Pincer-Search: An Efficient Algorithm for Discovering the Maximum Frequent Set

Dao-I Lin* Zvi M. Kedem†

Telcordia Technologies, Inc. New York University

July 15, 1999

Abstract

Discovering frequent itemsets is a key problem in important data mining applications, such as the discovery of association rules, strong rules, episodes, and minimal keys. Typical algorithms for solving this problem operate in a bottom-up, breadth-first search direction. The computation starts from frequent 1-itemsets (the minimum length frequent itemsets) and continues until all maximal (length) frequent itemsets are found. During the execution, every frequent itemset is explicitly considered. Such algorithms perform well when all maximal frequent itemsets are short. However, performance drastically decreases when some of the maximal frequent itemsets are relatively long. We present a new algorithm which combines both the bottom-up and the top-down searches. The primary search direction is still bottom-up, but a restricted search is also conducted in the top-down direction. This search is used only for maintaining and updating a new data structure, the maximum frequent candidate set. It is used to prune early candidates that would normally encountered in the bottom-up search. A very important characteristic of the algorithm is that it does not require explicite examination of every frequent itemset. Therefore the algorithm performs well even when some maximal frequent itemsets are long. As its output, the algorithm produces the maximum frequent set, i.e., the set containing all maximal frequent itemsets, thus specifying immediately all frequent itemsets. We evaluate the performance of the algorithm using well-known synthetic benchmark databases and real-life census and

*Applied Research, Telcordia Technologies, Inc., 445 South Street, Morristown, NJ 07960 +1 973 829 4740, tlin@research.telcordia.com.

†Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, +1 212 998 3101, kedem@cs.nyu.edu.

stock market databases. The improvement in performance can be up to several orders of magnitude, compared to the best current algorithms.

1 Introduction

Knowledge discovery in databases (KDD) has received increasing attention and has been recognized as a promising new field of database research. It is defined by Fayyad *et al.* [8] as "the non-trivial process of identifying valid, novel, potentially useful, and ultimately understandable patterns in data". The key step in the knowledge discovery process is the data mining step, "consisting of applying data analysis and discovery algorithms that, under acceptable computational efficiency limitations, produce a particular enumeration of patterns over the data" [8]. This paper addresses a pattern discovery problem, which is important in many data mining applications.

A key component of many data mining problems is formulated as follows. Given a large database of sets of items (representing market basket data, alarm signals, etc.), discover all frequent *itemsets* (sets of items), where a frequent itemset is one that occurs in at least a user-defined percentage (minimum *support*) of the database. Depending on the semantics attached to the input database, the frequent itemsets, and the term "occurs," we get the key components of different data mining problems such as the discovery of association rules (e.g., [3, 10, 19]), theories (e.g., [9]), strong rule (e.g., [22]), episodes (e.g., [17]), and minimal keys (e.g., [9]).

Typical algorithms for finding the *frequent set*, i.e., the set of all frequent itemsets, operate in a *bottom-up*, breadth-first fashion (e.g., [2, 3, 7, 10, 18, 19, 21, 23, 26]). The computation starts from frequent 1-itemsets (the minimum length frequent itemsets) at the bottom, and then extends one level up in every pass until all maximal (length) frequent itemsets are discovered. *All* frequent itemsets are *explicitly examined* and discovered by these algorithms. When *all* maximal frequent itemsets are short, these algorithms perform well. However, performance drastically decreases when *any* of the maximal frequent itemsets becomes longer, because a maximal frequent itemset of size l implies the presence of $2^l - 2$ additional frequent itemsets (its nontrivial subsets) as well, each of which is explicitly examined by such algorithms. In data mining applications where items are correlated, maximum frequent itemsets could be long [7].

Therefore, instead of examining and "assembling" all the frequent itemsets, an alternative approach might be to "shortcut" the process and attempt to search for maximal frequent itemsets "more directly," as they immediately specify all frequent itemsets. Furthermore, it suffices to know only the maximal frequent set in many data mining applications, such as the minimal key discovery [9] and the theory extraction [9].

Finding the *maximum frequent set* (or MFS), the set of all maximal frequent itemsets, is essentially a search problem in a hypothesis search space (a lattice of subsets). The search for the maximum frequent set can proceed from the 1-itemsets to n-itemsets (bottom-up) or from the n-itemsets to 1-itemsets (top-down).

We present a novel *Pincer-Search* algorithm, which searches for the MFS from *both bottom-up and top-down directions*. It performs well even when the maximal frequent itemsets are long.

The bottom-up search is similar to *Apriori* [3] and *OCD* [19] algorithms. However, the top-down search is novel. It is implemented efficiently by introducing an auxiliary data structure, the *maximum frequent candidate set* (or MFCS), as explained later. By incorporating the computation of the MFCS in our algorithm, we are able to efficiently approach the MFS from both top-down and bottom-up directions. Unlike the bottom-up search that goes up one level in each pass, the MFCS can help the computation "move down" many levels in the top-down direction in one pass.

In this paper, we apply the MFCS concept to the association rule mining. In fact, the MFCS concept can be applied in solving other data mining problems if the problem has closure properties as discussed later. The monotone specialization relation discussed in [18, 15] addressed the same closure properties.

Popular benchmark databases designed by Agrawal and Srikant [3] have been used in [4, 21, 23, 26, 29]. We use these same benchmarks to evaluate the performance of our algorithm. In most cases, our algorithm not only reduces the number of passes of reading the database but also reduces the number of candidates (for whom support is counted). In such cases, both I/O time and CPU time are reduced by eliminating the candidates that are subsets of maximal frequent itemsets found in the MFCS.

The organization of the rest of the paper is as follows. The problem of association rule mining and the importance of frequent itemsets are sketched. Section 2. The importance of maximum frequent set, structural properties for maximal frequent itemsets and how they lead to algorithms for their discovery, together with an important representative algorithm are presented in Section 3. Our new Pincer-Search Algorithm is presented in Section 4. Performance

evaluation is presented in Section 5. Related work is presented in Section 6. Concluding remarks are given in Section 7.

2 Association Rule Mining

This section briefly introduces the association rule mining problem, following as feasible the terminology of [2].

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of m (distinct) items. We assume that the items are drawn from some totally ordered domain. For instance, if they are strings, they could be ordered lexicographically. So for convenience, an itemset will be stored as a sequence following the order of the domain. The itemsets could represent different items in a supermarket or different alarm signals in telecommunication networks [12]. A *transaction* T is a set of items in I. A transaction could represent some customer purchases of some items from a supermarket or the set of alarm signals occurring within a time interval. A database D is just a set of transactions. A set of items is called an *itemset*. The number of items in an itemset is called the *length* of an itemset. Itemsets of some length k are referred to as k-itemsets.

A transaction T is said to *support* an itemset $X \subseteq I$ if and only if $X \subseteq T$. The fraction of the transactions in D that support X is called the *support* of X, denoted as support(X). There is a user-defined *minimum support* threshold, which is a fraction, i.e., a number in [0,1]. An itemset is *frequent* iff its support is at least the minimum support. Otherwise, it is *infrequent*.

An *association rule* has the form $R : X \rightarrow Y$, where X and Y are two non-empty and non-intersecting itemsets. The *support for rule* R is defined as support$(X \cup Y)$. A *confidence* factor for such a rule (customarily represented by percentage) is defined as $100 \cdot \text{support}(X \cup Y)/\text{support}(X)$ (assume support$(X) > 0$) and is used to evaluate the strength of such association rules. The confidence of a rule indicates how often it can be expected to apply, while its support indicates how trustworthy it is.

The goal of association rule mining is to discover all rules that have support and confidence greater than some user-defined minimum support and minimum confidence thresholds, respectively.

The normally followed scheme for mining association rules consists of two stages [3]:

1. the discovery of frequent itemsets

2. the generation of association rules.

As the second step is rather straightforward and as the first step is dominates the processing time, we explicitly focus the paper on the first step: the discovery of frequent itemsets.

3 Frequent Itemsets: Structural Properties and Basic Discovery Approaches

3.1 The Maximum Frequent Set

Among all the frequent itemsets, some will be *maximal frequent itemsets*: they have no proper supersets that are themselves frequent. The *maximum frequent set* (or MFS) is the set of all the maximal frequent itemsets. The problem of discovering the frequent set can be reduced to the problem of discovering the MFS. The MFS immediately specifies of frequent itemsets; these are precisely the non-empty subsets of its elements. The MFS forms a border between frequent and infrequent sets. Once the MFS is known, the supports of all the frequent itemsets can be computed by reading the database once.

3.2 Closure Properties

A typical frequent set discovery process follows a standard scheme. Throughout the execution, the set of all itemsets is partitioned, perhaps implicitly, into three sets:

1. *frequent:* This is the set of those itemsets that have been discovered so far as frequent

2. *infrequent:* This is the set of those itemsets that have been discovered so far as infrequent

3. *unclassified:* This is the set of all the other itemsets.

Initially, the frequent and the infrequent sets are empty. Throughout the execution, they grow at the expense of the unclassified set. The execution terminates when the unclassified set becomes empty, and then, of course all the

maximal frequent itemsets are discovered.

Consider *any process* for classifying itemsets and some point in the execution where some itemsets have been classified as frequent, some as infrequent, and some are still unclassified. Two *closure properties* can be used to immediately classify some of the unclassified itemsets:

Property 1: If an itemset is infrequent, all it supersets must be infrequent, and they need not be examined further

Property 2: If an itemset is frequent, all its subsets must be frequent, and they need not be examined further

3.3 Discovering Frequent Itemsets

In general, it is possible to search for the maximal frequent itemsets either *bottom-up* or *top-down*. If all maximal frequent itemsets are expected to be short (close to 1 in size), it seems efficient to search for them bottom-up. If all maximal frequent itemsets are expected to be long (close to n in size) it seems efficient to search for them top-down.

We first sketch a realization of the most commonly used approach of discovering the MFS: a *bottom-up* approach. It consists of repeatedly applying a *pass*, itself consisting of two *steps*. At the end of pass k all frequent itemsets of size k or less have been discovered.

As the *first step* of pass $k+1$, itemsets of size $k+1$ each having two frequent k-subsets with the same first $k-1$ items are generated. Itemsets that are supersets of infrequent itemsets are pruned (and discarded), as of course they are infrequent (by Property 1). The remaining itemsets form the set of *candidates* for this pass.

As the *second step*, the support of the candidates is computed (by reading the database), and they are classified as either frequent or infrequent.

Example Consider a database containing five distinct items, 1, 2, 3, 4, and 5. There are four transactions in this database: {1,2,3,4,5}, {1,3}, {1,2}, and {1,2,3,4}. The minimum support is set to 0.5. Fig. 1 shows an example of this bottom-up approach. All five 1-itemsets ({1}, {2}, {3}, {4}, {5}) are candidates in the first pass. After the support counting phase, the 1-itemset {5} is determined to be infrequent. By Property 1, all the supersets of {5} need not be considered. So the candidates for the second pass are {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, {3,4}. The same procedure repeats until all the maximal frequent itemsets are obtained; in this example, only one: {1,2,3,4}. □

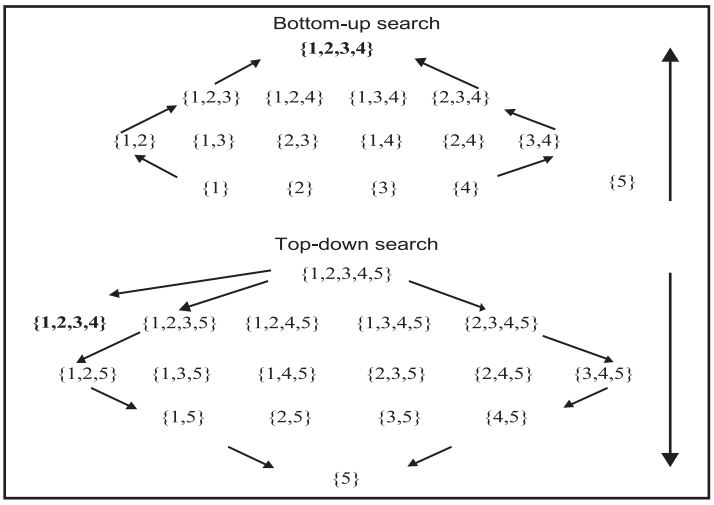

Figure 1: One-Way Searches

In this bottom-up approach, *every* frequent itemset must have been a candidate at some pass and is therefore also *explicitly* considered. When *some* maximal frequent itemsets happen to be long, this method will be inefficient. In such a case, it might be more efficient to search for the long maximal frequent itemsets using a top-down approach.

A *top-down* approach starts with the single n-itemset and decreases the size of the candidates by one in every pass. When a k-itemset is determined to be infrequent, all of its $(k-1)$-subsets will be examined in the next pass. However, if a k-itemset is frequent, then all of its subsets must be frequent and need not be examined (by Property 2).

Example See Fig. 1 and consider the same database as the previous example. The 5-itemset {1,2,3,4,5} is the only candidate in the first pass. After the support counting phase, it is infrequent. The candidates for the second pass are all the 4-subsets of itemset {1,2,3,4,5}. In this example, itemset {1,2,3,4} is frequent and all the others are infrequent. By Property 2, all subsets of {1,2,3,4} are frequent (but not maximal) and need not be examined. The same procedure repeats until all maximal frequent itemsets are obtained (i.e., after all infrequent itemsets are visited). □

In this top-down approach, *every* infrequent itemset is *explicitly* examined. As shown in Fig. 1, every infrequent itemset (itemset {5} and its supersets) needs to be visited before the maximal frequent itemsets are obtained.

Note that, in a "pure" bottom-up approach, only Property 1 above is used to prune candidates. This is the technique that many algorithms (e.g., [3, 4, 7, 10, 18, 19, 21, 23, 26]) use to decrease the number of candidates. In a "pure"

top-down approach, only Property 2 is used to prune candidates. This is the technique used in [29, 18].

3.4 The Apriori Algorithm

The Apriori algorithm [3] is a typical bottom-up approach algorithm. We describe it in some detail, as we will find it helpful to rely on this in presenting our results. The Apriori algorithm repeatedly uses *Apriori-gen* algorithm to generate candidates and then count their supports by reading the entire database once. The algorithm is described in Fig. 2.

Algorithm: Apriori algorithm
Input: a database and a user-defined minimum support
Output: all frequent itemsets
1. $L_0 := \emptyset$; $k := 1$;
2. $C_1 := \{\{i\} \mid i \in I\}$
3. Answer := $\emptyset$
4. **while** $C_k \neq \emptyset$
5. read database and count supports for C_k
6. L_k := {frequent itemsets in C_k}
7. C_{k+1} := *Apriori-gen*(L_k)
8. $k := k + 1$
9. Answer := Answer $\cup$ L_k
10. **return** Answer

Figure 2: Apriori Algorithm

Apriori-gen relies on Property 1 mentioned above. The candidate generation algorithm consists of a *join* procedure and a *prune* procedure

The *join* procedure combines two frequent k-itemsets, which have the same $(k-1)$-prefix, to generate a $(k+1)$-itemset as a new preliminary candidate. Following the *join* procedure, the *prune* procedure is used to remove from the preliminary candidate set all itemsets c such that some k-subset of c is not a frequent itemset. See Fig. 3 for details.

The Apriori-gen algorithm has been very successful in reducing the number of candidates and has been used in many subsequent algorithms, such as *DHP* [23], *Partition* [26], *Sampling* [27], *DIC* [7], and *Clique* [29].

```
Algorithm: The join procedure of the Apriori-gen algorithm
Input: L_k, the set containing frequent itemsets found in pass k
Output: preliminary candidate set C_{k+1}
/* The itemsets in L_k are sorted */
1. for i from 1 to |L_k − 1|
2.   for j from i + 1 to |L_k|
3.     if L_k.itemset_i and L_k.itemset_j have the same (k − 1)-prefix
4.       C_{k+1} := C_{k+1} ∪ {L_k.itemset_i ∪ L_k.itemset_j}
5.     else
6.       break
```

```
Algorithm: The prune procedure of the Apriori-gen algorithm
Input: Preliminary candidate set C_{k+1} generated from the join procedure above
Output: final candidate set C_{k+1} which does not contain any infrequent subset
1. for all itemsets c in C_{k+1}
2.   for all k-subsets s of c
3.     if s ∉ L_k
4.       delete c from C_{k+1}
```

Figure 3: Join and Prune Procedures

4 Fast Algorithms for Discovering the Maximum Frequent Set

4.1 Our Approach to Reducing the Number of Candidates and the Number of Passes

As discussed in the last section, the bottom-up approach is good for the case when *all* maximal frequent itemsets are short and the top-down approach is good when *all* maximal frequent itemsets are long. If some maximal frequent itemsets are long and some are short, then both one-way search approaches will not be efficient.

To design an algorithm that can efficiently discover both long and short maximal frequent itemsets, one might think of simply running both bottom-up and top-down programs at the same time. However, this naive approach is not good enough. We can actually do much better than that.

Recall that the bottom-up approach described above uses only Property 1 to reduce the number of candidates and the top-down approach uses only Property 2 to reduce the number of candidates.

In our *Pincer-Search* approach (first presented in [14]) we combine the top-down and the bottom-up searches, we synergistically rely on *both* properties to prune candidates. A key component of the approach is the use of information gathered in the search in one direction to prune more candidates during the search in the other direction.

If some maximal frequent itemset is found in the top-down direction, then this itemset can be used to eliminate (possibly many) candidates in the bottom-up direction. The subsets of this frequent itemset can be pruned because they are frequent (Property 2). Of course, if an infrequent itemset is found in the bottom-up direction, then it can be

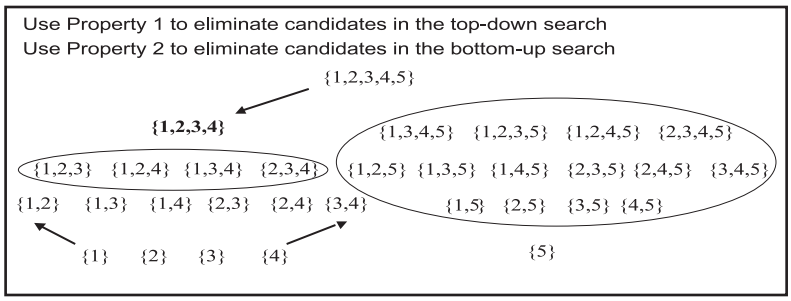

Figure 4: Two-Way Search

used to eliminate some candidates in the top-down direction (Property 1). This "two-way search approach" can fully make use of both properties and thus speed up the search for the maximum frequent set.

Example See Fig. 4, which considers the database of Fig. 1. In the first pass, all five 1-itemsets are the candidates for the bottom-up search and the 5-itemset {1,2,3,4,5} is the candidate for the top-down search. After the support counting phase, infrequent itemset {5} is discovered by the bottom-up search and this information is shared with the top-down search. This infrequent itemset {5} not only allows the bottom-up search to eliminate its supersets as candidates but also allows the top-down search to eliminate its supersets as candidates in the second pass. In the second pass, the candidates for the bottom-up search are {1,2}, {1,3}, {1,4}, {2,3}, {2,4}, and {3,4}. Itemsets {1,5}, {2,5}, {3,5}, and {4,5} are not candidates, since they are supersets of {5}. The only candidate for the top-down search in the second pass is {1,2,3,4}, since all the other 4-subsets of {1,2,3,4,5} are supersets of {5}. After the second support counting phase, {1,2,3,4} is discovered to be frequent by the top-down search. This information is shared with the bottom-up search. All of its subsets are frequent and need not be examined. In this example, itemsets {1,2,3}, {1,2,4}, {1,3,4}, and {2,3,4} will not be candidates for our bottom-up or top-down searches. After that, the program can terminate, since there are no candidates for either bottom-up or top-down searches. □

In this example, the number of candidates considered, was smaller than required by either bottom-up or top-down search. We also needed fewer passes of reading the database than either bottom-up or top-down searches. The "pure" bottom-up approach would have taken four passes and the "pure" top-down approach would have taken five passes for this database—we needed *only* two. In fact, Pincer-Search will always use at most as many passes as the minimum

of the passes used by bottom-up approach and top-down approach.

Reducing the number of candidates is of critical importance for the efficiency of the frequent set discovery process, since the cost of the entire process comes from reading the database (I/O time) to generate the supports of candidates (CPU time) and the generation of new candidates (CPU time). The support counting of the candidates is the most expensive part. Therefore, the number of candidates dominates the entire processing time. Reducing the number of candidates not only can reduce the I/O time but also can reduce the CPU time, since fewer candidates need to be counted and generated.

Therefore, it is important that Pincer-Search reduces both the number of candidates and the number of passes. A realization of this two-way search algorithm is discussed next.

4.2 Two-Way Search by Using the MFCS

We have designed a combined two-way search algorithm for discovering the maximum frequent set. It relies on a new data structure during its execution, the *maximum frequent candidate set*, or MFCS for short, which we define next.

Definition 1 *Consider some point during the execution of an algorithm for finding the* MFS. *Let* FREQUENT *be the set of the itemsets known to be frequent, and let* INFREQUENT *be the set of the itemsets known to be infrequent. Then the maximum frequent candidate set (MFCS) is the minimum cardinality set of items satisfying the conditions.*

$$\text{FREQUENT} \subseteq \cup\{2^X \mid X \in \text{MFCS}\}$$

$$\text{INFREQUENT} \cap \{2^X \mid X \in \text{MFCS}\} = \emptyset$$

In other words, MFCS is the set of all maximal itemsets that are *not known* to be infrequent at this state of the algorithm. Thus obviously at any point of the algorithm MFCS is a superset of the MFS. When the algorithm terminates, the MFCS and the MFS are equal.

The computation of our algorithm follows the bottom-up breadth-first search approach. We base our presentation on the Apriori algorithm, and for greatest ease of exposition we present our algorithm as a modification to that algorithm.

Briefly speaking, in each pass, in addition to counting supports of the candidates in the bottom-up direction, the algorithm also counts supports of the itemsets in the MFCS: this set is adapted for the top-down search. This will help in pruning candidates, but will also require changes in candidate generation, as explained later.

Consider a pass k, during which, in the bottom-up direction, itemsets of size k are to be classified. If, during the top-down direction some itemset that is an element of the MFCS of cardinality greater than k is found to be frequent, then all its subsets of cardinality k can be pruned from the set of candidates considered in the bottom-up direction in this pass. They, and their supersets will never be candidates throughout the rest of the execution, potentially improving performance. But of course, as the maximum frequent set is ultimately computed, they "will not be forgotten."

Similarly, when a new infrequent itemset is found in the bottom-up direction, the algorithm will use it to update the MFCS. The subsets of the MFCS must not contain this infrequent itemset.

Fig. 5 conceptually shows the combined two-way search. The MFCS is initialized to contain a single element, the itemset of cardinality n containing all the elements of the database. As an example of its utility, consider the first pass of the bottom-up search. If some m 1-itemsets are infrequent after the first pass (after reading the database once), the MFCS will have one element of cardinality $n - m$. This itemset is generated by removing the m infrequent items from the initial element of the MFCS. In this case, the top-down search goes down m levels in one pass. In general, unlike the search in the bottom-up direction, which goes up one level in one pass, *the top-down search can go down many levels in one pass*.

Notice that the bottom up and the top down searches do not proceed in a symmetrical fashion. The reason is that by a general assumption there are no extremely long frequent itemsets. If this assumption is not likely to hold, one can easily reverse the roles of the searches in the two directions.

By using the MFCS, we may be able to discover some maximal frequent itemsets in early passes. This early discovery of the maximal frequent itemsets can reduce the number of candidates and the passes of reading the database which in turn can reduce the CPU time and I/O time. This is especially significant when the maximal frequent itemsets discovered in the early passes are long.

For our approach to be effective, we need to address two issues. First, how to update the MFCS efficiently? Second, once the subsets of the maximal frequent itemsets found in the MFCS are removed, how do we generate the

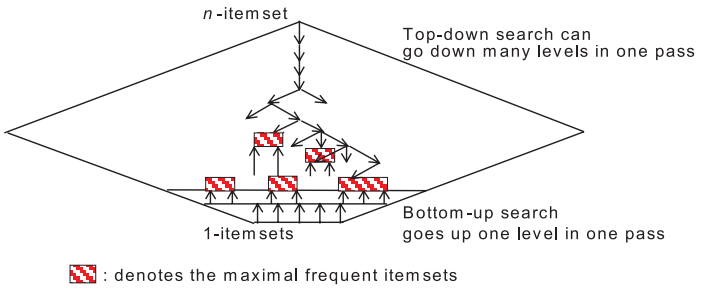

Figure 5: The Search Space of Pincer-Search

correct candidate set for the subsequent passes in the bottom-up direction?

4.3 Updating the MFCS Efficiently

Consider some itemset Y that has been "just" classified as infrequent. By the definition of the MFCS, it will be a subset of one or more itemsets in the MFCS, and we need to update the MFCS such that its subsets no longer contain Y. To update the MFCS, we will do the following process for every superset of Y that is in the MFCS. We replace every such itemset (say X) by $|Y|$ itemsets, each obtained by removing from X a single item (element) of Y. Such newly generated itemset is added to the MFCS only when it is not already a subset of any itemset in the MFCS. Formally, we have the *MFCS-gen* algorithm as in Fig. 6 (shown here for pass k).

Algorithm: *MFCS-gen*
Input: Old MFCS and the infrequent set S_k found in pass k
Output: New MFCS
1. **for** all itemsets $s \in S_k$
2. **for** all itemsets $m \in$ MFCS
3. **if** s is a subset of m
4. MFCS := MFCS $\setminus \{m\}$
5. **for** all items $e \in$ itemset s
6. **if** $m \setminus \{e\}$ is not a subset of any itemset in the MFCS
7. MFCS := MFCS $\cup \{m \setminus \{e\}\}$
8. **return** MFCS

Figure 6: MFCS-gen Algorithm

Example See Fig. 7. Suppose {{1,2,3,4,5,6}} is the current ("old") value of the MFCS and two new infrequent itemsets {1,6} and {3,6} are discovered. Consider first the infrequent itemset {1,6}. Since the itemset {1,2,3,4,5,6} (element of the MFCS) contains items 1 and 6, one of its subsets will be {1,6}. By removing item 1 from itemset

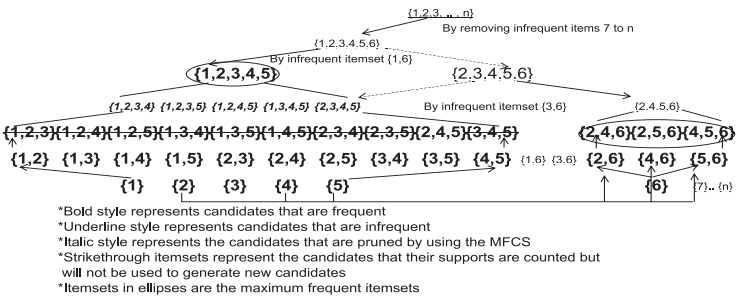

Figure 7: Pincer-Search

{1,2,3,4,5,6}, we get {2,3,4,5,6}, and by removing item 6 from itemset {1,2,3,4,5,6} we get {1,2,3,4,5}. After considering itemset {1,6}, the MFCS becomes {{1,2,3,4,5}, {2,3,4,5,6}}. Itemset {3,6} is then used to update this MFCS. Since {3,6} is a subset of {2,3,4,5,6}, two itemsets {2,3,4,5} and {2,4,5,6} are generated to replace {2,3,4,5,6}. Itemset {2,3,4,5} is a subset of itemset {1,2,3,4,5} in the new MFCS, and it will not be added to the MFCS. Therefore, the MFCS becomes {{1,2,3,4,5}, {2,4,5,6}}. The top-down arrows in Fig. 7 show the updates of the MFCS. □

Lemma 1 *The algorithm MFCS-gen correctly updates the* MFCS.

Proof: The algorithm excludes all the infrequent itemsets, so the final set will not contain any infrequent itemsets as subsets of its elements. Step 7 removes only one item from the itemset m: the longest subset of the itemset m that does not contain the infrequent itemset s. Since this algorithm always generates longest itemsets, the number of the itemsets will be minimum at the end. □

4.4 New Candidate Generation Algorithms

Recall that, as discussed in Section 3.4, a preliminary candidate set will be generated after the *join* procedure is called. In our algorithm, after a maximal frequent itemset is added to the MFS, all of its subsets in the frequent set (computed so far) will be removed. We show by example that if the original *join* procedure of the Apriori-gen algorithm is applied, some of the needed itemsets could be missing from the preliminary candidate set. Consider Fig. 7. Suppose that the original frequent itemset L_3 is {{1,2,3}, {1,2,4}, {1,2,5}, {1,3,4}, {1,3,5}, {1,4,5}, {2,3,4}, {2,3,5}, {2,4,5},

{2,4,6}, {2,5,6}, {3,4,5}, {4,5,6}}. Assume itemset {1,2,3,4,5} in the MFCS is determined to be frequent. Then all 3-itemsets of the original frequent set L_3 will be removed from it by our algorithm, except for {2,4,6}, {2,5,6}, and {4,5,6}. Since the Apriori-gen algorithm uses a $(k-1)$-prefix test on the frequent set to generate new candidates, and no two itemsets in the current frequent set {{2,4,6}, {2,5,6}, {4,5,6}} share a 2-prefix, no candidate will be generated by applying the *join* procedure on this frequent set. However, the correct preliminary candidate set should be {{2,4,5,6}}.

Based on the above observation, we need to recover some missing candidates.

4.4.1 New Preliminary Candidate Set Generation Procedure

In our new preliminary candidate set generation procedure, the *join* procedure of the Apriori-gen algorithm is first called to generate a temporary candidate set, which might be incomplete. In such a case, a *recovery* procedure will be called to recover the missing candidates.

All missing candidates can be obtained by restoring some itemsets to the current frequent set. The restored itemsets are extracted from the MFS of the current pass, which implicitly maintains all frequent itemsets discovered so far.

The first group of itemsets that needs to be restored contains those k-itemsets that have the same $(k-1)$-prefix as some itemset in the current frequent set. Consider then in pass k, an itemset X in the MFS and an itemset Y in the current frequent set such that $|X| > k$. Suppose that the first $k-1$ items of Y are in X and the $(k-1)$st item of Y is equal to the jth item of X. We obtain the k-subsets of X that have the same $(k-1)$-prefix as Y by taking one item of X that has an index greater than j and combining it with the first $k-1$ items of Y, thus getting one of these k-subsets. After these k-itemsets are found, we recover candidates by combining them with itemset Y as shown in Fig. 8.

Example See Fig. 7. The MFS is {{1,2,3,4,5}} and the current frequent set is {{2,4,6}, {2,5,6}, {4,5,6}}. The only 3-subset of {{1,2,3,4,5}} that needs to be restored for itemset {2,4,6} to generate a new candidate is {2,4,5}. This is because it is the only subset of {{1,2,3,4,5}} that has the same length and the same 2-prefix as itemset {2,4,6}. By combining {2,4,5} and {2,4,6}, we recover the missing candidate {2,4,5,6}. No itemsets need to be restored for itemsets {2,5,6} and {4,5,6}. □

```
Algorithm: The recovery procedure
Input: C_{k+1} from join procedure, L_k, and current MFS
Output: a complete candidate set C_{k+1}
1. for all itemsets l in L_k
2.   for all itemsets m in MFS
3.     if the first k-1 items in l are also in m
4.       /* suppose m.item_j = l.item_{k-1} */
5.       for i from j+1 to |m|
6.         C_{k+1} := C_{k+1} ∪ {{l.item_1, l.item_2, ..., l.item_k, m.item_i}}
```

Figure 8: Recovery Algorithm

The second group of itemsets that need to be restored consists of those k-subsets of the MFS having the same $(k-1)$-prefix but having no common superset in the MFS. A similar *recovery* procedure can be applied after they are restored.

4.4.2 New Prune Procedure

After the recovery stage, a preliminary candidate set will be generated. We can then proceed to the prune stage. Instead of checking to see if all k-subsets of an itemset X are in L_k, we can simply check to see if X is a subset of an itemset in the current MFCS as shown in Fig. 9. In comparison with the *prune* procedure of Apriori-gen, we use one loop less.

```
Algorithm: New prune procedure
Input: current MFCS and C_{k+1} after join and recovery procedures
Output: final candidate set C_{k+1}
1. for all itemsets c in C_{k+1}
2.   if c is not a subset of any itemset in the current MFCS
3.     delete c from C_{k+1}
```

Figure 9: New Prune Algorithm

4.4.3 Correctness of the New Candidate Generation Algorithm

In summary, our candidate generation process contains three steps as described in Fig. 10.

```
Algorithm: New candidate generation algorithm
Input: L_k, current MFCS, and current MFS
Output: new candidate set C_{k+1}
1. call the join procedure as in the Apriori algorithm
2. call the recovery procedure if necessary
3. call the new prune procedure
```

Figure 10: New Candidate Generation Algorithm

Lemma 2 *The new candidate generation algorithm generates the correct candidate set.*

Proof: Recall the candidate generation process as in Apriori-gen. There are four possible cases when we combine two frequent k-itemsets, say I and J, which have the same $(k-1)$-prefix, to generate a $(k+1)$-itemset as a new preliminary candidate. In this proof, we will show that, even though that we remove the subsets of the MFS from the current frequent set, our new candidate generation algorithm will handle all these cases correctly.

Case 1: I and J are not subsets of any elements of MFS. Both itemsets are in the current frequent set. The *join* procedure will combine them and generate a preliminary candidate.

Case 2: I is a subset of some element of MFS but J is not a subset of any element of MFS. I is removed from the current frequent set. However, combining I and J will generate a candidate that needs to be examined. The *recovery* procedure discussed above will recover candidates in this case.

Case 3: I and J are subsets of some element of MFS. We do not combine them, since the candidate that they generate will be a subset of X and must be frequent.

Case 4: I and J are both subsets of element of MFS, but there is no single element of MFS of which they are both subsets. Both I and J are removed from the current frequent set. However, by combining them, a necessary candidate will be generated. Similar *recovery* procedure as discussed above will recover missing candidates in this case.

Our preliminary candidate generation algorithm considers the same combinations of the frequent itemsets as does the Apriori-gen algorithm. This preliminary candidate generation algorithm will generate all the candidates, as the Apriori-gen algorithm does, except those that are subsets of the MFS. By the *recovery* procedure, some subsets of the MFS will be restored in the later passes when necessary.

Lemma 1 showed that MFCS will be maintained correctly in every pass. Therefore, our new *prune* procedure will make sure that no superset of infrequent itemsets is in the preliminary candidate set. Therefore, the *new* candidate generation algorithm is correct. □

4.5 The Basic Pincer-Search Algorithm

We now present our complete algorithm (see Fig. 11), *The Pincer-Search Algorithm*, which relies on the combined approach for determining the maximum frequent set. Lines 9 to 12 constitute our *new* candidate generation procedure.

```
Algorithm: The Pincer-Search algorithm
Input: a database and a user-defined minimum support
Output: MFS which contains all maximal frequent itemsets
 1. L_0 := ∅; k := 1; C_1 := {{i} | i ∈ I }
 2. MFCS := {{1, 2, ..., n}}; MFS := ∅
 3. while C_k ≠ ∅
 4.  read database and count supports for C_k and MFCS
 5.  remove frequent itemsets from MFCS and add them to MFS
 6.  L_k := {frequent itemsets in C_k} \ {subsets of MFS}
 7.  S_k := {infrequent itemsets in C_k}
 8.  call the MFCS-gen algorithm if S_k ≠ ∅
 9.  call the join procedure to generate C_{k+1}
10.  if any frequent itemset in C_k is removed in line 6
11.   call recovery procedure to recover candidates to C_{k+1}
12.  call new prune procedure to prune candidates in C_{k+1}
13.  k := k + 1
14. end-while
15. return MFS
```

Figure 11: The Pincer-Search algorithm

The MFCS is initialized to contain one itemset, which consists of all the database items. The MFCS is updated whenever new infrequent itemsets are found (line 8). If an itemset in the MFCS is found to be frequent, then its subsets will not participate in the subsequent support counting and candidate set generation steps. Line 6 will exclude those itemsets that are subsets of any itemset in the current MFS, which contains the frequent itemsets found in the MFCS. If some itemsets in L_k are removed, the algorithm will call the *recovery* procedure to recover missing candidates (line 11).

Theorem 1 *The Pincer-Search algorithm generates all maximal frequent itemsets.*

Proof: Lemma 2 showed that our candidate generation algorithm will generate candidate set correctly. The Pincer-Search algorithm will explicitly or implicitly discover all frequent itemsets. The frequent itemsets are *explicitly* discovered when they are discovered by the bottom-up search (i.e., they were in the L_k set at some point). The frequent itemsets are *implicitly* discovered when the top-down search discovers their frequent supersets (which are maximal) earlier than the bottom-up search reaches them. Furthermore, only the maximal frequent itemsets will be added to the MFS in Line 5. Therefore, the Pincer-Search algorithm generates all maximal frequent itemsets. □

4.6 The Adaptive Pincer-Search Algorithm

In general, one may not want to use the "basic" version of the Pincer-Search algorithm. For instance, in some cases, there may be too many infrequent 2-itemsets. In such cases, it may be too costly to maintain the MFCS. The algorithm we have implemented is in fact an adaptive version of the algorithm. This adaptive version does not maintain the MFCS, when doing so would be counterproductive. It delays the maintenance of the MFCS until a later pass when the expected cost of calculating the MFCS is acceptable. This is also the algorithm whose performance is being evaluated in Section 5. Thus the very small overhead of deciding when to use the MFCS is accounted in the performance evaluation of our adaptive Pincer-Search algorithm.

Another adaptive approach is to generate all candidates as the Apriori algorithm, but not to count the support of the candidates that are subsets of any itemset in the current MFS. This approach simplifies the basic Pincer-Search algorithm in such a way that it need not do the candidate recovery process mentioned in Section 4.4. A flag, indicating whether a candidate should be counted or not, can be easily maintained. Based on how the recovery process is done, its cost can be estimated by the number of the current frequent set and the number of the current MFS. When the estimated cost exceeds some threshold, we can switch from the recovery procedure to the candidate generation procedure that generates all candidates. This way, we can still avoid the support counting phase, which is the most time-consuming process.

5 Performance Evaluation

For the Pincer-Search algorithm to be effective, the top-down search needs to reach the maximal frequent itemsets faster than the bottom-up search. A reasonable question can be informally stated: "Can the search in the top-down direction proceed fast enough to reach a maximal frequent itemset faster than the search in the bottom-up direction?" There can be no categorical answer, as this really depends on the distribution of the frequent and infrequent itemsets. Encouragingly, according to both [3] and our experiments, a large fraction of the 2-itemsets will usually be infrequent. These infrequent itemsets will cause the MFCS to go down the levels very fast, allowing it to reach some maximal frequent itemsets after only a few passes. Indeed, in our experiments, we have found that, in most cases, many of the

maximal frequent itemsets are found in the MFCS in very early passes. For instance, in the experiment on database T20.I15.D100K (Fig. 13), all maximal frequent itemsets containing up to 17 items are found in 3 passes only!

The performance evaluation presented compares our adaptive Pincer-Search algorithm to the Apriori algorithm. Although, later in related work, many variants of the Apriori algorithm will be described, we limit ourselves this performance comparison because it is sufficiently instructive to understand the characteristics of the new algorithm's performance. Furthermore, these algorithms, except *A-Random-MFS* [9] and Max-Miner [6], discover the entire frequent set. Their improvement over Apriori usually is less than an order of magnitude.

5.1 Preliminary Discussion

5.1.1 Auxiliary Data Structures Used

The databases used in performance evaluation, are the synthetic databases used in [3], the census databases similar to [7], and the stock transaction databases from New York Stock Exchange, Inc. [20]. The experiments are run on Intel Pentium Pro 200 with 64 MB RAM running Linux.

Also, as done in [21, 24], we used a one-dimensional array and a two-dimensional array to speed up the process of the first and the second pass correspondingly. The support counting phase runs very fast by using an array, since no searching is needed. No candidate generation process for 2-itemsets is needed because we use a two-dimensional array to store the support of all combinations of those frequent 1-itemsets. We start using the link-list data structure after the third pass. For a fair comparison, in all the cases, the number of candidates shown in the figures does not include the candidates in the first two passes. The number of the candidates in the Pincer-Search algorithm includes the candidates in the MFCS.

5.1.2 Scattered and Concentrated Distributions

We first concentrated on the synthetic databases, since they allow experimenting with different distributions of the frequent itemsets. For the same number of frequent itemsets, their distribution can be *concentrated* or *scattered*. In concentrated distribution, the frequent itemsets, having the same length, contain many common items: the frequent items tend to cluster. If the frequent itemsets do not have many common elements, the distribution is scattered. In

other words, the distribution is concentrated if there are only a few long maximal frequent itemsets, and is scattered if there are many short maximal frequent itemsets. By using the synthetic data generation program as in [3], we can generate databases with different distributions by adjusting various parameters. We will present experiments to examine the impact of the distribution type on the performance of the two algorithms.

In the first set of experiments, the number of the maximal frequent itemsets $|L|$ is set to 2000, as in [3]. The frequent

itemsets found in this set of experiments are rather scattered. To produce databases having a concentrated distribution of the frequent itemsets, we decrease the parameter $|L|$ to 50 in the second set of experiments. The minimum supports are set to higher values so that the execution time will not be too long.

5.1.3 Non-Monotone Property of the Maximum Frequent Set

For a given database, both the number of candidates and the number of frequent itemsets increase as the minimum support decreases. However, this is *not* the case for the number of the maximal frequent itemsets. For example, when minimum support is 9%, the maximum frequent set may be {{1,2}, {1,3}, {2,3}}. When the minimum support decreases to 6%, the maximum frequent set could become {{1,2,3}}. The number of the maximal frequent itemsets decreased from three to one.

This "nonmonotonicity" does not help bottom-up breadth-first search algorithms. They will have to discover the entire frequent itemsets before the maximum frequent set is discovered. Therefore, in those algorithms, the time, the number of candidates, and the number of passes will monotonically increase when the minimum support decreases.

However, when the minimum support decreases, the length of some maximal frequent itemsets may increase and our MFCS may reach them faster. Therefore, our algorithm *does have* the potential to benefit from this nonmonotonicity.

5.2 Experiments

The test databases are generated synthetically by an algorithm designed by the *IBM Quest* project [1]. The synthetic data generation procedure is described in detail in [3], whose parameter settings we follow. The number of items

N is set to 1000. $|D|$ is the number of transactions. $|T|$ is the average size of transactions. $|I|$ is the average size of maximal frequent itemsets. Thus, e.g., T10.I4.D100K specifies that the average size of transactions is ten, the average size of maximal frequent itemsets is four, and the database contains one hundred thousand transactions.

5.2.1 Scattered Distributions

The results of the first set of experiments, for scattered distributions, are shown in Fig. 12. In these experiments, Pincer-Search sometimes used more candidates than Apriori. That is because of the number of additional candidates used in the MFCS is more than the number of extra candidates pruned relying on the MFCS. The maximal frequent itemsets, found in the MFCS, are so short that not too many subsets can be pruned. However, the I/O time saved may compensate for the extra cost. Therefore, we may still get some improvement. In these experiments, the best case occurs when the minimum support is 0.33% and the database is T10.I4.D100K. In this case, Pincer-Search ran about 17% faster than the Apriori algorithm.

Depending on the distribution of the frequent itemsets, it is also possible that our algorithm might spend time counting the support of the candidates in the MFCS while still not finding enough maximal frequent itemsets from the MFCS to cover the extra costs. For instance, in the worst case of all the experiments, Pincer-Search ran 28% slower than the Apriori algorithm when the minimum support is 0.25% and the database is T20.I6.D100K.

5.2.2 Concentrated Distributions

In the second set of experiments we study the relative performance of the two algorithms on databases with concentrated distributions. The results are shown in Fig. 13. In the first experiment, we use the same parameters as the T20.I6.D100K database in the first set of experiments, but the parameter $|L|$ is set to 50. The improvements of Pincer-Search begin to increase. When the minimum support is 17%, our algorithm runs about 43% faster than the Apriori algorithm.

The non-monotone property of the maximum frequent set, considered in Section 5.1.3, impacts on this experiment. When the minimum support is 12%, both Apriori algorithm and Pincer-Search algorithm took eight passes to discover the maximum frequent set. But, when the minimum support decreases to 11%, the maximal frequent itemsets become

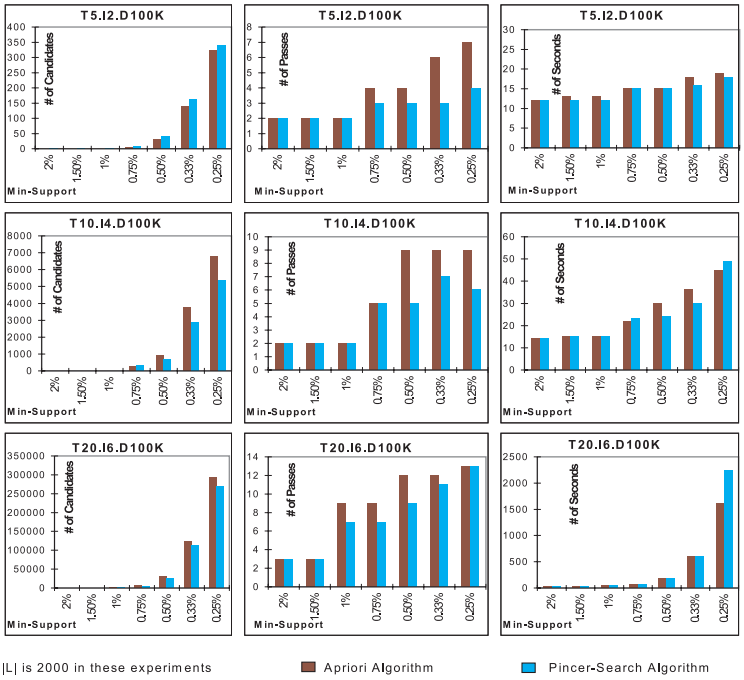

Figure 12: Scattered Distribution

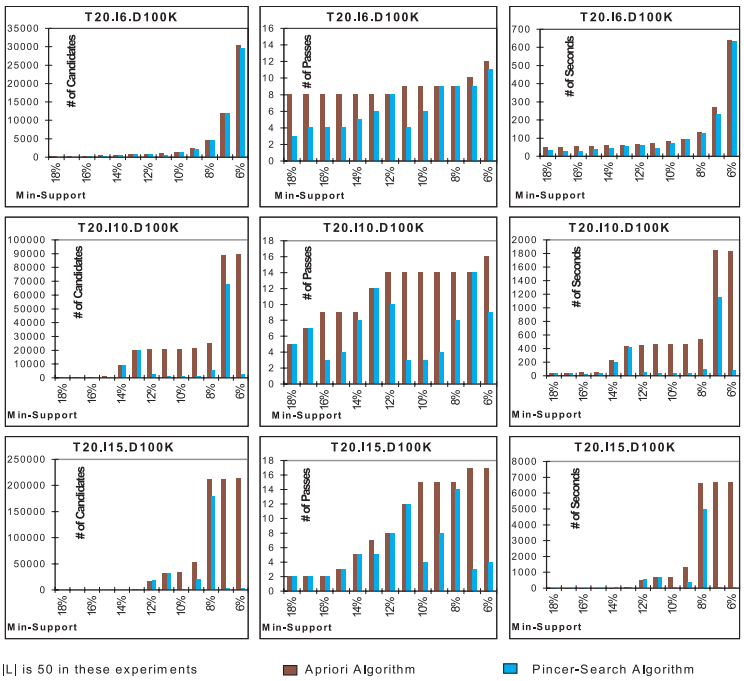

Figure 13: Concentrated Distribution

longer. This forced the Apriori algorithm to take more passes (nine passes) and consider more candidates to discover the maximum frequent set. In contrast, the MFCS allowed our algorithm to reach the maximal frequent itemsets faster. Pincer-Search took only four passes and considered fewer candidates to discover all maximal frequent itemsets.

We further increased the average size of the frequent itemsets in the next two experiments. The average size of the maximal frequent itemsets was increased to 10 in the second experiment and database T20.I10.D100K was used. The best case, in this experiment, is when the minimum support is 6%. Pincer-Search ran approximately 24 times faster than the Apriori algorithm. This improvement mainly came from the early discovery of maximal frequent itemsets which contain up to 16 items. Their subsets were not generated and counted in our algorithm. As shown in this experiment, the reduction of the number of candidates can significantly decrease both I/O time and CPU time.

The last experiment ran on database T20.I15.D100K. Pincer-Search took as few as three passes to discover all maximal frequent itemsets which contain as many as 17 items. This experiment shows improvements of more than two orders of magnitude when the minimum supports are 6% and 7%. One can expect even greater improvements when the average size of the maximal frequent itemsets is further increased.

5.2.3 Census Databases

A PUMS file, which contains public use microdata samples, was provided to us by Roberto Bayardo from IBM. This file contains actual census entries which constitute a five percent sample of a state that the file represents. This database is similar to the database looked at by Brin *et al.* [7]. As discussed in that paper, this PUMS database is quite hard. That is because a number of items in the census database appear in a large fraction of the database and therefore very many frequent itemsets will be discovered. From the distribution point of view, this means that some maximal frequent itemsets are quite long and the distribution of the frequent itemsets is quite scattered.

In order to compare the performance of the two algorithms within a reasonable time, we used the same approach as they proposed in the paper: we remove all items that have 80% or more support from the database. The experimented results are shown in Fig. 14.

In this real-life census database, the Pincer-Search algorithm also performs well. In all cases, the Pincer-Search algorithm used less time, fewer candidates, and fewer passes of reading the database. The overall improvement in

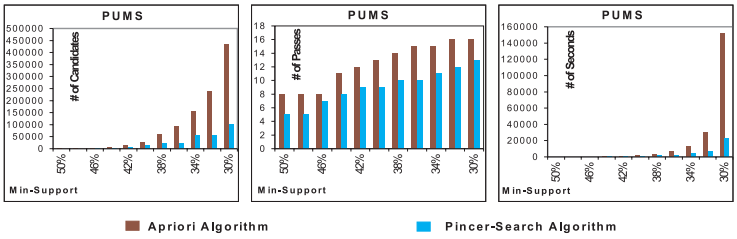

Figure 14: Census Database

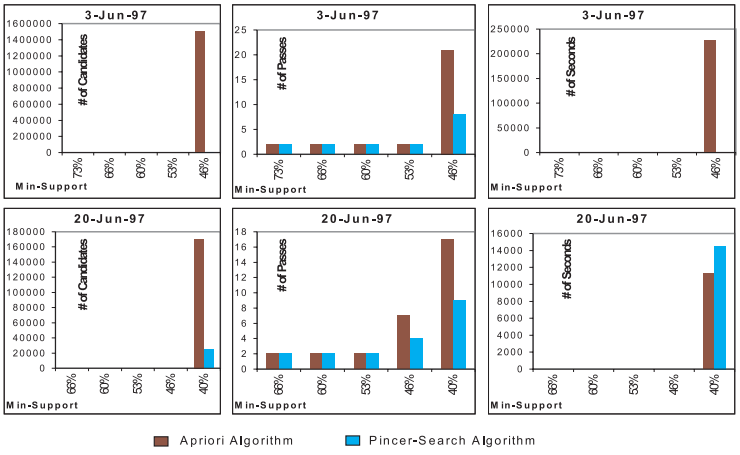

Figure 15: NYSE Databases (60-minute's interval)

performance ranges from 10% to 650%.

5.2.4 Stock Market Databases

We also conducted some experiments on the stock market databases. We used the trading data of June 1997, collected by the New York Stock Exchange, for the experiments. The trading data contains the symbol, price, volume, and time for every trade. Around 3000 stocks were chosen for these experiments. We are interested in discovering the pattern of price changes. A simplified problem can be stated as the followings: "What are those stocks the prices of which go up and down together during 2/3 of the time intervals in one day?"

Whether the price of a stock goes up or down is determined by looking at the first and the last trading price of the stock within an interval of time. If the last price is higher than the first price, then the price went up. If there is no trade for a stock during this period of time, then assume that the price is unchanged. Otherwise, the price went down.

Collect the stocks whose prices go up during this period of time and treat them as a transaction in our association rule mining problem. Do the same for the stocks whose prices went down during this period of time and form another transaction. Now, we can run the frequent set discovery algorithm to discover the price-changing patterns.

We ran experiments on every trading day of June 1997. Here, we only show the experiments on those days that reflect extreme cases when either Pincer-Search or Apriori performed best. The experiments on June 3, and June 20 are shown in Fig. 15. The minimum supports range from 40% to 73%. We cannot choose a fixed range of minimum support because of the data in each day are quite different from each other. The interval was set to 60 minutes.

For the experiments on June 3 data, Pincer-Search performed much better than the Apriori algorithm. For the experiments on June 20 data, even though the Pincer-Search algorithm used fewer passes and fewer candidates than the Apriori algorithm, the overhead of maintaining the MFCS costs too much. The Apriori algorithm performed better in this case. It is possible to adjust some of the parameters in the adaptive approach to reduce the differences.

Because of the number of records in the database is 15, which is very small, we have only a few available values on setting the minimum support. As we can see from these figures, there is a big jump in execution time when we decrease the minimum support down to some point. For instance, in the experiment of June 3, when the minimum support is decreased from 53% (8/15) to 46% (7/15), the execution time increases significantly. When we decrease the minimum support further, neither algorithm can complete the execution due to the limited size of the main memory. However, we suspect that Pincer-Search will do better when the minimum support is decreased (maximal frequent itemsets will be longer). In many cases, Pincer-Search did find many very long maximal frequent itemsets in very early passes. However, since the MFS is not complete, we cannot draw any definite conclusion.

In many of these experiments, Pincer-Search did use fewer passes. However, as the database is so small, the property of reducing the passes of reading the database for Pincer-Search algorithm did not contribute to the saving of the execution time. The improvement came purely from reducing the number of candidates.

In all the other experiments that are not shown here, both algorithms perform competitively, since the maximal frequent itemsets are all short. The totals of the execution time of all other experiments are 3,451,400 seconds and 3,176,103 seconds for Pincer-Search and Apriori respectively. The totals of the candidates used are 163,328 and 148,376, and the total passes used are 726 and 886 for Pincer-Search and Apriori respectively.

6 Related Work

6.1 Typical Algorithms for Frequent Set Discovery

We briefly discuss existing frequent set discovery algorithms in a roughly chronological order.

AIS and SETM Algorithms The problem of association rule mining was first introduced in [2]. An algorithm called AIS was given for discovering the frequent set. SETM algorithm [13] was later designed to use only standard SQL commands to find the frequent set. The Apriori algorithm [3], described above, performs much better than AIS and SETM.

The OCD Algorithm It is worth adding, that concurrently with the Apriori algorithm, OCD algorithm [19] used the same closure property to eliminate candidates.

DHP and Partition Algorithm DHP algorithm [23] extended the Apriori algorithm by introducing a hash filter for counting the upper-bound of the support of candidates in the next pass. Some candidates can be pruned before reading the database in the next pass.

Partition algorithm [26] proposed to divide the database into equal sized partitions. Each partition is processed independently to produce a *local frequent set* for that partition. After all local frequent sets are discovered, their union, the *global candidate set*, forms a superset of the actual frequent set. The database is then read again to produce the actual support for the global candidate set. The entire process takes only two (read) passes.

Sampling Algorithm Sampling Algorithm [27] proposed to consider first (small) samples of the database and discover an *approximate frequent set* by using a standard bottom-up approach algorithm. The approximate frequent set is then verified against the entire database. False frequent itemsets need to be removed and missing frequent itemsets need to be recovered.

A-Random-MFS, DIC, and MaxClique Algorithms *A-Random-MFS* algorithm [9] is a randomized algorithm for discovering the maximum frequent set. A single run of the algorithm cannot guarantee correct results. A complete

algorithm requires repeatedly calling the randomized algorithm until no new maximal frequent itemset can be found.

Dynamic itemset counting (DIC) algorithm [7] combines candidates of different lengths into one pass. The database is divided into partitions of equal size. In each pass, after the first i partitions are read, some itemsets containing up to $i + 1$ items may become candidates based on the database partitions read so far.

MaxClique [29] used a hybrid traversal which contains a look-ahead phase followed by a pure bottom-up phase. The look-ahead phase consists of extending the frequent 2-itemsets until the extended itemset becomes infrequent. After the look-ahead phase, an Apriori-like traversal is executed.

One of the most important differences between MaxClique and Pincer-Search is that MaxClique only looks ahead at some long candidate itemsets during the initialization stage (in the second pass). In contrast, the Pincer-Search algorithm repeatedly maintains the upper-bound of the frequent itemsets (MFCS) throughout the entire process. The look-ahead candidate itemsets are dynamically adjusted based on all available information discovered so far. In fact, the MFCS is the most accurate approximation one can get while no additional knowledge of the data is available.

Another important difference is that MaxClique used a bottom-up approach to calculate the look-ahead candidate itemsets. Conceptually, it keeps applying Apriori-gen until no more candidates can be generated. In contrast, Pincer-Search uses a top-down approach. As will be discussed in the next section, this top-down approach has the advantage that it is suitable for incremental updates. It updates the MFCS only when a new infrequent itemset is discovered. Ignoring implementation details, MaxClique can be viewed as a special case of Pincer-Search.

Max-Miner *Max-Miner* algorithm [6] was recently proposed to discover the maximum frequent set. This algorithm partitions the candidate set into groups with the same prefix. Like Pincer-Search, it looks ahead at some long candidate itemsets throughout the search. The main difference is the long candidate itemsets that it examines. Max-Miner looks ahead at longest itemsets that can be constructed from every group. A frequency heuristic is used to reorder the items such that the most frequent items appear in the most candidate groups. According to the experiments in the paper, this item-reordering heuristic improves the performance dramatically. So far, we only had the opportunity to perform preliminary comparison with the Max-Miner from the algorithmic point of view. We feel that Max-Miner and Pincer-Search could be complementary. One of the possibilities is to run Max-Miner in the first few passes and switch to

Pincer-Search for the later passes.

6.2 Other Related Work

In addition to the algorithms discussed so far, there has been extensive research relating to the problem of association rule mining such as [10, 21]. Similar candidate pruning techniques has been applied to discover sequential patterns (e.g., [17, 25]) and episodes (e.g., [17]). Some other papers concentrate on designing parallel algorithms on share-nothing parallel environment (e.g., [4, 11]) and share-memory parallel environment (e.g., [28]). The discovery of frequent set is a key process in solving these problems, thus speeding up this discover is important.

The complexity of (level-wise) bottom-up breadth-first search style algorithms was analyzed in [18]. As our algorithm does not fit in this model, their complexity low bound does not apply to it.

Our work was inspired by the notion of *version space* [15]. We found that if we treat a newly discovered frequent itemset as a new *positive training instance*, a newly discovered infrequent itemset as a new *negative training instance*, the candidate set as the *maximally specific generalization* (S), and the MFCS as the *maximally general generalization* (G), then we will be able to use a two-way approaching strategy to discover the maximum frequent set (*generalization*) efficiently.

7 Concluding Remarks

7.1 Summary

An efficient way to discover the maximum frequent set can be very useful in various data mining problems, such as the discovery of the association rules, theories, strong rules, episodes, and minimal keys. The maximum frequent set provides a unique representation of all the frequent itemsets. In many situations, it suffices to discover the maximum frequent set, and once it is known, all the required frequent subsets can be easily generated.

In this paper, we presented a novel algorithm that can efficiently discover the maximum frequent set. Our Pincer-Search algorithm could reduce both the number of times the database is read and the number of candidates considered. Experiments show that the improvement of using this approach can be very significant, especially when some maximal

frequent itemsets are long.

A popular assumption is that the maximal frequent itemsets are usually very short and therefore the computation of *all* (and not just maximal) frequent itemsets is feasible. Such assumption on maximal frequent itemsets does not need to be true in important applications. Consider for example the problem of discovering patters in price changes of individual stocks in a stock market. Prices of individual stocks are frequently quite correlated with each other. Therefore, the discovered patterns may contain many items (stocks) and the frequent itemsets are long. We expect our algorithm will be useful in these applications.

7.2 Possible Extensions

The performance of the Pincer-Search algorithm in applications of discovering other price changing patterns in stock markets will be studied. The maximal frequent itemsets in many instances of such applications are likely to be long. Therefore we expect the algorithm to provide dramatic performance improvements.

Many classification problems as discussed in [5] tend to have long patterns. It is worthwhile to study the performance of the Pincer-Search algorithm on these problems.

In general, if some maximal frequent itemsets are long and the maximal frequent itemsets are distributed in a scatterd manner, then the problem of discovering the MFS can be very hard. In this case, even Pincer-Search might not be able to solve it efficiently. Parallelizing the Pincer-Search algorithm might be a possible way to solve this hard problem. We propose to divide the candidate set in such a way that all the candidates that are subsets of an itemset in the MFCS are assigned to a same processor.

Although there might be some duplicate calculations, this way of partitioning the candidates can have the benefits that no synchronization or communication among processors is needed. Each processor can run totally independent. The issues to study would be the way to minimize the duplicate calculations and to maximize the use of available processors.

8 Acknowledgments

This research was partially supported by the National Science Foundation under grant number CCR-94-11590, by Intel Corporation, and by Microsoft. We thank Rakesh Agrawal, Roberto J. Bayardo, and Ramakrishnan Srikant for kindly providing us the experimental data. We thank Thomas Anantharaman and Sridhar Ramaswamy for their very valuable comments and suggestions. We thank also Sridhar Ramaswamy for his proposal for the term *Pincer-Search*.

References

[1] R. Agrawal, A. Arning, T. Bollinger, M. Mehta, J. Shafer, R. Srikant. The Quest Data Mining System. In *Proc. 2nd KDD*, Aug. 1996.

[2] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. SIGMOD*, May 1993.

[3] R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proc. 20th VLDB*, Sept. 1994.

[4] R. Agrawal and J. Shafer. Parallel mining of association rules. *IEEE Trans. on Knowledge and Data Engineering*, Jan. 1996.

[5] R. Bayardo. Brute-force mining of high-confidence classification rules. In *Proc. 3rd KDD*, Aug. 1997.

[6] R. Bayardo. Efficient Mining Long Patterns from Databases. In *Proc. SIGMOD*, June 1998.

[7] S. Brin, R. Motwani, J. Ullman, and S. Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proc. SIGMOD*, May 1997.

[8] U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthrusamy (Eds.). *Advances in Knowledge Discovery and Data Mining*. AAAI Press, Menlo Park, CA, 1996.

[9] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all most specific sentences by randomized algorithm. In *Proc. 13th ICDT*, Jan. 1997.

[10] J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proc. 21st VLDB*, Sept. 1995.

[11] E. Han, G. Karypis, and V. Kumar. Scalable parallel data mining for association rules. In *Proc. SIGMOD*, May 1997.

[12] K. Hätönen, M. Klemettinen, H. Mannila, P. Ronkainen, and H. Toivonen. Knowledge Discovery from Telecommunication Network Alarm Databases. In *Proc. 12th ICDE*, Feb. 1996.

[13] M. Houtsma and A. Swami. Set-oriented mining of association rules. *Research Report RJ 9567*, IBM Almaden Research Center, Oct. 1993.

[14] D. Lin and Z. Kedem. Pincer-Search: A new algorithm for discovering the maximum frequent set. In *Proc. 6th EDBT*. Mar. 1998.

[15] T. Mitchell. Generalization as search. *Artificial Intelligence*, Vol. 18, 1982.

[16] A. Mueller. Fast sequential and parallel algorithms for association rule mining: A comparison. *Technical Report No. CS-TR-3515* of CS Department, University of Maryland-College Park.

[17] H. Mannila and H. Toivonen. Discovering frequent episodes in sequences. In *Proc. 1st KDD*, Aug. 1995.

[18] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Technical Report TR C-1997-8*, Dept. of Computer Science, U. of Helsinki, Jan. 1997.

[19] H. Mannila, H. Toivonen, and A. Verkamo. Improved methods for finding association rules. In *Proc. AAAI Workshop on Knowledge Discovery*, July 1994.

[20] The TAQ Database Release 1.0 in CD-ROM, New York Stock Exchange, Inc., June 1997.

[21] B. Özden, S. Ramaswamy. and A. Silberschatz. Cyclic Association Rules. In *Proc. 14th ICDE*, Feb. 1998.

[22] G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. *Knowledge Discovery in Databases*, AAAI Press, 1991.

[23] J. Park, M. Chen, and P. Yu. An effective hash-based algorithm for mining association rules. In *Proc. ACM-SIGMOD*, May 1995.

[24] R. Srikant and R. Agrawal. Mining generalized association rules. In *Proc. 21st VLDB*. Sep. 1995.

[25] R. Srikant and R. Agrawal. Mining Sequential Patterns: Generalizations and Performance Improvements. In *Proc. 5th EDBT*. Mar. 1996.

[26] A. Sarasere, E. Omiecinsky, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. 21st VLDB*, Sept. 1995.

[27] H. Toivonen. Sampling large databases for association rules. In *Proc. 22nd VLDB*, Sept. 1996.

[28] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. Parallel data mining for association rules on shared-memory multi-processors. *Technical Report 618* of the Department of Computer Science, University of Rochester. May 1996.

[29] M. Zaki, S. Parthasarathy, M. Ogihara, and W. Li. New algorithms for fast discovery of association rules. In *Proc. 3rd KDD*, Aug. 1997.