GEOMETRICALLY INSPIRED ITEMSET MINING
TECHNICAL REPORT 588

FLORIAN VERHEIN AND SANJAY CHAWLA

JULY, 2006

Geometrically Inspired Itemset Mining

Florian Verhein, Sanjay Chawla
School of Information Technologies, the University of Sydney, Australia
{fverhein,chawla}@it.usyd.edu.au

Abstract

In our geometric view, an itemset is a vector (itemvector) in the space of transactions. The support of an itemset is the generalized dot product of the participating items. Linear and potentially non-linear transformations can be applied to the itemvectors before mining patterns. *Aggregation functions can be applied to the transformed vectors and pushed inside the mining process. We show that* **interesting** *itemset mining can be carried out by instantiating four abstract functions: a transformation (g), an algebraic aggregation operator ($\circ$) and measures (f and F). For* **frequent** *itemset mining (FIM), g and F are identity transformations, $\circ$ is intersection and f is the cardinality. Based on the geometric view we present a novel algorithm that uses space linear in the number of 1-itemsets to mine all interesting itemsets in a single pass over this data, with no candidate generation. It scales (roughly) linearly in running time with the number of interesting itemsets. FIM experiments show that it outperforms FP-Growth on realistic datasets above a small support threshold (0.29% and 1.2% in our experiments).*

1 Introduction

Traditional Association Rule Mining (ARM) considers a set of transactions T containing items I. Each transaction $t \in T$ is a subset of the items, $t \subset I$. The most time-consuming task of ARM is Frequent Itemset Mining (FIM), whereby all itemsets $i \subset I$ that occur in a sufficient number of transactions are generated. Specifically, if $\sigma(i) \geq minSup$, where $\sigma(i) = |\{t : i \subset t\}|$ is the number of transactions containing i (known as the *support* of i).

For *item enumeration* type algorithms, each transaction has generally been recorded as a row in the dataset. These algorithms make two or more passes, reading it one transaction at a time.

We consider the same data in its transposed format: Each row, x_i, (corresponding to an item i) contains the set of transaction identifiers (tids) of the transactions containing i. Specifically, $x_i = \{t.tid : t \in T \wedge i \in t\}$. We call x_i an *itemvector* because it represents an item in the space spanned by the transactions[1]. An example is provided in Figure 1(a).

Just as we can represent an item as an itemvector, we can represent an itemset $I' \subset I$ as an itemvector: $x_{I'} = \{t.tid : t \in T \wedge I' \subset t\}$. Figure 1(b) shows all itemsets that have support greater than one, represented as vectors in transaction space. For example, consider $x_4 = \{t_2, t_3\}$ located at g and $x_2 = \{t_1, t_2, t_3\}$ located at f. $x_{\{2,4\}}$ can be obtained using $x_{\{2,4\}} = x_2 \cap x_4 = \{t_2, t_3\}$, and so is located at g. It should be clear that $\sigma(x_{I'}) = |x_{I'}| = | \cap_{i \in I'} x_i|$.

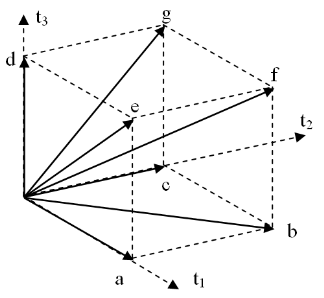
There are a three important things to note from the above: (1) We can represent an item by a vector (we used a set representation to encode its location in transaction space, but we could have equally well used an alternate representation in some other space). (2) We can create itemvectors that represent itemsets by performing a simple operation on the itemvectors (in the case above, set intersection). (3) We can evaluate a measure using a simple function on the itemvector (in the above case, we used set size and the support measure). These fundamental operations are all that are required for a mining algorithm. In Section 3, we generalise them to functions $g(\cdot)$, $f(\cdot)$, operator $\circ$ and an additional function $F(\cdot)$ for more complicated measures.

So far we have considered itemvectors as binary. There is no reason for this restriction. Provided that we have functions $f(\cdot)$ and $F(\cdot)$ and an operator $\circ$ that obey the requirements set out in Section 3, we can map the original itemvector into some other

[1]For simplicity we will use transactions and their tids interchangeably

Traditional	Transposed
$t_1 : 1, 2, 5$	$1 : t_1, t_2$
$t_2 : 1, 2, 3, 4$	$2 : t_1, t_2, t_3$
$t_3 : 2, 4, 5$	$3 : t_2$
	$4 : t_2, t_3$
	$5 : t_1, t_3$

(a) Transposing a dataset of three transactions ($tid \in \{t_1, t_2, t_3\}$) containing items $I = \{1, 2, 3, 4, 5\}$.



label	corresponding itemsets
a	{1,5}
b	{1},{1,2}
c	{3}, {1,3}, {1,4}, {2,3}, {3,4}, {1,2,3}, {1,2,4}, {1,3,4}, {2,3,4}, {1,2,3,4}
d	{4,5}
e	{5},{2,5}
f	{2}
g	{4},{2,4}

(b) Itemsets with support greater than 1 in transaction space.

Figure 1. Running itemvector example

space via $g(\cdot)$. This has potential for dimensionality and noise reduction or sketch algorithms. Suppose it is possible to perform Singular Value Decomposition (SVD) or Random Projections [1] to reduce the noise and dimensionality (number of transactions) before mining itemsets. This has previously been impossible since the transformation creates real valued vectors, and hence cannot be mined using exiting algorithms. Using our framework, all that is required are suitable $\circ$, $f(\cdot)$ and $F(\cdot)$. We can also use measures other than support. Any anti-monotonic (or prefix or weakly anti-monotonic) measure that fits into our framework can be used.

We briefly illustrate some of the ideas used in our **algorithm** using Figure 1(a). We simply wish to convey the importance of the transpose view to our technique, and introduce some of the challenges we solved. Too keep things simple, we use the instantiation of $g(\cdot)$, $\circ$, $f(\cdot)$ and $F(\cdot)$ required for traditional FIM. Our algorithm scans the transposed dataset row by row. Suppose we scan it bottom up[2] so we first read $x_5 = \{t_1, t_3\}$. Assume $minSup = 1$. We can immediately say that $\sigma(\{5\}) = 2 \geq minSup$ and so itemset $\{5\}$ is frequent. We then read the next row, $x_4 = \{t_2, t_3\}$, and find that $\{4\}$ is frequent. Since we now have both x_5 and x_4, we can create $x_{\{4,5\}} = x_4 \cap x_5 = \{t_3\}$. We have now checked all possible itemsets containing items 4 and 5. To progress, we read $x_3 = \{t_2\}$ and find that $\{3\}$ is frequent. We can also check more itemsets: $x_{\{3,5\}} = x_3 \cap x_5 = \emptyset$ and $x_{\{3,4\}} = x_3 \cap x_4 = \{t_2\}$ so $\{3, 4\}$ is frequent. Since $\{3, 5\}$ is not frequent, neither is $\{3, 4, 5\}$ by the anti-monotonic property of support [2]. We next read x_2 and continue the process. It should be clear from the above that (1) a single pass over the dataset is sufficient to mine all frequent itemsets, (2) having processed any n itemvectors corresponding to items in $J = \{1, ..., n\}$, we can generate all itemsets $L \subset J$ and (3) having the dataset in transpose format and using the itemvector concept allows this to work.

Each itemvector could take up significant space, we may need many of them, and operations on them could be expensive. We generate at least as many itemvectors as there are frequent itemsets[3]. Since the number of itemsets is at worst $2^{|I|} - 1$, clearly it is not feasible to keep all these in memory, nor do we need to. On the other hand, we do not want to recompute them as this is expensive. If there are n items we could use n itemvectors of space and create all itemsets, but we must recompute most itemvectors multiple times, so the time is not linear in the number of frequent itemsets - it will be exponential. For example, suppose we have created $x_{\{1,2,3\}}$. When we later need $x_{\{1,2,3,4\}}$, we do not want to have to recalculate it as $x_1 \cap x_2 \cap x_3 \cap x_4$. Instead, we would like to use the previously calculated $x_{\{1,2,3\}}$: $x_{\{1,2,3,4\}} = x_{\{1,2,3\}} \cap x_4$ being one option. The challenge is to use as little space as necessary, while avoiding re-computations.

We present an algorithm that uses time roughly linear in the number of interesting itemsets and at worst $n' + \lceil l/2 \rceil)$ itemvectors of space, where $n' \leq n$ is the number of interesting 1-itemsets and l is the size of the largest interesting itemset. This worst case scenario is only reached with extremely low support, and most practical situations require only a small fraction of n'. Based on these facts and the geometric inspiration provided by the itemvectors, we call it *Geometrically inspired Linear Itemset Mining In the Transpose*, or *GLIMIT*.

It is widely recognised that FP-Growth type algo-

[2]This is just so the ordering of items in the data structure of our algorithm is increasing.

[3]It is 'at least' because some itemsets are not frequent, but we only know this once we have calculated its itemvector.

rithms are the fastest know algorithms. We show experimentally that GLIMIT outperforms FP-Growth [5] when the support is above a small threshold.

GLIMIT is more than "just another ARM algorithm" because it departs significantly from previous algorithms. It is a new, and fast, class of algorithm. Furthermore, it opens possibilities for useful preprocessing techniques based on our itemvector framework, as well as new geometrically inspired interestingness measures.

We make the following **contributions**:

- We show interesting consequences of viewing transaction data as itemvectors in transaction-space. We develop a theoretical framework for operating on itemvectors. This abstraction allows a new class of algorithm to be developed, great flexibility in the measures used and opens up the potential for useful transformations on the data that were previously impossible.
- We present GLIMIT, a new class of algorithm that uses our framework to mine itemsets in one pass without candidate generation. It uses linear space and time linear in the number of interesting itemsets. It significantly departs from existing algorithms. Experiments show it beats FP-Growth above small support thresholds.

In Section 2 we put our framework and GLIMIT in context of previous work. Section 3 presents our itemvector framework. Section 4 gives the the two data structures that can be used by GLIMIT. In Section 5 we first give the main facts exploited by GLIMIT and follow up with a comprehensive example. We prove the space complexity and give the pseudo-code. Section 6 contains our experiments. We conclude in Section 7.

2 Previous Work

Many itemset mining algorithms have been proposed since association rules were introduced [2]. Recent advances can be found in [3] and [4]. Most algorithms can be broadly classified into two groups, the *item enumeration* (such as [2, 5, 9]) and the *row enumeration* (such as [7, 12]) techniques. Broadly speaking, item enumeration algorithms are most effective for datasets where $|T| >> |I|$, while row enumeration algorithms are effective for datasets where $|T| << |I|$, such as for microarray data [7].

Item enumeration algorithms mine subsets of an itemset I' before mining I'. Only those itemsets for which all subsets are frequent are generated - making use of the anti-monotonic property of support. Apriori-like algorithms [2] do this in a breadth first manner and use a candidate generation step. They use multiple passes, at most equal to the length of the longest frequent itemset. Our algorithm does *not* perform candidate generation, and generates association rules in a *depth first* fashion using a *single* pass over the *transposed* dataset.

FP-Growth type algorithms [5, 9] generates a compressed summary of the dataset using two passes in a highly cross referenced tree, the FP-tree, before mining itemsets by traversing the tree. Like our algorithm it does not perform candidate generation and mines the itemsets in a depth first manner while still mining all subsets of an itemset I' before mining I'. It is very fast at reading from the FP-tree, but the downside is that the FP-tree can become very large and is expensive to generate, so this investment does not always pay off. Our algorithm uses only as much space as is required.

Row enumeration techniques effectively intersect transactions and generate supersets of I' before mining I'. Although it is much more difficult for these algorithms to make use of the anti-monotonic property for pruning, they exploit the fact that searching the row space in data with $|T| << |I|$ becomes cheaper than searching the itemset-space. GLIMIT is similar to row enumeration algorithms since both search using the transpose of the dataset. However, where row enumeration intersects transactions (rows), we effectively intersect itemvectors (columns). But this similarity is tenuous at best. Furthermore, existing algorithms use the transpose for counting convenience rather than for any insight into the data, as we do in our itemvector framework. Since GLIMIT searches through the itemset space, it is classified as an item enumeration technique and is suited to the same types of data. However, it scans the original data column-wise (by scanning the transpose row-wise), while all other item enumeration techniques scan it row-wise. The transpose has *never*, to our best knowledge, been used in an item enumeration algorithm. In summary, we think it is about as similar to other item enumeration techniques as FP-Growth is to Apriori.

Efforts to create a framework for support exist. Steinbach et al. [11] present one such generalisation, but their goal is to extend support to cover continuous data. This is very different to transforming the original (non-continuous) data into a real vector-space (which is one of our motivations). Their work is geared toward existing item enumeration algorithms and so their *"pattern evaluation vector"* summarises transactions (that is, *rows*). Our framework operates on *columns* of the original data matrix. Furthermore, rather than generalising the support measures so as to cover more types of datasets, we generalise the operations on itemvector and the transformations on the *same* dataset that can be used to enable a wide range of measures, *not* just support.

To our best knowledge, Ratio Rules are the closest attempt at combining SVD (or similar techniques such as Principal Component Analysis) and rule mining. Korn et al. [6] consider transaction data where items have continuous values associated with them, such as price. A transaction is considered a point in the space spanned by the items. By performing SVD on such datasets, they observe that the axes (orthogonal basis vectors) produced define ratios between single items. We consider items (*and itemsets*) in *transaction space* (not the other way around) so when we talk of performing SVD, the new axes are linear combinations of transactions - not items. Hence I is unchanged. Secondly, we talk about mining itemsets, not just ratios between single items. Finally, SVD is just one possible instantiation of $g(\cdot)$.

By considering items as vectors in transaction space, we can interpret itemsets geometrically, which we do not believe has been considered previously. As well as inspiring our algorithm, this geometric view has the potential to lead to very useful preprocessing techniques, such as dimensionality reduction of the transactions space. Since GLIMIT uses only this framework, it will enable us to use such techniques - which are impossible using existing FIM algorithms.

3 Theoretical Itemvector Framework

In Section 1 we considered itemvectors as sets of transactions. We used the intersection operator to create itemvectors corresponding to itemsets. We then used the cardinality function on itemvectors to evaluate the support of the itemsets. That is, we used three main functions. These functions can clearly be abstracted, and we do so with $g(\cdot)$, $\circ$ and $f(\cdot)$ respectively. We also include an additional function $F(\cdot)$ for more complicated measures. But why do this? The short answer is that by viewing each itemset as a vector in transaction space, we can think of a few different instances of these functions that have merit. Before going into this we will formally define our functions. First, recall that we use $x_{I'}$ to denote the itemvector for itemset $I' \subset I$, and for simplicity we abbreviate single itemvectors $x_{\{i\}}$ to x_i. Recall also that $x_{I'}$ is the set of transactions that contain the itemset $I' \subset I$. Call X the space spanned by all possible $x_{I'}$.

Definition 1 *$g : X \rightarrow Y$ is a transformation on the original itemvector to a different representation $y_{I'} = g(x_{I'})$ in a new space Y.*

Even though $g(\cdot)$ is a transformation, it's output still 'represents' the itemvector. To avoid too many terms, we also refer to $y_{I'}$ as an itemvector.

Definition 2 *$\circ$ is an operator on the transformed itemvectors so that $y_{I' \cup I''} = y_{I'} \circ y_{I''} = y_{I''} \circ y_{I'}$.*

That is, $\circ$ is a commutative operator for combining itemvectors to create itemvectors representing larger itemsets. We do *not* require that $y_{I'} = y_{I'} \circ y_{I'}$ [4].

Definition 3 *$f : Y \rightarrow \mathbb{R}$ is a measure on itemsets, evaluated on transformed itemvectors. We write $m_{I'} = f(y_{I'})$.*

Suppose we have a measure if interestingness of an itemset that depends only on that itemset. We can represent this as follows, where $I' = \{i_1, ..., i_q\}$:

$$interestingness(I') = f(g(x_{i_1}) \circ g(x_{i_2}) \circ ... \circ g(x_{i_q})) \quad (1)$$

So the challenge is, given an $interestingness$ measure, find suitable and useful g,$\circ$ and f so that the above holds. For *support*, we know we can use $\circ = \cap$, $f = |\cdot|$ and g as the identity function.

We now return to our motivation. First assume that $g(\cdot)$ trivially maps $x_{I'}$ to a binary vector. Using $x_1 = \{t_1, t_2\}$ and $x_5 = \{t_1, t_3\}$ from Figure 1(a) we have $y_1 = g(x_1) = 110$ and $y_5 = g(x_5) = 101$. It should be clear that we can use bitwise AND as $\circ$ and $f = sum()$, the number of set bits. But notice that $sum(y_1 \, AND \, y_2) = sum(y_1 .* y_2) = y_1 \cdot y_2$, the dot product ($.*$ is the element-wise product[5]). That is, the dot product of two itemvectors is the support of the the 2-itemset. What makes this interesting is that this holds for any rotation about the origin. Suppose we have an arbitrary 3×3 matrix R defining a rotation about the origin (so R is orthogonal). This means we can define $g(x) = Rx^T$ because the dot product is preserved by R (hence $g(\cdot)$). For example, $\sigma(\{1,5\}) = y_1 \cdot y_5 = (Rx_1^T) \cdot (Rx_5^T)$. So we can perform an arbitrary rotation of our itemvectors before mining 2-itemsets. Of course this is much more expensive than bitwise AND, so why would we want to do this? Consider Singular Value Decomposition. If normalisation is skipped, it becomes a

[4] Equivalently, $\circ$ *may* have the restriction that $I' \cap I'' = \emptyset$.

[5] $(a. * b)[i] = a[i] * b[i]$ for all i, where [] indexes the vectors.

rotation about the origin, projecting the original data onto a new set of basis vectors pointing in the direction of greatest variance (incidentally, the covariance matrix calculated in SVD also defines the support of all 2-itemsets[6]). If we additionally use it for dimensionality reduction, it has the property that it roughly preserves the dot product. This means we should be able to use SVD for dimensionality reduction and or noise reduction prior to mining frequent 2-itemsets without introducing too much error. The drawback is that the dot product applies only to two vectors. That is, we cannot use it for larger itemsets because the *'generalised dot product'* satisfies $sum(Rx_1^T .* Rx_2^T .* .... * Rx_q^T) = sum(x_1 .* x_2 .* ... .* x_q)$ only for $q = 2$. However, this does not mean that there are not other useful $\circ$, $f(\cdot)$, $F(\cdot)$ and interestingness measures that satisfy Equation 1 and use $g(\cdot) = SVD$, some that perhaps will be motivated by this observation. Note that the transpose operation is crucial in applying dimensionality or noise reduction because it keeps the items intact. If we did not transpose the data, the itemspace would be reduced, and the results would be in terms of *linear combinations* of the original items, which cannot be interpreted. It also makes more sense to reduce noise in the transactions.

We could also choose $g(\cdot)$ as a set compression function or use approximate techniques, such as sketches, to give estimates rather than exact values of support or other measures. However, we think new measures inspired by the our view of itemvectors in transaction space will be the most promising. For example, angles between itemvectors are linked to the correlation between itemsets. Of course, we can also translate existing measures into our framework.

To complete our framework we now define $F(\cdot)$ and give an example.

Definition 4 *$F : \mathbb{R}^k \rightarrow \mathbb{R}$ is a measure on an itemset I' that supports any composition of measures (provided by $f(\cdot)$) on any number of subsets of I'. That is, $M_{I'} = F(m_{I'_1}, m_{I'_2}, ..., m_{I'_k})$ where $m_{I'_i} = f(y_{I'_i})$ and $I'_1, ..., I'_k$ are k arbitrary subsets of I'.*

Note that k is not fixed. We can now support more complicated interestingness functions that require more than a simple measure on one itemset:

$$interestingness(I') = F(m_{I'_1}, m_{I'_2}, ..., m_{I'_k}) \quad (2)$$

where the $m_{I'_i}$ are evaluated by $f(\cdot)$ as before. That is, $M_{I'} = F(\cdot)$ is evaluated over some k measures $m_{I'_i}$ where all $I'_i \subset I'$. We call $F(\cdot)$ *trivial* if $k = 1$ and $M_{I'} = F(m_{I'})$. In this case the function of $F(\cdot)$ can be performed by $f(\cdot)$ alone, as was the case in the examples we considered before introducing $F(\cdot)$.

Example 1 *The $minPI$ of an itemset $I' = \{1, ..., q\}$ is $minPI(I') = \min_i\{\sigma(I')/\sigma(\{i\})\}$. This measure is anti-monotonic and gives high value to itemsets where each member predicts the itemset with high probability. It is used in part for spatial colocation mining [10]. Using the data in Figure 1(a), $minPI(\{1, 2, 3\}) = \min\{1/2, 1/3, 1/1\} = 1/3$. In terms of our framework $g(\cdot)$ is the identity function, $\circ = \cap$, $f = |\cdot|$ so that $m_{I'} = \sigma(I')$ and $M_{I'} = F(m_{I'}, m_1, ..., m_q) = \min_i\{m_{I'}/m_i\}$.*

Our algorithm uses only the framework described above for computations on itemvectors. It also provides the arguments for the operators and functions very efficiently so it is flexible and fast. Because GLIMIT generates all subsets of an itemset I' before it generates the itemset I', an anti-monotonic property enables it to prune the search space. Therefore, to avoid exhaustive searches, our algorithm requires[7] that the function $F(\cdot)$ be anti-monotonic[8] in the underlying itemsets over which it operate (in conjunction with $\circ$, $g(\cdot)$ and $f(\cdot)$[9]).

Definition 5 *$F(\cdot)$ is* anti-monotonic *if $M_{I'} \geq M_{I"} \iff I' \subset I"$, where $M_{I'} = F(\cdot)$ is evaluated as per Definition 4.*

In the spirit this restriction, an itemset I' is considered interesting if $M_{I'} \geq minMeasure$, a threshold. We call such items *F-itemsets*.

4 Data Structures

In this section we outline two data structures that our algorithm (optionally) generates and uses.

We use the **PrefixTree** to efficiently store and build frequent itemsets. We represent an itemset $I' = \{i_1, ..., i_k\}$ as a sequence $\langle i_1, ..., i_k \rangle$ by choosing an ordering of the items (in this case $i_1 < ... < i_k$), and store the sequence in the tree. Since each node represents a sequence (ordered itemset) we can use the terms prefix node, itemset and sequence interchangeably. The prefix tree is built of *PrefixNodes*. Each PrefixNode is a tuple $(parent, depth, m, M, item)$

[6]That is, $C_M[i, j] = \sigma(\{i, j\})$.

[7]If there are few items then this constraint is not needed.

[8]Actually, prefix anti-monotonic[8] is necessary, which is a weaker constraint. With some modification, weakly anti-monotonic $F(\cdot)$ can also be supported

[9]Note that $f(\cdot)$ does *not* have to be anti-monotonic.

where $parent$ points to the parent of the node (so $n.parent$ represents the prefix of n), $depth$ is its depth of the node and therefore the length of the itemset at that node, m (M) is the measure of the itemset represented by the node as evaluated by $f(\cdot)$ ($F(\cdot)$) and $item$ is the last item in the sequence represented by the node. ϵ is the empty item so that $\{\epsilon\} \cup \alpha = \alpha$ where α is an itemset. The sequence (in reverse) represented by any node can be recovered by traversing toward the root. To make the link with our itemvector framework clear, suppose the itemset represented at a PrefixNode p is $I' = \{i_1, i_2, .., i_k\}$. Then $p.m = m_{I'} = f(g(x_{i_1}) \circ g(x_{i_2}) \circ ... \circ g(x_{i_k}))$ and $p.M = F(\cdot)$ where F is potentially a function of the m's of PrefixNodes corresponding to subsets of p.

The tree has the property that if s is in the PrefixTree, then so are all subsequences $s' \subset s$ by the anti-monotonic property of $F(\cdot)$. Hence we save a lot of space because the tree never duplicates prefixes. In fact, it contains exactly one node per *F-itemset*.

The PrefixTree is designed for efficient storage, and not for lookup purposes. This is why there are no references to child nodes. To facilitate the generation of association rules, a set of all nodes that have no children is maintained. We call this the *fringe*. An example of a PrefixTree is shown in Figure 2(a).

The fringe is useful for because (1) it contains all *maximal itemsets*[10], and (2) it can be used to efficiently generate all association rules:

Lemma 1 *Let $s =< i_1, ..., i_k >= \alpha\beta\gamma$ be the sequence corresponding to a prefix node n where $\alpha, \beta \neq \emptyset$. All association rules can be generated by creating all rules $\alpha \Rightarrow \beta$ and $\beta \Rightarrow \alpha$ for each fringe node.*

Proof: (Sketch) We don't generate all possible association rules that can be generated from itemset $\{i_1, ..., i_k\}$ by considering only n. Specifically, we miss (1) any rules $\alpha' \Rightarrow \beta'$ or $\beta' \Rightarrow \alpha'$ where α' is not a prefix of s, and (2) any such rules where there is a gap between α' and β'. However, by the construction of the tree there exists another node n' corresponding to the sequence $s' =< \alpha', \beta' >$ (since $s' \subset s$). If n' is not in the fringe, then by definition $s' \subset s$" where $s" =< \alpha', \beta', \gamma' >$ for some $\gamma' \neq \emptyset$ and n" (the node for s") is in the fringe. Hence $\alpha' \Rightarrow \beta'$ and $\beta' \Rightarrow \alpha'$ will be generated from node(s) other than n. Finally, the longest sequences are guaranteed to be in the fringe, hence all rules will be generated by induction.

[10]A *maximal itemset* is a F-itemset for which no superset is interesting (has measure above threshold $minMeasure$).

We use a **SequenceMap** to index the nodes in the PrefixTree so we can retrieve them for the following purposes: (1) to check *all* subsequences of a potential itemset for pruning (we automatically check two subsets without using the sequence map[11]), (2) to find the measures (m, M) for β in Lemma 1 when generating association rules[12], and (3) to find the the ms when we evaluate a *nontrivial* $F(\cdot)$.

First we introduce an abstract type - $Sequence$ - which represents a sequence of items. Equality testing and hash-code calculation is done using the most efficient iteration direction. PrefixNode can be considered a Sequence[13] and reverse iteration is the most efficient. By performing equality comparisons in reverse sequence order, we can map any Sequence to a PrefixNode by using a hash-table that stores the PrefixNode both as the key and the value. Hence we can search using any Sequence implementation (including a list of items), and the hash-table maps Sequences to the PrefixNodes that represents them. The space required is only that of the hash-table's bucket array, so it is very space efficient.

Finally, we can **avoid the use of *both* the PrefixTree and SequenceMap** without alteration to GLIMIT if (1) we output frequent itemsets when they are mined, (2) do not care if not *all* subsets are checked before we calculate a new itemvector, and (3) have a *trivial* $F(\cdot)$.

5 Algorithm

In this section we outline the main principles we use in GLIMIT and follow up with an illustrative example. We prove space complexity bounds before giving the algorithm in pseudo-code.

We exploit the following **facts** in order to use minimum space while avoiding any re-computations:

1. We can construct all itemvectors $y_{I'}$ by incrementally applying the rule $y_{I' \cup \{i\}} = y_{I'} \circ y_i$. That is, we only ever $\circ$ itemvectors corresponding to single items to the end of an itemvector. This means that given a PrefixNode p that is *not* the root, we only ever need to keep a single itemvector in memory for any child of p at

[11]Fact 3 in Section 5

[12]Since β is not a prefix of s, its measures are not stored along the path of PrefixNodes corresponding to s. So to get its measures we need to find its PrefixNode - by looking up β in the SequenceMap.

[13]It is can be considered as a reversed, singly linked list.

a time. If p *is* the root, we will need to keep its children's itemvectors in memory (the y_i).

2. It also means we use least space if we perform a depth first search. Then for any depth ($p.depth$), we will at most have only one itemvector in memory at a time.
3. We only ever check a new sequence by 'joining' siblings. That is, we check $\langle i_a, i_b, ..., i_i, i_j, i_k\rangle$ only if siblings $\langle i_a, i_b, ..., i_i, i_j\rangle$ and $\langle i_a, i_b, ..., i_i, i_k\rangle$, $k > j$ are in the prefix tree. Hence, we only try to expand nodes which have one or more siblings *below* it.
4. Suppose the items are $I = \{i_1, i_2, ..., i_n\}$. If we have read in k itemvectors y_{i_j} $j \in \{n, n-1, .., n-k-1\}$, then we can have completed all nodes corresponding to all subsets of $\{i_{n-k-1}, ..., i_n\}$. So if we use the depth first procedure, when a PrefixNode p is created all PrefixNodes corresponding to subsets of p's itemset will already have been generated. As well as being most space efficient, this is required to evaluate nontrivial $F(\cdot)$ and for more thorough (than Fact 3) pruning using the anti-monotonic requirement. This is what we call the 'bottom up' order of building the Prefix Tree.
5. When a PrefixNode p with $p.depth > 1$ (or $p.item$ is the top-most item) cannot have any children (because it has no siblings by Fact 3), its itemvector will no longer be needed.
6. When a topmost *sibling* (the topmost *child* of a node) is created (or we find its itemset is not frequent and hence don't need to create it), the itemvector corresponding to its *parent* p can be deleted. That is, we have just created the topmost (last) immediate child of p. This applies only when $p.depth > 1$ or when $p.item$ is the top-most *item*[14]. This is because we only ever need $y_{\{i_a, i_b, ..., i_i, i_j\}}$ until we generate $y_{\{i_a, i_b, ..., i_i, i_j, i_k\}} = y_{\{i_a, i_b, ..., i_i, i_j\}} \circ y_{i_k}$ where $i_a < i_b <, ..., < i_i < i_j < i_k$ (eg: $b - a$ and a may both greater then 1, etc) and $\{i_a, i_b, ..., i_i, i_q\} : j < q < k$ is not frequent. Indeed, we can write the result of $y_{\{i_a, i_b, ..., i_j\}} \circ y_{i_k}$ directly into the itemvector holding $y_{\{i_a, i_b, ..., i_j\}}$. Conversely, while there is still a child to create (or test) we cannot delete p's corresponding itemvector.
7. When we create a PrefixNode p on the topmost *branch* (eg: when all itemsets are frequent, p will correspond to $< i_1, i_2, .., i_k >$ for any $k \geq 1$), we can delete the itemvector corresponding to the single item $p.item$ (eg: i_k). Fact 6 will always apply in this case too (eg: we can also delete i_{k-1} if $k > 1$). The reason behind this is that by using the bottom up method (and the fact our itemsets are ordered), we know that if we have $y_{i_1, ..., i_k}$ we can only ever $\circ$ a y_{i_j} with $j > k$ onto the end.

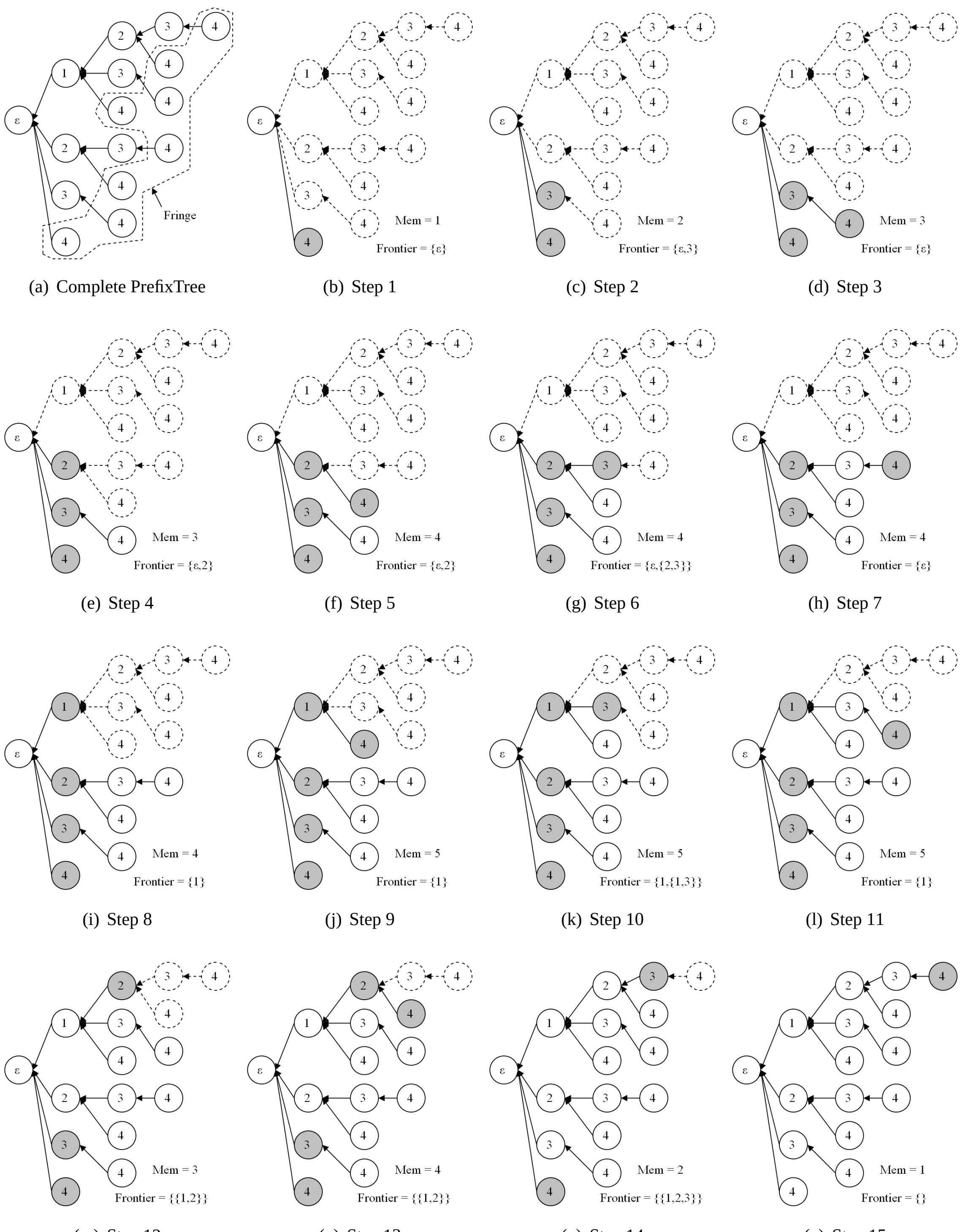
[14]By Fact 1 we cannot apply this to nodes with $p.depth = 1$ (unless it is the topmost node) as they correspond to single items and are still needed.

We now present an **example** of our algorithm to illustrate some of these facts.

Suppose we have the items $\{1, 2, 3, 4\}$ and the $minMeasure$ (we will use $minSup$) threshold is such that all itemsets are interesting (frequent). Figure 2 shows the target prefix tree and the steps in mining it. This example serves to show how we manage the memory while avoiding any re-computations. For now, consider the frontier list in the figure as a list of PrefixNodes that have not been completed. It should be clear that we use a bottom up and depth first procedure to mine the itemsets, as motivated by Facts 2 and 4. We complete all subtrees before moving to the next item. In (d) we calculate $y_{\{3,4\}} = y_3 \circ y_4$ as per Fact 1. Note we are also making use of Fact 3 - $\{3\}$ and $\{4\}$ are siblings. Once we have created the node for $\{3, 4\}$ in (d), we can delete $y_{\{3,4\}}$by Fact 5. It has no possible children because of the ordering of the sequences. The same holds for $\{2, 4\}$ in (f). In (g), the node for $\{2, 3\}$ is the topmost sibling (child). Hence we can apply Fact 6 in (h). Note that by Fact 1 we calculate $y_{\{2,3,4\}}$ as $y_{\{2,3,4\}} = y_{\{2,3\}} \circ y_4$. Note also that because we need the itemvectors of the single items in memory we have not been able to use Fact 7 yet. Similarly, Fact 6 is also applied in (l), (m), (o) and (p). However, note that in (m), (o) and (p) we also use Fact 7 to delete y_2, y_3, and y_4. In (l) we deleted y_1 for two reasons: Fact 6 and 7 (it is a special case in Fact 6). Finally, to better illustrate Fact 3, suppose $\{2, 4\}$ is not frequent. This means that $\{2, 3\}$ will have no siblings anymore, so we do not even need to consider $\{2, 3, 4\}$ by Fact 3.

We know already that the time **complexity** is roughly linear in the number of frequent itemsets, because we avoid re-computations of itemvectors. So the question now is, what is the maximum number of itemvectors that we have in memory at any time? There are two main factors that influence this. First, we need to keep the itemvectors for individual items in memory until we we have completed the node for the top-most item (Fact 1 and 7). Hence, the 'higher' up in the tree we are, the more this contributes. Secondly, we need to keep itemvectors in memory until

Figure 2. Building the PrefixTree (mining itemsets) Example. Nodes are labeled with their $item$ value. Shaded nodes have their corresponding itemvector in memory. Dotted nodes have not been mined yet. Solid lines are the parts of the tree that have been created.



(a) Complete PrefixTree (b) Step 1 (c) Step 2 (d) Step 3

(e) Step 4 (f) Step 5 (g) Step 6 (h) Step 7

(i) Step 8 (j) Step 9 (k) Step 10 (l) Step 11

(m) Step 12 (n) Step 13 (o) Step 14 (p) Step 15

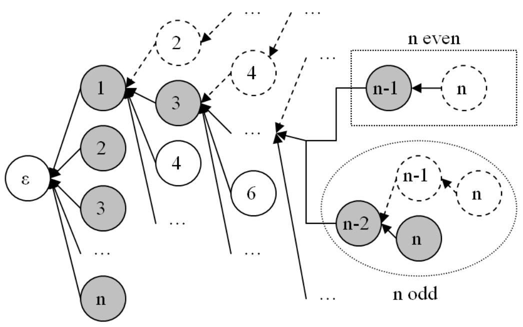


Figure 3. Maximum number of itemvectors. Two cases: n even or odd.

we complete their respective nodes. That is, check all their children (Fact 6) or if they can't have children (Fact 5). Now, the further we are up in the tree, or any subtree for that matter, without completing the node, the longer the sequence of incomplete nodes is and hence the the more itemvectors we need to keep. Considering both these factors leads to the situation in Figure 3 - that is, we are up to the top item and the topmost path from that item so that no node along the path is completed. If we have n items, the worst case itemvector usage is just the number of coloured nodes in Figure 3. There are n itemvectors $y_i : i \in \{1, ..., n\}$ corresponding to the single items (children of the root). There are a further $\lceil n/2 \rceil$ itemvectors along the path from node $\{1\}$ (inclusive) to the last coloured node (these are the uncompleted nodes)[15]. Therefore the total space required is just $n + \lceil n/2 \rceil - 1$, where the -1 is so that we do not double count the itemvector for $\{1\}$.

This is for the worst case when all itemsets are frequent. Clearly, a closer bound is if we let $n' \leq n$ be the number of frequent items. Hence, we need space linear in the number of frequent items. The multiplicative constant (1.5) is low, and in practice (with non-pathological support thresholds), we use far fewer than n itemvectors. If we know that the longest frequent itemset has size l, then we can additionally bound the space by $n' + \lceil l/2 \rceil - 1$. Furthermore, since the frontier contains all uncompleted nodes, we know from the above that its upper bound is $\lceil l/2 \rceil$. We have sketched the proof of:

Lemma 2 *Let n be the number of items, and $n' \leq n$ be the number of frequent items. Let $l \leq n'$ be the largest itemset. GLIMIT uses at most $n' + \lceil l/2 \rceil - 1$ itemvectors of space. Furthermore, $|frontier| \leq \lceil l/2 \rceil$.*

As an aside, note we could perform the transpose operation in memory before mining while still remaining within the worst case space complexity. However, on *average* and for practical levels of $minMeasure$ (eg: $minSup$), this would require more memory.

The **algorithm** is a depth first traversal through the PrefixTree. Any search can be implemented either recursively or using the *frontier* method, whereby a list (priority queue) of states (each containing a node that has yet to be completely expanded) is maintained. The general construct is to retrieve the first state, evaluate it for the search criteria, expand it (create some child nodes), and add states corresponding to the child nodes to the frontier. Using different criteria and frontier orderings leads to different search techniques. Our frontier contains any nodes that have not yet been completed, wrapped in $State$ objects. Algorithm 1 describes[16] the additional types we use (such as $State$) and shows the initialisation and the main loop - which calls $step(\cdot)$. It also describes the $check(\cdot)$ and $calculateF(\cdot)$ methods, used by $step(\cdot)$.

6 Experiments

We evaluated our algorithm on two publicly available datasets from the FIMI repository[17] - T10I4D100K and T40I10D100K. These datasets have $100,000$ transactions and a realistic skewed histogram of items. They have 870 and 942 items respectively. To apply GLIMIT we first transpose the dataset as a preprocessing step[18].

We compared GLIMIT to a publicly available implementation of FP-Growth and Apriori. We used the algorithms from ARtool[19] as it is written in Java, like our implementation, and it has been available for

[15]When n is even, the last node is $\{1, 3, 5, ..., n-3, n-1\}$ and when n is odd it is $\{1, 3, 5, ..., n-2, n\}$. The cardinality of both these sets, equal to the number of nodes along the path, is $\lceil n/2 \rceil$. Note that in the even case, the next step to that shown will use the same memory (the itemvector for node $\{1, 3, 5, ..., n-3, n-1\}$ is no longer needed once we create $\{1, 3, 5, ..., n-3, n-1, n\}$ by Fact 6, and we write $y_{\{1,3,5,...,n-3,n-1,n\}}$ directly into $y_{\{1,3,5,...,n-3,n-1\}}$ as we compute it so both need never be in memory at the same time).

[16]The pseudo-code in our algorithms is java-like and we assume a garbage collector which simplifies it. Indentation defines blocks and we ignore type casts.

[17]http://fimi.cs.helsinki.fi/data/

[18]This is cheap, especially for sparse matrices - precisely what the datasets in question typically are. Our data was transposed in 8 and 15 seconds respectively using a naive Java implementation, and without using sparse techniques.

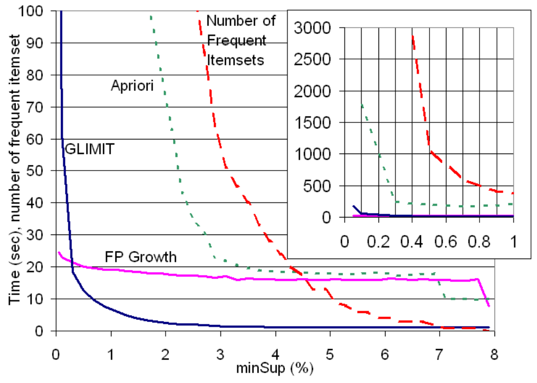
[19]http://www.cs.umb.edu/ laur/ARtool/. It was not used via the supplied GUI. The underlying classes were invoked directly.

Algorithm 1 Data-types, initialisation, main loop and methods.

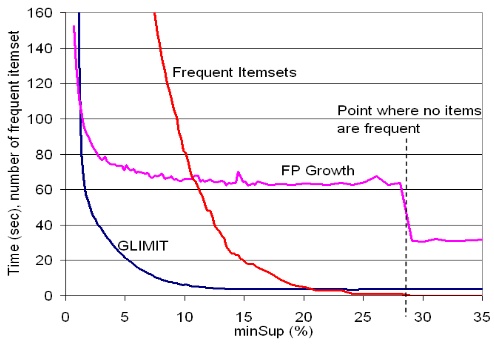
Input: (1) Dataset ($inputFile$) in transpose format (*may* have $g(\cdot)$ already applied). (2) $f(\cdot)$, $\circ$, $F(\cdot)$ and $minMeasure$.

Output: Completed $PrefixTree$ ($prefixTree$) and $SequenceMap$ (map) containing all *F-itemsets*.

```
$Pair:\ (Itemvector\, y_i,\ Item\, item)$
/*$y_i$ is the itemvector for $item$ and corresponds to $y_i$ in Fact 1. We keep reusing them through $buffer$:*/
$State:\ (PrefixNode\, node,\ Itemvector\, y_{I'},\ Iterator\, itemvectors,\ boolean\, top,\ Pair\, newPair,\ List\, buffer)$
/*$y_{I'}$ is the itemvector corresponding to $node$ (and $y_{I'}$ in Fact 1). $buffer$ is used to create the *Iterators* (such as
$itemvectors$) for the $State$s created to hold the children of $node$. We need $buffer$ to make use of Fact 3. $itemvectors$
provides the $y_i$ to join with $y_{I'}$ and $newPair$ helps us do this.*/

/***Initialisation**: initialise $prefixTree$ with its root. Initialise $map$ and $frontier$ as empty. Create initial state:*/
$Iterator\, itemvectors = new\, AnnotatedItemvetorIterator(inputFile)$; /*Iterator is over $Pair$ objects*/
/*Reads input one row at a time and annotates the itemvector with the item it corresponds to. could also apply $g(\cdot)$*/
$frontier.add(new\, State(prefixTree.getRoot(), null, itemvectors, false, null, new\, LinkedList()))$;
/***Main Loop***/
while (!$frontier.isEmpty()$) $step(frontier.getFirst())$;

/*Perform one expansion. $state.node$ is the parent of the new $PrefixNode$ ($newNode$) that we create if
$newNode.M \geq minMeasure$. $localTop$ is true iff we are processing the top sibling of *any* subtree. $nextTop$ becomes
$newNode.top$ and is set so that $top$ is true only for a node that is along the *topmost branch* of the prefix tree.*/
$void$ **step(**$State\, state$**)**
  if ($state.newPair \neq null$) /*see end of method ♣*/
    $state.buffer.add(state.newPair)$; $state.newPair = null$; /*so we don't add it again*/
  $Pair\, p = state.itemvectors.next()$; $boolean\, localTop = !state.itemvectors.hasNext()$;
  if ($localTop$) /*Remove $state$ from $frontier$ (and hence delete $state.y_{I'}$) as we are creating...*/
    $localFrontier.removeFirst()$; /*... the top child of $node$ in this step. Fact 6*/
  $Itemvector\, y_{I' \cup \{i\}} = null$; $double\, m, M$; $boolean\, nextTop$; /*$top$ in the next $State$ we create.*/
  if ($state.node.isRoot()$) /*we are dealing with itemsets of length 1 (so $I' = \{\epsilon\}$)*/
    $m = f(p.y_i)$; $M = calculateF(null, \{p.item\}, m)$; $y_{I' \cup \{i\}} = p.y_i$;
    $state.top = localTop$; $nextTop = localTop$; /*initialise $top$s.*/
  else
    $nextTop = localTop\, \&\&\, state.top$;
    if ($check(state.node, p.item)$) /*make use of anti-monotonic pruning property*/
      if ($localTop\, \&\&\, (state.node.getDepth() > 1\, ||\, state.top)$) /*Fact 6 or 7*/
        /*No longer need $state.y_{I'}$ as this is the last child we can create under
        * $state.node$ (and it is not a single item other than perhaps the topmost)*/
        $y_{I' \cup \{i\}} = state.y_{I'}$; $y_{I' \cup \{i\}} \circ = p.y_i$; /*can write result directly into $y_{I' \cup \{i\}}$*/
      else $y_{I' \cup \{i\}} = state.y_{I'} \circ p.y_i$; /*need to use additional memory for the child ($y_{I' \cup \{i\}}$).*/
      $m = f(y_{I' \cup \{i\}})$; $M = calculateF(state.node, \{p.item\}, m)$;
    else $m = M = -\infty$ /*don't need to calculate, we know $M < minMeasure$*/
  if ($M \geq minMeasure$) /*Found an interesting itemset - create $newNode$ for it.*/
    $PrefixNode\, newNode = prefixTree.createChildUnder(state.node)$;
    $newNode.item = p.item$; $newNode.m = m$; $newNode.M = M$;
    $sequenceMap.put(newNode)$;
    if ($state.buffer.size() > 0$) /*there is potential to expand $newNode$. Fact 5*/
      $State\, newState = new\, State(newNode, y_{I' \cup \{i\}}, state.buffer.iterator(), nextTop, new\, LinkedList())$;
      /*add to front of frontier (ie: in front of $state$ if it's still present) so depth first search. Fact 2.*/
      $frontier.addFront(newState)$; $state.newPair = p$; /*if $state.node$ is not complete, we
      * will add $p$ to $state.buffer$ *after* $newState$ has been completed. See ♣*/

/*Let $\alpha$ be the itemset corresponding to $node$. $\alpha \cup \{item\}$ is the itemset represented by a child $p$ of $node$ so that
$p.item = item$. $m$ would be $p.m$. This method calculates $p.M$ by using $map$ to look up the $PrefixNode$s
corresponding to the $k$ required subsets of $\alpha \cup \{item\}$ to get their $m$ values, $m_1, ..., m_k$. Then it return $F(m_1, ..., m_k)$.*/
$double$ **calculateF(**$PrefixNode\, node, Item\, item, double\, m$**)** /*details depend on $F(\cdot)$*/

/*Check whether the itemset $\alpha \cup \{item\}$ could be interesting by exploiting the anti-monotonic property of $F(\cdot)$: use $map$
to check whether subsets of $\alpha \cup \{item\}$ (except $\alpha$ and $(\alpha - node.item) \cup \{item\}$ by Fact 3) exist.*/
$boolean$ **check(**$PrefixNode\, node, Item\, item$**)** /*details omitted*/
```



(a) Runtime and frequent itemsets. T10I4D100K. Inset shows detail for low support.



(b) Runtime and frequent itemsets. T40I10D100K.

Figure 4. Results

some time. There are some implementation differences that at worst give ARtool a small constant factor advantage[20]. This is fine since in this section we really only want to show that GLIMIT is quite fast and efficient when compared to existing algorithms on the traditional FIM problem. Our contribution is the itemvector framework that allows operations that previously could not be considered, and a flexible and new class of algorithm that uses this framework to efficiently mine data cast into different and useful spaces. *The fact that it is also fast when applied to traditional FIM is secondary*. To represent itemvectors for traditional FIM, we used bit-vectors[21] so that each bit is set if the corresponding transaction contains the item(set). Therefore g creates the bit-vector, $\circ = AND$, $f(\cdot) = sum(\cdot)$ and $F(m) = m$.

[20]We use ASCII input format, so we need to parse the transaction numbers. ARtool uses a binary format, where items are integers encoded in 4 bytes. We use counting algorithms via an abstract framework and generally make use of many layers of inheritance and object level operations for flexibility while the ARtool algorithms are rigid and use low level operations.

[21]We used the *Colt* (http://dsd.lbl.gov/~hoschek/colt/) BitVector implementation.

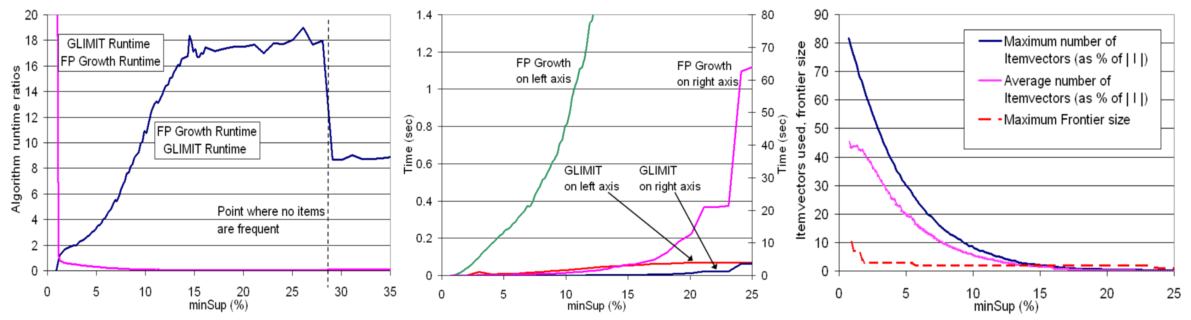
Figure 4(a) shows the runtime[22] of FP-Growth, GLIMIT and Apriori[23] on T10I4D100K, as well as the number of frequent items. The analogous graph for T40I10D100K is shown in Figure 4(b) - we did not run Apriori as it is too slow. These graphs clearly show that when the support threshold is below a small value (about 0.29% and 1.2% for the respective datasets), FP-Growth is superior to GLIMIT. However, above this threshold GLIMIT outperforms FP-Growth significantly. Figure 5(a) shows this more explicitly by presenting the runtime ratios for T40I10D100K. FP-Growth takes at worst 19 times as long as GLIMIT. We think it is clear that GLIMIT is superior above the threshold. Furthermore, this threshold is very small and practical applications usually mine with much larger thresholds than this.

GLIMIT scales roughly linearly in the number of frequent itemsets. Figure 5(b) demonstrates this experimentally by showing the average time to mine a single frequent itemset. The value for GLIMIT is quite stable, rising slowly toward the end (as there we still need to check itemsets, but very few of these turn out to be frequent). FP-Growth on the other hand, clearly does not scale linearly. The reason behind these differences is that FP-Growth first builds an FP-tree. This effectively stores the entire Dataset (minus infrequent single items) in memory. The FP-tree is also highly cross-referenced so that searches are fast. The downside is that this takes significant time and a lot of space. This pays off extremely well when the support threshold is very low, as the frequent itemsets can read from the tree very quickly. However, when $minSup$ is larger, much of the time and space is wasted. GLIMIT uses time and space as needed, so it does not waste as many resources, making it fast. The downside is that the operations on bit-vectors (in our experiments, of length $100,000$) can be time consuming when compared to the search on the FP-tree, which is why GLIMIT cannot keep up when $minSup$ is very small. Figure 5(c) shows the maximum and average[24] number of itemvectors our algorithm uses as a percentage of the number of items. At worst, this can be interpreted as the percentage of the dataset in memory. Although the worst case space is 1.5 times the number of items, n (Lemma 2), the figure clearly shows this is never reached in our experiments. Our maximum was ap-

[22]Pentium 4, 2.4GHz with 1GB RAM running WindowsXP Pro.

[23]Apriori was not run for extremely low support as it takes longer than 30 minutes for $minSup \leq 0.1\%$

[24]over the calls to $step(\cdot)$.



(a) Runtime ratios. T10I4D100K.

(b) Average time taken per frequent itemset shown on two scales. T10I4D100K.

(c) Number of Itemvectors needed and maximum frontier size. T10I4D100K.

Figure 5. Results

proximately $0.82n$. By the time it gets close to $1.5n$, $minSup$ would be so small that the runtime would be unfeasibly large anyhow. Furthermore, the space required drops quite quickly as $minSup$ is increased (and hence the number of frequent items decreases). Figure 5(c) also shows that the maximum frontier size is very small.

Finally, we reiterate that we can avoid using the prefix tree and sequence map, so the only space required are the itemvectors and the $frontier$. That is, the space required is truly linear.

7 Conclusion and Future Work

We showed interesting consequences of viewing transaction data as itemvectors in transaction-space, and developed a framework for operating on itemvectors. This abstraction gives great flexibility in the measures used and opens up the potential for useful transformations on the data. Our future work will focus on finding useful geometric measures and transformations for itemset mining. One problem is to find a way to use SVD prior to mining for itemsets larger than 2. We also presented GLIMIT, a novel algorithm that uses our framework and significantly departs from existing algorithms. GLIMIT mines itemsets in one pass without candidate generation, in linear space and time linear in the number of interesting itemsets. Experiments showed that it beats FP-Growth above small support thresholds. Most importantly, it allows the use of transformations on the data that were previously impossible.

References

[1] D. Achlioptas. Database-friendly random projections. In *Symposium on Principles of Database Systems*, 2001.

[2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of 20th International Conference on Very Large Data Bases VLDB*, pages 487–499. Morgan Kaufmann, 1994.

[3] Workshop on frequent itemset mining implementations 2003. http://fimi.cs.helsinki.fi/fimi03.

[4] Workshop on frequent itemset mining implementations 2004. http://fimi.cs.helsinki.fi/fimi04.

[5] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *2000 ACM SIGMOD Intl. Conference on Management of Data*, pages 1–12. ACM Press, May 2000.

[6] F. Korn, A. Labrinidis, Y. Kotidis, and C. Faloutsos. Quantifiable data mining using ratio rules. *VLDB Journal: Very Large Data Bases*, 8(3–4):254–266, 2000.

[7] F. Pan, G. Cong, A. Tung, J. Yang, and M. Zaki. Carpenter: Finding closed patterns in long biological datasets. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Morgan Kaufmann, 2003.

[8] J. Pei, J. Han, and L. Lakshmanan. Pushing convertible constraints in frequent itemset mining. *Data Mining and Knowledge Discovery: An International Journal*, 8:227–252, May 2004.

[9] J. Pei, J. Han, and R. Mao. CLOSET: An efficient algorithm for mining frequent closed itemsets. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 21–30, 2000.

[10] S. Shekhar and Y. Huang. Discovering spatial colocation patterns: A summary of results. *Lecture Notes in Computer Science*, 2121:236+, 2001.

[11] M. Steinbach, P.-N. Tan, H. Xiong, and V. Kumar. Generalizing the notion of support. In *The Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining KDD'04*, 2004.

[12] J. Wang and G. Karypis. Harmony: Efficiently mining the best rules for classification. In *SIAM International Conference on Data Mining*, pages 205–215, 2005.

School of Information Technologies, J12
The University of Sydney
NSW 2006 AUSTRALIA
T +61 2 9351 4917
F +61 2 9351 3838
www.it.usyd.edu.au

ISBN 1 86487 854 1