LPMiner: An Algorithm for Finding Frequent Itemsets Using Length-Decreasing Support Constraint*

Masakazu Seno and George Karypis

Department of Computer Science and Engineering, Army HPC Research Center

University of Minnesota

4-192 EE/CS Building, 200 Union Street SE, Minneapolis, MN 55455

Fax: (612) 625-0572

Technical Report #01-026

{`seno, karypis`}`@cs.umn.edu`

August 7, 2001

Abstract

Over the years, a variety of algorithms for finding frequent itemsets in very large transaction databases have been developed. The key feature in most of these algorithms is that they use a constant support constraint to control the inherently exponential complexity of the problem. In general, itemsets that contain only a few items will tend to be interesting if they have a high support, whereas long itemsets can still be interesting even if their support is relatively small. Ideally, we desire to have an algorithm that finds all the frequent itemsets whose support decreases as a function of their length. In this paper we present an algorithm called LPMiner, that finds all itemsets that satisfy a length-decreasing support constraint. Our experimental evaluation shows that LPMiner is up to two orders of magnitude faster than the FP-growth algorithm for finding itemsets at a constant support constraint, and that its runtime increases gradually as the average length of the transactions (and the discovered itemsets) increases.

1   Introduction

Data mining research during the last eight years has led to the development of a variety of algorithms for finding frequent itemsets in very large transaction databases [1, 2, 4, 9]. These itemsets can be used to find association rules or extract prevalent patterns that exist in the transactions, and have been effectively used in many different domains and applications.

The key feature in most of these algorithms is that they control the inherently exponential complexity of the problem by finding only the itemsets that occur in a sufficiently large fraction of the transactions, called the *support*. A limitation of this paradigm for generating frequent itemsets is that it uses a constant value of support, irrespective of the length of the discovered itemsets. In general, itemsets that contain only a few items will tend to be interesting if they have a high support, whereas long itemsets can still be interesting even if their support is relatively small. Unfortunately, if constant-support-based frequent itemset discovery algorithms are used to find some of the longer but infrequent itemsets, they will end up generating an exponentially large number of short itemsets. Maximal frequent itemset discovery algorithms [9] can potentially be used to find some of these longer itemsets, but these algorithms can still generate a very large number of short infrequent itemsets if these itemsets are maximal. Ideally, we desire to have an algorithm that finds all the frequent itemsets whose support decreases as

---

a function of their length. Developing such an algorithm is particularly challenging because the downward closure property of the constant support constraint cannot be used to prune short infrequent itemsets.

In this paper we present another property, called *smallest valid extension* (SVE), that can be used to prune the search space of potential itemsets in the case where the support decreases as a function of the itemset length. Using this property, we developed an algorithm called LPMiner, that finds all itemsets that satisfy a length-decreasing support constraint. LPMiner uses the recently proposed FP-tree [4] data structure to compactly store the database transactions in main memory, and the SVE property to prune certain portions of the conditional FP-trees, that are being generated during itemset discovery. Our experimental evaluation shows that LPMiner is up to two orders of magnitude faster than the FP-growth algorithm for finding itemsets at a constant support constraint, and that its runtime increases gradually as the average length of the transactions (and the discovered itemsets) increases.

The rest of this paper is organized as follows. Section 2 provides some background information and related research work. Section 3 describes the FP-growth algorithm [4], on which LPMiner is based. In Section 4, we describe how the length-decreasing support constraint can be exploited to prune the search space of frequent itemsets. The experimental results of our algorithm are shown in Section 5, followed by the conclusion in Section 6.

2 Background and Related Works

The problem of finding frequent itemsets is formally defined as follows: Given a set of transactions T, each containing a set of items from the set I, and a support σ, we want to find all subsets of items that occur in at least $\sigma|T|$ transactions. These subsets are called *frequent itemsets*.

Over the years a number of efficient algorithms have been developed for finding all frequent itemsets. The first computationally efficient algorithm for finding itemsets in large databases was Apriori [1], which finds frequent itemsets of length l based on previously generated $(l-1)$-length frequent itemsets. The key idea of Apriori is to use the downward closure property of the support constraint to prune the space of frequent itemsets. The FP-growth algorithm [4] finds frequent itemsets by using a data structure called FP-tree that can compactly store in memory the transactions of the original database, thus eliminating the need to access the disks more than twice. Another efficient way to represent transaction database is to use *vertical tid-list database* format. The vertical database format associates each item with all the transactions that include the item. Eclat in [7] uses this data format to find all frequent itemsets.

Even though to our knowledge no work has been published for finding frequent itemsets in which the support decreases as a function of the length of the itemset, there has been some work in developing itemset discovery algorithms that use multiple support constraints. Liu *et al.* [5] presented an algorithm in which each item has its minimum item support (or MIS). The minimum support of an itemset is the lowest MIS among those items in the itemset. By sorting items in ascending order of their MIS values, the minimum support of the itemset never decreases as the length of itemset grows, making the support of itemsets downward closed. Thus the Apriori-based algorithm can be applied. Wang *et al.* [6] allow a set of more general support constraints. In particular, they associate a support constraint for each one of the itemsets. By introducing a new function called Pminsup that has "Apriori-like" property, they proposed an Apriori-based algorithm for finding the frequent itemsets. Finally, Cohen *et al.* [3] adopt a different approach in that they do not use any support constraint. Instead, they search for similar itemsets using probabilistic algorithms, that do not guarantee that all frequent itemsets can be found.

3 FP-growth Algorithm

In this section, we describe how the FP-growth algorithm works because our approach is based on this algorithm. The description here is based on [4].

The key idea behind FP-growth is to use a data structure called FP-tree to obtain a compact representation of the original transactions so that they can fit into the main memory. As a result, any subsequent operations that are required to find the frequent itemsets can be performed quickly, without having to access the disks. The FP-growth algorithm achieves that by performing just two passes over the transactions. Figure 1 shows how the FP-tree generation algorithm works given an input transaction database that has five transactions with a total
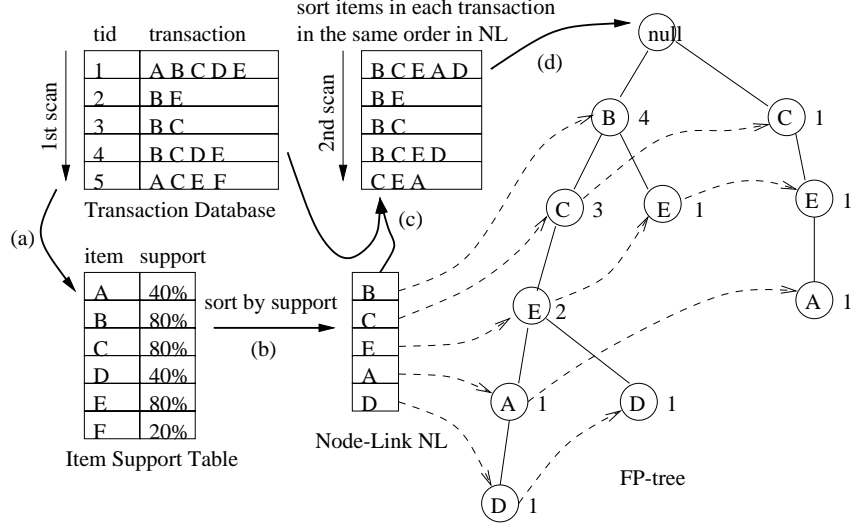
Figure 1: Flow of FP-tree generation

of six different items. First, it scans the transaction database to count how many times each item occurs in the database to get an "Item Support Table" (step (a)). The "Item Support Table" has a set of (item-name, support) pairs. For example, item A occurs twice in the database, namely in a transaction with tid 1 and another one with tid 5; therefore its support is $2/5 = 40\%$. In step (b), those items in the Item Support Table are sorted according to their support. The result is stored in item-name field of Node-Link header table NL. Notice that item F is not included in NL because the support of item F is less than the minimum support constraint 40%. In step (c), items in each transaction in the input transaction database are sorted in the same order as items in the Node-Link header table NL. While transaction tid 5 is sorted, item F is discarded because the item is infrequent and has no need of consideration. In step (d), the FP-tree is generated by inserting those sorted transactions one by one. The initial FP-tree has only its root. When the first transaction is inserted, nodes that represent item B, C, E, A, and D are generated, forming a path from the root in this order. The count of each node is set to 1 because each node represents only one transaction (tid 1) so far. Next, when the second transaction is inserted, a node representing item B is *not* generated. Instead, the node already generated is reused. In this case, because the root node has a child that represents item B, the count of the node is incremented by 1. As for item E, since there is no child representing item E under the current node, a new node with item-name E is generated as a child of the current node. Similar processes are repeated until all the sorted transactions are inserted into the FP-tree.

Once an FP-tree is generated from the input transaction database, the algorithm mines frequent itemsets from the FP-tree. The algorithm generates itemsets from shorter ones to longer ones adding items one by one to those itemsets already generated. It divides mining the FP-tree into mining smaller FP-trees, each of which is based on an item on the Node-Link header table in Figure 1. Let us choose item D as an example. For item D, we generate a new transaction database called *conditional pattern base*. Each transaction in the conditional pattern base consists of items on the paths from parent nodes whose child nodes have item-name D to the root node. The conditional pattern base for item D is shown in Figure 2. Each transaction in the conditional pattern base also has its count of occurrence corresponding to the count of the node with item-name D in the original FP-tree. Note that item D itself is a frequent itemset consisting of one item. Let us call this frequent itemset "D" a *conditional pattern*. A conditional pattern base is a set of transactions each of which includes the conditional pattern. What we do next is to forget the original FP-tree in Figure 1 for a while and then focus on the conditional pattern base we got just now to generate frequent itemsets that include this conditional pattern "D". For this purpose, we generate a smaller FP-tree than the original one, based on the conditional pattern "D". This new FP-tree, called *conditional FP-tree*,
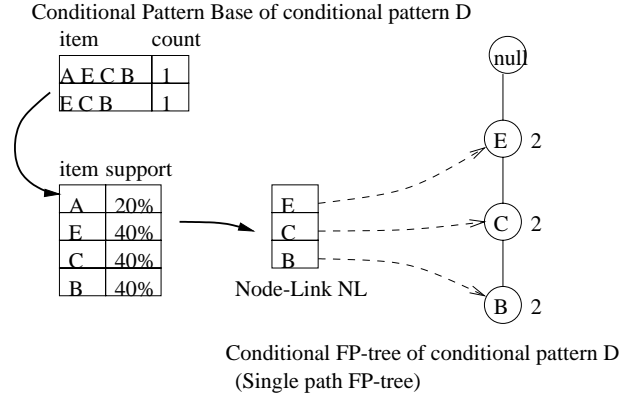
3

Figure 2: Conditional FP-tree

is generated from the conditional pattern base using the FP-tree generation algorithm again. If the conditional FP-tree is not a single path tree, we divide mining this conditional FP-tree to mining even smaller conditional FP-trees recursively. This is repeated until we obtain a conditional FP-tree with only a single path. During those recursively repeated processes, all selected items are added to the conditional pattern. Once we obtain a single path conditional FP-tree like the one in Figure 2, we generate all possible combinations of items along the path and combine each of these sets of items to the conditional pattern. For example, from those three nodes in the conditional FP-tree in Figure 2, we have $2^3 = 8$ combinations of item B, C, and E: " " (no item), "B", "C", "E", "BC", "CE", "EB", and "BCE". Then we obtain frequent itemsets based on conditional pattern base "D": "D", "DB", "DC", "DE", "DBC", "DCE", "DEB", and "DBCE".

4 LPMiner Algorithm

LPMiner is an itemset discovery algorithm, based on the FP-growth algorithm, which finds all the itemsets that satisfy a particular length-decreasing support constraint $f(l)$; here l is the length of the itemset. More precisely, $f(l)$ satisfies $f(l_a) \geq f(l_b)$ for any l_a, l_b such that $l_a < l_b$. The idea of introducing this kind of support constraint is that by using a support that decreases with the length of the itemset, we may be able to find long itemsets, that may be of interest, without generating an exponentially large number of shorter itemsets. Figure 3 shows a typical length-decreasing support constraint. In this example, the support constraint decreases linearly to the minimum value and then stays the same for itemsets of longer length. Our problem is restated as finding those itemsets located above the curve determined by length-decreasing support constraint $f(l)$.

A simple way of finding such itemsets is to use any of the traditional constant-support frequent itemset discovery algorithms, in which the support was set to $\min_{l>0} f(l)$, and then discard the itemsets that do not satisfy the length-decreasing support constraint. This approach, however, does not reduce the number of infrequent itemsets being discovered, and as our experiments will show, requires a large amount of time.

As discussed in the introduction, finding the complete set of itemsets that satisfy a length-decreasing support function is particularly challenging because we cannot use the downward closure property of the constant support frequent itemsets. This property states that in order for an itemset of length l to be frequent, all of its subsets have to be frequent as well. As a result, once we find that an itemset of length l is infrequent, we know that any longer itemsets that include this particular itemset cannot be frequent, and thus eliminate such itemsets from further consideration. However, because in our problem the support of an itemset decreases as its length increases, an itemset *can* be frequent even if its subsets are infrequent.

A key property, regarding the itemset whose support decreases as a function of their length, is the following. Given a particular itemset I with a support of σ_I, such that $\sigma_I < f(|I|)$, then $f^{-1}(\sigma_I) = \min(\{l|f(l) = \sigma_I\})$ is

4

support(%)

0.5

0.01
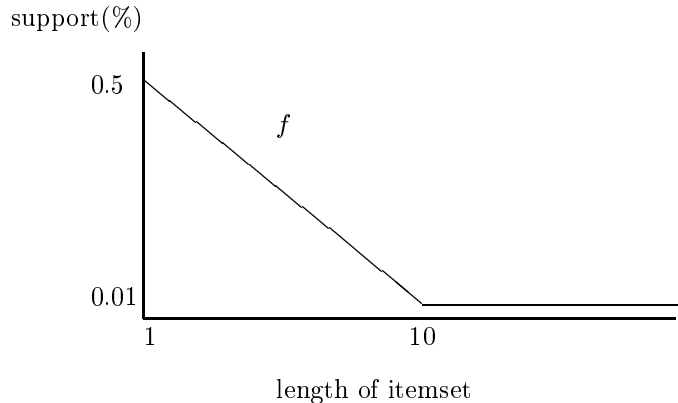
f

1   10

length of itemset

Figure 3: An example of typical length-decreasing support constraint

the minimum length that an itemset I' such that $I' \supset I$ must have before it can potentially become frequent. Figure 4 illustrates this relation graphically. The length of I' is nothing more than the point at which a line parallel to the x-axis at $y = \sigma_I$ intersects the support curve; here, we essentially assume the best case in which I' exists and it is supported by the same set of transactions as its subset I. We will refer to this property as the *smallest valid extension* property or *SVE* for short.

LPMiner uses this property as much as it can to prune the conditional FP-trees, that are generated during the itemset discovery phase. In particular, it uses three different pruning methods that, when combined, substantially reduce the search space and the overall runtime. These methods are described in the rest of this section.

4.1   Transaction Pruning, TP

The first pruning scheme implemented in LPMiner uses the smallest valid extension property to eliminate entire candidate transactions of a conditional pattern base. Recall from Section 3 that, during frequent itemset generation, the FP-growth algorithm builds a separate FP-tree for all the transactions that contain the conditional pattern currently under consideration. Let CP be that conditional pattern, $|CP|$ be its length, and $\sigma(CP)$ be its support. If CP is infrequent, we know from the SVE property that in order for this conditional pattern to grow to something indeed frequent, it must have a length of at least $f^{-1}(\sigma(CP))$. Using this requirement, before building the FP-tree corresponding to this conditional pattern, we can eliminate any transactions whose length is shorter than $f^{-1}(\sigma(CP)) - |CP|$, as these transactions cannot contribute to a valid frequent itemset in which CP is part of it. We will refer to this as the *transaction pruning* method and denote it by *TP*.

We evaluated the complexity of this method in comparison with the complexity of inserting a transaction to a conditional pattern base. There are three parameters we have to know to prune a transaction: the length of each transaction being inserted, $f^{-1}(\sigma(CP))$, and $|CP|$. The length of each transaction is calculated in a constant time added to the original FP-growth algorithm, because we can count each item when the transaction is actually being generated. As $f^{-1}(\sigma(CP))$ and $|CP|$ are common values for all transactions in a conditional pattern base, these values need to be calculated only once for the conditional pattern base. It takes a constant time added to the original FP-growth algorithm to calculate $|CP|$. As for $f^{-1}(\sigma(CP))$, evaluating f^{-1} takes $O(\log(|I|))$ to execute binary search on the support table determined by $f(l)$. Let cpb be the conditional pattern base and $m = \sum_{tran \in cpb} |tran|$. The complexity per inserting a transaction is $O(\log(|I|)/m)$. Under an assumption that all items in I are contained in cpb, this value is nothing more than $O(1)$. Thus, the complexity of this method is just a constant time per inserting a transaction.

5

support(%)

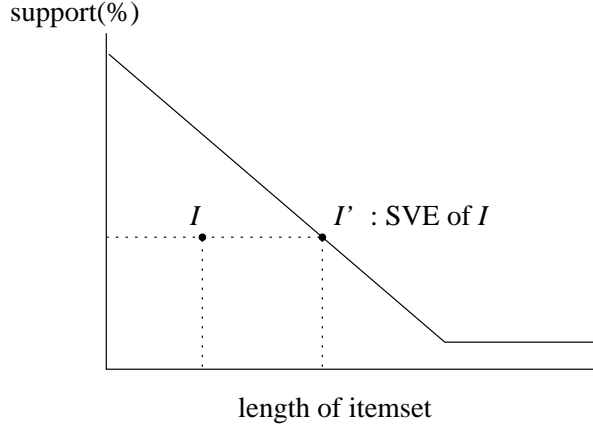I I' : SVE of I

length of itemset

Figure 4: Smallest valid extension (SVE)

4.2   Node Pruning, NP

The second pruning method focuses on pruning certain nodes of a conditional FP-tree, on which the next conditional pattern base is about to be generated. Let us consider a node v of the FP-tree. Let $I(v)$ be the item stored at this node, $\sigma(I(v))$ be the support of the item in the conditional pattern base, and $h(v)$ be the height of the longest path from the root through v to a leaf node. From the SVE property we know that the node v will contribute to a valid frequent itemset if and only if

$$h(v) + |CP| \geq f^{-1}(\sigma(I(v))) \tag{1}$$

where $|CP|$ is the length of conditional pattern of the current conditional FP-tree. The reason that equation (1) is correct is because, among the transactions that go through node v, the longest itemset that $I(v)$ can participate in has a length of $h(v)$. Now, if the support of $I(v)$ is small such that it requires an itemset whose length $f^{-1}(\sigma(I(v)))$ is greater than $h(v) + |CP|$, then that itemset cannot be supported by any of the transactions that go through node v. Thus, if equation (1) does not hold, node v can be pruned from the FP-tree. Once node v is pruned, then $\sigma(I(v))$ will decrease as well as the height of the nodes through v, possibly allowing further pruning. We will refer to this as the *node pruning* method, or *NP* for short.

A key observation to make is that both the TP and NP methods can be used together as each one of them prunes portions of the FP-tree that the other one does not. In particular, the NP methods can prune a node in a path that is longer than $f^{-1}(\sigma(CP)) - |CP|$, because the item of that node has lower support than CP. On the other hand, TP reduces the frequency of some itemsets in the FP-tree by removing entire short transactions. For example, consider two transactions; (A, B, C, D) and (A, B). Let's assume that $f^{-1}(\sigma(CP)) - |CP| = 4$, and each one of the items A,B,C,D has a support equal to that of CP. In that case, the NP will not remove any nodes, whereas TP will eliminate the second transaction.

In order to perform the node pruning, we need to compute the height of each node and then traverse each node v to see if it violates equation (1). If it does, then the node v can be pruned, the height of all the nodes whose longest path goes through v must be decremented by 1, and the support of $I(v)$ needs to be decremented to take account of the removal of v. Every time we make such changes in the tree, nodes that could not have been pruned before may now become eligible for pruning. In particular, all the rest of the nodes that have the same item $I(v)$ needs to be rechecked, as well as all the nodes whose height was decremented upon the removal of v. Our initial experiments with such an implementation showed that the cost of performing the pruning was quite often higher than the saving we achieved when used in conjunction with the TP scheme. For this reason we implemented an approximate but fast version of this scheme that achieves a comparable degree of pruning.

6

Our approximate NP algorithm initially sorts the transactions of the conditional pattern base in decreasing transaction length, then traverses each transaction in that order, and tries to insert them in the FP-tree. Let t be one such transaction and $l(t)$ be its length. When t is inserted into the FP-tree, it may share a prefix with some transactions already in the FP-tree. However, as soon as the insertion of t results in a new node being created, we check to see if we can prune it using equation (1). In particular, if v is that newly created node, then $h(v) = l(t)$, because the transactions are inserted into the FP-tree in decreasing length. Thus v can be pruned if

$$l(t) + |CP| < f^{-1}(\sigma(I(v))) \quad . \tag{2}$$

If that can be done, the new node is eliminated and the insertion of t continues to the next item. Now if one of the next items inserts a new node u, then that one may be pruned using equation (2). In equation (2), we use the original length of the transaction $l(t)$, not the length after the removal of the item previously pruned. The reason is that $l(t)$ is the correct upper bound of $h(u)$, because one of the transactions inserted later may have a length of at most $l(t)$, the same as the length of the current transaction, and can modify its height.

The above approach is approximate because (I) the elimination of a node affects only the nodes that can be eliminated in the subsequent transactions, not the nodes already in the tree; (II) we use pessimistic bounds on the height of a node (as discussed in the previous paragraph). This approximate approach, however, does not increase the complexity of generating the conditional FP-tree, beyond the sorting of the transactions in the conditional pattern base. Since the length of the transaction falls within a small range, they can be sorted using bucket sort in linear time.

4.3   Path Pruning, PP

Once the tree becomes a single path, the original FP-growth algorithm generates all possible combinations of items along the path and concatenates each of those combinations with its conditional pattern. If the path contains k items, there exist a total of 2^k such combinations. However, using the SVE property we can limit the number of combinations that we may need to consider.

Let $\{i_1, i_2, \ldots, i_k\}$ be the k items such that $\sigma(i_j) \geq \sigma(i_{j+1})$. One way of generating all possible 2^k combinations is to grow them incrementally as follows. First, we create two sets, one that contains i_1, and the other that does not. Next, for each of these sets, we generate two new sets such that, in each pair of them, one contains i_2 and the other does not, leading to four different sets. By continuing this process a total of k times, we will obtain all possible 2^k combinations of items. This approach essentially builds a binary tree with k levels of edges, in which the nodes correspond to the possible combinations. One such binary tree for $k = 4$ is shown in Figure 5.

To see how the SVE property can be used to prune certain subgraphs of this tree (and hence combinations to be explored), consider a particular internal node v of that tree. Let $h(v)$ be the height of the node (root has a height of zero), and let $\beta(v)$ be the number of edges that were one on the path from the root to v. In other words, $\beta(v)$ is the number of items that have been included so far in the set. Using the SVE property we can stop expanding the tree under node v if and only if

$$\beta(v) + (k - h(v)) + |CP| < f^{-1}(\sigma(I_{h(v)})) \quad .$$

Essentially, the above formula states that, based on the frequency of the current item, the set must have a sufficiently large number of items before it can be frequent. If the number of items that were already inserted in the set ($\beta(v)$) is small plus the number of items that are left for possible insertion ($k - h(v)$) is not sufficiently large, then no frequent itemsets can be generated from this branch of the tree, and hence it can be pruned. We will refer to this method as *path pruning* or *PP* for short.

The complexity of PP per one binary tree is $k \log |I|$ because we need to evaluate f^{-1} for k items. On the other hand, the original FP-growth algorithm has the complexity of $O(2^k)$ for one binary tree. The former is much smaller for large k. For small k, this analysis tells that PP may cost more than the saving. Our experimental result, however, suggests that the effect of pruning pays the price.
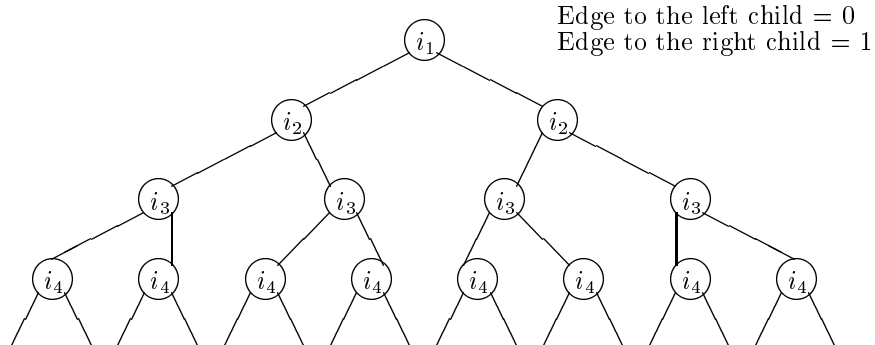
Edge to the left child = 0
Edge to the right child = 1

Figure 5: Binary tree when $k = 4$

5 Experimental Results

We experimentally evaluated the various search space pruning methods of LPMiner using a variety of datasets generated by the synthetic transaction generator that is provided by the IBM Quest group and was used in evaluating the Apriori algorithm [1]. All of our experiments were performed on Intel-based Linux workstations with Pentium III at 600MHz and 1GB of main memory. All the reported runtimes are in seconds.

We used two classes of datasets DS1 and DS2. Both of two classes of datasets contained 100K transactions. For each of the two classes we generated different problem instances, in which we varied the average size of the transactions from 3 items to 35 items for DS1, obtaining a total of 33 different datasets, DS1.3, ..., DS1.35, and from 3 items to 30 items for DS2, obtaining DS2.3, ..., DS2.30. For each problem instance in both of DS1.x and DS2.x, we set the average size of the maximal long itemset to be $x/2$, so as x increases, the dataset contains longer frequent itemsets. The difference between DS1.x and DS2.x is that each problem instance DS1.x consists of 1K items, whereas each problem instance DS2.x consists of 5K items. The characteristics of these datasets are summarized in Table 1.

parameter	DS1	DS2
$\|D\|$ Number of transactions	100K	100K
$\|T\|$ Average size of the transactions	3 to 35	3 to 30
$\|I\|$ Average size of the maximal potentially long itemsets	$\|T\|/2$	$\|T\|/2$
$\|L\|$ Number of maximal potentially large itemsets	10000	10000
N Number of items	1000	5000

Table 1: Parameters for datasets used in our tests

In all of our experiments, we used minimum support constraint that decreases linearly with the length of the frequent itemsets. In particular, for each of the DS1.x datasets, the initial value of support was set to 0.5 and it was decreased linearly down to 0.01 for itemsets up to length x. For the rest of the itemsets, the support was kept fixed at 0.01. The left graph of Figure 6 shows the shape of the support curve for DS1.20. In the case of the DS2 class of datasets, we used a similar approach to generate the constraint, however instead of using 0.01 as the minimum support, we used 0.005. The right graph of Figure 6 shows the shape of the support curve for DS2.20.

| | | LPMiner | | | | | | |
Dataset	FP-growth	NP	TP	PP	NP+TP	NP+PP	TP+PP	NP+TP+PP
DS1.3	3.664	3.559	3.695	3.672	3.614	3.598	3.706	3.572
DS1.4	4.837	3.816	4.423	4.828	3.764	3.871	4.407	3.775
DS1.5	7.454	5.035	6.361	7.467	4.904	4.993	6.369	4.865
DS1.6	11.164	6.813	8.810	11.149	6.324	6.829	8.813	6.421
DS1.7	15.316	8.778	11.827	15.329	8.065	8.798	11.842	8.051
DS1.8	22.079	12.153	15.666	22.065	10.701	12.155	15.630	10.667
DS1.9	28.122	15.260	19.676	28.025	13.519	15.245	19.695	13.559
DS1.10	40.427	21.369	25.035	40.387	18.322	21.291	25.038	18.342
DS1.11	49.420	25.276	29.291	49.583	22.320	25.767	29.805	22.178
DS1.12	71.091	32.806	35.726	70.920	27.886	32.648	35.595	27.874
DS1.13	86.639	38.489	41.226	86.282	32.921	38.271	41.203	32.805
DS1.14	130.604	47.867	48.314	125.701	40.552	47.590	48.261	40.389
DS1.15	155.171	54.868	54.903	154.612	46.734	54.727	54.839	47.934
DS1.16	255.528	67.794	68.522	253.890	56.468	67.161	64.066	60.442
DS1.17	289.600	73.841	70.428	285.373	63.333	77.307	70.126	61.611
DS1.18	409.961	85.851	80.079	404.513	71.296	84.641	79.170	71.043
DS1.19	488.898	95.666	89.101	483.596	79.276	94.794	88.480	78.827
DS1.20	730.399	113.983	105.252	711.947	93.823	110.499	101.096	89.358
DS1.21	856.614	125.378	117.470	837.304	102.944	122.580	114.886	100.077
DS1.22	1224.417	145.259	141.530	1180.976	117.607	137.180	133.186	109.376
DS1.23	1430.478	153.676	156.277	1385.205	124.548	150.661	151.419	121.270
DS1.24	1840.375	183.516	191.363	1739.318	142.728	174.060	184.608	134.174
DS1.25	2147.452	199.894	219.430	2038.823	155.002	193.338	210.911	148.172
DS1.26	3465.201	287.813	306.509	3134.160	212.427	226.667	259.939	166.956
DS1.27	3811.978	296.645	336.420	3479.318	217.086	253.775	302.121	185.205
DS1.28	7512.347	2142.169	1911.442	4646.935	1733.971	300.822	362.577	210.955
DS1.29	8150.431	1748.402	1552.467	5271.311	1288.414	337.896	412.495	233.016
DS1.30	8884.682	431.021	534.117	7370.503	338.811	397.331	489.129	266.111
DS1.31	9744.785	489.858	604.189	8073.919	347.581	447.265	568.864	302.462
DS1.32	31063.532	11001.177	8289.842	12143.147	7943.063	547.121	676.441	361.113
DS1.33	29965.612	4750.367	1789.832	14037.153	1423.910	615.470	760.411	408.505
DS1.34	51420.519	16214.516	10990.934	18027.933	10446.444	751.236	905.894	487.831
DS1.35	64473.916	11282.476	6828.611	21458.692	6426.131	856.127	1024.330	561.449

Table 2: Comparison of pruning methods using DS1

| | | LPMiner | | | | | | |
Dataset	FP-growth	NP	TP	PP	NP+TP	NP+PP	TP+PP	NP+TP+PP
DS2.3	11.698	11.436	12.708	12.680	13.392	11.579	11.354	11.277
DS2.4	16.238	15.178	15.060	16.558	15.768	14.762	15.219	14.243
DS2.5	20.230	16.781	17.701	20.406	16.627	16.712	17.516	17.004
DS2.6	33.859	21.293	22.972	33.719	20.705	21.411	23.700	20.691
DS2.7	42.712	23.419	27.253	43.554	22.864	23.583	26.654	23.009
DS2.8	71.215	29.089	33.553	70.947	26.878	28.848	33.619	26.846
DS2.9	90.909	30.675	38.187	89.857	29.446	30.496	38.669	29.732
DS2.10	146.919	37.372	47.848	147.161	34.559	37.153	47.757	35.100
DS2.11	181.040	40.243	54.862	182.041	38.316	39.713	55.119	37.986
DS2.12	275.834	47.299	66.480	274.819	43.653	46.978	66.040	43.281
DS2.13	329.967	49.697	75.979	329.018	47.775	49.714	76.343	47.594
DS2.14	475.752	58.445	90.502	471.671	53.758	56.396	88.981	52.975
DS2.15	542.815	62.627	104.307	539.249	60.567	61.607	103.873	60.503
DS2.16	812.486	80.111	125.099	798.523	72.078	77.162	122.502	70.391
DS2.17	936.694	85.838	142.994	926.153	80.798	84.775	140.097	78.362
DS2.18	1280.641	100.254	165.018	1252.841	93.058	91.616	160.608	86.791
DS2.19	1437.460	106.910	183.748	1409.567	99.812	101.294	181.314	97.464
DS2.20	2359.507	143.242	223.244	2282.950	125.456	116.602	207.559	112.240
DS2.21	2563.249	154.079	249.584	2483.045	135.950	130.427	234.743	125.072
DS2.22	3592.047	229.332	315.034	3388.120	186.411	150.000	267.465	139.401
DS2.23	3935.333	236.802	336.882	3725.836	191.559	166.241	300.465	156.602
DS2.24	5137.134	313.264	373.711	4676.638	208.681	186.624	336.514	173.389
DS2.25	5898.104	293.610	392.530	5424.018	208.909	208.689	375.901	194.778
DS2.26	12974.804	2297.732	2094.524	10022.341	1884.838	241.356	426.627	221.592
DS2.27	13411.080	2351.364	2053.704	10314.877	1823.076	263.164	466.550	241.366
DS2.28	-	8431.519	7149.525	-	6977.563	328.289	551.046	296.884
DS2.29	7980.772	6288.037	-	6050.794	334.178	581.189	299.912	
DS2.30	-	4564.717	2243.066	-	1905.217	367.330	639.672	322.922

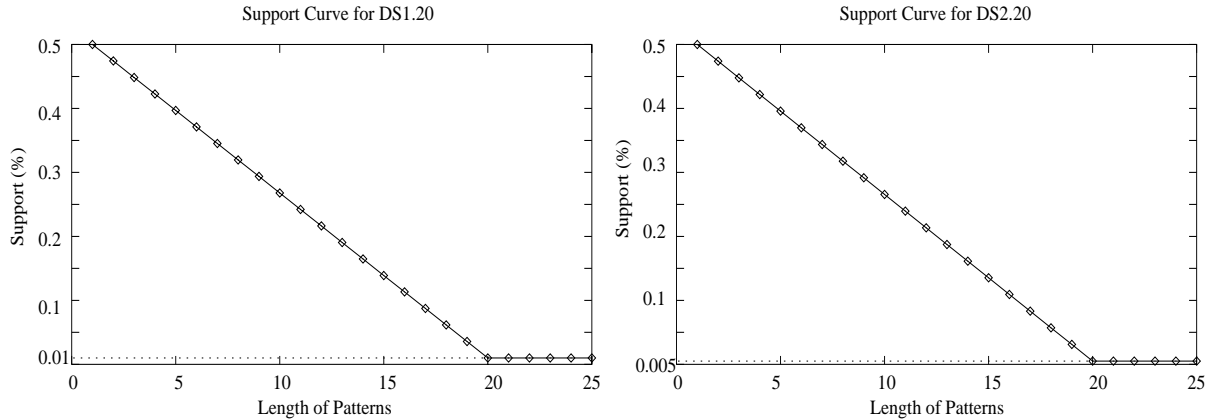Table 3: Comparison of pruning methods using DS2

Figure 6: Support curve for DS1.20 and DS2.20

5.1 Results

Tables 2 and 3 show the experimental results that we obtained for the DS1 and DS2 datasets, respectively. Each row of the tables shows the results obtained for a different DS1.x or DS2.x dataset, specified on the first column. The remaining columns show the amount of time required by different itemset discovery algorithms. The column labeled "FP-growth" shows the amount of time taken by the original FP-growth algorithm using a constant support constraint that corresponds to the smallest support of the support curve, 0.01 for DS1, and 0.005 for DS2. The columns under the heading "LPMiner" show the amount of time required by the proposed itemset discovery algorithm that uses the decreasing support curve to prune the search space. A total of seven different varieties of the LPMiner algorithm are presented, that are different combinations of the pruning methods described in Section 4. For example, the column label "NP" corresponds to the scheme that uses only node pruning (Section 4.2), whereas the column labeled "NP+TP+PP" corresponds to the scheme that uses all the three different schemes described in Section 4. Note that values with a "-" correspond to experiments that were aborted because they were taking too long time.

A number of interesting observations can be made from the results in these tables. First, either one of the LPMiner methods performs better than the FP-growth algorithm. In particular, the LPMiner that uses all three pruning methods does the best, requiring substantially smaller time than the FP-growth algorithm. For DS1, it is about 2.2 times faster for DS1.10, 8.2 times faster for DS1.20, 33.4 times faster for DS1.30, and 115 times faster for DS1.35. Similar trends can be observed for DS2, in which the performance of LPMiner is 4.2 times faster for DS2.10, 21.0 times faster for DS2.20, and 55.6 times faster for DS2.27.

Second, the performance gap between FP-growth and LPMiner increases as the length of the discovered frequent itemset increases (recall that, for both DS1.x and DS2.x, the length of the frequent itemsets increases with x). This is due to the fact that the overall itemset space that LPMiner can prune becomes larger, leading to improved relative performance.

Third, comparing the different pruning methods in isolation, we can see that NP and TP lead to the largest runtime reduction and PP achieves the smallest reduction. This is not surprising as PP can only prune itemsets during the late stages of itemset generation.

Finally, the runtime with three pruning methods increases gradually as the average length of the transactions (and the discovered itemsets) increases, whereas the runtime of the original FP-growth algorithm increases exponentially.

6   Conclusion

In this paper we presented an algorithm that can efficiently find all frequent itemsets that satisfy a length-decreasing support constraint. The key insight that enabled us to achieve high performance was the smallest valid extension property of the length decreasing support curve.

References

[1] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules. *VLDB 1994*, 487-499.

[2] R.C. Agarwal, C. Aggarwal, V.V.V. Prasad, and V. Crestana. A Tree Projection Algorithm for Generation of Large Itemsets for Association Rules. *IBM Research Report RC21341*, Nov, 1998.

[3] E. Cohen, M. Datar, S. Fujiwara, A. Gionis, P. Indyk, R. Motwani, J. D. Ullman, and C. Yang. Finding Interesting Associations without Support Pruning. *ICDE 2000*, 489-499.

[4] J. Han, J. Pei, and Y. Yin. Mining Frequent Patterns without Candidate Generation. *SIGMOD 2000*, 1-12.

[5] B. Liu, W. Hsu, Y. Ma. Mining association rules with multiple minimum supports. *SIGKDD 1999*, 125-134

[6] K. Wang, Y. He, and J. Han. Mining Frequent Itemsets Using Support Constraints. *VLDB 2000*, 43-52

[7] M. J. Zaki. Scalable algorithms for association mining. *IEEE Transactions on Knowledge and Data Engineering*, 12(3):372-390, 2000

[8] M. J. Zaki and C. Hsiao. CHARM: An efficient algorithm for closed association rule mining. *RPI Technical Report 99-10*, 1999.

[9] M. J. Zaki and K. Gouda. Fast Vertical Mining Using Diffsets. *RPI Technical Report 01-1*, 2001