

AFOPT: An Efficient Implementation of Pattern Growth Approach*

Guimei Liu Hongjun Lu
Department of Computer Science
Hong Kong University of
Science & Technology
Hong Kong, China
{cslgm, luhj}@cs.ust.hk

Jeffrey Xu Yu
Department of Systems Engineering and
Engineering Management
The Chinese University of Hong Kong
Hong Kong, China
{yu}@se.cuhk.edu.hk

Wei Wang Xiangye Xiao
Department of Computer Science
Hong Kong University of
Science & Technology
Hong Kong, China
{fervvac, xiaoxy}@cs.ust.hk

Abstract

In this paper, we revisit the frequent itemset mining (FIM) problem and focus on studying the pattern growth approach. Existing pattern growth algorithms differ in several dimensions: (1) item search order; (2) conditional database representation; (3) conditional database construction strategy; and (4) tree traversal strategy. They adopted different strategies on these dimensions. Several adaptive algorithms were proposed to try to find good strategies for general situations. In this paper, we described the implementation techniques of an adaptive pattern growth algorithm, called AFOPT, which demonstrated good performance on all tested datasets. We also extended the algorithm to mine closed and maximal frequent itemsets. Comprehensive experiments were conducted to demonstrate the efficiency of the proposed algorithms.

1 Introduction

Since the frequent itemset mining problem (FIM) was first addressed [2], a large number of FIM algorithms have been proposed. There is a pressing need to completely characterize and understand the algorithmic performance space of FIM problem so that we can choose and integrate the best strategies to achieve good performance in general cases.

Existing FIM algorithms can be classified into two categories: the candidate generate-and-test approach and the pattern growth approach. In each iteration of the candidate generate-and-test approach, pairs of frequent k -itemsets are joined to form candidate $(k+1)$ -itemsets, then the database is scanned to verify their supports. The resultant frequent $(k+1)$ -itemsets will be used as the input for next iteration. The drawbacks of this approach are: (1) it needs scan database multiple times, in worst case, equal to the maximal length of the frequent itemsets; (2) it needs generate lots of

candidate itemsets, many of which are proved to be infrequent after scanning the database; and (3) subset checking is a cost operation, especially when itemsets are very long. The pattern growth approach avoids the cost of generating and testing a large number of candidate itemsets by growing a frequent itemset from its prefix. It constructs a conditional database for each frequent itemset t such that all the itemsets that have t as prefix can be mined only using the conditional database of t .

The basic operations in the pattern growth approach are counting frequent items and new conditional databases construction. Therefore, the number of conditional databases constructed during the mining process, and the mining cost of each individual conditional database have a direct effect on the performance of a pattern growth algorithm. The total number of conditional databases mainly depends on in what order the search space is explored. The traversal cost and construct cost of a conditional database depends on the size, the representation format (tree-based or array-based) and construction strategy (physical or pseudo) of the conditional database. If the conditional databases are represented by tree structure, the traversal strategy of the tree structure also matters. In this paper, we investigate various aspects of the pattern growth approach, and try to find out what are good strategies for a pattern growth algorithm.

The rest of the paper is organized as follows: Section 2 revisits the FIM problem and introduces some related works; In Section 3, we describe an efficient pattern growth algorithm—AFOPT; Section 4 and Section 5 extend the AFOPT algorithm to mine frequent closed itemsets and maximal frequent itemsets respectively; Section 6 shows experiment results; finally, Section 7 concludes this paper.

2 Problem Revisit and Related Work

In this section, we first briefly review FIM problem and the candidate generate-and-test approach, then focus on studying the algorithmic performance space of the pattern growth approach.

*This work was partly supported by the Research Grant Council of the Hong Kong SAR, China (Grant HKUST6175/03E, CUHK4229/01E).

2.1 Problem revisit

Given a transactional database D , let I be the set of items appearing in it. Any combination of the items in I can be frequent in D , and they form the search space of FIM problem. The search space can be represented using set enumeration tree [14, 1, 4, 5, 7]. For example, given a set of items $I = \{a, b, c, d, e\}$ sorted in lexicographic order, the search space can be represented by a tree as shown in Figure 1. The root of the search space tree represents the empty set, and each node at level l (the root is at level 0, and its children are at level 1, and so on) represents an l -itemset. The *candidate extensions* of an itemset p is defined as the set of items after the last item of p . For example, items d and e are candidate extensions of ac , while b is not a candidate extension of ac because b is before c . The frequent extensions of p are those candidate extensions of p that can be appended to p to form a longer frequent itemset. In the rest of this paper, we will use $cand_exts(p)$ and $freq_exts(p)$ to denote the set of candidate extensions and frequent extensions of p respectively.

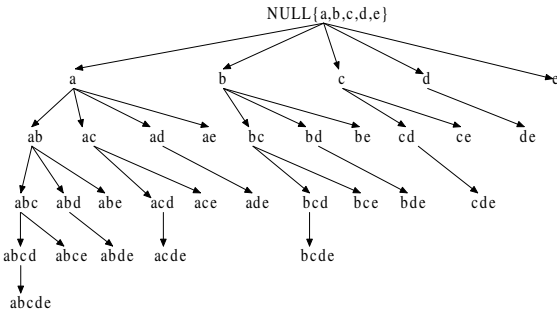


Figure 1. Search space tree

2.2 Candidate generate-and-test approach

Frequent itemset mining can be viewed as a set containment join between the transactional database and the search space of FIM. The candidate generate-and-test approach essentially uses block nested loop join, i.e. the search space is the inner relation and it is divided into blocks according to itemset length. Different from simple block nested loop join, in candidate generate-and-test approach the output of the previous pass is used as seeds to generate next block. For example, in the k -th pass of the Apriori algorithm, the transaction database and the candidate k -itemsets are joined to generate frequent k -itemsets. The frequent k -itemsets are then used to generate next block—candidate $(k+1)$ -itemsets. Given the large amount of memory available nowadays, it is a waste of memory to put only a single length of itemsets into memory. It is desirable to fully utilize available memory by putting some longer and possibly frequent itemsets into memory in earlier stage to reduce the number of database scans. The first FIM algorithm AIS [2] tries to estimate the frequencies of longer itemsets using the output of current pass, and includes those itemsets

that are estimated as frequent or themselves are not estimated as frequent but all of its subsets are frequent or estimated as frequent into next block. The problem with AIS algorithm is that it does not fully utilize the pruning power of the Apriori property, thus many unnecessary candidate itemsets are generated and tested. DIC algorithm [3] makes improvements based on Apriori algorithm. It starts counting the support of an itemset shortly after all the subsets of that itemset are determined to be frequent rather than wait until next pass. However, DIC algorithm cannot guarantee the full utilization of memory. The candidate generate-and-test approach faces a trade-off: on one hand, the memory is not fully utilized and it is desirable to put as many as possible candidate itemsets into memory to reduce the number of database scans; on the other hand, set containment test is a costly operation, putting itemsets into memory in earlier stage has the risk of counting support for many unnecessary candidate itemsets.

2.3 Pattern growth approach

The pattern growth approach adopts the divide-and-conquer methodology. The search space is divided into disjoint sub search spaces. For example, the search space shown in Figure 1 can be divided into 5 disjoint sub search spaces: (1) itemsets containing a ; (2) itemsets containing b but no a ; (3) itemsets containing c but no a, b ; (4) itemsets containing d but no a, b and c ; and (5) itemsets containing only e . Accordingly, the database is divided into 5 partitions, and each partition is called a conditional database. The conditional database of item i , denoted as D_i , includes all the transactions containing item i . All the items before i are eliminated from each transaction. All the frequent itemsets containing i can be mined from D_i without accessing other information. Each conditional database is divided recursively following the same procedure. The pattern growth approach not only reduces the number of database scans, but also avoids the costly set-containment-test operation.

Two basic operations in pattern growth approach are *counting frequent items* and *new conditional databases construction*. Therefore, the total number of conditional databases constructed and the mining cost of each individual conditional database are key factors that affect the performance of a pattern growth algorithm. The total number of conditional databases mainly depends on in what order the search space is explored. This order is called *item search order* in this paper. Some structures for representing conditional databases can also help reduce the total number of conditional databases. For example, if a conditional database is represented by tree-structure and there is only one branch, then all the frequent itemsets in the conditional database can be enumerated directly from the branch. There is no need to construct new conditional databases. The mining cost of a conditional database depends on the size, the

Datasets	Asc			Lex			Des		
	#cdb	time	max_mem	#cdb	time	max_mem	#cdb	time	max_mem
T10I4D100k (0.01%)	53688	4.52s	5199 kb	47799	4.89s	5471 kb	36725	5.32s	5675 kb
T40I10D100k (0.5%)	311999	30.42s	17206 kb	310295	33.83s	20011 kb	309895	43.37s	21980 kb
BMS-POS (0.05%)	115202	27.83s	17294 kb	53495	127.45s	38005 kb	39413	147.01s	40206 kb
BMS-WebView-1 (0.06%)	33186	0.69s	731 kb	65378	1.12s	901 kb	79571	2.16s	918 kb
chess (45%)	312202	2.68s	574 kb	617401	8.46s	1079 kb	405720	311.19s	2127 kb
connect-4 (75%)	12242	1.31s	38 kb	245663	2.65s	57 kb	266792	14.27s	113 kb
mushroom (5%)	9838	0.34s	1072 kb	258068	3.11s	676 kb	464903	272.30s	2304 kb
pumsb (70%)	272373	3.87s	383 kb	649096	12.22s	570 kb	469983	16.62s	1225 kb

Table 1. Comparison of Three Item Search Orders (Bucket Size=0)

representation and construction strategy of the conditional database. The traversal strategy also matters if the conditional database is represented using a tree-structure.

Item Search Order. When we divide the search space, all items are sorted in some order. This order is called *item search order*. The sub search space of an item contains all the items after it in *item search order* but no item before it. Two item search orders were proposed in literature: static lexicographic order and dynamic ascending frequency order. Static lexicographic order is to order the items lexicographically. It is a fixed order—all the sub search spaces use the same order. Tree projection algorithm [15] and H-Mine algorithm[12] adopted this order. Dynamic ascending frequency order reorders frequent items in every conditional database in ascending order of their frequencies. The most infrequent item is the first item, and all the other items are its candidate extensions. The most frequent item is the last item and it has no candidate extensions. FP-growth [6], AFOPt [9] and most of maximal frequent itemsets mining algorithms [7, 1, 4, 5] adopted this order.

The number of conditional databases constructed by an algorithm can differ greatly using different item search orders. Ascending frequency order is capable of minimizing the number and/or the size of conditional databases constructed in subsequent mining. Intuitively, an itemset with higher frequency will possibly have more frequent extensions than an itemset with lower frequency. We put the most infrequent item in front, though the candidate extension set is large, the frequent extension set cannot be very large. The frequencies of successive items increase, at the same time the size of candidate extension set decreases. Therefore we only need to build smaller and/or less conditional databases in subsequent mining. Table 1 shows the total number of conditional databases constructed (#cdb column), total running time and maximal memory usage when three orders are adopted in the framework of AFOPt algorithm described in this paper. The three item search orders compared are: dynamic ascending frequency order (Asc column), lexicographic order (Lex column) and dynamic descending frequency order (Des column). The minimum support threshold on each dataset is shown in the first column. On the

first three datasets, ascending frequency order needs to build more conditional databases than the other two orders, but its total running time and maximal memory usage is less than the other two orders. It implies that the conditional databases constructed using ascending frequency order are smaller. On the remaining datasets, ascending frequency order requires to build less conditional databases and needs less running time and maximal memory usage, especially on dense datasets connect-4 and mushroom.

Agrawal et al proposed an efficient support counting technique, called *bucket counting*, to reduce the total number of conditional databases[1]. The basic idea is that if the number of items in a conditional database is small enough, we can maintain a counter for every combination of the items instead of constructing a conditional database for each frequent item. The bucket counting can be implemented very efficiently compared with conditional database construction and traversal operation.

Conditional Database Representation. The traversal and construction cost of a conditional database heavily depends on its representation. Different data structures have been proposed to store conditional databases, e.g. tree-based structures such as FP-tree [6] and AFOPt-tree [9], and array-based structure such as Hyper-structure [12]. Tree-based structures are capable of reducing traversal cost because duplicated transactions can be merged and different transactions can share the storage of their prefixes. But they incur high construction cost especially when the dataset is sparse and large. Array-based structures incur little construction cost but they need much more traversal cost because the traversal cost of different transactions cannot be shared. It is a trade-off in choosing tree-based structures or array-based structures. In general, tree-based structures are suitable for dense databases because there can be lots of prefix sharing among transactions, and array-based structures are suitable for sparse databases.

Conditional Database Construction Strategy Constructing every conditional database physically can be expensive especially when successive conditional databases do not shrink much. An alternative is to pseudo-construct them, i.e. using pointers pointing to transactions in upper

Algorithms	Item Search Order	CondDB Format	CondDB Construction	Tree Traversal
Tree-Projection [15]	static lexicographic	array	adaptive	-
FP-growth [6]	dynamic frequency	FP-tree	physical	bottom-up
H-mine [12]	static lexicographic	hyper-structure	pseudo	-
OP [10]	adaptive	adaptive	adaptive	bottom-up
PP-mine [17]	static lexicographic	PP-tree	pseudo	top-down
AFOPT [9]	dynamic frequency	adaptive	physical	top-down
CLOSET+ [16]	dynamic frequency	FP-tree	adaptive	adaptive

Table 2. Pattern Growth Algorithms

level conditional databases. However, pseudo-construction cannot reduce traversal cost as effectively as physical construction. The item ascending frequency search order can make the subsequent conditional databases shrink rapidly, consequently it is beneficial to use physical construction strategy with item ascending frequency order together.

Tree Traversal Strategy The traversal cost of a tree is minimal using top-down traversal strategy. FP-growth algorithm [6] uses ascending frequency order to explore the search space, while FP-tree is constructed according to descending frequency order. Hence FP-tree has to be traversed using bottom-up strategy. As a result, FP-tree has to maintain parent links and node links at each node for bottom-up traversal, which increases the construction cost of the tree. AFOPT algorithm [9] uses ascending frequency order both for search space exploration and prefix-tree construction, so it can use the top-down traversal strategy and do not need to maintain additional pointers at each node. The advantage of FP-tree is that it can be more compact than AFOPT-tree because descending frequency order increases the possibility of prefix sharing. The ascending frequency order adopted by AFOPT may lead to many single branches in the tree. This problem was alleviated by using arrays to store single branches in AFOPT-tree.

Existing pattern growth algorithms mainly differ in the several dimensions aforementioned. Table 2 lists existing pattern growth algorithms and their strategies on four dimensions. AFOPT [9] is an efficient FIM algorithm developed by our group. We will discuss its technical details in next three sections.

3 Mining All Frequent Itemsets

We discussed several trade-offs faced by a pattern growth algorithm in last section. Some implications from above discussions are: (1) Use tree structure on dense database and use array structure on sparse database. (2) Use dynamic ascending frequency order on dense databases and/or when minimum support threshold is low. It can dramatically reduce the number and/or the size of the successive conditional databases. (3) If dynamic ascending frequency order is adopted, then use physical construction strategy because the size of conditional databases will shrink quickly. In this section, we describe our algorithm AFOPT which takes the

above three implications into consideration. The distinct features of our AFOPT algorithm include: (1) It uses three different structures to represent conditional databases: arrays for sparse conditional databases, AFOPT-tree for dense conditional databases, and buckets for counting frequent itemsets containing only top- k frequent items, where k is a parameter to control the number of buckets used. Several parameters are introduced to control when to use arrays or AFOPT-tree. (2) It adopts the dynamic ascending frequency order. (3) The conditional databases are constructed physically on all levels no matter whether the conditional databases are represented by AFOPT-tree or arrays.

3.1 Framework

Given a transactional database D and a minimum support threshold, AFOPT algorithm scans the original database twice to mine all frequent itemsets. In the first scan, all frequent items in D are counted and sorted in ascending order of their frequencies, denoted as $F = \{i_1, i_2, \dots, i_m\}$. We perform another database scan to construct a *conditional database* for each $i_j \in F$, denoted as D_{i_j} . During the second scan, infrequent items in each transaction t are removed and the remaining items are sorted according to their orders in F . Transaction t is put into D_{i_j} if the first item of t after sorting is i_j . The remaining mining will be performed on conditional databases only. There is no need to access the original database.

We first perform mining on D_{i_1} to mine all the itemsets containing i_1 . Mining on individual conditional database follows the same process as mining on the original database. After the mining on D_{i_1} is finished, D_{i_1} can be discarded. Because D_{i_1} also contains other items, the transactions in it will be inserted into the remaining conditional databases. Given a transaction t in D_{i_1} , suppose the next item after i_1 in t is i_j , then t will be inserted into D_{i_j} . This step is called push-right. Sorting the items in ascending order of their frequencies ensures that every time, a small conditional database is pushed right. The pseudo-code of AFOPT-all algorithm is shown in Algorithm 1.

3.2 Conditional database representation

Algorithm 1 is independent of the representation of conditional databases. We choose proper representations ac-

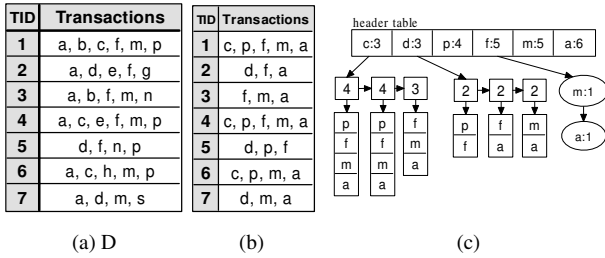
Algorithm 1 AFOPT-all Algorithm

Input:

p is a frequent itemset
 D_p is the conditional database of p
 min_sup is the minimum support threshold;

Description:

- 1: Scan D_p count frequent items, $F=\{i_1, i_2, \dots, i_n\}$;
 - 2: Sort items in F in ascending order of their frequencies;
 - 3: **for all** item $i \in F$ **do**
 - 4: $D_p \cup \{i\} = \phi$;
 - 5: **for all** transaction $t \in D_p$ **do**
 - 6: remove infrequent items from t , and sort remaining items according to their orders in F ;
 - 7: let i be the first item of t , insert t into $D_p \cup \{i\}$.
 - 8: **for all** item $i \in F$ **do**
 - 9: Output $s = p \cup \{i\}$;
 - 10: AFOPT-all(s, D_s, min_sup);
 - 11: PushRight(D_s);
-


Figure 2. Conditional DB Representation

according to the density of conditional databases. Three structures are used: (1) array, (2) AFOPT-tree, and (3) buckets. As aforementioned, these three structures are suitable for different situations. Bucket counting technique is proper and extremely efficient when the number of distinct frequent items is around 10. Tree structure is beneficial when conditional databases are dense. Array structure is favorable when conditional databases are sparse. We use four parameters to control when to use these three structures as follows: (1) frequent itemsets containing only top- $bucket_size$ frequent items are counted using buckets; (2) if the minimum support threshold is greater than $tree_min_sup$ or average support of all frequent items is no less than $tree_avg_sup$, then all the rest conditional databases are represented using AFOPT-tree; otherwise (3) the conditional databases of the next $tree_alphabet_size$ most frequent items are represented using AFOPT-tree, and the rest conditional databases are represented using arrays.

Figure 2 shows a transactional database D and the initial conditional databases constructed with $min_sup=40\%$. There are 6 frequent items $\{c:3, d:3, p:4, f:5, m:5, a:6\}$. Figure 2(b) shows the projected database after removing infrequent items and sorting. The values of the parameters for conditional database construction are set as follows: $bucket_size=2$, $tree_alphabet_size=2$, $tree_min_sup=50\%$, $tree_avg_sup=60\%$. The frequent itemsets containing only m and a are counted using buck-

ets of size 4 ($=2^{bucket_size}$). The conditional databases of f and p are represented by AFOPT-tree. The conditional databases of item c and d are represented using arrays. From our experience, the $bucket_size$ parameter can choose a value around 10. A value between 20 and 200 will be safe for $tree_alphabet_size$ parameter. We set $tree_min_sup$ to 5% and $tree_avg_sup$ to 10% in our experiments.

Table 3 shows the size, construction time (build column) and push-right time if applicable, of the initial structure constructed from original database by AFOPT, H-Mine and FP-growth algorithms. We set $bucket_size$ to 8 and $tree_alphabet_size$ to 20 for AFOPT algorithm. The initial structure of AFOPT includes all three structures. The array structure in AFOPT algorithm simply stores all items in a transaction. Each node in hyper-structure stores three pieces of information: an item, a pointer pointing to the next item in the same transaction and a pointer pointing to the same item in another transaction. Therefore the size of hyper-structure is approximately 3 times larger than the array structure used in AFOPT. A node in AFOPT-tree maintains only a child pointer and a sibling pointer, while a FP-tree node maintains two more pointers for bottom-up traversal: a parent pointer and a node link. AFOPT consumes the least amount of space on almost all tested datasets.

4 Mining Frequent Closed Itemsets

The complete set of frequent itemsets can be very large. It has been shown that it contains many redundant information [11, 18]. Some works [11, 18, 13, 16, 8] put efforts on mining frequent closed itemsets to reduce output size. *An itemset is closed if all of its supersets have a lower support than it.* The set of frequent closed itemsets is the minimum informative set of frequent itemsets. In this section, we describe how to extend Algorithm 1 to mine only frequent closed itemsets. For more details, please refer to [8].

4.1 Removing non-closed itemsets

Non-closed itemsets can be removed either in a postprocessing phase, or during mining process. The second strategy can help avoid unnecessary mining cost. Non-closed frequent itemsets are removed based on the following two lemmas (see [8] for proof of these two lemmas).

Lemma 1 *In Algorithm 1, an itemset p is closed if and only if two conditions hold: (1) no existing frequent itemsets is a superset of p and is as frequent as p ; (2) all the items in D_p have a lower support than p .*

Lemma 2 *In Algorithm 1, if a frequent itemset p is not closed because condition (1) in Lemma 1 does not hold, then none of the itemsets mined from D_p can be closed.*

We check whether there exists q such that $p \subset q$ and $sup(p)=sup(q)$ before mining D_p . If such q exists, then there is no need to mine D_p based on Lemma 2 (line 10).

Datasets	AFOPT			H-Mine			FP-growth	
	size	build	pushright	Size	build	pushright	size	build
T10I4D100k (0.01%)	5116 kb	0.55s	0.37s	11838 kb	0.68s	0.19s	20403 kb	1.83s
T40I10D100k (0.5%)	16535 kb	1.85s	1.91s	46089 kb	2.10s	1.42s	104272 kb	6.16s
BMS-POS (0.05%)	17264 kb	2.11s	1.43s	38833 kb	2.58s	1.00s	47376 kb	6.64s
BMS-WebView-1 (0.06%)	711 kb	0.12s	0.01s	1736 kb	0.17s	0.01s	1682 kb	0.27s
chess (45%)	563 kb	0.04s	0.01s	1150 kb	0.05s	0.03s	1339 kb	0.12s
connect-4 (75%)	35 kb	0.73s	0.01s	22064 kb	1.15s	0.55s	92 kb	2.08s
mushroom (5%)	1067 kb	0.08s	0.04s	2120 kb	0.10s	0.03s	988 kb	0.17s
pumsb (70%)	375 kb	0.82s	0.02s	17374 kb	1.15s	0.43s	1456 kb	2.26s

Table 3. Comparison of Initial Structures

Thus the identification of a non-closed itemsets not only reduces output size, but also avoids unnecessary mining cost. Based on pruning condition (2) in Lemma 1, we can check whether an item $i \in F$ appears in every transaction of D_p . If such i exists, then there is no need to consider the frequent itemsets that do not contain i when mining D_p . In other words, we can directly perform mining on $D_p \cup \{i\}$ instead of D_p (line 3-4). The efforts for mining $D_p \cup \{j\}$, $j \neq i$ are saved. The pseudo-code for mining frequent closed itemsets is shown in Algorithm 2.

Algorithm 2 AFOPT-close Algorithm

Input:

- p is a frequent itemset
- D_p is the conditional database of p
- min_sup is the minimum support threshold;

Description:

- 1: Scan D_p count frequent items, $F = \{i_1, i_2, \dots, i_n\}$;
- 2: Sort items in F in ascending order of their frequencies;
- 3: $I = \{i | i \in F \text{ and } support(p \cup \{i\}) = support(p)\}$;
- 4: $F = F - I$; $p = p \cup I$;
- 5: **for all** transaction $t \in D_p$ **do**
- 6: remove infrequent items from t , and sort remaining items according to their orders in F ;
- 7: let i be the first item of t , insert t into $D_p \cup \{i\}$;
- 8: **for all** item $i \in F$ **do**
- 9: $s = p \cup \{i\}$;
- 10: **if** s is closed **then**
- 11: Output s ;
- 12: AFOPT-close(s, D_s, min_sup);
- 13: PushRight(D_s);

CFP-tree to check whether an itemset is closed. An example of CFP-tree is shown in Figure 3(b) which stores all the frequent closed itemsets in Figure 3(a). They are mined from the database shown in Figure 2(a) with support 40%.

Each CFP-tree node is a variable-length array, and all the items in the same node are sorted in ascending order of their frequencies. A path in the tree starting from an entry in the root node represents a frequent itemset. The CFP-tree has two properties: *the left containment property* and *the Apriori property*. The *Apriori Property* is that the support of any child of a CFP-tree entry cannot be greater than the support of that entry. The *Left Containment Property* is that the item of an entry E can only appear in the subtrees pointed by entries before E or in E itself. The superset of an itemset p with support s can be efficiently searched in the CFP-tree based on these two properties. The apriori property can be exploited to prune subtrees pointed by entries with support less than s . The left containment property can be utilized to prune subtrees that do not contain all items in p . We also maintain a hash-bitmap in each entry to indicate whether an item appears in the subtree pointed by that entry to further reduce searching cost. The superset search algorithm is shown in Algorithm 3. BinarySearch($cnode, s$) returns the first entry in a CFP-tree node with support no less than s . Algorithm 3 do not require the whole CFP-tree to be in main memory because it is also very efficient on disk. Moreover, the CFP-tree structure is a compact representation of the frequent closed itemsets, so it has a higher chance to be held in memory than flat representation.

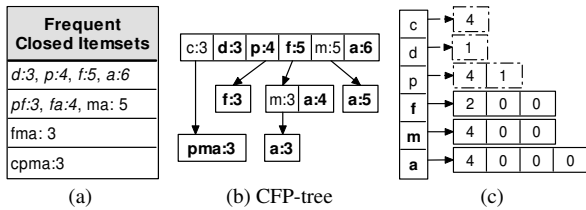


Figure 3. CFP-tree and Two-layer Hash Map

4.2 Closed itemset checking

During the mining process, we store all existing frequent closed itemsets in a tree structure, called Condensed Frequent Pattern tree or CFP-tree for short [8]. We use the

Although searching in CFP-tree is very efficient, it is still costly when CFP-tree is large. Inspired by the two-layer structure adopted by CLOSET+ algorithm[16] for subset checking, we use a two-layer hash map to check whether an itemset is closed before searching in CFP-tree. The two-layer hash map is shown in Figure 3(c). We maintain a hash map for each item. The hash map of item i is denoted by $i.hashmap$. The length of the hash map of an item i is set to $\min\{sup(i)-min_sup, max_hashmap_len\}$, where $max_hashmap_len$ is a parameter to control the maximal size of the hash maps and $min_sup = \min\{sup(i) | i \text{ is frequent}\}$. Given an itemset

Algorithm 3 Search_Superset Algorithm

Input:

l is a frequent itemset
 $cnode$ the CFP+-tree node pointed by l
 s is the minimum support threshold
 I is a set of items to be contained in the superset

Description:

```
1: if  $I = \phi$  then
2:   return true;
3:  $\bar{E}$  = the first entry of  $cnode$  such that  $\bar{E}.item \in I$ ;
4:  $E'$  = BinarySearch( $cnode, s$ );
5: for all entry  $E \in cnode, E$  between  $E'$  and  $\bar{E}$  do
6:    $l' = l \cup \{E.item\}$ ;
7:   if  $E.child \neq \text{NULL}$  AND all items in  $I - \{E.item\}$  are in
       $E.subtree$  then
8:     found = Search_Superset( $l', E.child, s, I - \{E.item\}$ );
9:     if found then
10:      return true;
11:   else if  $I - \{E.item\} = \phi$  then
12:     return true;
13: return false;
```

$p = \{i_1, i_2, \dots, i_l\}$, p is mapped to $i_j.hashmap[(sup(p) - min_sup) \% max_hashmap_len]$, $j = 1, 2, \dots, l$. An entry in a hash map records the maximal length of the itemsets mapped to it. For example, itemset $\{c, p, m, a\}$ set the first entry of $c.hashmap$, $p.hashmap$, $m.hashmap$ and $a.hashmap$ to 4. Figure 3(c) shows the status of the two-layer hash map before mining D_f . An itemset p must be closed if any of the entry it mapped to contains a lower value than its length. In such cases there is no need to search in CFP-tree. The hash map of an item i can be released after all the frequent itemsets containing i are mined because they will not be used in later mining. For example, when mining D_f , the hash map of items c , d and p can be deleted.

5 Mining Maximal Frequent Itemsets

The problem of mining maximal frequent itemsets can be viewed as given a minimum support threshold min_sup , finding a border through the search space tree such that all the nodes below the border are infrequent and all the nodes above the border are frequent. The goal of maximal frequent itemsets mining is to find the border by counting support for as less as possible itemsets. Existing maximal algorithms [19, 7, 1, 4, 5] adopted various pruning techniques to reduce the search space to be explored.

5.1 Pruning techniques

The most effective techniques are based on the following two lemmas to prune a whole branch from search space tree.

Lemma 3 *Given a frequent itemset p , if $p \cup cand_exts(p)$ is frequent but not maximal, then none of the frequent itemsets mined from D_p and from p 's right sibling's conditional databases can be maximal because all of them are subsets of $p \cup cand_exts(p)$.*

Lemma 4 *Given a frequent itemset p , if $p \cup freq_exts(p)$ is frequent but not maximal, then none of the frequent itemsets mined from D_p can be maximal because all of them are subsets of $p \cup cand_exts(p)$.*

Based on Lemma 3, before mining D_p , we can first check whether $p \cup cand_exts(p)$ is frequent but not maximal. This can be done by two techniques.

Superset Pruning Technique: It is to check whether there exists some maximal frequent itemset such that it is a superset of $p \cup cand_exts(p)$. Like frequent closed itemset mining, subset checking can be challenging when the number of maximal itemsets is large. We will discuss this issue in next subsection.

Lookahead Technique: It is to check whether $p \cup cand_exts(p)$ is frequent when count frequent items in D_p . If D_p is represented by AFOPT-tree, the lookahead operation can be accomplished by simply looking at the left-most branch of AFOPT-tree. If $p \cup cand_exts(p)$ is frequent, then the length of the left-most branch is equal to $|cand_exts(p)|$, and the support of the leaf node of the left-most branch is no less than min_sup .

If the superset pruning technique and lookahead technique fail, then based on Lemma 4 we can use superset pruning technique to check whether $p \cup freq_exts(p)$ is frequent but not maximal. Two other techniques are adopted in our algorithm.

Excluding items appearing in every transaction of D_p from subsequent mining: Like frequent closed itemset mining, if an item i appears in every transaction of D_p , then a frequent itemset q mined from D_p and not containing i cannot be maximal because $q \cup \{i\}$ is frequent.

Single Path Trimming: If D_p is represented by AFOPT-tree and it has only one child i , then we can append i to p and remove it from subsequent mining.

5.2 Subset checking

When do superset pruning, to check against all frequent maximal itemsets can be costly when the number of maximal itemsets is large. Zaki et. al proposed a progressive focusing technique for subset checking [5]. The observation behind the progressive focusing technique is that only the maximal frequent itemsets containing p can be a superset of $p \cup cand_exts(p)$ or $p \cup freq_exts(p)$. The set of maximal frequent itemsets containing p is called the local maximal frequent itemsets with respect to p , denoted as $LMFI_p$. When check whether $p \cup cand_exts(p)$ or $p \cup freq_exts(p)$ is a subset of some existing maximal frequent itemsets, we only need to check them against $LMFI_p$. The frequent itemsets in $LMFI_p$ can either come from p 's parent's LMFI, or from p 's left-siblings' LMFI. The construction of LMFI is very similar to the construction of conditional databases. The construction consists of two steps: (1) projecting: after all frequent items F in D_p

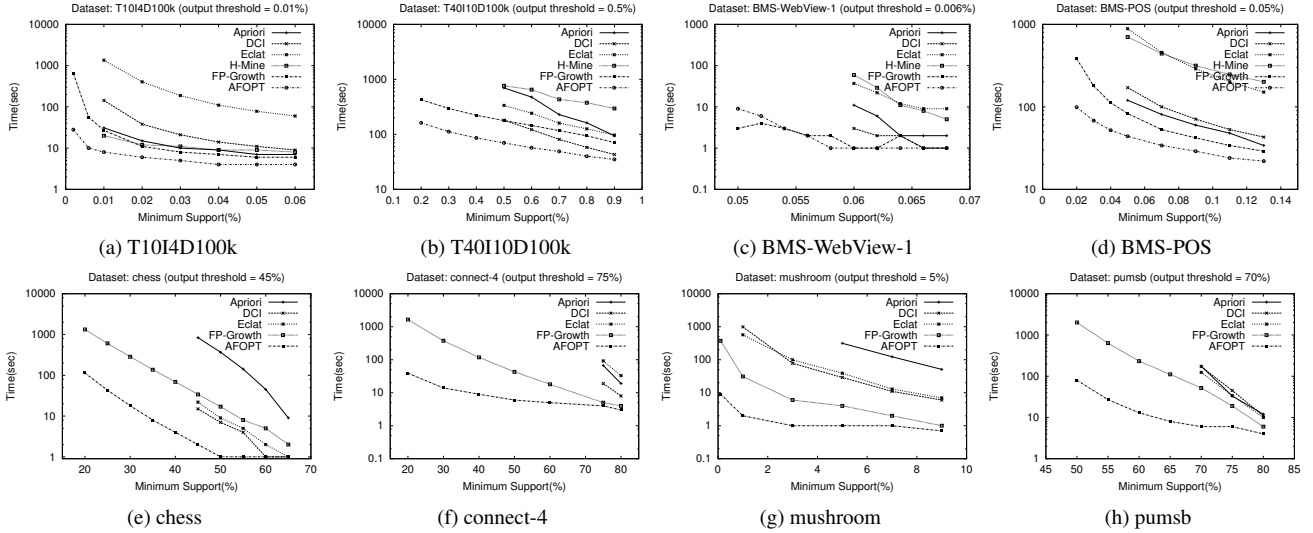


Figure 4. Performance Comparison of FI Mining Algorithms

Data Sets	Size	#Trans	#Items	MaxTL	AvgTL
T10I4D100k (0.01%)	3.93M	100000	870	30	10.10
T40I10D100k (0.5%)	15.12M	100000	942	78	39.61
BMS-POS (0.05%)	11.62MB	515597	1657	165	6.53
BMS-WebView-1 (0.06%)	1.28M	59601	497	267	2.51
chess (45%)	0.34M	3196	75	37	37.00
connect-4 (75%)	9.11M	67557	129	43	43.00
mushroom (5%)	0.56M	8124	119	23	23.00
pumsb (70%)	16.30M	49046	2113	74	74.00

Table 4. Datasets

are counted, $\forall s \in \text{LMFI}_p$, s is put into $\text{LMFI}_p \cup \{i\}$, where i is the first item in F appears in s ; (2) push-right: after all the maximal frequent itemsets containing p are mined, $\forall s \in \text{LMFI}_p$, s is put into LMFI_q if q is the first right sibling of p containing an item in s . In our implementation, we use pseudo projection technique to generate LMFIs, i.e. LMFI_p is a collection of pointers pointing to those maximal itemsets containing p .

6 Experimental results

In this section, we compare the performance of our algorithms with other FIM algorithms. All the experiments were conducted on a 1Ghz Pentium III with 256MB memory running Mandrake Linux.

Table 4 shows some statistical information about the datasets used for performance study. All the datasets were downloaded from FIMI'03 workshop web site. The fifth and sixth columns are maximal and average transaction length. These statistics provide some rough description of the density of the datasets.

6.1 Mining all frequent itemsets

We compared the efficiency of AF OPT-all algorithm with Apriori, DCI, FP-growth, H-Mine and Eclat algo-

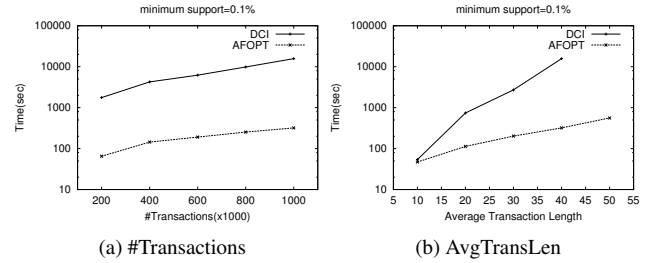


Figure 5. Scalability Study

rithms. The Apriori and Eclat algorithms we used are implemented by Christian Borgelt. DCI was downloaded from its web site. We obtained the source code of FP-growth from its authors. H-Mine was implemented by ourselves. We ran H-Mine only on several sparse datasets since it was designed for sparse datasets and it changes to use FP-tree on dense datasets. Figure 4 shows the running time of all algorithms over datasets shown in Table 4. When the minimum support threshold is very low, an intolerable number of frequent itemsets can be generated. So when minimum support threshold reached some very low value, we turned off the output. This minimum support value is called **output threshold**, and they are shown on top of each figure.

With high minimum support threshold, all algorithms showed comparable performance. When minimum support threshold was lowered, the gaps between algorithms increased. The two candidate generate-and-test approaches, Apriori and DCI, showed satisfactory performance on several sparse datasets, but took thousands of seconds to terminate on dense datasets due to high cost for generating and testing a large number of candidate itemsets. H-Mine demonstrated similar performance with FP-growth on dataset T10I4D100k, but it was slower than FP-growth on the other three sparse datasets. H-Mine uses pseudo con-

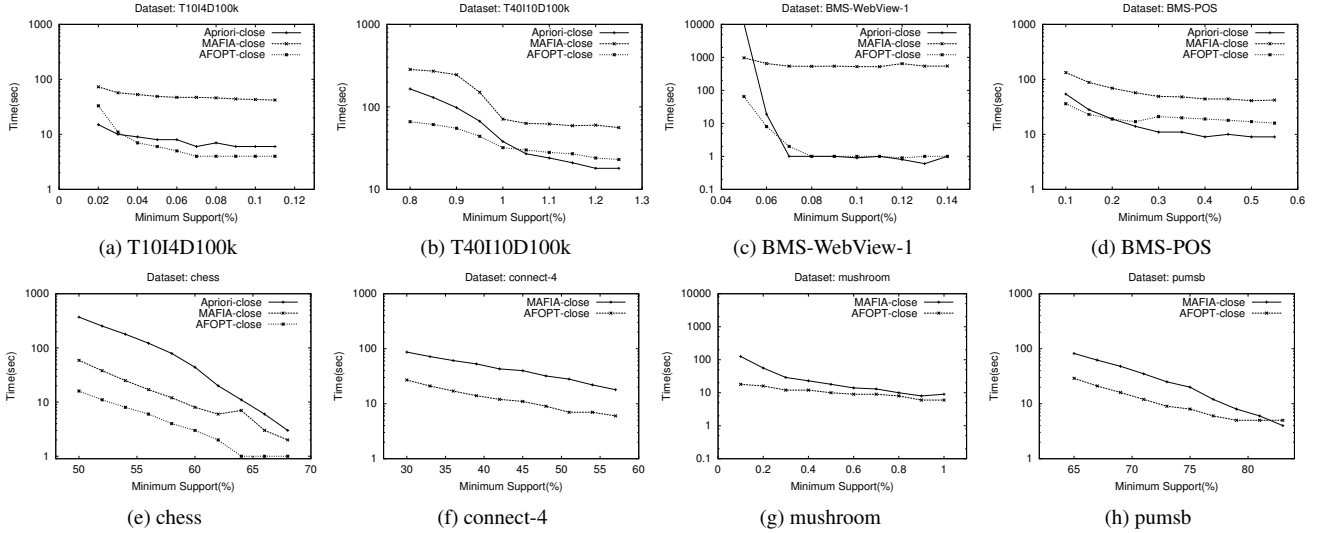


Figure 6. Performance Comparison of FCI Mining Algorithms

struction strategy, which cannot reduce traversal cost as effective as physical construction strategy. Eclat uses vertical mining techniques. Support counting is performed efficiently by transaction id list join. But Eclat is not scale well with respect to the number of transactions in a database. The running time of AFOPT-all was rather stable over all tested datasets, and it outperformed other algorithms.

6.2 Scalability

We studied the scalability of our algorithm by perturbing the IBM synthetic data generator along two dimensions: the number of transactions was varied from 200k to 1000k and the average transaction length was varied from 10 to 50. The default values of these two parameters were set to 1000k and 40 respectively. We compared our algorithm with algorithm DCI. Other algorithms took long time to finish on large datasets, so we exclude them from comparison. Figure 5 shows the results when varying the two parameters.

6.3 Mining frequent closed itemsets

We compared AFOPT-close with MAFIA [4] and Apriori algorithms. Both algorithms have an option to generate only closed itemsets. We denoted these two algorithms as Apriori-close and MAFIA-close respectively in figures. MAFIA was downloaded from its web site. We compared with Apriori-close only on sparse datasets because Apriori-close requires a very long time to terminate on dense datasets. On several sparse datasets, AFOPT-close and Apriori-close showed comparable performance. Both of them were orders of magnitude faster than MAFIA-close. MAFIA-close uses vertical mining technique. It uses bitmaps to represent tid lists. AFOPT-close showed better performance on tested dense datasets due to its adaptive nature and the efficient subset checking techniques described in Section 4. On dense datasets, AFOPT-close uses tree

structure to store conditional databases. The tree structure has apparent advantages on dense datasets because many transactions share their prefixes.

6.4 Mining maximal frequent itemsets

We compared AFOPT-max with MAFIA and Apriori algorithms. The Apriori algorithm also has an option to produce only maximal frequent itemsets. It is denoted as “Apriori-max” in figures. Again we only compare with it on sparse datasets. Apriori-max explores the search space in breadth-first order. It finds short frequent itemsets first. Maximal frequent itemsets are generated in a post-processing phase. Therefore Apriori-max is infeasible when the number of frequent itemsets is large even if it adopts some pruning techniques during the mining process. AFOPT-max and MAFIA generate frequent itemsets in depth-first order. Long frequent itemsets are mined first. All the subsets of a long maximal frequent itemsets can be pruned from further consideration by using the superset pruning and lookahead technique. AFOPT-max uses tree structure to represent dense conditional databases. The AFOPT-tree introduces more pruning capability than tid list or tid bitmap. For example, if a conditional database can be represented by a single branch in AFOPT-tree, then the single branch will be the only one possible maximal itemset in the conditional database. AFOPT-max also benefits from the progressive focusing technique for superset pruning. MAFIA was very efficient on small datasets, e.g chess and mushroom when the length of bitmap is short.

7 Conclusions

In this paper, we revisited the frequent itemset mining problem and focused on investigating the algorithmic performance space of the pattern growth approach. We identified four dimensions in which existing pattern growth al-

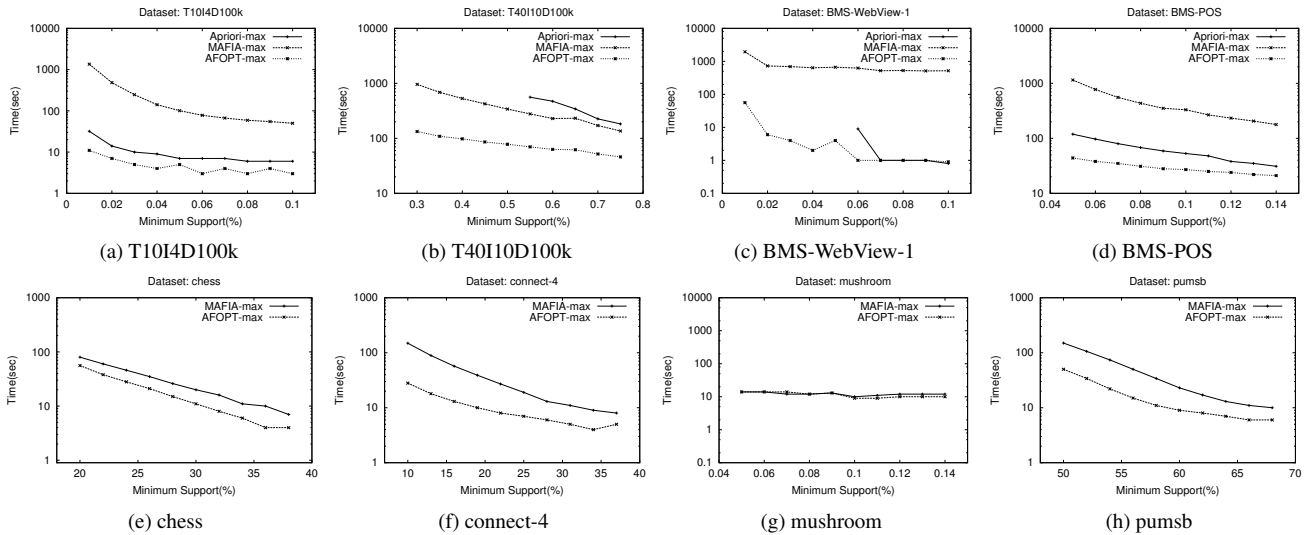


Figure 7. Performance Comparison of MFI Mining Algorithms

gorithms differ: (1) item search order: static lexicographical order or ascending frequency order; (2) conditional database representation: tree-based structure or array-based structure; (3) conditional database construction strategy: physical construction or pseudo construction; and (4) tree traversal strategy: bottom-up or top-down. Existing algorithms adopted different strategies on these four dimensions in order to reduce the total number of conditional databases and the mining cost of each individual conditional database.

we described an efficient pattern growth algorithm AFOPT in the paper. It adaptively uses three different structures: arrays, AFOPT-tree and buckets, to represent conditional databases according to the density of a conditional database. Several parameters were introduced to control which structure should be used for a specific conditional database. We showed that the adaptive conditional database representation strategy requires less space than using array-based structure or tree-based structure solely. We also extended AFOPT algorithm to mine closed and maximal frequent itemsets, and described how to incorporate pruning techniques into AFOPT framework. Efficient subset checking techniques for both closed and maximal frequent itemsets mining were presented. A set of experiments were conducted to show the efficiency of the proposed algorithms.

References

- [1] R. Agrawal, C. Aggarwal, and V. Prasad. Depth first generation of long patterns. In *SIGKDD*, 2000.
- [2] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *SIGMOD*, 1993.
- [3] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *SIGMOD*, 1997.
- [4] D. Burdick, M. Calimlim, and J. Gehrke. Mafia: A maximal frequent itemset algorithm for transactional databases. In *ICDE*, 2001.
- [5] K. Gouda and M. J. Zaki. Genmax: Efficiently mining maximal frequent itemsets. In *ICDM*, 2001.
- [6] J. Han, J. Pei, and Y. Yin. Mining frequent patterns without candidate generation. In *SIGMOD*, 2000.
- [7] R.J. Bayardo. Jr. Efficiently mining long patterns from databases. In *SIGMOD*, 1998.
- [8] G. Liu, H. Lu, W. Lou, and J. X. Yu. On computing, storing and querying frequent patterns. In *SIGKDD*, 2003.
- [9] G. Liu, H. Lu, Y. Xu, and J. X. Yu. Ascending frequency ordered prefix-tree: Efficient mining of frequent patterns. In *DASFAA*, 2003.
- [10] J. Liu, Y. Pan, K. Wang, and J. Han. Mining frequent item sets by opportunistic projection. In *SIGKDD*, 2002.
- [11] N. Pasquier, Y. Bastide, R. Taouil, and L. Lakhal. Discovering frequent closed itemsets for association rules. In *ICDT*, 1999.
- [12] J. Pei, J. Han, H. Lu, S. Nishio, S. Tang, and D. Yang. Hmine: Hyper-structure mining of frequent patterns in large databases. In *ICDM*, 2001.
- [13] J. Pei, J. Han, and R. Mao. Closet: An efficient algorithm for mining frequent closed itemsets. In *DMKD*, 2000.
- [14] R. Raymon. Search through systematic set enumeration. In *Proc. of KR Conf.*, 1992.
- [15] R.C. Agarwal, C.C. Aggarwal, and V.V.V. Prasad. A tree projection algorithm for finding frequent itemsets. *Journal on Parallel and Distributed Computing*, 61(3):350–371, 2001.
- [16] J. Wang, J. Pei, and J. Han. Closet+: Searching for the best strategies for mining frequent closed itemsets. In *SIGKDD*, 2003.
- [17] Y. Xu, J. X. Yu, G. Liu, and H. Lu. From path tree to frequent patterns: A framework for mining frequent patterns. In *ICDM*, pages 514–521, 2002.
- [18] M. J. Zaki and C. Hsiao. Charm: An efficient algorithm for closed itemset mining. In *SDM*, 2002.
- [19] M. J. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *SIGKDD*, 1997.