Frequent Closed Itemset Mining Using Prefix Graphs with an Efficient Flow-Based Pruning Strategy

H. D. K. Moonesinghe, Samah Fodeh, Pang-Ning Tan
Department of Computer Science & Engineering
Michigan State University
East Lansing, MI 48824
(moonesin, fodehsam, ptan)@cse.msu.edu

Abstract

This paper presents PGMiner, a novel graph-based algorithm for mining frequent closed itemsets. Our approach consists of constructing a prefix graph structure and decomposing the database to variable length bit vectors, which are assigned to nodes of the graph. The main advantage of this representation is that the bit vectors at each node are relatively shorter than those produced by existing vertical mining methods. This facilitates fast frequency counting of itemsets via intersection operations. We also devise several inter-node and intra-node pruning strategies to substantially reduce the combinatorial search space. Unlike other existing approaches, we do not need to store in memory the entire set of closed itemsets that have been mined so far in order to check whether a candidate itemset is closed. This dramatically reduces the memory usage of our algorithm, especially for low support thresholds. Our experiments using synthetic and real-world data sets show that PGMiner outperforms existing mining algorithms by as much as an order of magnitude and is scalable to very large databases.

1. Introduction

Frequent closed itemset mining is the task of discovering frequent itemsets whose support counts are different than those of their supersets. Frequent closed itemsets provide a compact yet lossless representation of the frequent itemsets. Numerous algorithms have been developed to improve the efficiency of the closed itemset mining task [1, 2, 5, 7, 8, 9, 10, 12]. There are two typical strategies adopted by these algorithms: (1) an effective pruning strategy to reduce the combinatorial search space of candidate itemsets and (2) a compressed data representation to facilitate in-core processing of the itemsets. *Item merging* and *sub-itemset pruning* are two of the most commonly used strategies employed by current algorithms [10]. These strategies ensure that many of the non-closed itemsets will not be examined when searching for frequent closed itemsets, thereby reducing the runtime of the algorithms.

Apart from the pruning strategies used, having a condensed representation of the database is also vital to achieve good efficiency. Existing algorithms such as *FPclose* [2] and *CLOSET+* [10] construct a frequent pattern tree (FP-tree) structure [3] to encode the relevant itemsets and frequency information. In this representation, each transaction in the database is represented as a path from the root of the FP-tree. Compression is achieved by merging prefix paths of transactions that share the same items. A vertical database representation is another popular strategy, in which each item is associated with a column of values indicating the transactions that contain the item. For example, algorithms such as *CHARM* [12] use vertical tid-lists as their data representation while *MAFIA* [1] and *DCI-Close* [5] use vertical bit vectors.

Although an FP-tree often provides a compact representation of the data, our analysis shows that there are situations where the storage requirements of the tree may exceed even the database size, especially when the support threshold is low or when the database is very sparse. This is because each tree node encodes not only the item label, but also the support count and pointers to the parent, sibling, and child nodes. As shown in Table 1, the size of the initial FP-tree exceeds the database size for databases such as *Chess* and *Kosarak*. Algorithms that use the FP-tree structure must also recursively build smaller subtrees and traverse multiple branches of the trees to collect frequency information during the mining process. The overhead of reconstructing and traversing the FP-trees may degrade the overall performance of the algorithm [2].

Table 1 shows that the memory requirements for storing vertical bit vectors are generally less than that for FP-tree and vertical tid-lists, with the exception of the *Kosarak* data set. Vertical bit vectors also allow for fast support counting using simple bitwise AND operations.

Table 1. Characteristics of various condensed representations

Database Name	Database Size	Min Support	FP-Tree		Vertical Bit Vector Size	Vertical TID-list Size	PrefixGraph Size
			Num Nodes	Size			
Chess	474.4Kb	25%	31812	621Kb	19.9Kb	435.3Kb	239.6Kb
Pumsb	14.0Mb	45%	183349	3.5MB	377.2Kb	8.58MB	4.28MB
WebDocs	1150.4Mb	10%	50313644	959.6MB	52.8MB	296.5MB	111.6MB
Kosarak	36.8Mb	0.08%	3425391	65.3MB	189.3MB	26.1MB	9.2MB

However, when the database is large and sparse, the handling of long bit-vectors is quite inefficient since there are lots of zeros in the vectors.

Regardless of the representation, current closed itemset mining algorithms must compare the support of a candidate itemset against the support of its supersets. To perform this task more efficiently, many algorithms such as *CLOSET+* [10], *FPclose* [2], and *CHARM* [12] store their intermediate result set, which contains all the closed itemsets that have been mined so far, in another data structure (FP-Tree, hash table etc.). Such a storage method is feasible as long as the number of closed itemsets is small. When the number of closed itemsets is large, it will consume considerable memory to store and time to search the itemsets. In fact, our analysis shows that in some cases the amount of memory occupied by the result set is several orders of magnitude larger than the size of initial FP-tree or vertical database representation. For example, in the *Chess* database with 25% support threshold, storing the result set takes up to 102MB, even though the size of the initial FP-Tree is only 621Kb! The cost for searching the result set (to determine whether a candidate itemset is closed) can also be very expensive. Our analysis on the *Chess* database shows that *FPclose* spends about 70% of its overall computation time searching the result-set.

In summary, both FP-Tree and vertical representations have their own strengths and limitations. In this paper, we introduce a novel representation called *PrefixGraph,* which leverages some of the positive aspects of existing representations. From FP-tree, it borrows the idea of projecting a database onto different nodes of a graph—but without the extra cost of traversing multiple branches of the tree to collect frequency information. A *PrefixGraph* also uses bit vectors to encode the database projection at each node. However, the length of its bit vector is considerably shorter than that used by existing vertical bit vector representations. We will discuss in more details how the graph is constructed in Section 2. From Table 1, note that the size of the prefix graph structure is moderate compared to other representations. For the *Kosarak* data set, it yields the most compact representation.

Using efficient pruning strategies derived from network flow analysis, a novel algorithm called *PGMiner* is developed. Our experimental results show that the memory usage for *PGMiner* does not grow quite as rapidly as other algorithms during the mining process. Furthermore, *PGMiner* outperforms *FPclose* [2], *DCI-Close* [5] and *CHARM* [12], three state-of-the-art closed itemset mining algorithms, by an order of magnitude both in time and memory requirements.

The rest of the paper is organized as follows: Section 2 introduces the *PrefixGraph* representation. Section 3 describes the *PGMiner* algorithm. Section 4 presents the experimental results while Section 5 concludes the paper.

2. PrefixGraph Representation

In this section, we describe the *PrefixGraph* representation and show in detail its construction.

2.1 Preliminaries

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct *items*. An *itemset* X is a non-empty subset of items; i.e. $X \subseteq I$. An itemset with k items is called a *k-itemset*. Items in a given itemset are assumed to be sorted according to some total order, $\prec$. We use the notation $x \prec y$ to indicate x precedes y according to the total order. The *support* of an itemset X, denoted as $\sigma(X)$, is defined as the fraction of total transactions that contain X. An itemset is called *frequent* if its support is greater than or equal to a minimum support threshold ξ. An itemset X is *closed* if none of its proper supersets has exactly the same support count as X. Given a database D and a support threshold ξ, the problem of mining closed itemsets is to find all closed itemsets that pass the support threshold.

A *PrefixGraph* consists of a set of nodes and a set of directed edges connecting pairs of nodes. Any item in the database that satisfies the support threshold is represented as a node in the *PrefixGraph*. Each node is also associated with a projected bit vector database (see Figure 2). Before illustrating the *PrefixGraph* structure further, we give some useful definitions.

Definition 1 (Prefix 2-Item) At a node k, an item i is called its prefix 2-item if $i \prec k$ and $\{i, k\}$ is a frequent 2-itemset.

Definition 2 (Prefix Itemset) Consider an itemset $X = \{i_1, i_2, \ldots, i_{j-1}, i_j, i_{j+1}, \ldots, i_n\}$. A prefix itemset of X with respect to node i_j is defined as all the items $\{i_1, i_2, \ldots i_{j-1}\}$.

Definition 3 (Suffix Node) Let S be a set of nodes sorted based on frequency descending order. A suffix node with respect to node j is any node $k \in S$ such that $j \prec k$.

Definition 4 (Suffix Link) A directed edge between node i and its suffix node k is called a suffix link.

Definition 5 (Farthest-Node) Let S be a set of nodes sorted based on frequency descending order and T(j) be a set of suffix nodes for node j. Let W(j) $\subseteq$ T(j) be a subset of the suffix nodes such that $\forall n \in$ W(j), jn is a suffix link in the *PrefixGraph* and $\{j, n\}$ is a frequent 2-itemset. The Farthest-Node of node j is defined as the suffix node k, such that $\forall n \in$ W(j), $n \prec k$.

Example 1: Consider the *PrefixGraph* shown in Figure 2 for the sample database in Table 2. The total order of the nodes are $a \prec b \prec d \prec e \prec c$. The prefix 2-item for node b is a, while the prefix 2-items for node c are a and b. The nodes *d, e,* and c are suffix nodes of b because b precedes these nodes. The edges bd, be and bc are examples of suffix links associated with node b. Finally, c is the Farthest-Node of b.

Table 2. Sample database

Transaction ID	Items	Frequent Items
1	*a, b, c, d,*	*a, b, d, c*
2	*b, d, a, e, f, g*	*a, b, d, e*
3	*d, a, e*	*a, d, e*
4	*i, a, c, b*	*a, b, c*
5	*b, c, e*	*b, e, c*
6	*d, e, h*	*d, e*

2.2 PrefixGraph Construction

We now illustrate the construction of the *PrefixGraph* using the transaction database given in Table 2 with support threshold $\xi = 2$.

First, we scan the database to identify the set of frequent items and their corresponding support counts. These frequent items form the nodes of the *PrefixGraph*. For the sample database, the list of frequent items are <(a:4), (b:4), (c:3), (d:4), (e:4)>, where (m:n) denotes an item m with support count n.

Once the nodes are identified, we order them based on the descending order of support count as shown in Figure 1(a). Next, we scan the database again to identify the prefix 2-items for each node. For example, the frequent 2-itemsets for nodes a, b, d, e, and c are (ab, ad, ac, ae), (ba, bd, be, bc), (da, db, de), (ea, eb, ed), and (ca, cb), respectively. The prefix 2-items and their corresponding support counts for these nodes are {}, {a:3}, {a:3, b:2}, {a:2, b:2, d:3}, and {a:2, b:3} respectively. For each node, we store its set of prefix 2-items in a header table as shown in Figure 1(b).

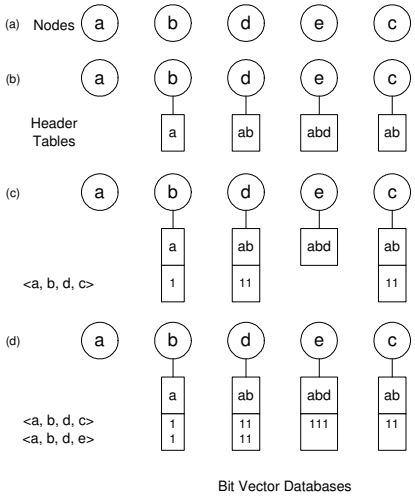

Figure 1. *PrefixGraph* construction-a running example

The next stage of the graph construction is to store the transactions as bits in the projected bit vector database of the nodes. We scan the database, and for each transaction, infrequent items are removed and the remaining items are sorted based on the frequency descending order. Let T be the resulting itemset. Now for each item k in T, we select the corresponding node k and compare its prefix 2-items against the prefix itemset of T. If there is a match then these matching items are stored as bits in the projected bit vector database of node k.

For example, consider the first transaction of the sample database. After removing the infrequent items, the remaining items are: T= {a, b, d, c}. Since the transaction has 4 frequent items we need to consider the nodes a, b, *d,* and c. Node a has no prefix 2-items and therefore nothing is stored. For node b, item a of the transaction matches with its prefix 2-item (i.e. a) and therefore bit <1> is stored in its projected bit vector DB. For node d, items {a, b} of T match with its prefix 2-items and therefore bits <11> are stored in the projected bit vector DB. Similarly for node c, the bits for items {a, b} of the transaction are stored in the projected bit vector DB. Figures 1(c) and 1(d) show the *PrefixGraph* after storing the first two transactions. When storing a transaction such as {d, e} at node e, we need to store bits <001> in node e, since only item d of the transaction matches with the prefix 2-items of node e.

In the *PrefixGraph* structure, suffix links are created based on the transactions. For each item k in the transaction T, a suffix node m is selected such that $m \in T$ and $\forall n \in$ suffix nodes of k, $m \prec n$. A suffix link is then created from node k to node m. For example, consider the transaction {a, b, d, c}. For node b we select the suffix node d (out of d and c) and add a suffix link from b to d. Figure 2 shows the complete *PrefixGraph* structure after storing all the transactions in the sample database.

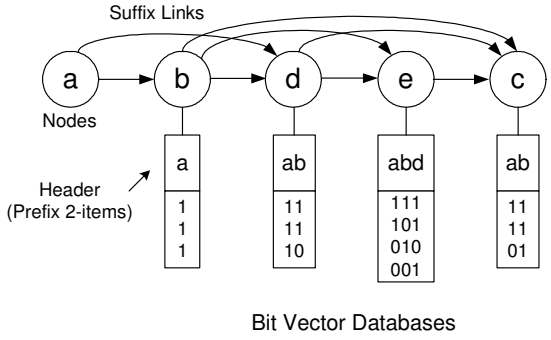

Figure 2. *PrefixGraph* representation

Instead of explicitly creating links, the links are incorporated directly into the projected bit vector database. More specifically, we can group the bit vectors of the transactions that have the same suffix link together and store them contiguously in the projected bit vector database of the node. For this purpose, the projected bit vector database of each node is partitioned into bins, and the set of bit vectors in each bin corresponds to a suffix link. For example, transactions {a, b, d, e} and {a, d, e} both have the same suffix link (*de*) at node *d*, and thus, can be stored together in a bin. If a transaction has no suffix link beyond a given node, these transactions are stored in an additional bin called the *terminating bin*. For example, the transaction {a, d, e} is stored in the *terminating bin* of node *e*. All bins must be arranged contiguously, so that the intersection of bit vectors (item wise) can be done fast as a one large chunk of words. Also, we need to keep track of the starting location of each bin in order to identify the suffix links.

A summary of the *PrefixGraph* construction procedure is given in Algorithm 1.

Algorithm 1 (*PrefixGraph* Construction)

Input: A transaction database D and support threshold ξ
Output: *PrefixGraph* structure

Method:

1. Scan the database D and find the frequent 1-itemsets (nodes) and their supports.
2. Sort the nodes in descending order of support.
3. Find the frequent 2-itemsets for each node and create the header tables.
4. For each transaction T:
   a. Sort the frequent items in T in descending order of their support.
   b. For each item k in T, select node k and match the prefix 2-items of node k with the items in T and if there is a match, store the matching items as a bit vector in the bit vector database of node k.

2.3 Analysis of *PrefixGraph* Structure

The *PrefixGraph* construction algorithm requires three scans of the database. The first two scans are necessary to find frequent 1-itemset and 2-itemset, while the third scan is needed to construct the projected bit vector databases.

Proposition 1 *The size of the projected bit vector database of a node is bounded by the support count of the node times the number of prefix 2-items of that node.*

Due to space limitations, readers may refer to [6] for the proof of this proposition.

3. Frequent Closed Itemset Mining

In this section, we will study how to efficiently mine frequent closed itemsets from the *PrefixGraph* structure. The algorithm proceeds in two phases: first, we find the frequent closed itemsets for each node (these are known as *local closed* itemsets). We then check whether the local closed itemsets are also *globally closed* using various inter-node pruning techniques. Here we give the formal definitions of local and global closed itemsets.

Definition 6 (Local Closed itemset) An itemset X, derived under node n is defined as locally closed, if there is no itemset Y ($\supset X$) derived under the same node n with $\sigma(Y) = \sigma(X)$.

Definition 7 (Global Closed itemset) An itemset X, derived under node n is defined as globally closed, if there is no itemset Y ($\supset X$) derived under any node k, ($k \in$ *set of all n*odes) with $\sigma(Y) = \sigma(X)$.

3.1 Intra-Node Closed Itemset Mining

In intra-node closed itemset mining we mine the locally closed itemsets from the projected bit vector database for each node. As shown in the next proposition, itemsets that are not locally closed are guaranteed to be non-globally closed. Such itemsets can therefore be excluded from further consideration.

Proposition 2: *For any given itemset X, derived under node n, if X is not locally closed then it is not globally closed.*

Proof. Since X is not locally closed, $\exists$ Y derived under node n, s. t. $X \subset Y$ and $\sigma(Y) = \sigma(X)$. Therefore, by the Definition 7, X cannot be globally closed. ■

In general, to generate frequent itemsets of a node, bit vectors of all distinct pairs of the itemsets are intersected and the cardinality of the resulting bit vector is checked. This is carried out recursively in a depth first manner until all the itemsets are enumerated. For example, in the itemset enumeration tree given in Figure-3, for itemset {a}, we generate all its combinations ({ab},{ac},{ad}). Then starting from {ab}, ({abc},{abd}) are generated. Similarly, itemsets {abcd}, {acd}, ({bc},{bd}), {bcd} and {cd} are enumerated in depth first manner. Note that any itemset $\{i_1, i_2, \ldots, i_k\}$ generated under a node n must have its node label appended as $\{i_1, i_2, \ldots, i_k, n\}$. We have omitted item n from the set enumeration tree in Figure 3 for brevity. Similar to several past algorithms [1, 5, 8, 12], we use two additional pruning techniques to rapidly identify the local closedness of the frequent itemset once it is generated.

Proposition 3: (sub-itemset pruning) For a *frequent itemset X and an already found closed itemset Y, if* $X \subset Y$ *and* $\sigma(X)=\sigma(Y)$*, then X and all X's descendent itemsets in the set enumeration tree are not closed.*

Proposition 4: (item merging) *For a frequent itemset X and an already found frequent itemset Y, if the tid-set(X)*$\subseteq$ *tid-set(Y) and Y*$\not\subset$ *X, then X and all X's descendent itemsets in the set enumeration tree are not closed.*

A direct implementation of sub–itemset pruning requires storing possibly a large set of closed itemsets and performing subset checking to determine whether an itemset is set-included in a superset. To reduce these overheads, we limit the applicability of this pruning strategy to itemsets between two successive levels of the depth first search space. For example, itemset {ab} at level-2 generates its level-3 itemsets ({abc}, {abd}). Based on Proposition 3, if {abc} and {ac} have identical support counts, we can prune {ac} and its sub-tree.

The applicability of the item merging proposition to an itemset X requires that we perform subset checking of X's bitmap with the bitmaps of all the processed (i.e. already mined) local itemsets of level-1 down to the parent level of itemset X. For example, if the itemset is {b, c, d} we need to check whether its bitmap is a subset of the bitmap of a. If it is a subset of bitmap(a), then {b, c, d} is not closed according to Proposition 4. In our vertical bit vector representation, such bitmap subset checking can be performed very fast.

The main advantage of local closed itemset mining is that itemset generation and support counting are very fast, since the projected database contains short bit vectors. Unlike *FPclose* and *CLOSET+*, the information needed for support counting is locally available and there is no need to traverse any other nodes.

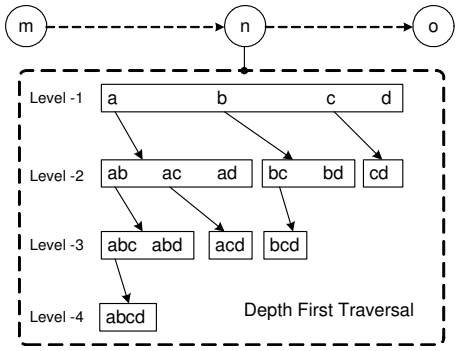

Figure 3. Itemset enumeration tree of a node

Application of both propositions ensures that we generate only the complete set of local closed itemsets for that node. In the next section, we develop an efficient flow based pruning strategy to verify whether the local closed itemsets are also globally closed.

3.2 Inter-Node Pruning

In order to develop inter-node closed itemset pruning, we consider *PrefixGraph* structure as a network with transactions flowing through the nodes. Therefore, the problem of discovering a global closed itemset can be mapped to a network flow problem.

Let us first analyze the suffix links of the *PrefixGraph* structure. For a node n in the *PrefixGraph* G, we define the out-neighborhood and in-neighborhood of n by $N^+(n) = \{m \in V(G) \mid (n, m) \in E(G)\}$ and $N^-(n) = \{m \in V(G) \mid (m, n) \in E(G)\}$, respectively (here V(G) and E(G) are the set of nodes and edges respectively).

For an edge (n, m) of the *PrefixGraph* G, $f(n,m)$ is the flow along the edge and is considered as the set of transactions that flows through the edge (n, m). Further more we have, $0 \le |f(n,m)| \le \sigma(\{nm\})$, where $|f(n,m)|$ denotes the number of transactions.

Based on this, for any node n, its outgoing flow can be defined as $OutF(n) = \bigcup_{m \in N^+(n)} f(n,m)$ and incoming flow can be defined as $InF(n) = \bigcup_{m \in N^-(n)} f(m,n)$. More specifically, we denote $f_X(n,m)$ as all the transactions containing itemset X that flow from node n to m. Then, for a given itemset X derived under node n, the outgoing

flow of X can be defined as $OutF_X(n) = \bigcup_{m \in N^+(n)} f_X(n,m)$. Similarly, the incoming flow of itemset X derived under node n can be defined as $InF_X(n) = \bigcup_{m \in N^-(n)} f_X(m,n)$. Here we give some properties of this flow based representation.

Postulate 1: *Given an itemset X derived under node n, where $|X| \geq 2$, the following properties hold:*

I. $\sigma(X) = |InF_X(n)|$

II. $InF_X(n) \supseteq OutF_X(n)$

III. $\forall m$: $OutF_X(n) \supseteq InF_{Xm}(m)$, *where* $n \prec m$ *and* $Xm = X \cup \{m\}$.

Example 2: Consider the *PrefixGraph* shown in Figure 4, for the database given in Table 2. The out-neighborhood of node *b,* $N^+(b) = \{d, e, c\}$ and the in-neighborhood of node *d,* $N^-(d) = \{b, a\}$. Transaction flow along edges (b,d) and (d,e) are $f(b,d) = \{t_1, t_2\}$ and $f(d,e) = \{t_2, t_3\}$ respectively, where t_i is the transaction ID. The flows can also be defined with respect to a given itemset. For example, $f_{\{a,b\}}(d,e) = \{t_2\}$. The incoming flow for itemset {a, d, e} at node *e,* $InF_{\{a,d,e\}}(e) = \{t_2, t_3\}$ and $|InF_{\{a,d,e\}}(e)| = 2 = \sigma(ade)$.

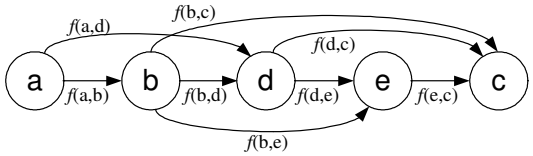

Figure 4. Transaction flow network

Under the network flow representation, a closed itemset can be defined as follows.

Theorem 1: *An itemset X derived under node n is globally closed if* $\forall m$: $InF_X(n) \neq InF_{Xm}(m)$, *where* $n \prec m$ *and* $Xm = X \cup \{m\}$.

Proof. This theorem is simply a re-statement of the definition of closed itemset that no supersets of X have the same support as X. Note that each immediate superset Xm must be generated at some node m in the *PrefixGraph* structure and $\sigma(Xm) = |InF_{Xm}(m)|$. ■

Corollary 1: The following conditions hold if X is not globally closed.

I. $\exists m$: $InF_X(n) = InF_{Xm}(m)$, where $n \prec m$.

II. $\exists m$: $InF_X(n) \subseteq InF(m)$, where $n \prec m$.

Proof. Condition I follows directly from the contra positive of Theorem 1. Condition II holds because $InF_{Xm}(m) \subseteq InF(m)$ (from the definition of incoming flow). ■

Based on this flow based representation, we develop several theorems that will assist us in identifying whether a given local closed itemset is globally closed.

Theorem 2: *For any itemset X derived under node n if,*

I. X is locally closed and

II. $\exists X'$ *s. t.* $X \subset X'$ *and* X' *is known to be a globally closed itemset under the same node n*

then X must also be globally closed.

Proof. To construct the proof, by contradiction, assume that X is not globally closed even though X' is globally closed. By Corollary 1, $\exists m$: $InF_X(n) \subseteq InF(m)$. Since $X \subset X'$, $InF_{X'}(n) \subseteq InF_X(n)$. Due to the transitive property of subset relation, $InF_{X'}(n) \subseteq InF(m)$, which contradicts the previous statement that X' is a globally closed itemset. Thus, X must be globally closed. ■

This theorem states that if we have a globally closed itemset derived under some node, then all locally closed subsets of the itemset are also globally closed (*upward closure* property). For example, in the search space given in Figure 3, suppose we found itemset {a, c, d} is globally closed; then all of the locally closed subsets of {a, c, d} derived under node n are guaranteed to be globally closed.

Efficient implementation of this theorem requires keeping all of the globally closed itemsets of a particular node in memory for subset checking. To avoid this, we devise two optimization methods: First, when checking the global closedness of local closed itemsets, we start from the maximal closed itemset (leaf) of the enumeration tree. That way, if we determine the leaf itemset as globally closed (using other techniques described later), then all the local closed itemsets in its path (to the root) become globally closed. Second, based on our analysis, we found that there is temporal locality which can be exploited during the search, i.e., most local closed itemsets are subsets of the most recently found globally closed itemset. Therefore, each time we discovered a new globally closed itemset, we keep a copy of this itemset in memory. Then, when a new local closed itemset is found we compare it against this copy.

Theorem 3: *For any itemset X derived under node n if,*

I. X is locally closed and

II. $|InF_X(n)| - |OutF_X(n)| > 0$

then X is a globally closed itemset.

Proof. To construct the proof, by contradiction, assume that X is not globally closed but $|InF_X(n)| > |OutF_X(n)|$. By Corollary 1, $\exists m$: $InF_X(n) = InF_{Xm}(m)$. From the third property of Proposition 1, $|OutF_X(n)| \geq |InF_{Xm}(m)|$. Putting them together, it follows that $|InF_X(n)| \leq$

$|OutF_X(n)|$, which contradicts our initial assumption. Thus, X must be globally closed. ■

According to this theorem if the bitmap of a local closed itemset X, derived under node n, has at least one transaction that terminates at node n (i.e. those transactions do not flow to other nodes), then X is globally closed. In our *PrefixGraph* structure, all we need to do is to examine the bits in the *terminating* bin of the corresponding itemset's bitmap. If at least one bit is '1' in the *terminating* bin of the itemset, then that itemset is globally closed. This is a very fast operation that requires checking the itemset's own bit vector to determine the global closedness.

Theorem 4: *For any itemset X derived under node n if,*

I. X is locally closed and
II. $InF_X(n) = OutF_X(n)$ and
III. X has exactly one suffix link to node m

then X is not a globally closed itemset.

Proof. Let $InF_X(n) = OutF_X(n)$. Since X has exactly one suffix link to a node m, $OutF_X(n) = InF_{Xm}(m)$. Putting them together, we obtain $InF_X(n) = InF_{Xm}(m)$, which according to Corollary 1 means that X is not globally closed. ■

Theorem 4 suggests that if all of the transactions that belong to itemset X flow to exactly one other node, then X is not closed. In the *PrefixGraph* representation, once an itemset is generated its links can be analyzed by checking the bins of the bit vector. Based on the number of links, we can decide whether the itemset is not closed.

For the remaining local closed itemsets in which the previous theorems are inapplicable, we need to test whether they are globally closed. In order to determine the global closedness of a local closed itemset, we need to visit every suffix node and compare the support of its corresponding superset, which is a very expensive operation. The following theorem reduces the number of such nodes that need to be visited.

Theorem 5: *Let X be any itemset derived under node n and let t be the Farthest-Node of n w. r .t. itemset X. Then for any itemset X' s. t.* $X' \supset X$ *derived under node m,* $n \prec m \prec t$, $\sigma(X') \neq \sigma(X)$.

The proof is given in [6]. For a given itemset, this theorem identifies the first possible node that can generate a superset itemset with identical support. So all of the nodes between the current node, where the itemset is generated, and the farthest node w.r.t. the itemset (excluding the farthest node itself) can be ignored. Using this theorem, we can identify the set of nodes that can possibly generate a superset itemset with an identical support for a given itemset X, derived under node n, as: $GEN_X(n) = \{m \in$ set of nodes | Farthest-Node$_X(n) \prec m$ and $\forall$ items j of X, $j \in$ prefix 2-items$(m)\}$.

To identify the global closedness of an itemset X, generated under node n, we visit each node in $GEN_X(n)$ until we determine its global closedness. Once we visit a node $k \in GEN_X(n)$, we can generate the itemset Xk using its bit vector database and compare the support count with X. There are several optimization strategies that can be employed here. We found temporal locality property that can be exploited during the subsequent generation of itemsets in a node. In order to facilitate fast subsequent itemset generation, we keep the bit vectors of the most common subsets of the itemset, once they have been generated under a node for reuse. Due to space limitation more details are given in [6].

This itemset regeneration based closedness check is efficient because of the following reasons: first our bit vectors are shorter in length, so that intersection is fast. Second, we keep track of the bit vectors of most common itemsets in memory, which avoids complete regeneration. Third and more importantly, after applying Theorems 2-4, the remaining percentage of itemsets that needs global closedness is much smaller. We have analyzed this in Section 4.

3.3 Mining Algorithm

Based on the above discussion, we have the following algorithm for frequent closed itemset mining.

Algorithm 2 (*PGMiner*)

Input: *PrefixGraph* structure G and support threshold ξ
Output: The complete set of frequent closed itemsets

Method:

1. Starting from the node with highest support count call MineNode(n) for each node $n \in$ V(G).

Procedure MineNode (n)

1. With the depth first search paradigm, mine the local closed itemsets at node n in a top down manner by intersecting its bit vectors. Use Propositions 3 and 4 to prune non-closed local itemsets.
2. Once at a leaf itemset X of the search path, use Theorems 2, 3 and 4 to detect global closedness for the local closed itemset found. If not detected, search the nodes in $GEN_X(n)$ (Theorem 5) using the regeneration method.
3. If an itemset is globally closed, mark all the local closed itemsets in the search path to the root as globally closed (by Theorem 2). Output any global closed itemset found.
4. Stop when all prefix 2-items in the node have been processed, and reclaim memory of the bit vector DB of that node.

4. Experimental Evaluation

4.1 Evaluating Environment

We compared the performance of *PGMiner* against three state of the art algorithms: *FPclose* [2], *DCI-Close* [5], and *CHARM* [12], which uses the *DiffSet* [11]. We experimented with variety of databases as shown in Table 3. Most of the real-world databases were obtained from the *FIMI* repository[†]. Synthetic databases were generated using the IBM data generator [4]. Our machine consists of a 2.8 GHz Intel *Pentium* 4 processor with 1 GB of memory running *Linux*. All recorded execution times refer to real time (that includes CPU and I/O time).

Table 3. Characteristics of the databases

Dataset	No. of Transactions	No. of Items
Medical	5,939,734	5,912
WebView2	77,513	3,340
Chess	3,196	75
WebDocs	1,692,082	5,267,656
Pumsb	49,046	2,113
Kosarak	990,002	41,270
T40I10D100K	100,000	1,000
T100I20D100K	100,000	997
T20I8D500K	500,000	8,612
T50I10D*x*K	25,000-50,000,000	25,000

4.2 Experimental Results

4.2.1 ***Performance Comparisons***. Execution time comparison of *PGMiner* against other algorithms is shown in Figure 5. When an algorithm took considerably longer time compared to the rest, it was eventually terminated. Our analysis shows that in seven out of nine databases tested, *PGMiner* shows the best runtime when compared to all other algorithms at low support thresholds. For the remaining two databases (*Chess*, and *Pumsb*), although *PGMiner* outperforms both *FPclose* and *CHARM*, *DCI-Close* shows better runtime. This is because their search space enumeration method seems better suited for these smaller databases. We found that in some cases all other algorithms fail to mine databases at low support thresholds, while *PGMiner* can still run for even smaller levels of support thresholds.

In summary, *PGMiner* shows better run time performance because it has very low overhead due to the effectiveness of its flow based pruning strategies. Unlike other algorithms, *PGMiner* does not need to store the entire result set in memory. The *PrefixGraph* structure also has shorter bit vectors and this significantly reduces the bit vector intersection cost for large databases. Thus, *PGMiner* has better runtime and can scale to very lower levels of support thresholds.

4.2.2 Memory Usage. The memory usage for all the algorithms on several databases is shown in Figure 6. We found that *PGMiner* mines all of the databases with low memory usage when compared with the other algorithms. In all these cases, *FPclose* shows higher memory consumption because of its storage based pruning strategy and the large FP-Tree structure it has to build for larger databases. However, *DCI-Close* shows better memory usage when compared with *FPclose* and *CHARM*. But in some cases (e.g. *T40I10D100K*), its memory consumption gets suddenly high when the threshold is gradually lowered. Note that the memory usage of *PGMiner* does not grow quite as rapidly as other algorithms during the mining process.

4.2.3 Scalability. We have also measured the execution time of all the algorithms by increasing the number of transactions gradually. We use the *T50I10DxK* data set, where *x* is varied from 25,000 transactions (DB size 6.9MB) to 50 million transactions (DB size 13.9GB), with minimum support threshold 0.1%. For this experiment we used a server (2 GHz) with 4 GB of memory, since these databases are of gigabyte size. The experimental results (see Figure 7) revealed that *CHARM*, *FPclose*, and *DCI-Close* could not reach more than 1 million transactions (*1000K*) of this database set. *FPclose* and *DCI-Close* crashed for the *5000K* dataset. Analysis of memory usage for these algorithms revealed that they consume high memory space. See Figure 8. In the 5000K dataset, the *FPclose* algorithm fails because it has consumed all the available memory. Memory usage of *DCI-Close* is high even for the *1000K* dataset. Note that *PGMiner* was able to reach 50 million transactions easily showing remarkably low memory usage.

4.2.4 Effectiveness of the Flow Based Pruning. In our algorithm, when a local closed itemset is discovered, we first apply Theorem 2 and then if it cannot discover the closedness of the itemset, we apply Theorem 3. In Table 4 we have shown the percentage of itemsets discovered by both these theorems.

Table 4. Evaluation of global closedness check

Data Set (min. sup.)	Theorem 2	Theorem 3
Chess (30%)	91.6%	8.0%
WebDocs (10%)	94.1%	85.4%
Pumsb (45%)	91.9%	30.5%
Kosarak (0.08%)	65.5%	68.1%
T20I8D500K (0.01%)	91.7%	26.4%
T40I10D100K (0.1%)	67.6%	40.3%

[†] *http://fimi.cs.helsinki.fi*

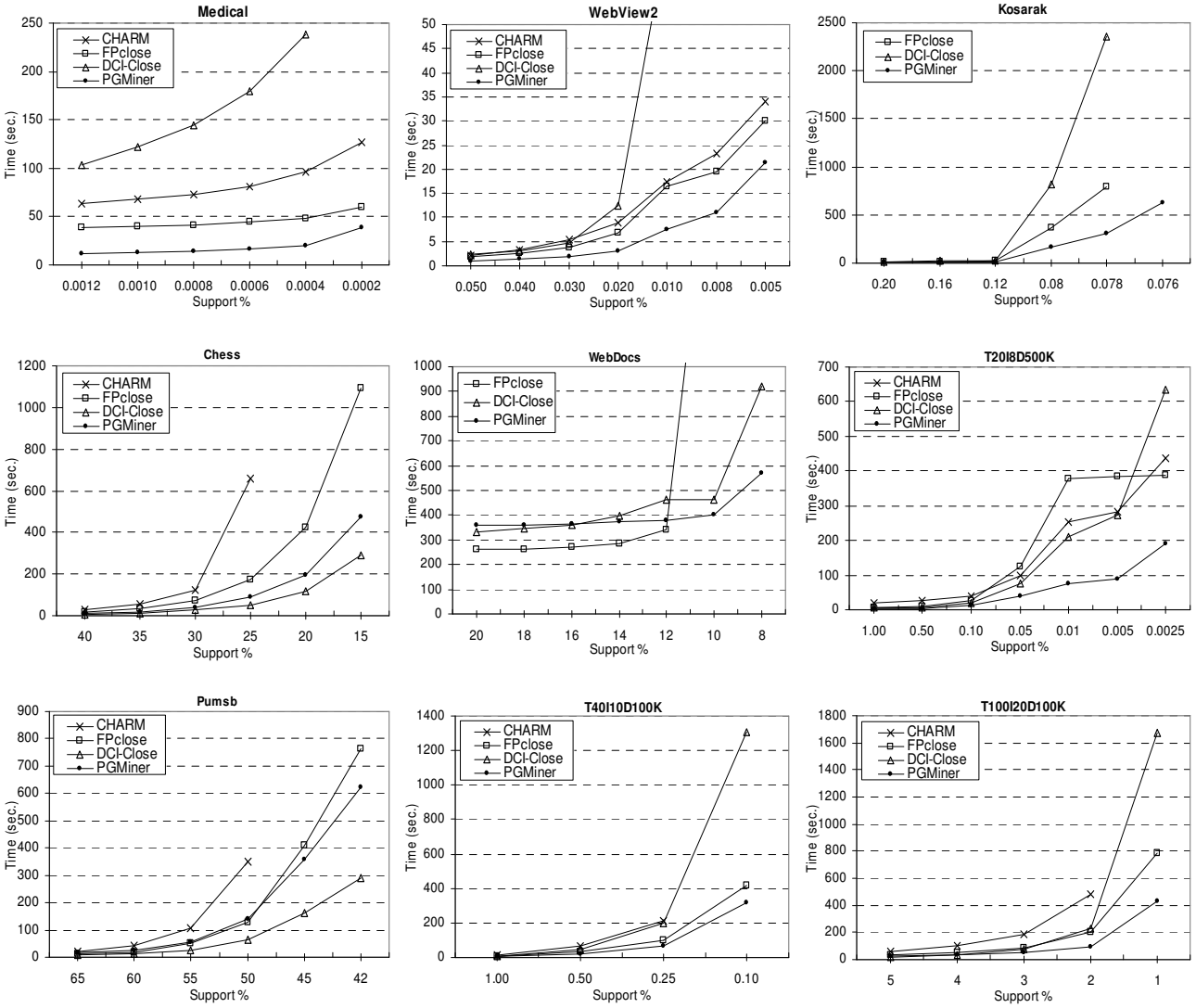

Figure 5. Execution time (in seconds) for CHARM, FPclose, DCI-Close and PGMiner

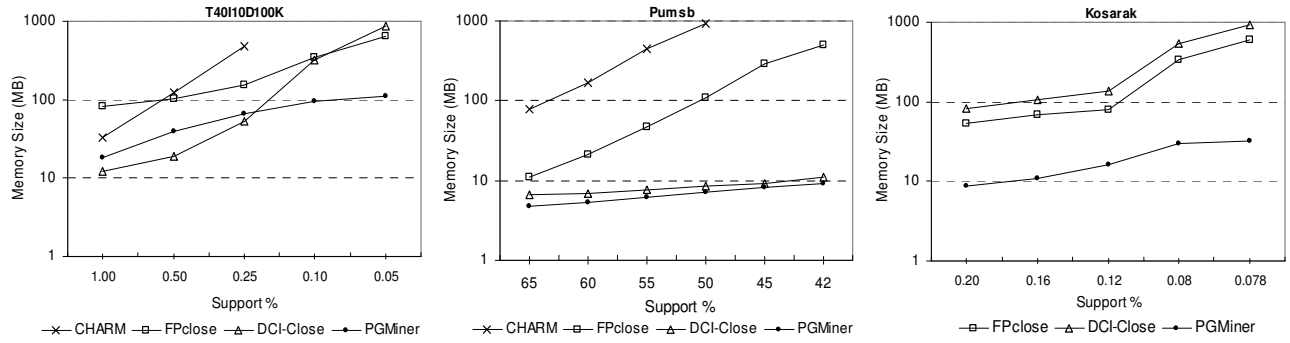

Figure 6. Amount of memory (in MB) required at various support levels

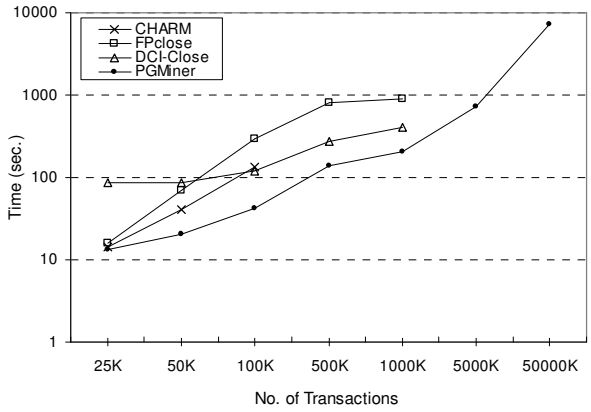

Figure 7. Execution time versus number of transactions (*K=1000*)

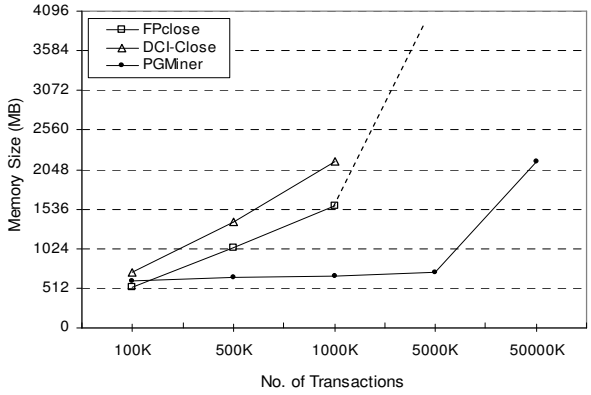

Figure 8. Memory usage of algorithms for large databases

For example, In *WebDocs* dataset we were able to discover 94.1% of the total local closed itemsets as either globally closed or not by using Theorem 2. From the remaining percentage (i.e. 5.9%), 85.4% of itemsets were discovered by Theorem 3. Table 4 clearly shows that both Theorem 2 and Theorem 3 are capable of detecting global closedness of many local closed itemsets of the database. Moreover, these two techniques can be easily implemented and it is one of the key factors to achieve faster performance in our algorithm.

5. Conclusions

This paper introduces a *PrefixGraph* representation for mining frequent closed itemsets. The key advantage of our representation is that it leverages the positive aspects from both FP-tree and vertical bit vector representations. The size of the *PrefixGraph* structure is quite moderate and its memory requirements do not grow as rapidly as other algorithms. Our proposed algorithm called *PGMiner* employs several effective itemset pruning strategies derived from network flow analysis. These strategies can be adapted to other existing algorithms (such as *CLOSET* [8]) that use projected databases to prune their non-closed itemsets.

For future work, we plan to extend our current work to mine even larger databases measured in billions of transactions using the secondary memory.

Acknowledgements

We would like to thank Dr. Mohammed J. Zaki for providing the source code of the *CHARM* algorithm. Thanks also go to the authors of the *FPclose* and the *DCI-Close* algorithms for sharing their source codes with us. We are grateful to Michael Zaroukian and Henry Barry from the college of Human Medicine at MSU for providing us anonymized medical database.

References

[1] D. Burdick, M. Calimlim, and J. Gehrke. MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases. In *Proc. of ICDE*, 2001.

[2] G. Grahne, J. Zhu. Efficiently Using Prefix-trees in Mining Frequent Itemsets. In *Proc. of FIMI'03*, 2003.

[3] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. In *proc. of ACM SIGMOD*, 2000.

[4] IBM Almaden. *Synthetic Data Generation Code for Associations and Sequential Patterns*. http://www.almaden.ibm.com/cs/quest/syndata.html.

[5] C. Lucchese, S. Orlando, and R. Perego. Fast and Memory Efficient Mining of Frequent Closed Itemsets. *TKDE*, 2006.

[6] H. D. K. Moonesinghe, S. Fodeh and P.-N. Tan. Frequent Closed Itemset Mining Using Prefix Graphs. *Michigan State University*, *Technical Report*, 2006.

[7] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT'99*, 1999.

[8] J. Pei, J. Han, and R. Mao. CLOSET: An Efficient Algorithm for Mining Frequent Closed Itemsets. In *Proc. of DMKD'00*, 2000.

[9] N. G. Singh, S. R. Singh, and A. K. Mahanta. CloseMiner: Discovering Frequent Closed Itemsets Using Frequent Closed Tidsets. In *Proc. of ICDM*, 2005.

[10] J. Wang, J. Han, and J. Pei. CLOSET+: Searching for the Best Strategies for Mining Frequent Closed Itemsets. In *Proc of ACM SIGKDD*, 2003.

[11] M. J. Zaki, K. Gouda. Fast vertical mining using diffsets. In *Proc. of ACM SIGKDD,* 2003.

[12] M. J. Zaki, C. J. Hsiao. CHARM: An Efficient Algorithm for Closed Itemset Mining. In *Proc of SDM'02,* 2002.