# Pincer-Search: A New Algorithm for Discovering the Maximum Frequent Set

Dao-I Lin* Zvi M. Kedem†
New York University New York University

September 11, 1997

**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 (minimal 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 reasonably 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 top-down directions. The main search direction is still bottom-up but a restricted search is conducted in the top-down direction. This search is used only for maintaining and updating a new data structure we designed, the maximum frequent candidate set. It is used to prune candidates in the bottom-up search. As a very important characteristic of the algorithm, it is not necessary to explicitly examine every frequent itemset. Therefore it 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, which therefore specifies immediately all frequent itemsets. We evaluate the performance of the algorithm using a well-known benchmark database. The improvements can be up to several orders of magnitude, compared to the best current algorithms.

## 1 Introduction

A key component of many data mining problems can be formulated as follows. Given a large database of sets of items (representing market basket data, episodes, etc.), discover all the frequent *itemsets* (sets of items), where a frequent itemset is one that occurs more than a user-defined number of times (minimum *support* in 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 association rules (e.g., [3] [12]), strong rules (e.g., [14]), episodes (e.g., [10]) and minimal keys (e.g., [11]).

Typical algorithms for finding the *frequent set*, i.e., the set of all frequent itemsets (e.g., [3] [12]), operate in a *bottom-up* breadth-first fashion. In other words, the computation starts from frequent 1-itemsets (minimal 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

*Department of Computer Science, Courant Institute of Mathematical Sciences, New York University, 251 Mercer St., New York, NY 10012-1185, +1 908 872 7760, lindaoi@cs.nyu.edu.

†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.

algorithms perform reasonably well. However, performance drastically decreases when any of the maximal frequent itemsets becomes longer. This is due to the fact that a maximal frequent itemset of size $l$ implies the presence of $2^l - 2$ non-trivial frequent itemsets (its nontrivial subsets) as well, each of which is explicitly examined by such algorithms.

Of course the set of the maximal (length) frequent itemsets (called *maximum frequent set*) uniquely defines the entire frequent itemsets: frequent itemsets are precisely all the non-empty subsets of its elements. Thus trivially, discovering the maximum frequent set implies immediate discovery of all the maximal frequent itemsets. In many situations, it suffices to know the supports of the maximal frequent itemsets, and the supports of a few subsets of the maximal frequent itemsets, rather than the supports of *all* the frequent itemsets. We will use MFS to denote the maximum frequent set.

Therefore, instead of examining and "assembling" all the frequent itemsets, an alternative approach might be to "shortcut" the process and attempt to search for MFS "more directly." Discovering the maximum frequent set is essentially a search problem in a hypothesis search space (a binomial graph). 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).

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

In our algorithm, the bottom-up search is similar to the *Apriori* [3] and the *OCD* [12] algorithms. However, the top-down search is unique. The top-down search is implemented efficiently by introducing a set that we call the *maximum-frequent-candidate-set* or MFCS. This set consists of the *minimum* number of itemsets such that the union of all their subsets contains all the known (discovered so far) frequent itemsets but not any of the known infrequent itemsets. The known frequent and infrequent itemsets are those determined by the supports of the itemsets that have been discovered until the most recent pass. Thus obviously at any point of the algorithm MFCS is a superset of MFS. When the algorithm terminates, MFCS and MFS are equal.

By incorporating the computation of MFCS in our algorithm, we are able to efficiently approach MFS from both top-down and bottom-up directions. Unlike the bottom-up search that goes up one level in each pass, MFCS set can "move down" many level in one pass, as we explain in Section 3.

In this paper, we apply the MFCS idea to association rules mining and to the discovery of frequent itemsets in market basket data. Popular benchmark databases designed by Agrawal and Srikant [3] have been used in [9], [16], and [18]: we use these 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 can reduce 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 MFCS.

The organization of the rest of the paper is as follows. The procedures for mining association rules, the cost of the processes, and the properties that can be used to reduce the cost will be discussed in Section 2. Section 3 will describe our algorithm and the ideas behind it. Two technical issues and the techniques to address them will also be discussed in this section. Section 4 presents the results of our experiments. Section 5 briefly discusses the related research. Finally, Section 6 concludes this paper.

## 2 Association Rule Mining

This section briefly introduces the association rule mining problem. To the extent feasible, we follow the terminology of [1].

## 2.1 The Setting of the Problem

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of $m$ distinct items. A *transaction* $T$ is defined as any subset of items in $I$. A database $D$ is 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 it contains all items of $X$, i.e., $X \subseteq T$. The fraction of the transactions in $D$ that support $X$ is called the *support* of $X$, denoted as support($X$). An itemset is *frequent* if its support is above some user-defined minimum support threshold. Otherwise, it is *infrequent*.

An *association rule* has the form $R : X \rightarrow Y$, where $X$ and $Y$ are two non-empty and non-interesecting itemsets. The *support for rule* $R$ is defined as support($X \cup Y$). A *confidence* factor defined as support($X \cup Y$)/support($X$), is used to evaluate the strength of such association rules. The semantics of the confidence of a rule indicates how often it can be expected to apply, while its support indicates how trustworthy this rule is.

The problem 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. Association rules that satisfy these requirements are *interesting*.

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

1. the discovery of frequent itemsets, followed by

2. the generation of association rules.

The *maximum frequent set* (MFS) is the set of all the *maximal frequent itemsets*. (An itemset is a maximal frequent itemset if it is frequent and no proper superset of it is frequent.) Obviously, an itemset is frequent if and only if it is a subset of a maximal frequent itemset. Thus, it is necessary to discover only the maximum frequent set during the first stage. Of course, an algorithm for that stage may explicitly discover and store some other frequent itemsets as a necessary part of its execution—but minimizing such effort may increase efficiency.

It is also interesting to note that an efficient way of generating interesting association rules is by examining the maximum frequent set first, and then proceeding to their subsets if the generated rules have the required confidence. Usually, only a few rules will be interesting. Therefore, while generating rules, all one needs to know is the support of the maximal frequent itemsets and of the itemsets "a little" shorter. If the maximum frequent set is known, one can easily generate the required subsets and count their supports by reading the database once, which is quite straightforward.

It follows, that in the vast majority of cases, the discovery of the maximum frequent set dominates the performance of the whole process. Therefore, we explicitly focus the paper on the discovery of such set.

## 2.2 A Common Approach to the Discovery of Frequent Itemsets

A typical frequent itemsets discovery process follows a standard scheme. Throughout the execution, the set of all itemsets is partitioned, perhaps implicitly, into 3 sets:

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

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

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

Initially the frequent and the infrequent sets are empty. The process terminates when every itemset is either in the frequent set or in the infrequent set.

We now briefly sketch a realization of this process as in, e.g., [3]. This is 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 $k+1$ such that two of whose subsets of size $k$ are frequent are generated. Some of these itemsets are *pruned*, as they do not need to be processed further. Specifically, itemsets that are supersets of infrequent itemsets are pruned (and discarded), as of course they are infrequent. The remaining itemsets form the set of *candidates* for this pass. As the second step the support for these itemsets is computed, and they are classified as either frequent or infrequent. Note that *every* frequent itemset is a candidate at some pass, and is *explicitly* considered..

The support of the candidate is computed by reading the database. The cost of the frequent itemsets discovery process comes from the reading of the database (I/O time) and the generation of new candidates (CPU time). 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. Thus reducing the number of candidates is of critical importance for the efficiency of the process.

### 2.3 Our Approach to Reducing the Number of Candidates

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 observations can be used to immediately classify some of the unclassified itemsets:

*Observation 1:* If an itemset is infrequent, all it supersets must be infrequent, and they do not need to be examined further

*Observation 2:* If an itemset is frequent, all its subsets must be frequent, and they do not need to be examined further

Note that the bottom-up process described above uses only the first observation to reduce the number of candidates. Conceivably, a process that relies on both observation to prune candidates could be much more efficient than a process that relies on *only* the first or the second.

## 3 A New Algorithm for Discovering the Maximum Frequent Set

### 3.1 Combining Top-down and Bottom-up Searches

It is possible to search for maximal frequent itemsets either *bottom-up* or *top-down*. If all maximal frequent itemsets are expected to be small (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.

In a "pure" bottom-up approach, only Observation 1 above is used to prune candidates. This is the technique that existing algorithms ([3] [4] [6] [9] [11] [12] [15] [16] [17]) use to decrease the number of candidates. In a "pure" top-down approach, only Observation 2 is used to prune candidates. We will show that by combining both approaches we will be able to make use of the information gathered 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 itemsets can be pruned because they are frequent (Observation 2). Of course, if an infrequent itemset is found in the bottom-up direction, then this infrequent itemset can be used to eliminate the candidates in the top-down direction found so far (Observation 1). This "two-way approaching" method can fully make use of both observations and thus speed up the search for the maximum frequent set.

We have designed a combined 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* MFS*. Some itemsets are frequent, some infrequent, and some unclassified. The maximum-frequent-candidate-set* (MFCS) *is a minimum cardinality set of itemsets such that the union of all the subsets of its elements contains all the frequent itemsets but does not contain any infrequent itemsets, that is, it is a minimum cardinality set satisfying the conditions*

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

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

*where* FREQUENT *and* INFREQUENT*, stand respectively for all frequent and infrequent itemsets (classified as such so far).*

Thus obviously at any point of the algorithm MFCS is a superset of MFS. When the algorithm terminates, MFCS and MFS are equal.

The computation of our algorithm follows the bottom-up (breadth-first) search approach. We base our presentation on the *Apriori* algorithm [3], 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 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 in detail later in this paper.

Consider now some pass $k$, during which itemsets of size $k$ are to be classified. If some itemset that is an element of MFCS, say $X$ of cardinality greater than $k$ is found to be frequent in this pass, then all its subsets must be frequent. Therefore, all of 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 finally 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 MFCS. The subsets of MFCS must not contain this infrequent itemset.

Figure 1 conceptually shows the combined two-way search in the search space. 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), MFCS will have one element of cardinality $n - m$. This itemset is generated by removing the $m$ infrequent items from the initial element of 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.*

By using the MFCS, we will 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 work efficiently, we need to address two issues. First, how to update MFCS efficiently? Second, once the subsets of the maximal frequent itemsets found in the MFCS are removed, how do we generate the correct candidate set for the subsequent passes in the bottom-up direction?

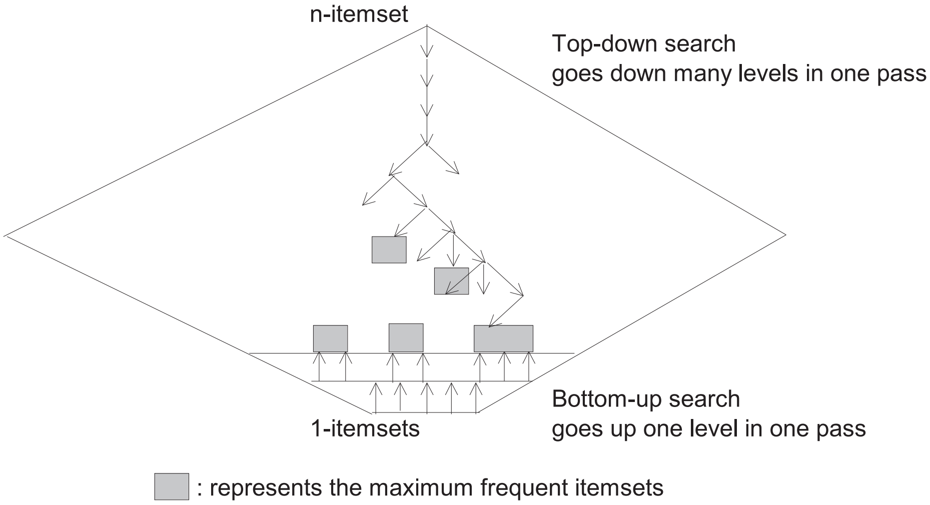



**Figure 1. The search space**

### 3.2 Updating MFCS Efficiently

Consider some itemset $Y$ that has been "just" classified as infrequent and assume that it is a subset of some itemset that is an element of MFCS. To update MFCS, we replace $X$ by $|Y|$ itemsets, each obtained by removing from $X$ a single item (element) of $Y$. We do this for each newly discovered infrequent itemset and each of its supersets that is an element of MFCS. Formally, we have the following *MFCS-gen* algorithm (shown here for pass $k$).

**Algorithm**: *MFCS-gen*
Input: Old MFCS and the infrequent set $S_k$ discovered in pass $k$
Output: New MFCS

```
for all itemsets s ∈ S_k begin
    for all itemsets m ∈ MFCS begin
        if s is a subset of m begin
            MFCS := MFCS \ {m}
            for all items e ∈ itemset s
                if m \ {e} is not a subset of any itemset in the MFCS set
                    MFCS := MFCS ∪ {m \ {e}}
        end
    end
end
return MFCS
```

**Example** Suppose $\{\{1,2,3,4,5,6\}\}$ is the current ("old") value of 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 MFCS) contains items 1 and 6, one of its subsets will be $\{1,6\}$. By removing item 1 from itemset $\{1,2,3,4,5,6\}$, 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 $\{1,6\}$, 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 $\{2,3,4,5,6\}$. The itemset $\{2,3,4,5\}$ is a subset of the itemset $\{1,2,3,4,5\}$ in the new MFCS, and it will be removed from MFCS. Therefore,

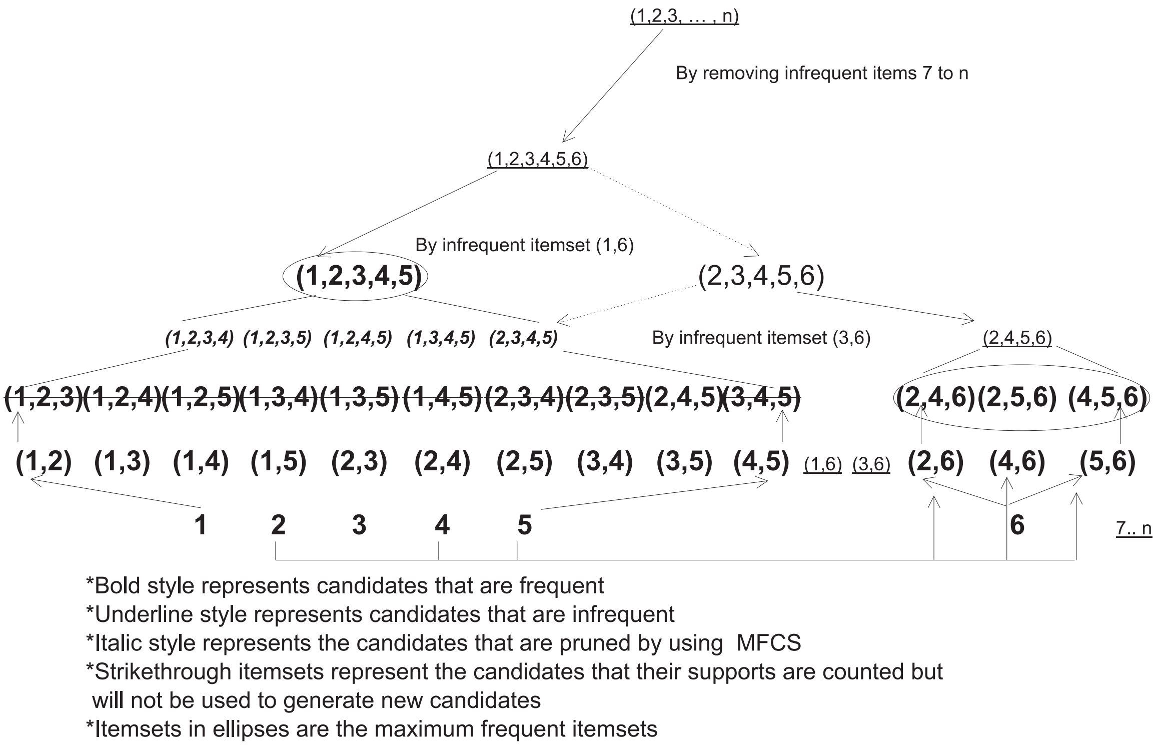



**Figure 2. Two-way search**

MFCS becomes {{1,2,3,4,5}, {2,4,5,6}}. The top-down arrows in Figure 2 show the updates of MFCS.

**Lemma 1** *The algorithm MFCS-gen correctly updates* 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. Therefore, this algorithm generates MFCS correctly.

## 3.3 Some Salient Features of the Apriori-gen Algorithm

We first recall that itemsets are maintained as sequences in sorted lexicographical order, and the algorithm relies on this fact.

The candidate generation algorithm is schematically as follows:

**Algorithm** (from [3]): Candidate generation algorithm
Input: $L_k$, the set containing frequent itemsets found in pass $k$
Output: new candidate set $C_{k+1}$
1. call the *join* procedure (see below)
2. call the *prune* procedure (see below)

The *join* procedure of the Apriori-gen algorithm is to combine two frequent $k$-itemsets, which have the same $(k-1)$-prefix, to generate a $(k+1)$-itemset as a new candidate. The *join* procedure works as follows:

**Algorithm**(from [3]): 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 */
for i from 1 to |L_k - 1| begin
    for j from i + 1 to |L_k| begin
        if L_k.itemset_i and L_k.itemset_j have the same (k - 1)-prefix
            C_{k+1} := C_{k+1} ∪ {L_k.itemset_i ∪ L_k.itemset_j}
        else
            break
    end
end
```

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 in the frequent set $L_k$. In other words, the supersets of an infrequent itemset are pruned. The procedure is specified by

**Algorithm**(from [3]): 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

```
for all itemsets c in C_{k+1}
    for all k-subsets s of c
        if s ∉ L_k
            delete c from C_{k+1}
```

## 3.4 A New Candidate Generation Method

In our algorithm, after a maximal frequent itemset is added to MFCS, all of its subsets in the frequent set (as computed so far) will be removed. We show by example that if the original *join* procedure is applied some of the needed itemsets could be missing from the candidate set. Consider Figure 2. Suppose 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 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 candidate set should be {{2,4,5,6}}.

The full set of required candidates can be obtained by restoring some itemsets to the current frequent set. They will be extracted from MFCS, which implicitly maintains all frequent itemsets discovered so far. The itemsets that need to be restored are precisely those $k$-itemsets that have the same $(k-1)$-prefix as an itemset in the current frequent set.

Consider then in pass $k$, an itemset $X$ in MFCS 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 $j$'th item of $X$. We determine 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 $Y$ to get one of these $k$-subsets. After these $k$-itemsets are found, we generate candidates by combining them with the itemset $Y$. This is specified by the following *recovery* procedure.

```
Algorithm: The recovery procedure
Input: C_{k+1}, L_k, and current MFS containing the maximal frequent itemsets discover until pass k
Output: a complete candidate set C_{k+1}
for all itemsets l in L_k
    for all itemsets m in MFS
        if the first k − 1 items in l are also in m
            /* suppose m.item_j = l.item_{k−1} */
            for i from j + 1 to |m|
                C_{k+1} := C_{k+1} ∪ {{l.item_1, l.item_2, ..., l.item_k, m.item_i}}
```

**Example** See Figure 2. MFCS 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 the 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 the itemset $\{2,4,6\}$. By combining $\{2,4,5\}$ and $\{2,4,6\}$, we recover the missing candidate $\{2,4,5,6\}$. No itemset needs to be restored for itemsets $\{2,5,6\}$ and $\{4,5,6\}$.

As stated before, we will not consider the subsets of a maximal frequent itemset as candidates. Therefore, the *prune* procedure in our candidate generation algorithm will remove those subsets.

```
Algorithm: The new prune procedure
Input: L_k, current MFS, and C_{k+1} generated from the join and recovery procedures 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.     if c is a subset of any itemset in current MFS
3.             delete c from C_{k+1}
4.     else
5.         for all k-subsets s of c
6.             if s ∉ L_k
7.                 delete c from C_{k+1}
```

In addition to eliminating the supersets of infrequent itemsets (line 7) from the candidate set, we also eliminate the candidates that are subsets of elements of current MFS (line 3). In summary, our candidate generation process contains the following three steps:

**Algorithm**: New candidate generation algorithm
Input: $L_k$ and current MFS
Output: new candidate set $C_{k+1}$

1. call the *join* procedure of the Apriori-gen algorithm
2. call the *recovery* precedure if necessary
3. call the new *prune* procedure

**Lemma 2** *The new candidate generation algorithm generates complete candidate set.*

**Proof:** Since this algorithm considers the same combinations of the frequent itemsets as the Apriori-gen algorithm, all candidates will be generated.

### 3.5 The Pincer-Search Algorithm

We now present our complete algorithm, *The Pincer-Search Algorithm*, which relies on the combined approach for determining the maximum frequent set.

**Algorithm**: The Pincer-Search algorithm
Input: a database and a user-defined minimum support
Output: MFS which contains all maximal frequent itemsets

1. $L_0 := \emptyset$; $k := 1$
2. $C_1 := \{\{i\} \mid i \in I\}$
3. MFCS $:= \{\{1, 2, \ldots, n\}\}$
4. MFS $:= \emptyset$
5. **while** $C_k \neq \emptyset$ **begin**
6.     read database and count supports for itemsets in $C_k$ and MFCS
7.     MFS := MFS $\cup$ {frequent itemsets in MFCS}
8.     $L_k :=$ {frequent itemsets in $C_k$} $\setminus$ {subsets of itemsets in MFS}
9.     $S_k :=$ {infrequent itemsets in $C_k$}
10.     call the *join* procedure of the Apriori-gen to generate $C_{k+1}$
11.     **if** any frequent itemset in $C_k$ is a subset of an itemset in MFS (test in line 8)
12.         call the *recovery* procedure to recover candidates to $C_{k+1}$
13.     call new *prune* procedure to prune candidates in $C_{k+1}$
14.     call the *MFCS-gen* algorithm when $S_k$ is not empty
15.     $k := k + 1$
16. **end**
17. **return** MFS

MFCS is initialized to contain one itemset which consists of all the database items. MFCS is updated whenever new infrequent itemsets are found (line 14). If an itemset in MFCS is found to be frequent, then its subsets will not participate in the subsequent support counting and candidate set generation steps. Line 8 and line 13 will exclude those itemsets that are subsets of any itemset in the current MFS set, 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 12).

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

**Proof:** Immediate from Lemma 1 and Lemma 2.

In general, one may not want to use the "pure" version of the Pincer Search algorithm. For instance, in some case there may be many 2-itemsets, but only a few of them are frequent. In this case it may not be worthwhile to maintain the MFCS, since there will not be many frequent itemsets to discover. In that case, we may simply count candidates of different sizes in one pass, as in [3] and [12]. The algorithm we have implemented is in fact an adaptive version of the algorithm described above. This adaptive version does not maintain the MFCS, when doing so would be counterproductive. This is also the algorithm whose performance is being evaluated in Section 4. Thus the very small overhead of deciding when to use the MFCS is accounted in the performance evaluation of our Pincer-Search algorithm.

# 4 Performance Evaluation

A key 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. However, according to both [3] and our experiments, a large fraction the 2-itemsets will usually be infrequent. These infrequent itemsets will cause 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 MFCS in very early passes. For instance, in the experiment on database T20.I15.D100K (Figure 4), 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 [3]. We restrict this comparison both for space limitation and because it is sufficiently instructive to understand the characteristics of the new algorithm's performance.

## 4.1 Preliminary Discussion

### 4.1.1 Auxiliary Data Structures Used

Since we are interested in studying the effect of using MFCS to reduce the number of candidates and the number of passes, we didn't use more efficient data structures, such as hash tables, to store the itemsets. We simply used a link-list data structure to store the frequent set and the candidate set. Some of the execution time (for both algorithms), shown in Figure 3, is greater than the execution time shown in [3]. However, since both Apriori and Pincer-Search algorithms are using the same data structure, the comparison is fair. The databases used in performance evaluation, are the synthetic databases used in [3].

Also, as suggested by Özden *et al.*'s [13], 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. We start using a link-list data structure after the third pass. For a fair comparison, 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 MFCS.

### 4.1.2 Scattered and Concentrated Distributions

For the same number of frequent itemsets, their distribution can be *concentrated* or *scattered*. In concentrated distribution, on each level the frequent itemsets have many common items: the frequent items tend to cluster. If the frequent itemsets do not have many common elements, the distribution is scattered. We will present experiments to examine the impact of the distribution type on the performance of the two algorithms.

The number of the maximal frequent itemsets $|L|$ is set to 2000, as in [3], in the first set of experiments. The frequent itemsets found in this set of experiments are rather scattered. To produce databases having a concentrated distribution of the frequent itemsets, we adjust the parameter $|L|$ to a smaller value. The value of $|L|$ is set 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.

### 4.1.3 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 can become $\{\{1,2,3\}\}$. The number of the maximal frequent itemsets decreased from 3 to 1.

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 be monotonically increased 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.

## 4.2 Experiments

The test databases are generated synthetically by an algorithm designed by the **IBM Quest** project. 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.

**Scattered Distributions.** The results of the first set of experiments are shown in Figure 3. The best improvement shown in Figure 3 occurs for T10.I4.D100K database and the minimum support of 0.5%. Pincer-Search runs 1.7 times faster than the Apriori. The improvement came from reducing the number passes of reading the database and the number of candidates.

In the experiment on database T5.I2.D100K, Pincer-Search used more candidates than Apriori. That is because of the number of additional candidates used in MFCS is more than the number of extra candidates pruned relying on MFCS. The maximal frequent itemsets, found in the MFCS, are so short that they don't have too many subsets to be pruned. However, the I/O time saved more than compensated for the extra cost. Therefore, we still get improvements.

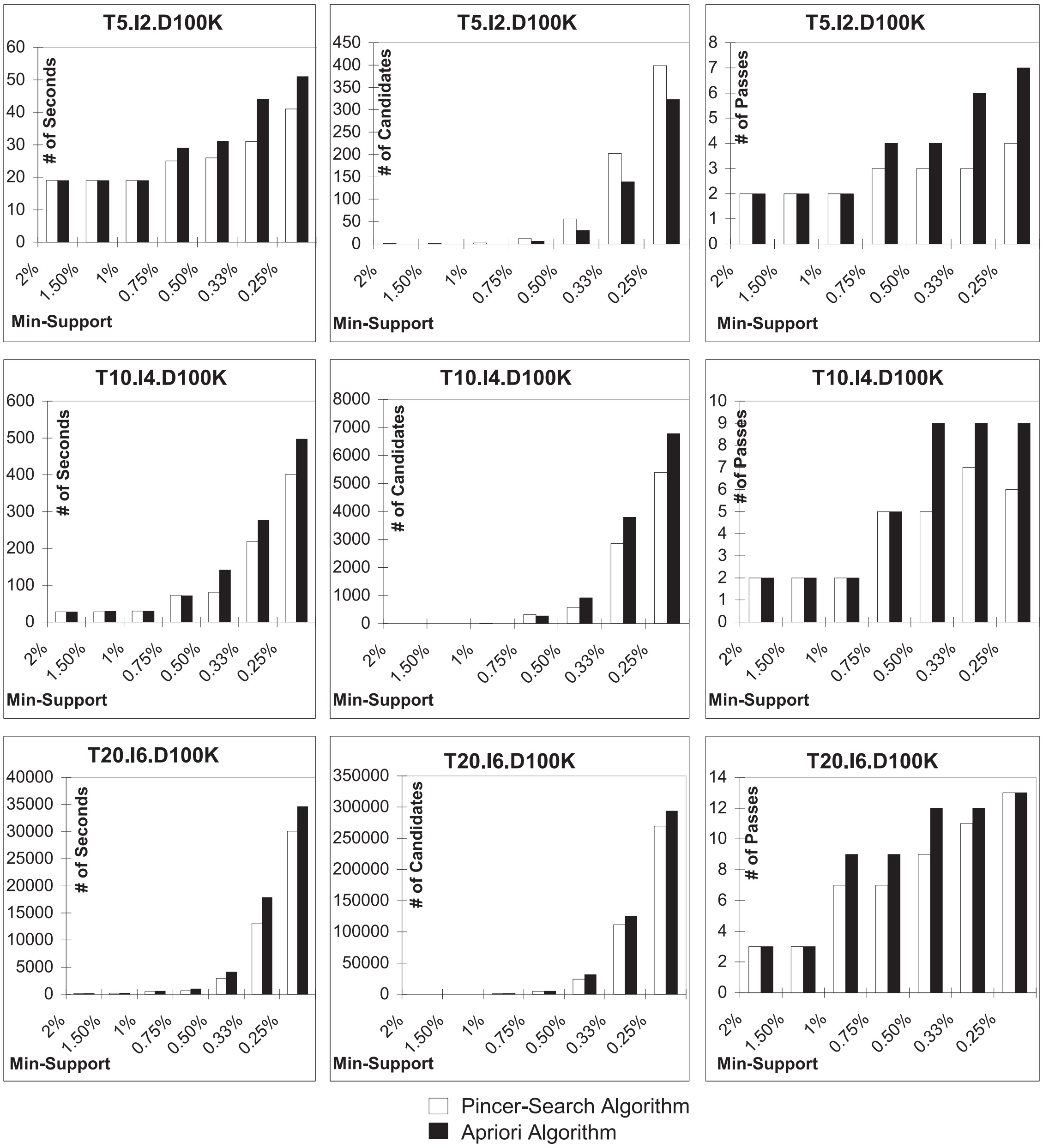
Depending on the distribution of the frequent itemsets, it is also possible that our algorithm might spend efforts on counting the support of the candidates in MFCS but does not find any maximal frequent itemset from MFCS. For instance, our algorithm took more time than the Apriori algorithm on the case when the minimum support is set to 0.75% and the database is T10.I4.D100K. However, since there were only a few candidates in MFCS, the difference is quite small.

**Concentrated Distributions.** In the second set of experiments we study the relative performance of the two algorithms on databases with such distributions. The results are shown in Figure 4. In the first experiment, we use the same T20.I6.D100K database as 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 18%, our algorithm runs about 2.3 times faster than the *Apriori* algorithm.

The non-monotone property of the maximum frequent set, considered in Section 4.1.3, reflects on this experiment. When the minimum support is 12%, both the Apriori and the Pincer-Search algorithms took 8 passes to discover the maximum frequent set. But, when the minimum support decreases to 11%, the maximal frequent itemsets become longer. This forced the Apriori algorithm to take more passes (9 passes) and consider more candidates to discover the maximum frequent set. In contrast, MFCS allowed our algorithm to reach the maximal frequent itemsets faster. Pincer-Search took only 4 passes and considered fewer candidates to discover all maximal frequent itemsets.

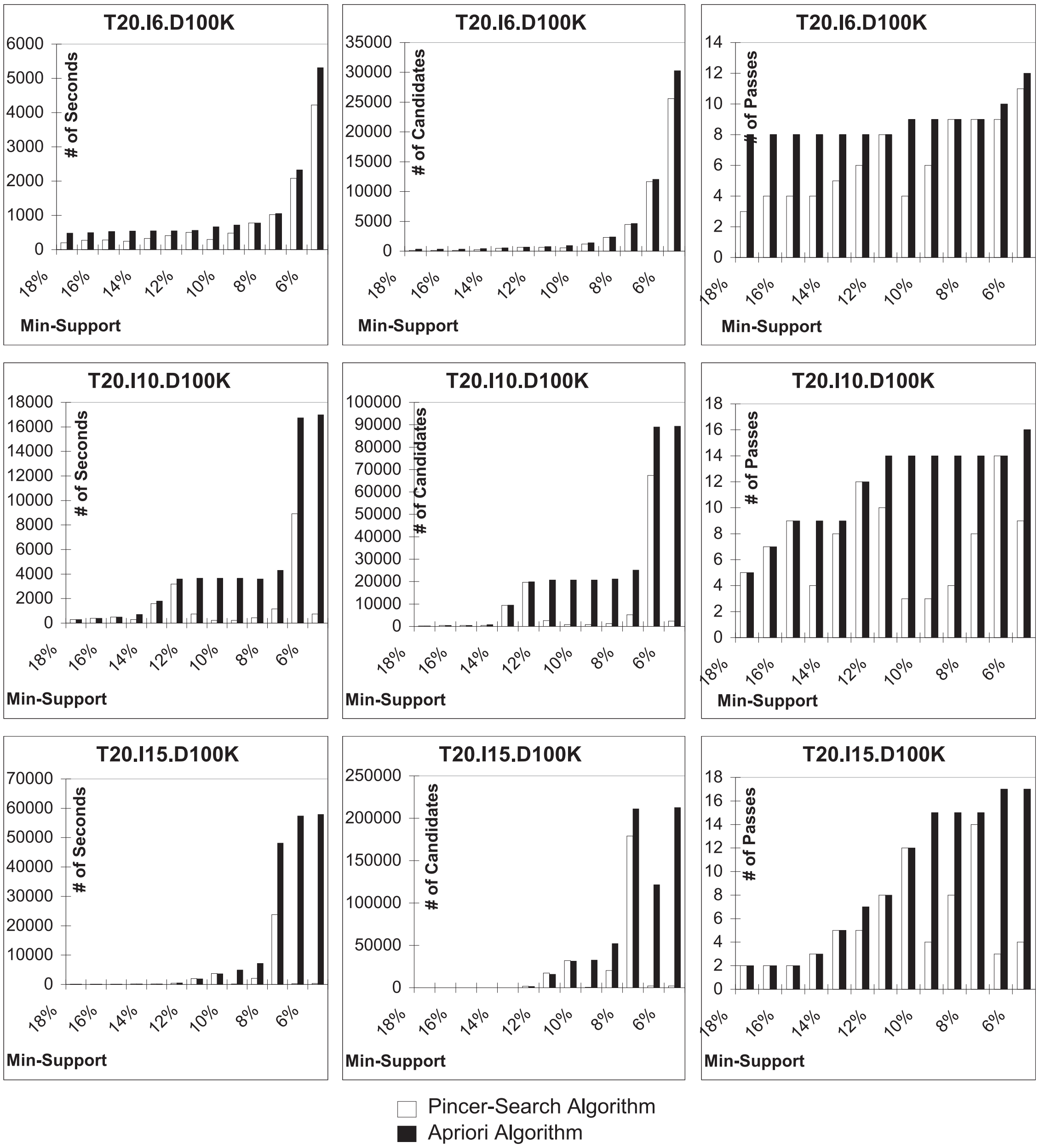
We further increased the average size of the frequent itemsets in the next two experiments. The average size of the maximal frequent itemsets was increased to 10 in the second experiment and database T20.I10.D100K was used. The best case, in this experiment, is when the minimum support is 6%. Pincer-Search ran approximately 23 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. As shown in the last row of Figure 4, Pincer-Search took as few as 3 passes to discover all maximal frequent itemsets which contain as many as 17 items. This experiment shows improvements of more than 2 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.




*|L| is 2000 in these experiments
*The number of candidates for both algorithms does not include the candidates in the first two passes
*The number of candidates for the Pincer algorithm includes the candidates in the MFCS

**Figure 3. Relative time, candidates, and passes (scattered distribution)**




*|L| is 50 in these experiments
*The number of candidates for both algorithms does not include the candidates in the first two passes
*The number of candidates for the Pincer algorithm includes the candidates in the MFCS

**Figure 4. Relative time, candidates, and passes (concentrated distribution)**

# 5 Related Work

There has been extensive research on designing association rule mining algorithms ([1] [3] [7] [9] [12] [16] [18]). Some papers concentrate on parallel algorithms ([4] [9] [16]). Other papers focus on mining generalized association rules ([6] [15]). Another direction, as described in [2] and [11], is to provide a unified model for the process of classification, association, and sequence rules discovery. The discovery of frequent set is an important process in solving these problems. Our approach can be applied to solve these problems.

A randomized algorithm for discovering the maximum frequent set was presented by Gunopulos *et al.* [5]. We present a deterministic algorithm for solving this problem.

Reading the database repeatedly can be very time consuming. Attempts have been made to reduce the number of passes in which the database is read. Some ([3] [12] [9]) proposed combining candidates from different levels to reduce the number of database readings. However, this technique is only useful in the later passes of the frequent itemsets discovery process. Others, like *Partition* [16] and *Sampling* [18]), proposed effective ways to reduce the I/O time. However, they are still inefficient when the maximal frequent itemsets are long. The Pincer-Search algorithm presents a new approach that can reduce both I/O and CPU time. This algorithm is useful in the early passes and is good for cases when the maximal frequent itemsets are long.

Mannila and Toivonen [11] analyze the complexity of the bottom-up breadth-first search style algorithms. 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* in Mitchell's machine learning paper [8]. 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* in his terminology) efficiently.

# 6 Concluding Remarks

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, 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 sets can be easily generated.

In this paper, we presented a new algorithm that can efficiently discover the maximum frequent set. Our Pincer-Search algorithm can reduce both the number of times the database is read and the number of candidates considered. Experiments shown 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 chages of individual stocks in a stock market. Prices of individual stocks are frequently quite correlated with each other (the market as a whole, goes up or down). Therefore, the discovered patterns may contain many items (stocks) and the frequent itemsets are long. Here, our algorithm could of great importance.

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

# 7 Acknowledgments

We would like to thank Rakesh Agrawal and Ramakrishnan Srikant for kindly providing us the synthetic data generation program. We would like to thank Sridhar Ramaswamy for his valuable comments and suggestions.

# References

[1] R. Agrawal, T. Imielinski, and R. Srikant. Mining association rules between sets of items in large databases. *SIGMOD*, May 1993.

[2] R. Agrawal, T. Imielinski, and R. Srikant. Database mining: A performance perspective. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 5, No. 6, Dec. 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. C. Shafer. Parallel Mining of association rules: Design, Implementation and Experience. *IBM Research Report RJ10004*, Feb. 1996.

[5] D. Gunopulos, H. Mannila, and S. Saluja. Discovering all most specific sentences by randomized algorithm. In *Proc. International Conference of Database Theory*, Jan. 1997.

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

[7] M. Klemettinen, H. Mannila, P. Ronkainen, H. Toivonen, and A. I. Berkamo. Finding interesting rules from large sets of discovered association rules. In *Proc. Third International Conference on Information and Knowledge Management*, Nov. 1994.

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

[9] 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.

[10] H. Mannila and H. Toivonen. In *Proc. Second International Conference on Knowledge Discovery and Data Mining (KDD'96)* AAAI Press. 1996.

[11] H. Mannila and H. Toivonen. Levelwise search and borders of theories in knowledge discovery. *Technical Report C-1997-8* of the Department of Computer Science, University of Helsinki, Finland, 1997.

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

[13] B. Özden, S. Ramaswamy. and A. Silberschatz. Cyclic Association Rules. Personal communication.

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

[15] R. Srikant and R. Agrawal. Mining generalized association rules. In *21st VLDB*, Sept. 1995.

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

[17] H. Toivonen. Discovery of frequent patterns in large data collections. *Technical Report A-1996-5* of the Department of Computer Science, University of Helsinki, Finland, 1996.

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