. Yu, WAR: Weighted association rules for item intensities, *Knowledge Information Systems*, vol.6, no.2, pp.203-229, 2004.

[36] U. Yun, Efficient mining of weighted interesting patterns with a strong weight and/or support affinity, *Information Sciences*, vol.177, no.17, pp.3477-3499, 2007.
[37] *IBM Synthetic Dataset Generator, 2001*, http://miles.cnuce.cnr.it/∼palmeri/datam/DCI/datasets.php, 2014.